Recent Developments in the Design and Implementation of Programming Languages

Gabbrielli's Festschrift, November 27, 2020, Bologna, Italy

Edited by
Frank S. de Boer
Jacopo Mauro

Editors

Frank S. de Boer
Centrum Wiskunde & Informatica (CWI), Amsterdam, The Netherlands
F.S.de.Boer@cwi.nl

Jacopo Mauro
University of Southern Denmark, Odense, Syddanmark, Denmark
mauro.jacopo@gmail.com

ACM Classification 2012
Software and its engineering → General programming languages

ISBN 978-3-95977-171-9

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-171-9.

Publication date
November, 2020

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

License
This work is licensed under a Creative Commons Attribution 3.0 Unported license (CC-BY 3.0): https://creativecommons.org/licenses/by/3.0/legalcode.

In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:
- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/OASIcs.Gabbrielli.2020.0

OASIcs – OpenAccess Series in Informatics

OASIcs aims at a suitable publication venue to publish peer-reviewed collections of papers emerging from a scientific event. OASIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

Editorial Board

- Daniel Cremers (TU München, Germany)
- Barbara Hammer (Universität Bielefeld, Germany)
- Marc Langheinrich (Università della Svizzera Italiana – Lugano, Switzerland)
- Dorothea Wagner (*Editor-in-Chief*, Karlsruher Institut für Technologie, Germany)

ISSN 1868-8969

https://www.dagstuhl.de/oasics

To Maurizio, esteemed colleague, beloved friend, and inspiring mentor.

Contents

Preface
Frank S. de Boer and Jacopo Mauro 0:ix–0:xi

List of Reviewers
.... 0:xiii

Papers

Locally Static, Globally Dynamic Session Types for Active Objects
Reiner Hähnle, Anton W. Haubner, and Eduard Kamburjan 1:1–1:24

A Formal Analysis of the Bitcoin Protocol
Cosimo Laneve and Adele Veschetti 2:1–2:17

Deconfined Intersection Types in Java
Mariangiola Dezani-Ciancaglini, Paola Giannini, and Betti Venneri 3:1–3:25

Towards a Unifying Framework for Tuning Analysis Precision by Program Transformation
Mila Dalla Preda 4:1–4:22

The Servers of Serverless Computing: A Formal Revisitation of Functions as a Service
Saverio Giallorenzo, Ivan Lanese, Fabrizio Montesi, Davide Sangiorgi, and Stefano Pio Zingaro 5:1–5:21

A Logic Programming Approach to Reaction Systems
Moreno Falaschi and Giulia Palma 6:1–6:15

Abstract Interpretation, Symbolic Execution and Constraints
Roberto Amadini, Graeme Gange, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey 7:1–7:19

The Standard Model for Programming Languages: The Birth of a Mathematical Theory of Computation
Simone Martini 8:1–8:13

A Concurrent Language for Argumentation: Preliminary Notes
Stefano Bistarelli and Carlo Taticchi 9:1–9:22

Inseguendo Fagiani Selvatici: Partial Order Reduction for Guarded Command Languages
Frank S. de Boer, Einar Broch Johnsen, Rudolf Schlatte, Silvia Lizeth Tapia Tarifa, and Lars Tveito 10:1–10:18

Derivation of Constraints from Machine Learning Models and Applications to Security and Privacy
Moreno Falaschi, Catuscia Palamidessi, and Marco Romanelli 11:1–11:20

Adaptive Real Time IoT Stream Processing in Microservices Architecture
Luca Bixio, Giorgio Delzanno, Stefano Rebora, and Matteo Rulli 12:1–12:20

Recent Developments in the Design and Implementation of Programming Languages.
Editors: Frank S. de Boer and Jacopo Mauro

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

This festschrift celebrates the 60th anniversary of Professor Maurizio Gabbrielli.

Maurizio received his PhD in Computer Science in 1992 and after working at CWI in the Netherlands and at the University of Udine, he joined the Department of Computer Science and Engineering of the University of Bologna where he became a full professor of Computer Science in 2001, commuting almost every day from his home at the top of a hill in Prato, Tuscany. Maurizio is internationally acknowledged for his outstanding and fundamental contributions to the semantics of declarative programming languages, notably logic and constraint programming languages. His first major achievements concerned the development of new semantic foundations of logic and constraint programs for modelling various aspects like compositionality, synchronization in concurrent constraint calculi, and real-time. From mid-90s Maurizio contributed to the major problems of verification and transformation of concurrent and timed constraint programs. Currently, his general semantic interests on modularity and compositionality focus on choreographies and languages for controlling and specifying agent cooperation, with IoT applications and web apps.

A general motivation underlying his scientific career is the persistent interest in the fundamental semantic concepts which form the basis for the modular and compositional verification of computational systems. With his research, Maurizio contributed to our understanding and increased control of the complexity of such systems.

Maurizio has been (and still is) a very active member of the scientific community, both nationally and internationally. He has been Director of the European EIT Digital Doctoral School from 2015 to 2017 and the Director of Studies for the Master's in AI & Digital Technology Management at University of Bologna. Furthermore, he has been President of the Italian Association for Logic Programming (GULP), member of the advisory board of the journal Theory and Practice of Logic Programming (TPLP), member of the Executive Committee of the Association for Logic Programming, chair of the Steering Committee of the ACM conference Principles and Practice of Declarative Programming (PPDP,) member of the board of the European Association for Programming Languages and Systems (EAPLS). Moreover, his keen interest in innovation and applications led to a close collaboration with companies and start-ups. For example, Maurizio is a partner of italianaSoftware, a spin-off of the University of Bologna producing technology for microservices and Cloud computing.

Finally, his keen interest in innovation and applications led to a close collaboration with companies and start-ups. For example, Maurizio is a partner of italianaSoftware, a spin-off of the University of Bologna producing technology for microservices and Cloud computing.

On a more personal note, we believe that whoever has had the pleasure of having met Maurizio must have been impressed by his high moral standards and his integrity. Abstracting from the details, this is best illustrated by the episode, happened a long time ago, in which Maurizio was considering to apply for some academic position (somewhere). While doing so, his major concern on whether to apply or not was not on whether he would qualify, but that some other candidate would be better qualified and a better fit for the job, or simply more entitled to it because of seniority.

Maurizio survived all these years with his remarkable cool attitude, reminiscent of the American actor Humphrey Bogard, which explains why in the early days he was called the "Bogard of Computer Science". Despite this cool attitude we were still able to detect some vulnerabilities, mostly age related. One of the possibly most severe mental havocs he

Recent Developments in the Design and Implementation of Programming Languages.
Editors: Frank S. de Boer and Jacopo Mauro
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

experienced in his career may very well have occurred when, in a meeting with one of his PhD students, the PhD student mentioned that he was supposed to visit his mother for her birthday. Maurizio responded by asking naively the age of the mother, apparently not prepared for the answer. It required all his mental resources to recover from the realization that actually he could have been the grandfather of the student. In the end, Maurizio found inspiration and consolation from Terenzio's "senectus ipsa est morbus" which helped him to come to grips with the above incident. Celebrating his 60th birthday, we would like to conclude reminding him of "unus dies hominum eruditorum plus patet, quam imperiti longissima aetas" and wishing him many days of "hominum eruditorum" more.

Personal note by Moreno Falaschi: From music to logic in computer science, Artimino, and market shares

I first met Maurizio Gabbrielli when he started his doctoral program at the Dipartimento di Informatica of the University of Pisa. I was a researcher from a short time. Maurizio seemed to me an unusual student of computer science. I had the impression that he was more keen towards literature and humanistic studies. Some time later, when we became good friends, he told me that 'computer science' was a kind of makeshift for him, as it would have been anything except music. Indeed, his real passion was playing piano, and classical music, and he would have liked to devote his life to music. The story is that he was studying piano in the conservatorio when an unfortunate accident caused a damage to his acoustical nerve and he had to stop playing. Now, after seeing the brilliant career and wonderful scientific results that he obtained in computer science, I wonder what kind of fantastic piano player he could have been. Maurizio and I when we met were working in the field of logic programming and we enjoyed working in that area. We used to take jokes about ourselves saying that we were T_P-ologists, meaning that we were clever to define transformations on logic programs and find their least fixpoint semantics. We were critical about ourselves as we thought that our work was really abstract and we could not see immediate real applications. Later on he started working with constraint solving and developed lots of real applications. Maurizio is really eclectic and nice and is capable of doing many activities at the same time. He writes great scientific works, we know that. But my impression is that every time that you call him he is in a different country, presenting papers, participating in scientific meetings, or simply travelling. Nevertheless he can invite you at his place for a fantastic 'gourmet' dinner that he cooks, in the Tuscany countryside near Artimino Medici's Villa. Ah, I was forgetting... it will be with his own Carmignano wine and the dinner will be with the products of his land at km-0. Recently he got interested in playing with market shares, mainly for fun. So we might say that he found another real application after his initial fully theoretical approach. He uses a methodology that seems due to a scientific study (probably he applies the choreographies of constraint solvers) with some moves which appear more of the kind of a poker player. Most of the times he wins. I'm sure that Maurizio has a lot of other surprising applications to show us in the next few years, that I cannot imagine right now. For the moment I just wish Maurizio a happy birthday.

Personal note by Catuscia Palamidessi (in Italian)

Caro Maurizio,

Non sono mai stata brava ad esprimere i miei sentimenti per iscritto, quindi ti prego di accettare queste poche righe che, anche se semplici, sono sincere e sentite.

Sebbene viviamo lontani da tanto tempo, ti ho sempre considerato il mio migliore amico, e il mio punto di riferimento morale. Ripenso spesso a quando eravamo insieme a Pisa, condividendo i momenti belli e brutti delle prime esperienze di vita accademica. Ripenso alla nostra solidarietà di fronte alle cose che consideravamo ingiuste, alle nostre riflessioni e discussioni, e, perché no, ai nostri litigi. Ti ricordi? Litigavamo soprattutto sulle questioni etiche, perché tu sei sempre stato molto idealista, e io volevo riportanti sulla terra, ma in fondo, anche se non volevo ammetterlo, il più delle volte ero sostanzialmente d'accordo con te. E poi, siamo toscani: la franchezza e i litigi fanno parte dell'amicizia vera. In ogni caso è anche per questo che, come dicevo, ti considero il mio punto di riferimento morale, ed ogni volta che devo prendere una decisione che ha dei risvolti etici penso a quello che diresti tu.

Quanto anni sono passati... le nostre strade si sono separate, ma quei tempi di Pisa, penso, ci hanno unito per sempre.

Catuscia

List of Reviewers

Adele Veschetti

Betti Venneri

Catuscia Palamidessi

Cinzia Di Giusto

Davide Sangiorgi

Einar Broch Johnsen

Fabrizio Montesi

Gianluigi Zavattaro

Giorgio Delzanno

Ivan Lanese

Mariangiola Dezani-Ciancaglini

Mila Dalla Preda

Moreno Falaschi

Paola Mello

Reiner Hähnle

Roberto Amadini

Roberto Bagnara

Saverio Giallorenzo

Simone Martini

Stefano Bistarelli

Ugo Dal Lago

Locally Static, Globally Dynamic Session Types for Active Objects

Reiner Hähnle
Technical University Darmstadt, Germany
reiner.haehnle@tu-darmstadt.de

Anton W. Haubner
Technical University Darmstadt, Germany
anton.haubner@stud.tu-darmstadt.de

Eduard Kamburjan
Technical University Darmstadt, Germany
kamburjan@cs.tu-darmstadt.de

Abstract

Active object languages offer an attractive trade-off between low-level, preemptive concurrency and fully distributed actors: syntactically identifiable atomic code segments and asynchronous calls are the basis of cooperative concurrency, still permitting interleaving, but nevertheless being mechanically analyzable. The challenge is to reconcile local static analysis of atomic segments with the global scheduling constraints it depends on. Here, we propose an approximate, hybrid approach; At compile-time we perform a local static analysis: later, any run not complying to a global specification is excluded via runtime checks. That specification is expressed in a type-theoretic language inspired by session types. The approach reverses the usual (first global, then local) order of analysis and, thereby, supports analysis of open distributed systems.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Object oriented constructs; Theory of computation → Type structures

Keywords and phrases Session Types, Active Objects, Runtime Verification, Static Verification

Digital Object Identifier 10.4230/OASIcs.Gabbrielli.2020.1

Funding This research is supported by the *Constraint-Based Operational Consistency of Evolving Software Systems* (COCoS) project, funded by the DFG as project 351097374.

For Maurizio Gabbrielli:
"Les raisins, ou la mort!"

1 Introduction

Lately, programming languages based on actors and *active objects* (AO) attracted a lot of interest in both academia and industry. Active objects [11] are an object-oriented modeling formalism, extending the actor model of distributed systems [21]. One prominent representative, the *abstract behavioral specification* (ABS) [24] language, was successfully applied in a variety of domains, ranging from railway operations [31] to cloud-based systems [40].

One of the advantages of ABS is its rich analysis framework with tools based on dataflow and graph analyses [3], deductive verification [13], and behavioral types [18]. However, for the time being, there is no support for code generation from scheduling policies (except user-defined schedulers at the object level), or for runtime verification beyond simple **assert** statements. The reasons lie in the AO (ABS) concurrency model.

Communication between Active Objects. An Active Object is a strongly encapsulated entity whose fields can only be accessed by getter and setter methods. Like an object in standard OO, an AO declares a set of methods, including constructors. Its peculiarity is that

© Reiner Hähnle, Anton W. Haubner, and Eduard Kamburjan;
licensed under Creative Commons License CC-BY

each method consists of *syntactically* marked atomic segments whose execution cannot be preempted. At most one task, executing the code of an atomic segment, is active at any time on the object's single processor. The advantage is that each atomic segment functions as a sequential program and can be analyzed (method-)*locally* in a *modular* fashion. However, its scheduling condition may depend on the availability of results provided by other methods, not necessarily from the same object. But such synchronization patterns can only be understood from an *object*-local or even *global* (program-wide) perspective. This is bad news for the local analysis of atomic segments as well, because in general they require information about previously scheduled tasks in order to guarantee meaningful properties. This dependency is an obstacle to any modular, global analysis. While it is possible to write sufficiently strong local contracts [30], it is a difficult, manual task, which does not align well with a top-down design starting from global communication patterns. In consequence, the ability to verify closed systems (that do not interact with their environment) is limited. Even worse, the analysis of open `ABS` models is generally impossible.

Additionally to cooperative method contracts [30], `ABS` currently uses Session Types [28, 29] for the verification of communication patterns. Both approaches consist of a *local* part that analyzes the code of single methods and a *global* part, feeding into it, that analyzes scheduling, synchronization, and messages. The global part is significantly more imprecise than the local one, because it abstracts away from functional behavior. The local specification is related to the global specification via a process called projection (from objects down to methods and atomic segments): the composition principle of the analysis follows the composition principle of the AO concurrency model [19].

Locally Static, Globally Dynamic Approach. In this paper we reverse the analysis sequence and partially move it from compile-time to runtime, resulting in a *hybrid* verification method. Specifically, local analysis is done statically, at compile-time, while global analysis is performed *later* at runtime. This is achieved by a modification of the workflow of session types: Classically, projection ensures that messages always arrive in their correct order. We retain projection, but only infer the *correct message order per object*, then construct a scheduler that enforces this order.

Local static checks permit to verify open systems: a locally specified `ABS` model provides only methods that perform locally correct steps, while at runtime it is ensured that methods are called correctly and in correct order. The downside is, obviously, that global errors are only detected at runtime, however, in an open system this is the only option. The second limitation of our approach is that it is not designed to perform full functional verification of state invariants, unlike interactive, deductive verification [30]. We aim at a lightweight, fully automatic method that nevertheless allows to express non-trivial properties and facilitates top-down design of distributed systems.

Yet, our approach does not modify the `ABS` concurrency model and requires as the single extension the availability of user-defined schedulers, i.e., the ability to reject certain task sequences. From the point of modularity, we can now verify *object*-local behavior. We implemented and evaluated our approach and illustrate with a case study that it is possible to ensure an open system always follows a given protocol.

Structure. In Sect. 2 we introduce active objects, `ABS`, and a suitable notion of session types. Sect. 3 describes scheduler generation and instrumentation, Sect. 4 describes and evaluates the implementation. In Sect. 5 we discuss related approaches. Finally, Sect. 6 concludes and gives future work. For space reasons, here we can describe the main ideas only with a limited degree of precision. A fully formal treatment is found in the report [20].

2 Active Objects and Session Types

The concurrency model of AO, as explained above, rests on a syntactically identifiable notion of atomic code segments that cannot be preempted. Together with strong encapsulation, this ensures that an object's state can only be modified by its own methods (including setters) and any state change must adhere to the local specification of an atomic segment. This is the basis to establish an object's invariant by suitable, *cooperative scheduling* of methods and their atomic segments. To make this work it is necessary to call a method of another (or even the own) object *without blocking*.

All active object languages, therefore, feature non-blocking, *asynchronous* method calls that return a future [4], a handle to the task executing the call.

Grammar	
Prgm ::= $\overrightarrow{\mathsf{ID}}\ \overrightarrow{\mathsf{CD}}$ Main ID ::= **interface** I [**extends** $\overrightarrow{\mathtt{I}}$]?{$\overrightarrow{\mathsf{MS}}$}	Program, Interfaces
CD ::= **class** C [**implements** $\overrightarrow{\mathtt{I}}$]? [($\overrightarrow{\mathtt{T\ f}}$)]?{$\overrightarrow{\mathsf{FD}}\ \overrightarrow{\mathsf{Met}}$ Run} Main ::= {s}	Classes, Main
Run ::= `Unit run() {s}` FD ::= `T f = e`	Run Method, Fields
MS ::= T m($\overrightarrow{\mathtt{T\ v}}$) Met ::= MS {s; **return** e;}	Signatures, Methods
s ::= **while** (e) {s} | **if** (e) {s} [**else** {s}]? | s; s	
| **case** (e) {$\overrightarrow{\mathtt{e => s;}}$} | **await** g | [T? e]? = rhs	Statements
g ::= e? rhs ::= e | **new** C($\overrightarrow{\mathtt{e}}$) | e.**get** | e!m($\overrightarrow{\mathtt{e}}$)	Guards and RHS's

Figure 1 ABS grammar. T ranges over types, I over interfaces and C over classes.

The various AO languages differ in the details of how synchronization is performed, so we now turn to their specific realization in ABS. The syntax of ABS is given by the grammar in Fig. 1. With e we denote standard expressions over fields f, variables v and operators `|`, `&`, `>=`, `<=`, `+`, `-`, `*`, `/`. Additionally, we use an expression `destiny` to access the currently computed future. Types T are all interface names (ABS enforces programming to interfaces), type-generic futures `Fut<T>`, lists `List<T>`, `Int`, `Unit`, and `Bool`. We also assume the usual functions for lists, etc.

In the final expression of the rule for rhs, the syntax for asynchronous method calls is shown (for simplicity, we leave out standard synchronous calls). As usual, a "!" replaces the dot. Asynchronous calls are always executable. Their result is a *future* of type `Fut<T>`, where T is the return type of m. The *effect* of an asynchronous call is to create a task to execute m's body in e's object o, to be scheduled at some time in the future. In case o is also the caller, obviously the calling method must first suspend, before the callee can be scheduled. Asynchronous calls occur only as right-hand side expressions, so the future is stored in a field (or variable) f. Once the result of the computation performed by m($\overrightarrow{\mathtt{e}}$) is ready, it can be retrieved with the expression f.**get**. If the result is *not* ready, the **get** expression blocks the calling object. This can easily lead to a deadlock, so one typically *guards* a **get** expression with an **await** statement of the form **await** e?, where e's type is of the form `Fut<T>`. The effect is that execution of the current task is suspended and only rescheduled after the result of e is ready. The **await** statement and the syntactic end of a method block *are the only places, where task suspension in* ABS *can occur.* This justifies the following definition:

▶ **Definition 1** (Atomic Segment). *Code sequences starting either at the syntactic beginning of a method body or at the statement right after an* **await** *statement and ending either at the syntactic end of a method body or with an* **await** *statement, such that they contain at most the* **await** *statement at the end, are called* atomic segment.

Generally, ABS programs follow a simple, but standard OO paradigm in any other aspect: A program contains a main method Main, interfaces $\overrightarrow{\mathsf{ID}}$ and classes $\overrightarrow{\mathsf{CD}}$. Interfaces are standard, the main method contains a list of object creations. Classes can have parameters $\overrightarrow{\mathsf{Tf}}$, these are fields being initialized during object creation. The parameter type may have the form **Fut**<T>, i.e., futures may be passed as arguments to other methods. Classes have fields $\overrightarrow{\mathsf{FD}}$, methods $\overrightarrow{\mathsf{Met}}$, and a run method Run to start a process.

▶ **Example 2.** Let us illustrate cooperative scheduling in ABS with the example in Fig. 2. The program models the behavior of a mail server with notification service. It consists of three objects created in the main block: a mail server `m`, a user interface `u`, and a notification service `n`, which knows both the mail server and the user interface. Then the notification service's only method `init()` is run on `n`. The method has a single loop that periodically checks whether mail arrived and, if this is the case, notifies the user via interface `u`. Checking for mail and notifying the user require asynchronous calls to `m` and `u`, respectively, so we allocate suitable fields `fCheckMail` and `fPopup` of future type. Checking the mail must be finished before notification is handled. This is ensured by the **await** statement in Line 12. At this point, `init()` suspends. In the example, `init()` is the only method executing on `m`, so the processor will be simply idle, but it is conceivable that the main method starts other tasks on the mail server which at this point can be interleaved. Checking for mail is modelled by randomly choosing one of the literals `Mail` or `NoMail` as a return value.

Once the `response` is available, the user is notified in case there is mail, otherwise, nothing happens. Since the call to `popup()` is asynchronous, in the absence of a defined scheduler, the sequence of multiple calls to `popup()` is not necessarily in the order of mail arriving. However, the code ensures that the number of completed or active calls to `popup()` is always less than the completed calls to `checkMail()`, that there can be at most one call to `popup()` between any two calls to `checkMail()`, etc. We will show that session types are suitable to specify such global behavior in a succinct way, which then can be enforced at runtime.

Before we define session types for AO, we need to set up the machinery of user-defined schedulers needed to implement runtime checks.

A user-defined scheduler [5] is a side-effect free function in ABS that takes as parameters (1) a list of schedulable *processes* and (2) several fields of its class. It returns either `Nothing` or `Just(p)`, where `p` is one of the processes in the input list. The return value controls scheduling: if `Nothing` is returned, no process is scheduled, otherwise the chosen process is scheduled next. A process is represented as an abstract data type `Process`, i.e., an ADT that cannot be constructed manually. Instead one can access the future of a process with `destinyOf(p)` and its methodname as a `String` with `method(p)`.

Lists are also ADTs and `nth(input,i)` returns the `i`-th element. Keyword **def** is used to define a function with parameters that evaluates a result using standard expressions (and recursion). ABS does not support fully-fledged functional programming and only a fixed set of higher-order functions. For example, higher-order functions such as `map` and `filter` are part of the ABS standard library.

▶ **Example 3.** Let us consider the following scheduler and class.

```
def Maybe<Process> scheduler(List<Process> input, Int y, String m) =
  if ( y < 0 || y >= length(input) ) Nothing else
    if ( method(nth(input,y)) != m) Just(nth(input,y) ) else Nothing;

[Scheduler: scheduler(queue, y, m)]
class C(String m, Int y) { ... }
```

```
1 data Msg = Mail | NoMail;
2
3 class NotifyService
4       (MailServerI m, UII u)
5       implements NotifyServiceI {
6  Fut<Msg>  fCheckMail;
7  Fut<Unit> fPopup;
8  Unit init(int bound) {
9   Int i = 0;
10  while (i < bound) {
11   fCheckMail = m!checkMail();
12   await fCheckMail?;
13   Msg response = fCheckMail.get;
14   case (response) {
15    Mail => fPopup = u!popup(i);
16    NoMail => skip;
17   }
18   i = i + 1;
19  }
20 }
21 }
22 class MailServer
23       implements MailServerI {
24  Msg checkMail() {
25   Msg result = NoMail;
26   if (random(2) == 1)
27       result = Mail;
28   return result;
29  }
30 }
31 class UI implements UII {
32  Unit popup(int id) {
33   println("You got mail! Id " + id);
34  }
35 }
36 { // Main block
37  MailServerI m = new MailServer() ;
38  UII u = new UI();
39  NotifyServiceI n
40     = new NotifyService(m,u);
41  await n!init(42);
42 }
```

Figure 2 Mail server example in ABS.

`y` and `m`, the field names and their types, must be identical in class and scheduler function to use the scheduler. The code above selects the `y`-th element in the input list, unless it is out of range or a process executing a method named `m`. The annotation `[Scheduler: scheduler( queue, y, m)]` connects scheduler and class. Whenever the scheduler is invoked, the input list is guaranteed to be non-empty.

2.1 Session Types for Active Objects

Session types specify and verify the behavior of a closed unit of communication, called a *session*. A session type specification consists of three parts: (1) global types, a global specification of the session, (2) local types, specifications for the endpoints in a session, and (3) a projection mechanism that generates a local specification for each endpoint participating in the communication from a global type. For checking that the whole unit adheres to its global specification, it suffices to check the local endpoints and, possibly, side-conditions on the unit. Additionally, the session type system needs some kind of mechanism to ensure that the local endpoints adhere to their local type.

In the original formulation for the π-calculus [8] a session is centered around a channel, endpoints are the processes participating in the communication over the typed channel. To ensure that projection succeeds, a linearity check on the channel is performed as a side-condition of projection.

For Active Objects, the situation changes: there are no channels and endpoints participating in any communication are not uniform, because the target of a method call is an object, but the target of a future read is a (terminated) process. As there are no channels, a linearity check to ensure that messages arrive in the right order is impossible.

Session types for AO [28, 29] adopt and adapt the concepts of session types for channels:

Unit of Communication: The unit of communication is described by a set of objects that only contain pointers to each other.

Endpoints: The notion of endpoint is two-fold. Both objects and processes are endpoints and the projection of a global session type first projects on the object and then projects one more time on the processes inside that object. The result of the first projection is an *object-local* type and the result of the second projection is a *method-local* type.[1]

Order of Messages: To ensure messages arrive in the correct order, a static analysis can be used to determine whether the order of messages is total from the perspective of each object (but not globally).

Here we remove the check on message order at the level of the type system and instead enforce it at runtime using the structure provided by the object-local type. Before we introduce syntax of global and local types, it is worth mentioning that we are only concerned with *protocol adherence*: Does the system implement the protocol described by the global type? We ignore *deadlock freedom*, which can be approached either with session types [29] or a dedicated deadlock checker for AO [15, 18, 25]. The system we introducing below is a slight variation of the session types in [29].

2.1.1 Global Types

Global types follow the structure of regular expressions and allow Kleene star-style repetition, sequence and branching. Branching is guarded by a single role that determines which branch of the protocol to follow. As single actions, the type defines a certain kind of interaction between two roles or a role and a process/future. To keep track of processes and futures within a protocol, we use tracked futures: references to the future of a specified method call.

▶ **Definition 4.** *Let* $\mathbf{p}$*,* $\mathbf{q}$ *range over roles,* t *over tracked futures and* C *over ADT constructors. The syntax of global protocols* $\mathbf{GP}$ *and global types* $\mathbf{G}$ *is defined in Fig. 3. Specifications* $(\!|\cdot|\!)$ *are all optional.*

$\mathbf{GP} ::= \mathbf{0} \xrightarrow{\mathsf{t}} \mathbf{p}\!:\!\mathtt{m}\ .\ \mathbf{G}$	Global Protocol
$\mathbf{G} ::= \mathbf{p} \xrightarrow{\mathsf{t}} \mathbf{q}\!:\!\mathtt{m} \mid \mathbf{p}\!\downarrow\!\mathsf{t}(\!|\mathsf{C}|\!) \mid \mathbf{p}\!\uparrow\!\mathsf{t}(\!|\mathsf{C}|\!)$	Call, Termination and Synchronization Action
$\mid \mathsf{Rel}(\mathbf{p},\mathsf{t}) \mid \mathbf{skip}$	Suspension and Empty Action
$\mid \mathbf{p}\{\mathbf{G}_i\}_{i\in I} \mid (\mathbf{G})^* \mid \mathbf{G}\ .\ \mathbf{G}$	Branching, Repetition and Sequential Composition

Figure 3 Syntax of Global Session Types.

The global protocol starts with $\mathbf{0} \xrightarrow{\mathsf{t}} \mathbf{p}\!:\!\mathtt{m}$ and specifies how the session is started. The call action $\mathbf{p} \xrightarrow{\mathsf{t}} \mathbf{q}\!:\!\mathtt{m}$ specifies a call from the object with role $\mathbf{p}$ to the one with role $\mathbf{q}$ on method `m`. This process is tracked by t in the rest of the type. The termination action $\mathbf{p}\!\downarrow\!\mathsf{t}(\!|\mathsf{C}|\!)$ specifies that the object with role $\mathbf{p}$ terminates the process tracked by t and the return value has the outermost constructor C. The synchronization action $\mathbf{p}\!\uparrow\!\mathsf{t}(\!|\mathsf{C}|\!)$ specifies that the object with role $\mathbf{p}$ reads from the future tracked by t and reads a value with the outermost constructor C. The suspension action $\mathsf{Rel}(\mathbf{p},\mathsf{t})$ specifies that the object with role $\mathbf{p}$ suspends its currently active process until the future tracked by t is resolved. We stress

[1] The projection may also be done in one step [27], but this removes the object-local types which we are investigating in this paper.

that t is *not* the tracked future of the suspended process. The empty action specifies no action and is needed to specify, e.g., branches without visible actions. Branching $\mathbf{p}\{\mathbf{G}_i\}_{i\in I}$ specifies that the object with role **p** chooses one of the branches $\mathbf{G}_i$ to continue the protocol. Finally, repetition and sequential composition are analogous to regular expressions.

▶ **Example 5.** We continue Ex. 2. The roles of the protocol are named **NS** for the notification service, **Mail** for the mail server and **GUI** for the GUI. The intended behavior is specified by the following global type:

$$\mathbf{0} \xrightarrow{t_0} \mathbf{NS}:\texttt{init} \;.$$
$$\Big(\mathbf{NS} \xrightarrow{t_1} \mathbf{Mail}:\texttt{check} \;.\; \mathsf{Rel}(\mathbf{NS}, t_1) \;.$$
$$\mathbf{Mail}\left\{\begin{array}{l} \mathbf{Mail}\downarrow t_1(\!|\texttt{NewMail}|\!) \;.\; \mathbf{NS}\uparrow t_1(\!|\texttt{NewMail}|\!) \;.\; \mathbf{NS} \xrightarrow{t_2} \mathbf{GUI}:\texttt{show} \;.\; \mathbf{GUI}\downarrow t_2 \\ \mathbf{Mail}\downarrow t_1(\!|\texttt{NoMail}|\!) \;.\; \mathbf{NS}\uparrow t_1(\!|\texttt{NoMail}|\!) \end{array}\right\}$$
$$\Big)^* \;.\; \mathbf{NS}\downarrow t_0$$

The above example demonstrates the use of tracked futures and repetition, but it is strongly synchronized: The described synchronization structure enforces correct interaction order, no deviation due to the scheduler is possible. In contrast, consider the following scenario and global session type that is not strongly synchronized.

▶ **Example 6.** The protocol describes four roles: a student **S**, a service desk **D**, a computation server **C** and a report generator **R**. The computation server computes the grade of a student, and sends it to the report generator, which in turn generates a report that is send to the service desk. The computation server notifies the student that its grade has been computed. The service desk may only serve the student after the report has arrived. This is specified by the following global protocol:

$$\mathbf{0} \xrightarrow{t_0} \mathbf{C}:\texttt{compute} \;.\; \mathbf{C} \xrightarrow{t_1} \mathbf{R}:\texttt{toReport} \;.\; \mathbf{C} \xrightarrow{t_2} \mathbf{S}:\texttt{notify} \;.\; \mathbf{R} \xrightarrow{t_3} \mathbf{D}:\texttt{publish} \;.\; \mathbf{R}\downarrow t_1$$
$$.\; \mathbf{D}\downarrow t_3 \;.\; \mathbf{S} \xrightarrow{t_4} \mathbf{D}:\texttt{request} \;.\; \mathbf{D}\downarrow t_4 \;.\; \mathbf{S}\uparrow t_4 \;.\; \mathbf{S}\downarrow t_2 \;.\; \mathbf{C}\downarrow t_0$$

Note that **D** is called on `request` and `publish` but no synchronization ensures that those messages arrive in the specified order.

2.1.2 Local Types

We distinguish object-local types, method-local types and scheduling types. Method-local and object-local types differ syntactically only in their passive choice operator and the specification of synchronization. Scheduling types describe the actions of the scheduler of a role and share their syntax with method-local types.

▶ **Definition 7.** *Let* **p** *range over roles and* **0**, m *over method names,* t *over tracked futures and* C *over ADT constructors. The syntax of object-local types* **L** *is defined as follows:*

$\mathbf{L} ::= \mathbf{p}?_t\texttt{m} \mid \mathbf{p}!_t\texttt{m} \mid \mathsf{Put}\ t(\!|\mathsf{C}|\!)$	Receiving, Sending and Termination Action
$\mathsf{Get}\ t(\!|\mathsf{C}|\!) \mid \mathsf{Susp}(t,t) \mid \mathsf{React}\ t$	Synchronization, Suspension and Reactivation Action
$\&_t\{\mathbf{L}_i\}_{i\in I} \mid \oplus\{\mathbf{L}_i\}_{i\in I}$	Passive and Active Choice
$\mathbf{skip} \mid \mathbf{L}\,.\,\mathbf{L} \mid (\mathbf{L})^*$	Empty Action, Sequential Composition and Repetition

The syntax of method-local types is analogous, but (1) the synchronization action takes no C *specification and (2) passive choice takes the following form, called* guarded *passive choice:*

$$\&_t\{\mathsf{C}_i : \mathbf{L}_i\}_{i\in I}$$

The receiving action is the callee's view on the global call action, **p** is the caller. Note that a method in ABS has no access to the caller, but we may access it in the scheduler. The sending action is the caller's view on the global call action, **p** is the callee. The local termination action is the equivalent of the global termination action. The local suspension action $\mathsf{Susp}(\mathtt{t}_1, \mathtt{t}_2)$ specifies that the process computing $\mathtt{t}_1$ suspends until $\mathtt{t}_2$ is known. The reactivation action $\mathsf{React}\ \mathtt{t}$ specifies reactivation of the process computing $\mathtt{t}$. These two actions are the local view on the global suspension action, but (1) locally the suspending process is known and (2) one can infer, where the reactivation must happen. We use three choice operators:

- (Object-Local) Unguarded passive choice $\&_{\mathtt{t}}\{\mathbf{L}_i\}_{i\in I}$ specifies that the object reacts to the choice stored in $\mathtt{t}$. The choice is stored as the C parameter of the first Get action on $\mathtt{t}$ in the given branch.
- (Method-Local) Guarded passive choice $\&_{\mathtt{t}}\{\mathsf{C}_i : \mathbf{L}_i\}_{i\in I}$ specifies which constructor corresponds to which branch directly, as it is only indirectly encoded in the unguarded passive choice.
- Active choice $\oplus\{\mathbf{L}_i\}_{i\in I}$ specifies that the object or process in question chooses one of the branches to continue. It is not specified how the choice is made.[2]

The remaining actions are analogous to their global counterpart.

We introduce projection formally in the subsequent section, but provide examples of local types based on Ex. 5, 6 now to illustrate the differences among the various local types.

▶ **Example 8.** Below is the object-local type of **Mail** in Ex. 5 followed by the method-local type of $\mathtt{t}_1$ and the scheduling type. The differences between the former are that (1) the method-local type contains *no* repetition, because the repetition is not visible to a single process and (2) the receiving action is omitted, because it is redundant when the tracked future is known.

$$\left(\mathbf{NS}?_{\mathtt{t}_1}\mathtt{check}\ .\ \oplus\left\{\begin{array}{l}\mathsf{Put}\ \mathtt{t}_1(\!|\mathtt{NewMail}|\!)\\ \mathsf{Put}\ \mathtt{t}_1(\!|\mathtt{NoMail}|\!)\end{array}\right\}\right)^* \qquad \text{Object-local type}$$

$$\oplus\left\{\begin{array}{l}\mathsf{Put}\ \mathtt{t}_1(\!|\mathtt{NewMail}|\!)\\ \mathsf{Put}\ \mathtt{t}_1(\!|\mathtt{NoMail}|\!)\end{array}\right\} \qquad \text{Method-local type}$$

$$\left(\mathbf{NS}?_{\mathtt{t}_1}\mathtt{check}\right)^* \qquad \text{Scheduling type}$$

The method-local type contains only the actions performed by the processes of a single method. A scheduling type contains only the actions needed for scheduling: empty, reactivation and receiving actions, as well as both kinds of branching, repetition and sequential compositions. The following is the scheduling type of **D** in Ex. 6: $\mathbf{R}?_{\mathtt{t}_3}\mathtt{publish}\ .\ \mathbf{S}?_{\mathtt{t}_4}\mathtt{request}$

3 LSGD Session Types

The verification workflow of our system takes a global type and generates an instrumented ABS program and a proof that each method is following its method-local type.

- First, we establish certain well-formedness conditions of the global type to ensure it describes a protocol that is realizable in the AO concurrency model.

[2] For guarded active choice we refer to Kamburjan & Chen [28].

- Then, the global type is projected on each participating object. This results in an object-local type, describing the actions an object both expects and is obliged to perform.
- From the object-local type we generate (a) a session automaton that describes the order of scheduling actions and (b) a method-local type for each method. Scheduling actions include the receiving action (receiving method calls) and the rescheduling action (reacting to a resolved future).
- The session automaton is translated into a user-defined scheduler, which is added to the object together with fields and operations to keep track of the state.
- Each method is checked statically against its method-local type.

For brevity, we give a simplified account of the implemented system [20] and omit some features, e.g., allowing interactions with objects that do not participate in the session.

3.1 Session Automata

Before defining the workflow, we introduce Session Automata [7]. Session automata are a class of register automata [33]: finite automata over an infinite alphabet. General register automata allow to store read values of infinite alphabets in registers and compare the register contents by equality. Session automata have the restriction that only *fresh values* can be stored, i.e., values that have not been seen in the input word so far. This matches our model when futures are regarded as data and allows one to decide whether two session automata accept the same language. In our system, the alphabet is the set of futures and we only store futures upon receiving a method call. This guarantees their freshness upon storage.

▶ **Definition 9.** *Let Σ be a finite set of labels, D an infinite set of data equipped with equality and $k \in \mathbb{N}$. A k-Register Session Automaton is a tuple (Q, q_0, Φ, F), where Q is the set of states, $q_0 \in Q$ its start state, $F \subseteq Q$ the set of accepting states, and the transition relation is as follows:*

$$\Phi \subseteq \big(Q \times Q\big) \cup \big(Q \times (\Sigma \times D) \times \mathcal{P}(\{1, \dots, k\}) \times \{1, \dots, k\} \times Q\big)$$

Runs of session automata are defined over stores and data words. A transition either (1) only changes the state, but neither changes the store nor consumes a letter, or (2) changes the state upon reading the next letter by comparing the data with a register in its store and storing the read data.

▶ **Definition 10.** *A store $\sigma : \{1, \dots, k\} \mapsto D \cup \{\bot\}$ is a function from register identifiers to data or the special symbol $\bot$. The initial store σ_0 maps all register identifiers to $\bot$. A data word $w = (a_0, d_0), \dots, (a_n, d_n)$ is a finite sequence of pairs of labels and data. A run $(q_0, j_0, \sigma_0), \dots, (q_m, j_m, \sigma_j)$ of a k-register session automaton (Q, q_0, Φ, F) on a word w of length n is a sequence*

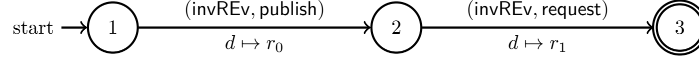
$$s \in (Q \times \mathbb{N} \times \{1, \dots, k\} \mapsto D)^*$$

where q_i is the current state, σ_i the current store and j_i the next letter. The sequence must start with $(q_0, 0, \sigma_0)$ and satisfy the following condition for each position $0 < i < m$:

$$\begin{aligned} &(q_i, q_{i+1}) \in \Phi \wedge (j_i = j_{i+1}) \wedge (\sigma_i = \sigma_{i+1}) \\ \vee &\Big((q_i, (a_{j_i}, d_{j_i}), I, k, q_{i+1}) \in \Phi \wedge (j_i = j_i + 1) \wedge \sigma_{i+1} = \sigma_i[k \setminus d_{j_i}] \wedge \forall l \in I.\ \sigma_i(l) = d_{j_i} \Big) \end{aligned}$$

In the following we set $D = \mathsf{Fut}$ and $\Sigma = \{\mathsf{invREv}\} \times \mathsf{Met} \cup \{\mathsf{condREv}\}$.

▶ **Example 11.** The following 2-register session automaton models the scheduling type of $\mathbf{D}$ in Ex. 8. The two stores of the futures in registers r_i are used to model reactivation.



For brevity, we write $(q, (\mathsf{invREv}, \mathtt{m}), q')$ and $(i, (\mathsf{condREv}), q')$ for transitions with the given label and say that register i is either written or read. We never write to or read from more than one register in a single transition.

3.2 Projection

Projection generates (1) a method-local type per participating method in the session and (2) a special object-local type, called *scheduling type*, for each role. The scheduling type describes the order of operations controlled by the scheduler, i.e., process start and rescheduling.

Projection consists of four steps: pre-analysis, projection on a role, projection on a tracked future, and generation of a scheduling type from an object-local type.

Pre-analysis: Reject obviously malformed types and annotate the global type with information used in later steps, for example, which future is currently being computed.

Projection on Role: Generate an object-local type that describes the view of a role on the global type.

Projection on Tracked Future: Generate a method-local type that describes the view of a process on the object-local type.

Generation of Scheduling Type: Generate the scheduling type that describes the operations performed by the scheduler of an object.

3.2.1 Pre-Analysis

Pre-analysis of a global type checks that it specifies a feasible protocol in the AO concurrency model. It generates an *annotated* global type $\mathbf{G}\langle\sigma\rangle$, where σ describes the specified state of a role before and after performing the specified action. We refrain from introducing all the formal details and only describe the checked properties of a global type.

Future Freshness: Each tracked future identifies exactly one call action. For example, the following type fails pre-analysis and is rejected, because t is not fresh in the second call.

$$\mathbf{0} \xrightarrow{\mathsf{t}} \mathbf{p}{:}\mathtt{m} \;.\; \mathbf{p} \xrightarrow{\mathsf{t}} \mathbf{q}{:}\mathtt{n} \;.\; \mathbf{p}{\downarrow}\mathsf{t} \;.\; \mathbf{q}{\downarrow}\mathsf{t}$$

Actor Activity: A call action can only be specified when the callee is not specified as currently executing a method and a suspending action can only suspend a process when it is specified as being active. For example, the following global type contains two errors: the call of t_2 must wait until $\mathbf{p}$ is terminated and the suspension action of $\mathbf{p}$ cannot suspend any process.

$$\mathbf{0} \xrightarrow{\mathsf{t}_0} \mathbf{p}{:}\mathtt{m} \;.\; \mathbf{p} \xrightarrow{\mathsf{t}_1} \mathbf{q}{:}\mathtt{n} \;.\; \mathbf{q} \xrightarrow{\mathsf{t}_2} \mathbf{p}{:}\mathtt{o} \;.\; \mathbf{p}{\downarrow}\mathsf{t}_0 \;.\; \mathsf{Rel}(\mathbf{p}, \mathsf{t}_1) \;.\; \mathbf{q}{\downarrow}\mathsf{t}_1 \;.\; \mathbf{p}{\downarrow}\mathsf{t}_2$$

The following is one possible "debugged" version that passes pre-analysis:

$$\mathbf{0} \xrightarrow{\mathsf{t}_0} \mathbf{p}{:}\mathtt{m} \;.\; \mathbf{p} \xrightarrow{\mathsf{t}_1} \mathbf{q}{:}\mathtt{n} \;.\; \mathsf{Rel}(\mathbf{p}, \mathsf{t}_1) \;.\; \mathbf{q} \xrightarrow{\mathsf{t}_2} \mathbf{p}{:}\mathtt{o} \;.\; \mathbf{p}{\downarrow}\mathsf{t}_2 \;.\; \mathbf{q}{\downarrow}\mathsf{t}_1 \;.\; \mathbf{p}{\downarrow}\mathsf{t}_0$$

Resolution Analysis: A future can only be read if it has terminated before and is accessible to the reading role.[3]

Scope Analysis: Repetition introduces *scopes* into the specification, as a process is started exactly once and terminated exactly once. For example, the following type is not correctly scoped, because it allows situations where (1) n is never called, and thus t_1 cannot be terminated and (2) where n is called multiple times and it is not specified how many of those processes are terminated and in which order:

$$\mathbf{0} \xrightarrow{t_0} \mathbf{p}:\mathtt{m} \ . \ (\mathbf{p} \xrightarrow{t_1} \mathbf{q}:\mathtt{n})^* \ . \ \mathbf{p}{\downarrow}t_1 \ . \ \mathbf{q}{\downarrow}t_0$$

The scope analysis checks that (1) every tracked future that is started within a repetition is resolved within the same repetition; (2) every tracked future that is resolved within a repetition is started within the same repetition; (3) every tracked future that is synchronized upon within a repetition is started within the same repetition; (4) for every role the active tracked future and the set of suspended tracked future before and after the repetition are the same. (5) every tracked future that is started within a branch is resolved within the same branch; (6) every tracked future that is resolved within a branch is started within the same branch;

During pre-analysis each global type, except sequence, is annotated with an abstract state σ. An abstract state is a mapping from roles to a pair $(\mathit{AState}, \mathit{SState})$, where AState is either $\mathsf{Active}(t)$, expressing that the role is currently specified as executing the process for t or Susp if it is currently specified inactive. SState is a set of pairs of tracked futures (t, t'), expressing that there is a suspended process for t waiting for t'. Pre-analysis ensures that there are no t_1, t_2 with $(t_1, t), (t_2, t) \in \mathit{SState}$ in any abstract state for any role, i.e., there are never two processes of one role waiting for the same future.[4]

3.2.2 Global Projection

The projection of global types on a role is defined in Fig. 4. Projection is a partial function $\mathbf{G}\!\upharpoonright\!\mathbf{p}$. It checks that any action is specified to happen when the role performing this action is active and has a process that can perform the communication. The result of projection is an object-local type, annotated with abstract states.

The initial action results in a receiving action for the callee and **skip** for any other role. Similarly, the projection of a call action is a receiving action for the callee and a sending action for the caller.

Projection of the termination action has three cases: (1) If projected on the terminating role, it is ensured that this role is active and can perform the action. The result is a local termination action. (2) If projected on a role waiting for the tracked future of the action, it is ensured that this role is inactive. The result is a reactivation action. (3) Projection on any other role results in **skip**. Projection fails if, for example, the terminating role is inactive.

Projection of synchronization results in a local synchronization action for the specified role and **skip** for any other role. It is checked that the specified role is active. The suspension action is analogous. Projection of **skip** is the identity, projection of branching results in an active choice for the specified role (which must be active) and a passive choice over the currently active future of the choosing role for any other role. Projection of the repetition

[3] On passing data in Session Types for Active Objects, we refer to [27].

[4] Because it is not *specified* in which order they should be reactivated. If such a specification were given, that order could be reflected in the projected object-local type.

$$\mathbf{0} \xrightarrow{\mathsf{t}} \mathbf{q}:\mathtt{m}\langle\sigma\rangle \restriction \mathbf{p} = \begin{cases} \mathbf{0}?_{\mathsf{t}}\mathtt{m}\langle\sigma\rangle & \text{if } \mathbf{p} = \mathbf{q} \\ \mathbf{skip} & \text{otherwise} \end{cases}$$

$$\mathbf{q} \xrightarrow{\mathsf{t}} \mathbf{r}:\mathtt{m}\langle\sigma\rangle \restriction \mathbf{p} = \begin{cases} \mathbf{r}!_{\mathsf{t}}\mathtt{m}\langle\sigma\rangle & \text{if } \mathbf{p} = \mathbf{q} \\ \mathbf{q}?_{\mathsf{t}}\mathtt{m}\langle\sigma\rangle & \text{if } \mathbf{p} = \mathbf{r} \\ \mathbf{skip} & \text{otherwise} \end{cases}$$

$$\mathbf{q}\downarrow\mathsf{t}(\!|\mathsf{C}|\!)\langle\sigma\rangle \restriction \mathbf{p} = \begin{cases} \mathsf{Put}\ \mathsf{t}(\!|\mathsf{C}|\!)\langle\sigma\rangle & \text{if } \mathbf{p} = \mathbf{q} \wedge \sigma(\mathbf{p})(\mathsf{Active}(\mathsf{t}), \mathit{SState}) \\ \mathsf{React}\ \mathsf{t}'\langle\sigma\rangle & \text{if } \mathbf{p} \neq \mathbf{q} \wedge \sigma(\mathbf{p})(\mathsf{Susp}, \mathit{SState}) \wedge (\mathsf{t}, \mathsf{t}') \in \mathit{SState} \\ \mathbf{skip} & \text{if } \mathbf{p} \neq \mathbf{q} \wedge \sigma(\mathbf{p})(\mathsf{Susp}, \mathit{SState}) \wedge \nexists \mathsf{t}'.\ (\mathsf{t}, \mathsf{t}') \in \mathit{SState} \end{cases}$$

$$\mathbf{q}\uparrow\mathsf{t}(\!|\mathsf{C}|\!)\langle\sigma\rangle \restriction \mathbf{p} = \begin{cases} \mathsf{Get}\ \mathsf{t}(\!|\mathsf{C}|\!)\langle\sigma\rangle & \text{if } \mathbf{p} = \mathbf{q} \wedge \sigma(\mathbf{p}) = (\mathsf{Active}(\mathsf{t}'), \mathit{SState}) \\ \mathbf{skip} & \text{otherwise} \end{cases}$$

$$\mathsf{Rel}(\mathbf{q}, \mathsf{t}) \restriction \mathbf{p} = \begin{cases} \mathsf{Susp}(\mathsf{t}', \mathsf{t})\langle\sigma\rangle & \text{if } \mathbf{p} = \mathbf{q} \wedge \sigma(\mathbf{p}) = (\mathsf{Active}(\mathsf{t}'), \mathit{SState}) \\ \mathbf{skip} & \text{if } \mathbf{p} \neq \mathbf{q} \end{cases}$$

$$\mathbf{q}\{\mathbf{G}_i\}_{i\in I}\langle\sigma\rangle \restriction \mathbf{p} = \begin{cases} \oplus\{\mathbf{G}_i \restriction \mathbf{p}\langle\sigma\rangle\}_{i\in I} & \text{if } \mathbf{p} = \mathbf{q} \wedge \sigma(\mathbf{p}) = (\mathsf{Active}(\mathsf{t}), \mathit{SState}) \\ \&_{\mathsf{t}}\{\mathbf{G}_i \restriction \mathbf{p}\langle\sigma\rangle\}_{i\in I} & \text{if } \mathbf{p} \neq \mathbf{q} \wedge \sigma(\mathbf{q}) = (\mathsf{Active}(\mathsf{t}), \mathit{SState}) \end{cases}$$

$$(\mathbf{G})^*\langle\sigma\rangle \restriction \mathbf{p} = \begin{cases} (\mathbf{L})^*\langle\sigma\rangle & \text{if } \mathbf{G} \restriction \mathbf{p} = \mathbf{L} \neq \mathbf{skip} \\ \mathbf{skip} & \text{otherwise} \end{cases}$$

$$(\mathbf{G}_1\ .\ \mathbf{G}_2) \restriction \mathbf{p} = (\mathbf{G}_1 \restriction \mathbf{p})\ .\ (\mathbf{G}_2 \restriction \mathbf{p}) \qquad \mathbf{skip} \restriction \mathbf{p} = \mathbf{skip}$$

Figure 4 Projection of global type on roles.

repeats the projection of the inner part if it performs some action. Otherwise, the repetition is replaced with an empty action. Finally, projection of sequential composition is sequential composition of the projected types. We assume that structural congruence is used to remove superfluous empty actions and branching.

3.2.3 Local Projection

Local projection generates a method-local type from an object-local type for each tracked future. Each tracked future is introduced by a call action, so we can easily connect method-local types to methods. For simplicity, we demand that each method has only one type.

Local projection must invert the relation between passive choice and synchronization. A global type specifies first the choice and marks the future of the choosing role during global projection. Afterwards, the future is resolved and may be synchronized upon. Locally, however, the method synchronizes *first* and *then* branches depending on the read value. Local projection handles this by pulling out the prefix of all branches from a passive choice up to the synchronization action over the choosing future.

▶ **Definition 12.** *Let* t *be a tracked future and* $\mathbf{L}_i$ *a set of object-local types. The prefix for* t *of some object-local* $\mathbf{L}$ *is defined as the shortest type* $\mathbf{L}_{\mathsf{t}}$ *that ends in* $\mathsf{Get}\ \mathsf{t}(\!|\mathsf{C}|\!)$*: The function* $\mathsf{split}_{\mathsf{t}}(\mathbf{L})$ *returns the prefix and the remaining postfix of a type.*

$$\mathsf{split}_{\mathsf{t}}(\mathbf{L}) = (\mathbf{L}_{\mathsf{head}}, \mathsf{C}, \mathbf{L}_{\mathsf{tail}}) \text{ such that}$$
$$\mathbf{L}_{\mathsf{head}} = \widehat{\mathbf{L}}\ .\ \mathsf{Get}\ \mathsf{t},\ \widehat{\mathbf{L}} \text{ contains no } \mathsf{Get}\ \mathsf{t}, \text{ and } \mathbf{L} \equiv \widehat{\mathbf{L}}\ .\ \mathsf{Get}\ \mathsf{t}(\!|\mathsf{C}|\!)\ .\ \mathbf{L}_{\mathsf{tail}}$$

The function $\mathsf{split}_{\mathsf{t}}(\{\mathbf{L}_i\}_{i\in I})$ *returns the common prefix and the remaining postfixes of all input types. Note that the function may be undefined.*

$$\mathsf{split}_{\mathsf{t}}(\{\mathbf{L}_i\}_{i\in I}) = (\mathbf{L}_{\mathsf{head}}, \{(\mathsf{C}_i, \mathbf{L}^i_{\mathsf{tail}}\})) \text{ such that } \mathsf{split}_{\mathsf{t}}(\mathbf{L}_i) = (\mathbf{L}_{\mathsf{head}}, \mathsf{C}_i, \mathbf{L}^i_{\mathsf{tail}})$$

$$\mathbf{L}\langle\sigma\rangle \restriction^{\mathbf{p}} \mathsf{t} = \mathbf{L} \text{ if } \sigma(\mathbf{p}) = ((\mathsf{Active}(\mathsf{t}), \mathit{SState})$$
$$\text{and } \mathbf{L} \in \{\mathbf{p}!_{\mathsf{t}'}\mathtt{m},\ \mathsf{Put}\ \mathsf{t}'(\!|\mathsf{C}|\!),\ \mathsf{Get}\ \mathsf{t}'(\!|\mathsf{C}|\!),\ \mathsf{Susp}(\mathsf{t}',\mathsf{t}''),\ \mathbf{skip}\}$$
$$\mathbf{p}?_{\mathsf{t}'}\mathtt{m} \restriction^{\mathbf{p}} \mathsf{t} = \mathsf{React}\ \mathsf{t}' \restriction^{\mathbf{p}} \mathsf{t} = \mathbf{skip}$$
$$(\mathbf{L}_1 \,.\, \mathbf{L}_2) \restriction^{\mathbf{p}} \mathsf{t} = (\mathbf{L}_1) \restriction^{\mathbf{p}} \mathsf{t} \,.\, (\mathbf{L}_2) \restriction^{\mathbf{p}} \mathsf{t}$$
$$(\mathbf{L})^* \restriction^{\mathbf{p}} \mathsf{t} = \begin{cases} \mathbf{L} \restriction^{\mathbf{p}} \mathsf{t} & \text{if } \mathsf{t} \text{ is introduced within } \mathbf{L} \\ (\mathbf{L}')^* & \text{if } \mathbf{L} \restriction^{\mathbf{p}} \mathsf{t} = \mathbf{L}' \neq \mathbf{skip} \text{ and } \mathsf{t} \text{ is not introduced within } \mathbf{L} \\ \mathbf{skip} & \text{otherwise} \end{cases}$$
$$\oplus\{\mathbf{L}_i\}_{i\in I} \restriction^{\mathbf{p}} \mathsf{t} = \oplus\{\mathbf{L}_i \restriction^{\mathbf{p}} \mathsf{t}\}_{i\in I}$$
$$\&_{\mathsf{t}'}\{\mathbf{L}_i\}_{i\in I} \restriction^{\mathbf{p}} \mathsf{t} = \begin{cases} \mathbf{L} \restriction^{\mathbf{p}} \mathsf{t} \,.\, \&_{\mathsf{t}'}\{\mathsf{C}_i : \widehat{\mathbf{L}_i} \restriction^{\mathbf{p}} \mathsf{t}\}_{i\in I} & \text{if } \mathsf{split}_{\mathsf{t}'}(\{\mathbf{L}_i\}_{i\in I}) = (\mathbf{L}, \{(\mathsf{C}_i, \widehat{\mathbf{L}_i})\}) \\ \&_{\mathsf{t}'}\{\mathbf{L}_i\}_{i\in I} \restriction^{\mathbf{p}} \mathsf{t} = \mathbf{L}_j \restriction^{\mathbf{p}} \mathsf{t} & \text{if } j \in J \text{ and } \mathsf{t} \text{ is introduced in } \mathbf{L}_j \end{cases}$$

Figure 5 Projection of local types on tracked futures.

Projection $\mathbf{L}\langle\sigma\rangle \restriction^{\mathbf{p}} \mathsf{t}$ of an annotated local type $\mathbf{L}\langle\sigma\rangle$ on t for role $\mathbf{p}$ is given in Fig. 5. It removes receiving and reactivation actions, is the identity on any other non-composed action and propagates on sequential composition and active choice. For passive choice, the above split is applied, unless the projection future is introduced in only one branch. Repetitions outside a single method run are removed.

3.2.4 Scheduling Type

Given a projected object-local type $\mathbf{L}$, the scheduling type $\mathcal{S}(\mathbf{L})$ is generated by replacing all termination, synchronization, suspension and sending actions with **skip** and using structural congruence (see Fig. 6) to simplify the result.

3.3 Locally Static

Method-local types are checked statically. This ensures that *if* every process is scheduled correctly, then the process will perform its local view on the protocol correctly. Before we present the type system itself, we define typing contexts and auxiliary functions.

The subtype relations $<$, $\leq$ and structural congruence are standard, see Fig. 6. Structural congruence allows to add and remove **skip** actions. An active choice with a single branch can be simplified to the content of the branch. The interesting rules for subtyping are the ones for branching: Active branching may drop branches, as the implementing role may never take a subset of its possible choices. Its dual, passive branching, may add branches instead.

We use two typing contexts: Δ maps locations (fields and variables) to roles, Γ maps tracked futures to pairs of locations or the symbol $\bot$. the Δ context ensures that a method interacts with the correct endpoints, while Γ keeps track of futures and their read values. We use some auxiliary functions and predicates:

- The function $\Gamma^{\mathfrak{A}}$ removes all fields from the pairs in the image of Γ.
- The function $\mathsf{constr}(\mathtt{e})$ returns the outermost constructor of expression $\mathtt{e}$.
- The function $\mathsf{def}(\mathtt{C})$ returns the declaration of class $\mathtt{C}$.
- The predicate $\mathsf{inter}(\mathtt{s}, \Gamma)$ holds if the statement $\mathtt{s}$ contains no **get**, no **return**, no **await**, and writes into no location that is in a pair in the image of Γ.

$$\oplus\{\mathbf{L}_i\}_{i\in I} < \oplus\{\mathbf{L}_i\}_{i\in I\cup J} \qquad \&_{\mathsf{t}}\{\mathsf{C}_i:\mathbf{L}_i\}_{i\in I} > \&_{\mathsf{t}}\{\mathsf{C}_i:\mathbf{L}_i\}_{i\in I\cup J}$$

$$(\mathbf{L})^* < (\widehat{\mathbf{L}})^* \qquad \text{if } \mathbf{L} < \widehat{\mathbf{L}}$$

$$\mathbf{L}_1.\mathbf{L}_2 < \widehat{\mathbf{L}_1}.\widehat{\mathbf{L}_2} \qquad \text{if } \mathbf{L}_i < \widehat{\mathbf{L}_i}$$

$$\oplus\{\mathbf{L}_i\}_{i\in I} < \oplus\{\widehat{\mathbf{L}_i}\}_{i\in I} \qquad \text{if } \mathbf{L}_i < \widehat{\mathbf{L}_i}$$

$$\&_{\mathsf{t}}\{\mathsf{C}_i:\mathbf{L}_i\}_{i\in I} < \&_{\mathsf{t}}\{\mathsf{C}_i:\widehat{\mathbf{L}_i}\}_{i\in I} \qquad \text{if } \mathbf{L}_i < \widehat{\mathbf{L}_i}$$

$$\mathbf{L}\equiv\oplus\{\mathbf{L}\} \quad \mathbf{L}\equiv\mathbf{skip}\ .\ \mathbf{L} \quad \mathbf{L}\equiv\mathbf{L}\ .\ \mathbf{skip}$$

$$\mathbf{L}\ .\ \oplus\{\mathbf{L}_i\}_{i\in I} \equiv \oplus\{\mathbf{L}\ .\ \mathbf{L}_i\}_{i\in I} \qquad \oplus\{\mathbf{L},\mathbf{skip}_i\}_{i\in I}\equiv\mathbf{L}$$

Figure 6 Subtype relation and structural congruence of method-local types.

- The predicate $\mathbf{p}\in\mathbf{G}$ or $\mathbf{p}\in\mathbf{L}$ holds if the role $\mathbf{p}$ occurs in the type.
- The predicate $\mathtt{e}\in\mathbf{im}\Gamma$ holds if the location $\mathtt{e}$ is in any pair in the image of Γ.

The type system is shown in Fig. 7. Rule T-**main** checks that the main block sets up the session correctly: Each role is assigned to exactly one object and the corresponding class is checked against the projected type on this role. Also, each parameter of a class is assigned such that the passed variable has the correct role (the f_{ij} are the fields declared in $\mathsf{def}(\mathtt{C}_i)$). Lastly, the sole called method is correctly specified and called on the correct object. The rule T-**class** checks that each role needed for the object-local type is available in some field and checks each method against its method-local type.

$$\text{T-}\mathbf{main}\ \frac{\forall i.\ \exists\mathbf{p}\in\mathbf{G}.\ \Delta(\mathtt{v}_i)=\mathbf{p} \qquad \forall\mathbf{p}\in\mathbf{G}.\ \exists i.\ \Delta(\mathtt{v}_i)=\mathbf{p} \qquad \Delta_i(f_{ij})=\Delta(\mathtt{v}_{ij}) \qquad \Delta_i\vdash\mathsf{def}(\mathtt{C}_i)\ :\ \mathbf{G}\upharpoonright\Delta(\mathtt{v}_i) \qquad \Delta(\mathtt{v}_k)=\mathbf{p}}{\vdash\{\mathtt{I}_i\ \mathtt{v}_i\ \mathtt{=}\ \mathbf{new}\ \mathtt{C}_i(\mathtt{v}_{ij})\mathtt{;}\mathtt{v}_k\mathtt{!m();}\}\ :\ \mathbf{0}\xrightarrow{\mathsf{t}}\mathbf{p}\mathtt{:m}\ .\ \mathbf{G}}$$

$$\text{T-}\mathbf{class}\ \frac{\forall\mathbf{p}\in\mathbf{L}.\ \exists i.\ \Delta(\mathtt{f}_i)=\mathbf{p} \qquad \Delta,\emptyset\vdash\mathtt{s}_k\ :\ \mathbf{L}\upharpoonright^{\mathbf{p}}\mathsf{t} \qquad \mathtt{m}_k \text{ is the method of } \mathsf{t} \text{ in } \mathbf{L}}{\Delta\vdash\mathbf{class}\ \mathtt{C(I}_i\ \mathtt{f}_i\mathtt{)\{T}_j\ \mathtt{f}_j\ \mathtt{=}\ \mathtt{e}_j\mathtt{;}\ \mathtt{T}_k\ \mathtt{m}_k\mathtt{(T}_{kl}\ \mathtt{v}_{kl}\mathtt{)\{s}_k\mathtt{\}\}}\ :\ \mathbf{L}}$$

$$\text{T-}\leq\ \frac{\Delta,\Gamma\vdash\mathtt{s}\ :\ \mathbf{L} \qquad \mathbf{L}\equiv\widehat{\mathbf{L}}'\leq\widehat{\mathbf{L}}}{\Delta,\Gamma\vdash\mathtt{s}\ :\ \widehat{\mathbf{L}}} \qquad \text{T-;}\ \frac{\Delta,\Gamma\vdash\mathtt{s}_2\ :\ \mathbf{L} \qquad \mathsf{inter}(\mathtt{s}_1,\Gamma)}{\Delta,\Gamma\vdash\mathtt{s}_1\mathtt{;}\mathtt{s}_2\ :\ \mathbf{L}}$$

$$\text{T-}\mathbf{return}\ \frac{\mathsf{constr}(\mathtt{e})=\mathsf{C}}{\Delta,\Gamma\vdash\mathbf{return}\ \mathtt{e;}\ :\ \mathsf{Put\ t}(\!|\mathsf{C}|\!)} \qquad \text{T-}\mathbf{skip}\ \frac{}{\Delta,\Gamma\vdash\mathbf{skip}\ :\ \mathbf{skip}}$$

$$\text{T-}\mathbf{if}\ \frac{\Delta,\Gamma\vdash\mathtt{s}_1\mathtt{;}\mathtt{s}_3\ :\ \mathbf{L} \qquad \Delta,\Gamma\vdash\mathtt{s}_2\mathtt{;}\mathtt{s}_3\ :\ \mathbf{L}}{\Delta,\Gamma\vdash\mathbf{if}\mathtt{(e)\{s}_1\mathtt{\}}\mathbf{else}\mathtt{\{s}_2\mathtt{\}s}_3\ :\ \mathbf{L}} \qquad \text{T-}\mathbf{while}\ \frac{\Delta,\widetilde{\Gamma}\vdash\mathtt{s}_1\ :\ \mathbf{L}_1 \qquad \Delta,\widetilde{\Gamma}\vdash\mathtt{s}_2\ :\ \mathbf{L}_2}{\Delta,\Gamma\vdash\mathbf{while}\mathtt{(e)\{s}_1\mathtt{\}s}_2\ :\ (\mathbf{L}_1)^*.\mathbf{L}_2}$$

$$\text{T-}\mathbf{await}\ \frac{\Delta,\Gamma^{\mathfrak{A}}\vdash\mathtt{s}\ :\ \mathbf{L} \qquad \Gamma(\mathsf{t}')=(\mathtt{e},_)}{\Delta,\Gamma\vdash\mathbf{await}\ \mathtt{e;}\ \mathtt{s}\ :\ \mathsf{Susp}(\mathsf{t},\mathsf{t}').\mathbf{L}}$$

$$\text{T-}\mathbf{get}\ \frac{\Gamma(\mathsf{t})=(\mathtt{e}_2,_) \qquad \Delta,\Gamma[\mathsf{t}\mapsto(\mathtt{e}_2,\mathtt{e}_1)]\vdash\mathtt{s}\ :\ \mathbf{L} \qquad \mathtt{e}_1\notin\mathbf{im}\Gamma}{\Delta,\Gamma\vdash\mathtt{e}_1\ \mathtt{=}\ \mathtt{e}_2\mathtt{.}\mathbf{get}\mathtt{;}\ \mathtt{s}\ :\ \mathsf{Get\ t}.\mathbf{L}}$$

$$\text{T-!}\ \frac{\Delta,\Gamma[\mathsf{t}\mapsto(\mathtt{e}_1,\bot)]\vdash\mathtt{s}\ :\ \mathbf{L} \qquad \Delta(\mathtt{e}_2)=\mathbf{p} \qquad \mathtt{e}_1\notin\mathbf{im}\Gamma}{\Delta,\Gamma\vdash\mathtt{e}_1\ \mathtt{=}\ \mathtt{e}_2\mathtt{!m(e);}\ \mathtt{s}\ :\ \mathbf{p}!_{\mathsf{t}}\mathtt{m}.\mathbf{L}}$$

$$\text{T-}\mathbf{case}\ \frac{\mathsf{C}_i=\mathsf{C}_j\rightarrow\Delta,\Gamma\vdash\mathtt{s}_i\ :\ \mathbf{L}_j.\mathbf{L} \qquad \forall j.\ \exists i.\ \mathsf{C}_i=\mathsf{C}_j \qquad \Gamma(\mathsf{t})=(_,\mathtt{e})}{\Delta,\Gamma\vdash\mathbf{case}\mathtt{(e)\{}\mathsf{C}_i\mathtt{=>s}_i\mathtt{\}}_{i\in I}\mathtt{s}\ :\ \&_{\mathsf{t}}\{\mathsf{C}_j:\mathbf{L}_j\}.\mathbf{L}}$$

Figure 7 Static Type System.

Rule T-$\leq$ is used for structural congruence and sub-typing. The construction of a syntax-directed variant of the type system without a special rule for subtying is standard. Rule T-; drops a prefix that performs no communication and modifies no location stored in Γ. Rule T-**return** checks that the sole remaining action is a Put action and that the correct constructor is returned. Rule T-**skip** closes the proof if the empty program **skip** is left and no further action is required. This is needed to typecheck loop bodies, where we can always add **skip** at the end. Rule T-**if** splits the derivation into two branches. The type is not changed. Rule T-**while** checks a loop against the Kleene star. The context $\widetilde{\Gamma}$ removes all fields and variables modified in the loop body. Rule T-**await** checks that the correct future is synchronized on and removes all fields from the context. Rule T-**get** checks that the correct future is read and stores the information where the read value is available in the context. It ensures that no relevant read value or future is overwritten. Rule T-! checks that the correct method on the correct role is called and stores the information where the future is available in the context. It ensures that no other relevant read value or future is overwritten. Rule T-**case** checks a **case** statement against a passive choice by mapping each branch of the statement against some branch of the type. It is ensured that for every specified choice an implemented branch exists and that the read value is indeed stemming from the future containing the choice.

A rule for assignments to copy futures or their read values is easily added, but requires to keep track of a pair of sets of locations and, for simplicity, we refrain from introducing this.

3.4 Globally Dynamic

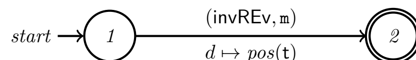
The globally dynamic part consists of two steps: first, we translate an object-local type into a session automaton, then we translate the session automaton into a user-defined scheduler.

3.4.1 Automaton Extraction

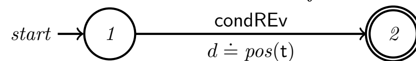
The structure of the translation follows the standard translation of regular expressions into finite automata.

▶ **Definition 13.** *Let* $\mathbf{L}$ *be a projected object-local type with* k *tracked futures. Let* $pos(\mathtt{t})$ *be the register assigned to* $\mathtt{t}$*. The translation of* $\mathbf{L}$ *into a* k*-register session automaton is denoted* $\mathcal{A}(\mathbf{L})$ *and defined as follows:*

- *A receiving type* $\mathbf{p}?_{\mathtt{t}}\mathtt{m}$ *is translated into an automaton with two states and a single transition that reads* invREv, $\mathtt{m}$ *and stores the read future in* $pos(\mathtt{t})$*:*



- *A reactivation type* React $(\mathtt{t})$ *is translated into an automaton with two states and a single transition that reads* condREv *and matches the read future with the one stored in* $pos(\mathtt{t})$*.*



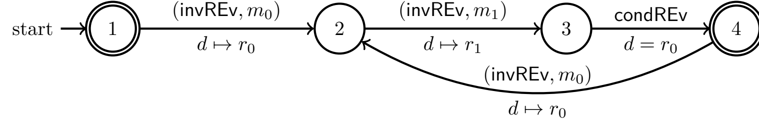
- *Branching, sequence and repetition are the standard translations of alternative, concatenation and Kleene star into finite automata.*

After this construction, standard ϵ*-transition elimination is performed.*

▶ **Example 14.** Consider the following scheduling type [29]:

$$\mathbf{L} = \left(\mathbf{p}?_{\mathtt{t}_0}\mathtt{m}_0 \ . \ \mathbf{p}?_{\mathtt{t}_1}\mathtt{m}_1 \ . \ \mathsf{React}\ (\mathtt{t}_0)\right)^*$$

Its translation $\mathcal{A}(\mathbf{L})$ is as follows (the translation yields an ϵ-transition from state 4 to state 1, which is eliminated to give the depicted automaton):



For formal soundness arguments, we again refer to [20]. Intuitively, the extraction is sound because the language accepted by the automaton is the same language as the one generated by the object-local type. Not every extracted session automaton is deterministic, because the input object-local type may not be deterministic, for example:

$$\&_{\mathtt{t}} \left\{ \begin{array}{l} \mathbf{p}?\mathtt{m} \\ \mathbf{p}?\mathtt{m} \;.\; \mathbf{q}?\mathtt{n} \end{array} \right\}$$

After receiving a call on `m`, this type cannot predict which branch to take. We do allow non-deterministic schedulers, but the implementation issues a warning. A simple syntax check on the automaton can exclude them.

3.4.2 Translation and Integration

Given a session automaton, we can finally extract a user-defined scheduler and add instrumentation code to ensure correctness.

▶ **Definition 15.** *Let* `C` *be a class that is checked against an object-local type that is transformed to a scheduling type* $\mathbf{L}$*. The instrumented class* $\mathtt{C}^{\mathcal{I}}$ *is constructed as follows:*

- *We add a field* `Int q = 0;` *that models the current state of the scheduling automaton.*
- *For each register* r_i *we add a field "*`Maybe<Fut<Any>> ri = Nothing;`*".*
- *The scheduler is as in Def. 16.*
- *For each method* `m` *we collect all transitions* $(q_i, (\mathsf{invREv}, \mathtt{m}), q_{i'})_{i \in I}$ *with written register* $reg(i)$ *and add the following as the first statement of* `m`*:*

```
case this.q {
  q_{i_1} => this.r reg(i_1) = Just(destiny); this.q = q_{i'_1};
  ⋮
  q_{i_m} => this.r reg(i_m) = Just(destiny); this.q = q_{i'_m};
}
```

  This statement saves its future in the given register and updates the automaton state. The generated scheduler ensures that no default branch is needed.
- *For each class* `C` *we collect all transitions* $(q_i, (\mathsf{condREv}), q_{i'})_{i \in I}$ *with read register* $reg(i)$ *and add the following as the first statement after each* **await** *statement in any method:*

```
case this.q { q_{i_1} => this.q = q_{i'_1}; ... q_{i_m} => this.q = q_{i'_m}; }
```

  Again, the generated scheduler ensures that no default branch is needed and the registers do not need to be checked against `destiny`*.*

▶ **Definition 16.** *The generated scheduler ensures that the initializing method with the hidden name* `.init()` *is always executed first. The function* `filter` *is one of the higher-order functions in ABS and takes a function of the form* `(params) => code` *as its first parameter and a list as its second.*

```
def Maybe<Process> scheduler(List<Process> list,
                             Int q,
                             Maybe<Fut<Any>> r1,
                             Maybe<Fut<Any>> r2) =
   if ( filter((Process p) => method(p) == ".init")(queue) != Nil )
       headOrNothing(filter((Process p) => method(p) == ".init")(queue))
   else case q {
     1 => headOrNothing(
        filter((Process p) => contains(set["publish"],method(p)))(list));
     2 => headOrNothing(
        filter((Process p) => contains(set["request"],method(p)))(list));
   }
```

Figure 8 Scheduler generated from Ex. 11.

```
def Maybe<Process> scheduler(List<Process> list, Int q,
                             Maybe<Fut<Any>> r1, ..., Maybe<Fut<Any>> rn) =
   if( filter((Process p) => method(p) == ".init")(queue) != Nil )
        headOrNothing(filter((Process p) => method(p) == ".init")(queue))
   else scheduler_body(list, q, r1, ..., rn);
```

After executing the initializer, the scheduler makes a case distinction over the states $1, \dots, m$ *of the scheduling automaton:*

```
def Maybe<Process> scheduler_body(List<Process> list, Int q,
                                  Maybe<Fut<Any>> r1, ..., Maybe<Fut<Any>> rn) =
  case q { 1 => transition1; ... m => transitionm; }
```

The transition $\texttt{transition}_i$ *from a state* i *is modeled as follows: Let* $\texttt{m}_1, \dots, \texttt{m}_{n_1}$ *be the method names that have outgoing transitions from* i *labeled with* invREv*. Let* $\texttt{r'}_1, \dots, \texttt{r'}_{n_2}$ *be the registers that the read future is compared with in outgoing transitions from* i *labeled with* condREv*. The first case checks that the future is allowed and not yet stored, the second case checks that the future is in one of the registers.*

```
headOrNothing(filter((Process p) =>
(contains(set[m1,...,mn1],method(p)) && !contains(set[r1,...,rn],destinyOf(p)))
|| contains(set[r'1,...,r'n2],destinyOf(p))
)(list))
```

We return the first process that is in the list and matches, a random scheduler is a straightforward modification.

▶ **Example 17.** The (beautified) scheduler generated from Ex. 11 is shown in Fig. 8:

3.5 Soundness and Stateful Session Types

Soundness. Soundness of the type system follows directly from the soundness theorem given for the original, purely static systems [27, 28]:

▶ **Theorem 18.** *Let* Prgm *be a well-typed ABS program and* **GP** *a global protocol. If* $\vdash$ Prgm : **GP** *and every object is instrumented with the scheduler type derived by the* $\vdash$ *relation, then every terminating and non-deadlocking run of* Prgm *has a trace where the communication events for each object are in the same order as specified in* **GP***.*

Our notion of soundness is not based on subject reduction and progress. Soundness of our system is *only* concerned with protocol adherence, not with deadlock freedom, as discussed above. Adding deadlock checks in session types complicates the system further [28] for little gain, as external tools can be used. We assume that the data types have been checked, so there is no need for a progress theorem. It is similar to *session fidelity* [22], which expresses the same intuition in terms of operational semantics.

Neither do we use a subject reduction theorem. Instead, we give a denotational semantics to session types and regard them as specifications of traces in monadic second-order logic: Any type $\mathbf{GP}$ can be translated into a formula $\mathbb{C}(\mathbf{GP})$ expressing that the communication events for each object are in the same order as specified in $\mathbf{GP}$. Soundness is then a *model-theoretic* notion that every trace $\mathbf{tr}$ generated by $\mathbf{GP}$ is a model of $\mathbb{C}(\mathbf{GP})$:

$$\vdash \mathsf{Prgm} : \mathbf{GP} \rightarrow \forall \mathbf{tr}.\ \mathsf{Prgm} \Downarrow \mathbf{tr} \rightarrow \mathbf{tr} \models \mathbb{C}(\mathbf{GP})$$

This model-theoretic treatment of session types allows an elegant connection to symbolic execution and dynamic logic [27] at the cost of an elaborate semantics [14] which we refrain to introduce for space reasons. This semantics is based on merging of local traces, which inhibits us from giving a straightfoward subject reduction theorem.

Stateful Session Types. So far, our session types do not constrain the execution state or passed data, except the outermost constructor of return values. We implemented an extension of the presented system, where each global call action is annotated with a property. This annotation is preserved during object-local projection and moved to the termination action during projection on a tracked future. Regarding instrumentiation, it results in a simple **assert** statement for the dynamic check.

▶ **Example 19.** We specify that a call of role $\mathbf{p}$ to a method `m` results in a postcondition that ensures the return value being larger than field `f`:

$$\ldots\ .\ \mathbf{p} \xrightarrow{\mathtt{t}} \mathbf{q}{:}\mathtt{m}(\!|\mathtt{this.f}\ \texttt{<}\ \mathtt{result}|\!)\ .\ \ldots$$

The return value is saved in a dedicated variable `result` and an assert is added afterwards. If the final statement was "`return e;`" before, it now is

```
Int result = e; assert(this.f < result); return result;
```

If it depends on the state of the scheduler which postcondition has to be checked, a **case** statement over the possible values of `q` is added. This approach is slightly less expressive than other stateful session types for AO [26, 28], but has the benefit that there is no need to translate first-order logic formulas into expressions.

4 Implementation and Evaluation

Our system is implemented on top of a slightly modified[5] version of the `ABS` compiler [43, 39]. Source code and all examples are accessible at `https://github.com/ahbnr/SessionTypeABS`.

As discussed, we do not handle full `ABS` and demand that the main block initializes a whole session, each interface plays exactly one role and no objects are created after initialization. The session type is specified in an ASCII variant of Def. 4 in a separate file alongside the other

[5] Blocking schedulers and access to the future of a process are not yet part of the master branch of `ABS`.

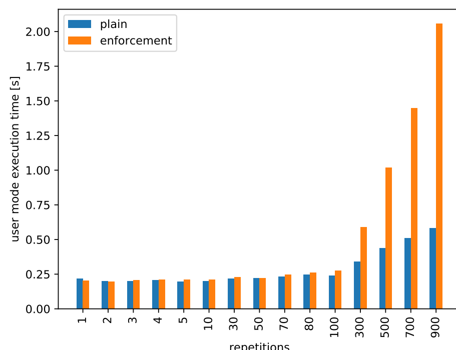


Figure 9 Execution times of the unmodified (blue) and modified (orange) model for different amounts of repetitions.

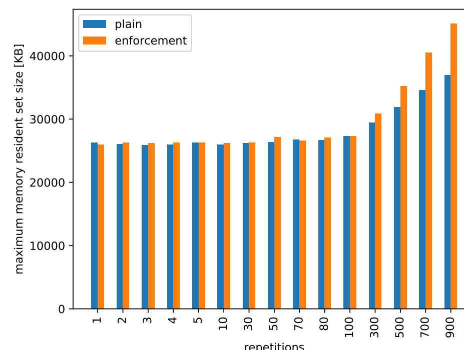


Figure 10 Maximum memory resident set size of the unmodified (blue) and modified (orange) model for different ammounts of repetitions.

model source files. The `ABS` compiler is used for parsing and (data-)typechecking the input model. The AST is then used for the static check and enriched with the instrumentation from the scheduling type. The resulting AST is passed back to the `ABS` compiler, which parses and typechecks it again.

We evaluate the impact of our modifications on the performance of Erlang-based simulations (the standard backend) of `ABS` models. The experiments are performed on a synthetic benchmark, where one object implementing the role **p** repeatedly calls two methods on another object of role **q** in a fixed order. The session is specified by the this global type:

$$\mathbf{0} \xrightarrow{\mathtt{t}} \mathbf{p}{:}\mathtt{init} \ . \ \left(\mathbf{p} \xrightarrow{\mathtt{t_{m1}}} \mathbf{q}{:}\mathtt{m1} \ . \ \mathbf{q}{\downarrow}\mathtt{t_{m1}} \ . \ \mathbf{p} \xrightarrow{\mathtt{t_{m2}}} \mathbf{q}{:}\mathtt{m2} \ . \ \mathbf{q}{\downarrow}\mathtt{t_{m2}}\right)^{*} \ . \ \mathbf{p}{\downarrow}\mathtt{t}$$

All reported data resulted from executing the model multiple times and averaging the measurements. Reported *execution times* designate the required run time in user-mode of the Erlang simulation of a model until termination on a Arch Linux system running Kernel 5.3.7 with a i5-4300U@2.9GHz CPU and 4GiB RAM.

Effect of Increased Object Communication. By changing the number of times the repeatable section of the session type is executed, we observe the behavior of the model simulation when the number of calls from **p** to **q** increases. We observe that the user-mode execution time of the simulations is nearly constant and mostly equivalent for the modified and unmodified version of the model for up to 100 repetitions, see Fig. 9. For higher numbers of repetitions, execution time increases for both version, but execution time of the modified model grows to increasing multiples of the execution time of the unmodified one. The maximum memory resident set size of the Erlang processes develops similarly, see Fig. 10, although the memory size of the modified model does not grow as rapidly as the execution time.

Comparison to Manual Synchronization. Instead of letting the generated schedulers enforce the execution order of methods, we now require **p** and **q** to synchronize every call by inserting an **await**-statement after each interaction. Even though execution time of the unmodified and modified model still increases for a high number of repetitions, there is now little difference between them, see Fig. 11. The overhead of synchronization is roughly equivalent to or lower than the version relying on the generated schedulers.

Testing the Reordering Capabilities of the Scheduler: In the previous experiments the scheduler never delayed activating a process, because there was always one in the queue which could immediately be scheduled. We now disable static verification, deliberately reverse the

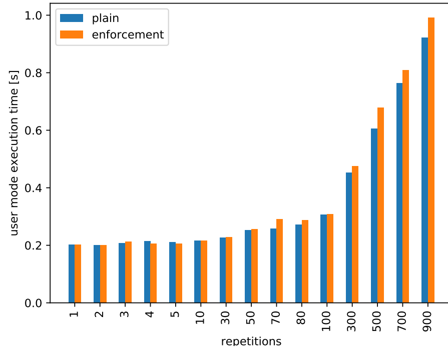


Figure 11 User-mode execution times when using **await** statements. Unmodified model in blue, modified model in orange.

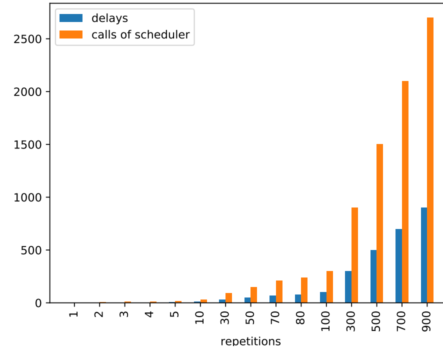


Figure 12 The number of times a scheduler has been invoked (orange) in contrast to the number of times it could not activate any waiting process (blue).

calls in the model source and put `duration` statements after each call, causing a delay in the execution.[6] We do not use synchronization and calls always arrive out of order at **q** and with enough inactivity in between them so that the scheduler of **q** frequently has to delay activation of a process until an acceptable one is available. Here, the modified and unmodified model always complete in almost the same execution time, presumably since the `duration` statements induce enough idle time to contain the overhead of the schedulers. However, we now observe that the scheduler successfully delays and reorders calls, see Fig. 12.

Discussion. A certain overhead must always be expected from instrumentation, but we deem the observed overhead acceptable. The generated schedulers only result in noteworthy overhead when a large number of processes is in the object queue. We conjecture that this effect is mostly an artifact of how the queue is represented for the user-defined scheduler.

5 Related Work

There is a considerable number of papers combining static and dynamic verification, a complete overview is out of scope for this work. We refer to, for example, the introduction of Ahrendt et al. [2] and only review directly related approaches here.

The `StaRVOOrS` [1, 9] tool combines static and dynamic verification of Java programs as follows: First, it attempts to prove certain properties statically using deductive verification and then it transforms failed proofs into runtime monitors. The static analysis is used to ensure that as little as possible is checked dynamically. `StarVOOrS` distinguishes between data and control-flow properties. The static analysis is mainly reducing the need for the computationally heavy data properties (e.g., all values of an array are non-zero) as far as possible, while monitoring control-flow properties can be done statically.

Our approach can be seen from a similar perspective: the object scheduler is handling the control flow inside an object, while the added **assert** statements are handling data properties. The type checker ensures that inside a method, only data properties need to be checked at

[6] Explicit time behavior is realized in Timed `ABS` [5] and here only used for evaluation.

runtime. It is straightforward to see how the ongoing integration of Session Types into the `Crowbar` prover using Behavioral Program Logic [26] can be used to discard as superflous **assert** statements statically.

The literature on session types includes approaches that handle protocols as (partially) dynamic types or mix static and dynamic checks otherwise. The conceptually closest to our approach is by Bocchi et at. [6], who also use distributed runtime enforcement, but introduce new components (for example, a queue) to do so. Completely dynamic approaches to session types are available for the Python language [12] and an actor model [35]. Other, less related, approaches are:

- Gradual session types [23] transform a dynamically checked *dyadic* session type for *channels* gradually to a statically checked one during development. The dynamic check for linearity that is central to gradual session types for channels has no direct counterpart in our system for AO, because the projection mechanisms differ on a technical level.
- Certain combined approaches, e.g., for Scala [38], draw the line between static and dynamic by performing only the linearity check at runtime and any other check statically.

A further type-based approach is *typestate* [41]. In contrast to session types, it was developed mainly for OO imperative programs. Typestate models that an object can change its interface, i.e., the set of exposed methods, over time. This was done statically in the original work and was subsequently gradualized [42] to combine static and dynamic type checking. A variant of typestate for concurrent Java, developed by Gerbo & Padovani [17], dynamically reports violations after injecting monitoring code. The object scheduler in our approach can be seen as a variant of typestate, but it is *generated*, not specified.

Choreographies [8] bear similarity to session types, being global specifications with a projection mechanism. However, they are mainly used to *generate* code via a correctness-by-construction approach. This also combines static and dynamic aspects, but reverses the direction: instead of dynamically ensuring that the static checks are sound, it is statically ensured (by code generation) that the dynamic behavior is structured correctly. The distinction between static and dynamic parts becomes even more prominent in the work of **Gabbrielli** et al. [16, 36, 37], where *dynamic choreographies* are used to generate a dynamic structure to update the structure of the application or include of new participants.

6 Conclusion

What should be the takeaway message from this work? First, the formalism of session types, first developed in the context of the π-calculus, and so far mainly used in theoretical investigations, appears in our context as a rather versatile and surprisingly practical specification mechanism. It is easily conceivable to find a more user-friendly, less mathematical notation for the global types in Fig. 3 and add IDE support.

Second, with the runtime checking approach, session types for AO can form the theoretical basis for top-down development of *open* distributed systems (with cooperative concurrency).

Third, as shown here and in [27], session types integrate well with static checking of logical properties. The semantic link is a straightforward translation from session types into logic , while the type systems syntactically ensures to place assertions at suitable locations.

Future Work. We plan to adopt the `StaRVOOrS` approach to partially reduce the need for **assert** statements on method-local level. We are investigating the use of the product line mechanism of `ABS` [10] to add the monitors, instead of using manual code injection. Using product lines enables a uniform treatment of code injection in `ABS` and the injection and removal of runtime monitors at runtime [40]. Furthermore, we plan to investigate the use of Timed Session Types [34] for Timed `ABS` and Hybrid `ABS` [32].

References

1 Wolfgang Ahrendt, Jesús Mauricio Chimento, Gordon J. Pace, and Gerardo Schneider. Verifying data- and control-oriented properties combining static and runtime verification: theory and tools. *Formal Methods Syst. Des.*, 51(1):200–265, 2017.

2 Wolfgang Ahrendt, Marieke Huisman, Giles Reger, and Kristin Yvonne Rozier. A broader view on verification: From static to runtime and back (track summary). In *ISoLA (2)*, volume 11245 of *Lecture Notes in Computer Science*, pages 3–7. Springer, 2018.

3 Elvira Albert, Puri Arenas, Antonio Flores-Montoya, Samir Genaim, Miguel Gómez-Zamalloa, Enrique Martin-Martin, German Puebla, and Guillermo Román-Díez. SACO: static analyzer for concurrent objects. In Erika Ábrahám and Klaus Havelund, editors, *TACAS 2014*, volume 8413 of *Lecture Notes in Computer Science*, pages 562–567. Springer, 2014. doi:10.1007/978-3-642-54862-8_46.

4 Henry G. Baker and Carl E. Hewitt. The incremental garbage collection of processes. In *Proceeding of the Symposium on Artificial Intelligence Programming Languages*, number 12 in SIGPLAN Notices, page 11, August 1977.

5 Joakim Bjørk, Frank S. de Boer, Einar Broch Johnsen, Rudolf Schlatte, and Silvia Lizeth Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *ISSE*, 9(1):29–43, 2013.

6 Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. Monitoring networks through multiparty session types. *Theor. Comput. Sci.*, 669:33–58, 2017. doi:10.1016/j.tcs.2017.02.009.

7 Benedikt Bollig, Peter Habermehl, Martin Leucker, and Benjamin Monmege. A fresh approach to learning register automata. In Marie-Pierre Béal and Olivier Carton, editors, *Developments in Language Theory: 17th Intl. Conf. DLT, Marne-la-Vallée, France*, volume 7907 of *LNCS*, pages 118–130. Springer, 2013.

8 Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8:1–8:78, 2012. doi:10.1145/2220365.2220367.

9 Jesús Mauricio Chimento, Wolfgang Ahrendt, Gordon J. Pace, and Gerardo Schneider. Starvoors: A tool for combined static and runtime verification of java. In Ezio Bartocci and Rupak Majumdar, editors, *RV 2015*, volume 9333 of *Lecture Notes in Computer Science*, pages 297–305. Springer, 2015. doi:10.1007/978-3-319-23820-3_21.

10 Dave Clarke, Radu Muschevici, José Proença, Ina Schaefer, and Rudolf Schlatte. Variability modelling in the ABS language. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *FMCO 2010*, volume 6957 of *Lecture Notes in Computer Science*, pages 204–224. Springer, 2010. doi:10.1007/978-3-642-25271-6_11.

11 Frank S. de Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. A survey of active object languages. *ACM Comput. Surv.*, 50(5):76:1–76:39, 2017. doi:10.1145/3122848.

12 Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Practical interruptible conversations: distributed dynamic verification with multiparty session types and python. *Formal Methods Syst. Des.*, 46(3):197–225, 2015. doi:10.1007/s10703-014-0218-8.

13 Crystal Chang Din, Richard Bubel, and Reiner Hähnle. Key-abs: A deductive verification tool for the concurrent modelling language ABS. In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 517–526. Springer, 2015. doi:10.1007/978-3-319-21401-6_35.

14 Crystal Chang Din, Reiner Hähnle, Einar Broch Johnsen, Ka I Pun, and Silvia Lizeth Tapia Tarifa. Locally abstract, globally concrete semantics of concurrent programming languages. In *TABLEAUX*, volume 10501 of *Lecture Notes in Computer Science*, pages 22–43. Springer, 2017.

15 Antonio Flores-Montoya, Elvira Albert, and Samir Genaim. May-happen-in-parallel based deadlock analysis for concurrent objects. In *FMOODS/FORTE*, volume 7892 of *LNCS*, pages 273–288. Springer, 2013.

16 Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Guess who's coming: Runtime inclusion of participants in choreographies. In Mário S. Alvim, Kostas Chatzikokolakis, Carlos Olarte, and Frank Valencia, editors, *The Art of Modelling Computational Systems: A Journey from Logic and Concurrency to Security and Privacy - Essays Dedicated to Catuscia Palamidessi on the Occasion of Her 60th Birthday*, volume 11760 of *Lecture Notes in Computer Science*, pages 118–138. Springer, 2019. `doi:10.1007/978-3-030-31175-9_8`.

17 Rosita Gerbo and Luca Padovani. Concurrent typestate-oriented programming in java. In Francisco Martins and Dominic Orchard, editors, *Proceedings Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2019, Prague, Czech Republic, 7th April 2019*, volume 291 of *EPTCS*, pages 24–34, 2019. `doi:10.4204/EPTCS.291.3`.

18 Elena Giachino, Cosimo Laneve, and Michael Lienhardt. A framework for deadlock detection in core ABS. *Software and Systems Modeling*, 15(4):1013–1048, 2016. `doi:10.1007/s10270-014-0444-y`.

19 Dilian Gurov, Reiner Hähnle, and Eduard Kamburjan. Who carries the burden of modularity? In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation, 9th Intl. Symp., ISoLA 2020, Rhodes, Greece*, LNCS. Springer, October 2020.

20 Anton W Haubner. Semi-dynamic session types for ABS. Bachelor thesis, Technical University of Darmstadt, 2019. URL: `https://github.com/ahbnr/SessionTypeABS/blob/master/thesis/thesis_final_pdfa.pdf`.

21 Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245. Morgan Kaufmann Publishers Inc., 1973. URL: `http://dl.acm.org/citation.cfm?id=1624775.1624804`.

22 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 273–284. ACM, 2008. `doi:10.1145/1328438.1328472`.

23 Atsushi Igarashi, Peter Thiemann, Vasco T. Vasconcelos, and Philip Wadler. Gradual session types. *Proc. ACM Program. Lang.*, 1(ICFP):38:1–38:28, 2017. `doi:10.1145/3110282`.

24 Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *FMCO 2010*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2010. `doi:10.1007/978-3-642-25271-6_8`.

25 Eduard Kamburjan. Detecting deadlocks in formal system models with condition synchronization. *ECEASST*, 76, 2018. `doi:10.14279/tuj.eceasst.76.1070`.

26 Eduard Kamburjan. Behavioral program logic. In *TABLEAUX*, volume 11714 of *Lecture Notes in Computer Science*, pages 391–408. Springer, 2019.

27 Eduard Kamburjan. *Modular Verification of a Modular Specification: Behavioral Types as Program Logics*. PhD thesis, Technische Universität Darmstadt, 2020.

28 Eduard Kamburjan and Tzu-Chun Chen. Stateful behavioral types for active objects. In Carlo A. Furia and Kirsten Winter, editors, *iFM 2018*, volume 11023 of *Lecture Notes in Computer Science*, pages 214–235. Springer, 2018. `doi:10.1007/978-3-319-98938-9_13`.

29 Eduard Kamburjan, Crystal Chang Din, and Tzu-Chun Chen. Session-based compositional analysis for actor-based languages using futures. In Kazuhiro Ogata, Mark Lawford, and Shaoying Liu, editors, *ICFEM 2016*, volume 10009 of *Lecture Notes in Computer Science*, pages 296–312, 2016. `doi:10.1007/978-3-319-47846-3_19`.

30 Eduard Kamburjan, Crystal Chang Din, Reiner Hähnle, and Einar Broch Johnsen. Asynchronous cooperative contracts for cooperative scheduling. In *SEFM*, volume 11724 of *Lecture Notes in Computer Science*, pages 48–66. Springer, 2019.

31 Eduard Kamburjan, Reiner Hähnle, and Sebastian Schön. Formal modeling and analysis of railway operations with active objects. *Sci. Comput. Program.*, 166:167–193, 2018. `doi:10.1016/j.scico.2018.07.001`.

32 Eduard Kamburjan, Stefan Mitsch, Martina Kettenbach, and Reiner Hähnle. Modeling and verifying cyber-physical systems with hybrid active objects. *CoRR*, abs/1906.05704, 2019. `arXiv:1906.05704`.

33 Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994. `doi:10.1016/0304-3975(94)90242-9`.

34 Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. Timed runtime monitoring for multiparty conversations. *Formal Asp. Comput.*, 29(5):877–910, 2017.

35 Rumyana Neykova and Nobuko Yoshida. Multiparty session actors. *Logical Methods in Computer Science*, 13(1), 2017.

36 Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies - safe runtime updates of distributed applications. In Tom Holvoet and Mirko Viroli, editors, *COORDINATION 2015*, volume 9037 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2015. `doi:10.1007/978-3-319-19282-6_5`.

37 Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies: Theory and implementation. *Log. Methods Comput. Sci.*, 13(2), 2017. `doi:10.23638/LMCS-13(2:1)2017`.

38 Alceste Scalas and Nobuko Yoshida. Lightweight session programming in scala. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*, pages 21:1–21:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. `doi:10.4230/LIPIcs.ECOOP.2016.21`.

39 Rudolf Schlatte and abstools Contributors. Modified branch of the abstools compiler version 1.8.1 - github source repository. `https://github.com/ahbnr/abstools/tree/thisDestiny`. Accessed: 2019-10-29.

40 Rudolf Schlatte, Einar Broch Johnsen, Jacopo Mauro, Silvia Lizeth Tapia Tarifa, and Ingrid Chieh Yu. Release the beasts: When formal methods meet real world data. In Frank S. de Boer, Marcello M. Bonsangue, and Jan Rutten, editors, *It's All About Coordination - Essays to Celebrate the Lifelong Scientific Achievements of Farhad Arbab*, volume 10865 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2018. `doi:10.1007/978-3-319-90089-6_8`.

41 Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986. `doi:10.1109/TSE.1986.6312929`.

42 Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In Mira Mezini, editor, *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*, volume 6813 of *Lecture Notes in Computer Science*, pages 459–483. Springer, 2011. `doi:10.1007/978-3-642-22655-7_22`.

43 Peter Y. H. Wong, Elvira Albert, Radu Muschevici, José Proença, Jan Schäfer, and Rudolf Schlatte. The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. *STTT*, 14(5):567–588, 2012.

A Formal Analysis of the Bitcoin Protocol

Cosimo Laneve
Dept. of Computer Science and Engineering, University of Bologna – INRIA Focus, Italy
cosimo.laneve@unibo.it

Adele Veschetti
Dept. of Computer Science and Engineering, University of Bologna – INRIA Focus, Italy
adele.veschetti2@unibo.it

Abstract

We study Nakamoto's Bitcoin protocol that implements a distributed ledger on peer-to-peer asynchronous networks. In particular, we define a principled formal model of key participants – the miners – as stochastic processes and describe the whole system as a parallel composition of miners. We therefore compute the probability that ledgers turn into a state with more severe inconsistencies, e.g. with longer forks, under the assumptions that messages are not lost and nodes are not hostile. We also study how the presence of hostile nodes mining blocks in wrong positions impacts on the consistency of the ledgers. Our theoretical results agree with the simulations performed on a probabilistic model checker that we extended with dynamic datatypes in order to have a faithful description of miners' behaviour.

2012 ACM Subject Classification Theory of computation → Formalisms

Keywords and phrases Bitcoin, Distributed consensus, Distributed ledgers, Blockchain, `PRISM`, forks

Digital Object Identifier 10.4230/OASIcs.Gabbrielli.2020.2

Related Version A full version of the paper with the proofs of the main statements is available at `http://cs.unibo.it/~laneve/papers/LaneveVeschetti.pdf`.

1 Introduction

Bitcoin is a distributed application that implements a ledger on peer-to-peer asynchronous networks that are dynamic (nodes may either join or leave) [20]. This technology is particularly critical because it manages and transfers relevant assets in the form of cryptocurrencies.

The basic problem of implementing a distributed ledger on a dynamic peer-to-peer asynchronous network is the management of inconsistent updates of the ledger performed by different nodes, which are called *forks*. This problem, known as distributed consensus in the literature, has been proved to be unsolvable since 1985 [9]. To overcome this shortcoming, the Bitcoin protocol uses an ingenious breakthrough: it guarantees a so-called *eventual consistency* whereby the various replicas of the ledger may be temporarily inconsistent in at most the last m blocks [10]. Overall, the protocol is very complex and the current research is actively involved in understanding all the critical points that a potential attacker might use. We refer to [24] for an overview of possible attacks to Bitcoin.

Following [13, 11, 22, 23], we study the foundational principles of the Bitcoin algorithm in a formal way, by defining a clean and principled model of the key participants – the *miners*. These miners are stateful nodes communicating with each other by means of asynchronous messages that either announce a transaction (which defines a particular event) or create and broadcast a block that contains (the encoding of) a set of transactions. Once blocks are received, the miners validate them (they verify the correctness of the transactions therein) and, if this process succeeds, add the block to the local copy of the ledger.

© Cosimo Laneve and Adele Veschetti;
licensed under Creative Commons License CC-BY

As said above, this setting cannot give a global consensus on miners' ledgers because the underlying system is distributed. In fact, in this context, blocks may be delivered in a wrong order, may be duplicated, may be lost, or may be produced by hostile nodes. The Bitcoin algorithm uses several expedients to overcome these issues. First of all, the algorithm controls the generation of new blocks in order to be much less frequent with respect to the broadcast delay (which is around 2 seconds). In particular, in Bitcoin, miners are committed to solve a computationally hard problem in order to mine a new block (and they are rewarded with new Bitcoins when this happens). The complexity of the problem is set in such a way that mining occurs every 10 minutes (the technique is known as *proof of work*). In addition, the Bitcoin algorithm uses *ledgers* that are *trees of blocks* with *a pointer to a leaf node at maximal depth* called *handle*. The (eventual) consistency is not guaranteed on the ledgers (that may be different), but on the *blockchain* of the ledgers, i.e. the chain of blocks starting from the handle to the root node, called *genesis block* (which is assumed to be always the same). In this context, the addition of a new block to the ledger is a critical operation because, besides connecting the block to its parent (every block records the parent node), it may also change the handle (and therefore the corresponding blockchain) if the height of the ledger increases.

In our modelling of the Bitcoin protocol we intentionally leave out a number of details, such as what can go into a transaction or into a block, the exact specifics of the proof of work algorithm, and the validation process. We also overlook standard issues of distributed systems, such as the loss of messages or miners that may become either inactive or may show up in the system at runtime. In our setting, the network and the miners are modelled by means of stochastic processes where actions have rates. These rates are the formal artifice we use for expressing the latency of the network and the time required by miners to solve the computational problem (which is inversely proportional to the so-called *hashing power*). Overall, our model is simple and rigorous, which are, in our opinion, fundamental criteria for reasoning about properties of blockchain-based algorithms and for gaining trust in their basic principles. Once the basic properties have been analyzed and understood, one can address other, possible more complex, scenarios of distributed systems.

Our contribution. The formal model for defining the Bitcoin protocol is an extension of `PRISM` [18]. `PRISM` has been chosen for two reasons. First, because it is a simple process calculus with a formal stochastic semantics that uses rates of actions as parameters of an exponential distribution, which is a standard feature of Bitcoin actions of mining and broadcasting [1, 26, 8, 4]. Secondly, because `PRISM` has a tool for analysing stochastic systems that can be used for complementing our theoretical results with practical simulations.

However, as it is, `PRISM` falls short to model faithfully the Bitcoin protocol because it misses the datatypes of blocks and ledgers. Therefore, in Section 2, following the description in [20], we have defined the values of blocks, queues and ledgers and the corresponding operations. The extension of `PRISM`, called `PRISM+`, with the foregoing datatypes is defined in Section 3. In `PRISM+` a system is a parallel composition of modules that interact on actions that have in common. These actions may update the internal states of the modules (including ledgers and queues) and the next state of the system is defined in terms of a *race condition* between possible actions. The operational semantics of `PRISM+` is also reported in Section 3.

The Bitcoin protocol is defined in Section 4 as a `PRISM+` system consisting of a `NETWORK` module and a set of `MINER` modules. The `NETWORK` has a set of queues, one for every miner, which store the new blocks created by the miners. The blocks in the queues of `NETWORK` are retrieved by the miners through an explicit action. These actions and the corresponding rates allow us to implement the delay and the nondeterminism of the broadcast. The `MINER`

may either (*i*) mine a new block, or (*ii*) retrieve a block from the network and add it to the local queue, or (*iii*) take a block from the local queue and try to add it to the ledger. In case (*i*) the block is added to the local ledger and sent to the `NETWORK` in order to be inserted to the queues of the other miners. In case of (*ii*), the block taken from the network is not inserted into the ledger because the parent block may be still missing. Instead, it is inserted in the local queue of the miner that extracts blocks from time to time with the action (*iii*).

The `PRISM+` system allows us to compute the probability of devolving into a "larger inconsistency", e.g. transiting from a state with a fork of length m to a state with a fork $m+1$. This work, which has required a time-consuming analysis of the stochastic transition system, has given a formula that is parametric with respect to the number of nodes, their hashing power and the latency of the network. Henceforth, it has been possible to analyze different scenarios by tuning the rates of the corresponding actions. For instance, given the current rate-values of the Bitcoin system, the probability of reaching a state of fork of length 2 is less than 10^{-3}.

In Section 5 we apply the same technique for studying an attack to Bitcoin that has been already discussed in [20]: the presence of hostile nodes mining new blocks in positions that are different from the correct one (blocks are not inserted at maximal depth). The probability that we compute depends on the hashing power of the attacker and the depth m of the fork created by the hostile node. For example, if `BTC.com`, which is a cluster currently retaining the 14,1% of the Bitcoin hashing power, decided to become hostile, then the probability to create an alternative attacker chain and achieving consensus from the other nodes is 8^{-m}.

In the companion paper [2] we discuss the implementation of the library for blocks, queues and ledgers that extends `PRISM` and we analyze the results of simulations with different values of the rate parameters of the `PRISM+` system in Section 4. Remarkably, the results of the simulations are compliant with the upper bounds defined by our formulas and, for completeness, they are also highlighted in our pictures. (Actually `PRISM+` has a scalability issue: due to the state explosion of the Bitcoin model, the simulations were performed on systems with about twenty nodes.)

We analyze related works in Section 6 and report our concluding remarks in Section 7.

For space constraints, the proofs of our main statements are not included. They are reported in the full paper at `http://cs.unibo.it/~laneve/papers/LaneveVeschetti.pdf`.

2 Blocks, queues and ledgers

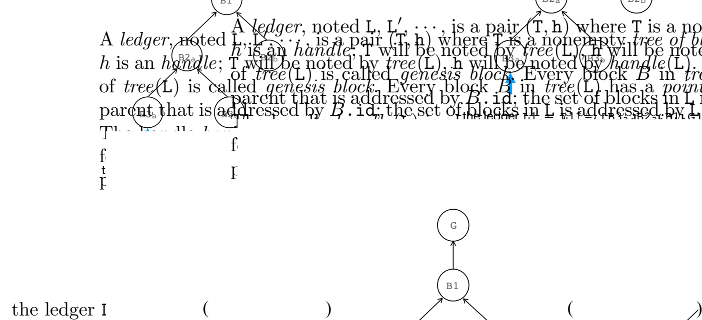
The Bitcoin protocol will be defined by a `PRISM` program, a process calculus with a stochastic semantics and an automatic analyzer of continuous-time Markov chains. However, datatypes used in Bitcoin cannot be modelled in `PRISM`; therefore we extend the language with `block`, `ledger` and `queue` data types and in the following we discuss this extension.

A basic component of the Bitcoin database is the `block`, which records transactions that are going to be certified, mining rewards, its hash value and a pointer to its parent. In this paper we abstract from many informations in blocks because they are not essential in the analysis of Sections 4 and 5 and we focus on the connections between blocks. Therefore, a `block` is a pair $(\texttt{name}, \texttt{father})$, where `name` uniquely identifies the block and `father` is the name of the previous block to which it is connected. Names will be represented by pairs $\texttt{mid}^n$, where `mid` is the name of the miner that mined the block and `n` is a number uniquely identifying the block. The operation that creates blocks is $\texttt{NewB}(\texttt{mid}, n, \texttt{p})$, which returns a block $(\texttt{mid}^{n+1}, \texttt{p})$, where `p` is the father name.

The Bitcoin protocol also uses a further datatype: the *bag of blocks*, namely sets of blocks that must still be appended to a ledger. We implement bags as *queues*, e.g. lists of blocks $[\mathtt{b}_0, \mathtt{b}_1, \cdots, \mathtt{b}_n]$ with the standard operations on queues:

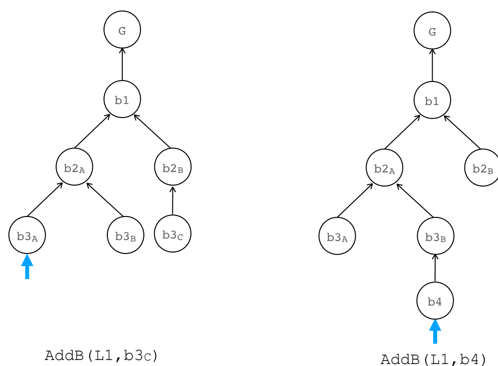
- the empty queue is noted `[]`;
- `isEmpty(Q)` returns *true* if `Q` = `[]`, *false* otherwise.
- the topmost block of a queue `Q` is given by `top(Q)`. When `Q` is empty, `top(Q)` returns *null*.
- the operation that inserts a block `b` at the end of a queue `Q` is `enqueue(Q, b)` (it returns a queue);
- the operation that removes the topmost block from a queue `Q` is `dequeue(Q)` (it returns a queue);
- the operation that removes the topmost block from a queue `Q` and inserts it at the end of the queue (because it cannot be added to the ledger) is `deq_enq(Q)` (it returns a queue). When `Q` is empty, `deq_enq` returns the empty queue.

The Bitcoin database is an append-only tree whose nodes are blocks and it is called *ledger*. A `ledger L` is a pair $\langle \mathtt{T}; \mathtt{p}\rangle$, where `T` is the *tree of blocks*, e.g. a set of blocks where each block points to its own parent, and `p`, called *handle*, is the name of a leaf at maximal depth. The root of the tree is the *genesis block* and noted $(\mathtt{gen}^0, \mathtt{gen}^0)$. The handle of a ledger `L` is given by `handle(L)`. The *blockchain of L*, noted $\mathtt{L}\uparrow$, is the sequence $(\mathtt{b}_0, \mathtt{b}_1, \mathtt{b}_2, \dots)$ such that $\mathtt{b}_0$ is the handle of `L` and, for every i, $\mathtt{b}_{i+1}$ is the parent of $\mathtt{b}_i$ (therefore the last block of the sequence is the genesis block). We illustrate ledgers by means of trees where nodes contain the name of the block and the unique exiting arrow is the pointer to its parent; the handle is represented by a tick arrow pointing to a leaf block at maximal depth. For example, the following picture illustrates two ledgers.



where $\mathtt{L1} = \langle\{\mathtt{G,(b1,gen^0),(b2_{\mathit{A}},b1),(b2_{\mathit{B}},b1),(b3_{\mathit{A}},b2_{\mathit{A}}),(b3_{\mathit{B}},b2_{\mathit{A}})}\}; \mathtt{b3}_A\rangle$ (`G` is the genesis block).

A key operation on ledgers is the *addition* of a new block to the ledger, written `AddB(L,b)`, that returns a ledger where `b` is connected to the block pointed by `b`. This operation may change the handle of the ledger. In particular, the handle of `AddB(L,b)` is equal to the handle of `L` if the new block has not changed the maximal depth of the tree; it is a pointer to `b` if this block has a depth strictly greater than the maximal one of `L`. For example, considering the ledgers `L1` and `L2` in the foregoing picture, let $\mathtt{b2}_B$ be the parent of $\mathtt{b3}_C$ and $\mathtt{b3}_B$ be the parent of `b4`. The ledgers $\mathtt{AddB(L1,b3}_C)$ and `AddB(L1,b4)` are



In these cases, the handle of AddB(L1,b3$_C$) is the same of L1, while this is not so for AddB(L1,b4) because the depth of the tree is changed.

It is also possible that a block cannot be added to a ledger because the parent block is not in the ledger. We use the boolean function canAdd(L, b) that returns *true* or *false* according to b can be added to L or not, respectively. canAdd(L, b) also returns *false* when b is *null*.

3 The modelling language: PRISM+

The language we use to analyze the Bitcoin protocol is an extension of PRISM [18] with the data types ledger, set and block. We call the language PRISM+.

To define PRISM+ we use a set of *action names* A, ranged over a, b, . . . , a set of *module names* ranged over M, M$_1$, . . . , and a set of *variables*, ranged over by x, y, z. Let α ranges over $A \cup \{\varepsilon\}$, where ε indicates *no-action*; let also ρ range over reals (called double).

A PRISM+ program P is a parallel composition of *modules*, that is

$$\mathtt{P} \;=\; \mathtt{M}_1 \parallel \cdots \parallel \mathtt{M}_n$$

where M ∥ M′ is the parallel composition of modules M and M′ synchronizing only on actions appearing in both M and M′. Let *actions*(M) be the set of actions in A that occur in M. A module M is defined by the syntax

$$\begin{array}{lcl} \mathtt{M} & ::= & \mathtt{module}\ \mathrm{M} :\ \mathtt{D}\ \mathtt{C}\ \mathtt{endmodule} \\ \mathtt{D} & ::= & \mathtt{T}\ x = v\ ; \quad | \quad \mathtt{T}\ x = v\ ;\ \mathtt{D} \\ \mathtt{T} & ::= & \mathtt{int} \ | \ \mathtt{double} \ | \ \mathtt{bool} \ | \ \mathtt{block} \ | \ \mathtt{ledger} \ | \ \mathtt{queue} \end{array}$$

That is, a module has a name, a sequence D of *local variable declarations* with initializations and a *set of commands* C. It is assumed that pairwise different modules in a PRISM+ program have different names and have also different local variables names.

Sets of commands C are written c_1 ; · · · ; c_m, where every c has the form:

$$\begin{array}{rcl} c & ::= & [\alpha]\, e \to \sum_{i\in I} \rho_i : \mathtt{upd}_i \\ \mathtt{upd} & ::= & \varepsilon \quad | \quad x' = e\ \&\ \mathtt{upd} \\ e & ::= & v \ | \ x \ | \ e\ \mathtt{op}\ e \ | \ !e \\ v & ::= & \mathit{true} \ | \ \mathit{false} \ | \ \mathit{integers} \ | \ \mathit{doubles} \ | \ \mathit{ledgers} \ | \ \mathit{queue} \\ & & | \ \ \mathit{blocks} \\ \mathtt{op} & ::= & - \ | \ + \ | \ * \ | \ = \ | \ \neq \ | \ \& \end{array}$$

In a command $[\alpha]\, e \to \sum_{i\in I} \rho_i : \mathtt{upd}_i$, α may be either empty or an action, e, called *guard*, is a boolean expression over all the variables in the program (including those belonging to other modules), and the right hand-side of the arrow describes a *transition*. In particular, when α is empty, if e is true then one of the corresponding updates may be performed. Each

update is defined by giving new values of *the variables in the module*, possibly as a function of other variables. Each update has also a rate, which will be given to the corresponding transition. Updates are written with the prime symbol: $x' = e$ means that, if v is the value of e in the current state then the value of x in the *next state* is v. We assume that, in an update ${x_1}' = e_1$ & ... & ${x_n}' = e_n$, left hand-side variables are all different.

When α is an action then the transition must be performed simultaneously with the other modules in parallel that have the same action (i.e. the modules *synchronize*). This is the standard CSP parallel composition [15]. The rate of the overall transition is equal to the product of the individual rates. Since the product of rates does not always meaningfully represent the rate of a synchronised transition, PRISM uses the technique to make exactly one action *active*, with a generic rate, and all the others *passive*, with rate 1. The rate of a synchronization is therefore defined by the unique active action.

Semantics. The semantics of a PRISM+ program is defined as a transition system whose states s are maps $[x_1 \mapsto v_1, \dots, x_n \mapsto v_n]$ where $\{x_1, \dots, x_n\}$ is the set of local variables of the program's modules. The transition relation uses the following auxiliary definitions:

- $s[x \mapsto v]$ is the state

$$(s[x \mapsto v])(y) \stackrel{\text{def}}{=} \begin{cases} v & \text{if } y = x \\ s(y) & \text{otherwise} \end{cases}$$

- $[\![e]\!](s)$ returns the value of an expression e in the state s. The value is computed by replacing the variables with their values in s and evaluating the operations. The formal definition is omitted because standard.
- $[\![\texttt{upd}]\!](s)$ returns the state s' defined as follows:

$$[\![{x_1}' = e_1 \ \& \ \dots \ \& \ {x_n}' = e_n]\!](s) \stackrel{\text{def}}{=} s[x_1 \mapsto [\![e_1]\!](s), \dots, x_n \mapsto [\![e_n]\!](s)]$$

The transition relation of PRISM+ is defined in Table 1 where we let $\mathcal{M}$ range over parallel compositions of modules and we assume $\parallel$ to be commutative. We use the judgment $\texttt{P} \Vdash s \xrightarrow{\alpha,\rho} s'$ meaning that the program P transits from s to s' with an action α and rate ρ. The auxiliary judgment $\mathcal{M} \Vdash s \xrightarrow{\alpha,\rho} \texttt{upd}$ collects all the updates in the synchronizing modules in $\mathcal{M}$ (according to our assumptions, different updates modify different variables). Rule [Upd] defines the semantics of a command. We write $c \in \texttt{M}$ if $\texttt{M} = \texttt{module}\ \textsc{m} :\ \texttt{D}\ \texttt{C}\ \texttt{endmodule}$ and $c \in \texttt{C}$. If e is true, then an update $\texttt{upd}_i$ is enabled with rate ρ_i and label α. The update $\texttt{upd}_i$ is a set of evaluated variables expressed as a conjunction of assignments. Rule [Sync] collects commands of synchronizing modules. We notice that the rate is the product of the rates of every single transition, which is actually the one of the unique active transition. Rule [Nosync] enables the interleaving of transitions (because of commutativity of $\parallel$, it also covers the symmetric rule). A PRISM+ program is a parallel composition of modules; its semantics is described in [Program].
PRISM+ supports different kinds of probabilistic formalisms; in this contribution we focus on CTMCs models [17], which are tuples $(\mathit{States}, s_{init}, \mathbf{R}, L)$ where:

- States is a countable set of states;
- $s_{init} \in \mathit{States}$ is the initial state; the initial state of a PRISM+ program

$$\texttt{P} = \prod_{i \in 1..n} \texttt{module}\ \textsc{m} :\ \texttt{D}_i\ \ \texttt{C}_i\ \texttt{endmodule}$$

  is $[\![\texttt{D}_1\ ;\ \cdots\ ;\ \texttt{D}_n]\!]$, where $[\![\texttt{T}_1\ x_1 = v_1\ ;\ \cdots\ ;\ \texttt{T}_k\ x_k = v_k]\!] = [x_1 \mapsto v_1, \dots, x_k \mapsto v_k]$;
- $\mathbf{R} : \mathit{States} \times \mathit{States} \to \mathbb{R}_{\geq 0}$ is a transition rate matrix,
- $L : \mathit{States} \to 2^{AP}$ is function which assigns to each state $s \in S$ the set $L(s)$ of atomic propositions that are valid in the state.

Table 1 The semantics of the PRISM language.

[Upd]
$$\frac{[\alpha]\, e \rightarrow \sum_{i\in I} \rho_i : \texttt{upd}_i \in \texttt{M} \qquad [\![e]\!](s) = \mathit{true}}{\texttt{M} \Vdash s \xrightarrow{\alpha,\rho_i} \texttt{upd}_i}$$

[Sync]
$$\frac{\mathcal{M} \Vdash s \xrightarrow{a,\rho} \texttt{upd} \qquad \mathcal{M}' \Vdash s \xrightarrow{a,\rho'} \texttt{upd}'}{\mathcal{M} \parallel \mathcal{M}' \Vdash s \xrightarrow{a,\rho\times\rho'} \texttt{upd\&upd}'}$$

[Nosync]
$$\frac{\mathcal{M} \Vdash s \xrightarrow{\alpha,\rho} \texttt{upd} \qquad \alpha \notin \mathit{actions}(\texttt{M})}{\mathcal{M} \parallel \texttt{M} \Vdash s \xrightarrow{\alpha,\rho} \texttt{upd}}$$

[Program]
$$\frac{\texttt{M}_1 \parallel \cdots \parallel \texttt{M}_n \Vdash s \xrightarrow{\alpha,\rho} \texttt{upd} \qquad \texttt{P} = \texttt{M}_1 \parallel \cdots \parallel \texttt{M}_n}{\texttt{P} \Vdash s \xrightarrow{\alpha,\rho} [\![\texttt{upd}]\!](s)}$$

A transition rate matrix assigns rates to each pair of states, which are used as parameters of the exponential distribution. A transition from state s to s' is possible only if $\mathbf{R}(s, s') > 0$. When multiple commands with the same update and that lead to the same state s' are enabled, the corresponding transitions are combined into a single transition whose rate is the sum of the individual rates. Furthermore, when there are several s' with $\mathbf{R}(s, s') > 0$, a *race condition* occurs: the transition triggered determines the next state. Technically, the time spent in s before a transition occurs is exponentially distributed with the *exit rate* of the state s:

$$E(s) = \sum_{s'\in S} \mathbf{R}(s, s')$$

Thus, the probability of leaving a state s within t seconds is $1 - e^{-tE(s)}$. Additionally, the choice between the transitions is independent of the time at which it occurs. This means that, if the state s has n outgoing transitions labeled with rates $\rho_1, \ldots, \rho_n$, then the probability that the j-th transition is taken is $\rho_j/(\sum_i \rho_i)$.

4 The abstract modelling of Bitcoin and its analysis

Bitcoin realises a distributed ledger on a peer-to-peer network of *miners*, which are processes that create blocks of the ledger and forward them to the nodes of the network. The Bitcoin system written in PRISM+ is

$$\texttt{MINER}_1 \parallel \cdots \parallel \texttt{MINER}_n \parallel \texttt{NETWORK}.$$

where $\texttt{MINER}_i$ and NETWORK are the modules defined in Listing 1.

In Listing 1, miners are defined from line 6 to 25. Every miner $\texttt{Miner}_i$ has five state variables: a state variable $\texttt{Miner}_i\texttt{_STATE}$, the last block $\texttt{b}_i$ added to the ledger; the local ledger $\texttt{L}_i$ that represents miner's view of the state of the system, a counter $\texttt{c}_i$ of the mined blocks, and a queue $\texttt{QMiner}_i$ that stores the blocks received by the network and that must be added to $\texttt{L}_i$. $\texttt{Miner}_i$ behaves as follows:

```
1  // states of Miner_i: Init = 0, Winner = 1
2  // mR = 1/600 is the Bitcoin mining rate
3  // hR_i is the percentage of hashing power of Miner_i, 0≤hR_i≤1
4  // r_i is the communication delay rate of Miner_i
5
6  module Miner_i
7    integer Miner_i_STATE = Init;
8    block b_i = (gen^0,gen^0);
9    ledger L_i = ⟨{(gen^0,gen^0)};gen^0⟩;
10   integer c_i = 0;
11   queue QMiner_i = [];
12
13   [] Miner_i_STATE=Init -> mR×hR_i : L_i'= AddB(L_i,NewB(Miner_i,c,handle(L_i))
14                                    & c_i'= c_i+1 & b_i' = NewB(Miner_i,c,handle(L_i))
15                                    & Miner_i_STATE'= Winner;
16
17   [] Miner_i_STATE=Init&canAdd(L_i,top(QMiner_i)) -> r : QMiner_i'= dequeue(QMiner_i)
18                                                          & L_i'= addB(L_i,top(QMiner_i));
19
20   [] Miner_i_STATE=Init&!canAdd(L_i,top(QMiner_i)) -> r : QMiner_i'= deq_enq(QMiner_i);
21
22   [addBlock_i] Miner_i_STATE=Init -> r_i : QMiner_i'= enqueue(QMiner_i,top(Q_i))
23
24   [addBlock_i] Miner_i_STATE=Winner -> r_i : Miner_i_STATE'= Init;
25 endmodule
26
27 module Network
28   integer n = numberOfMiners;
29   queue Q_1 = []; ...; queue Q_n = [];
30         ...
31   [addBlock_i] (Miner_i_STATE=Winner) -> 1:
32                               for ((j∈ 1..n) & (j≠i)) do (Q_j'= enqueue(Q_j,b_i)) ;
33
34   [addBlock_i] Miner_i_STATE=Init&!isempty(Q_i) -> 1 : Q_i'=dequeue(Q_i);
35         ...
36 endmodule
```

Listing 1 Simplified model of Bitcoin.

lines 13-15: it may mine a new block. This operation has a rate $\mathtt{mR} \times \mathtt{hR}_i$ that indicates the miner's rate of generating new blocks ($\mathtt{hR}_i$ is the miner's hashing power, while `mR` is the difficulty level of the crytopuzzle [20]; this is how we abstract away from the proof-of-work technique for mining blocks). When a block is created by a miner – operation `NewB` –, it is added to the local ledger – operation `AddB` – and it is stored in the variable $\mathtt{b}_i$. The state of the miner becomes `Winner`.

line 24: when `Miner`'s state is `Winner`, the miner synchronizes with `Network` using the action $\mathtt{addblock}_i$; the `Network` stores the new block in the bags of the other miners. The state of $\mathtt{Miner}_i$ is set back to `Init`.

lines 17-18: it may add a block to the ledger from the local queue. The predicate $\mathtt{canAdd}(\mathtt{L}_i, \mathtt{top}(\mathtt{QMiner}_i))$ verifies that the parent of the block on top of the queue $\mathtt{QMiner}_i$ is already stored in $\mathtt{L}_i$. The corresponding updates on $\mathtt{L}_i$ and $\mathtt{QMiner}_i$ are performed. The time spent in doing this action is simulated by the rate `r`. Clearly, this rate is much higher than the other rates because it corresponds to local management operations of the Miner (therefore, the probability that a Miner tries to add a block in his ledger is way higher than the probability of receiving a new block or mining).

lines 20: it may try to add a block to $\mathtt{L}_i$ that cannot be added either because parent's block is still not stored in $\mathtt{L}_i$ or because $\mathtt{QMiner}_i$ is empty. In this case, the block is enqueued in $\mathtt{QMiner}_i$ (thus guaranteeing a fair behaviour).

line 22: it may receive a block from the network through the action $\mathtt{addBlock}_i$. In this case the block is added to $\mathtt{QMiner}_i$. The synchronization on $\mathtt{addBlock}_i$ has a rate $\mathtt{r}_i$, which simulates the latency of the network. In fact, as explained in [6], the communication delay across the Bitcoin network can be also approximated by an exponential distribution.

The `NETWORK` module is defined in Listing 1, lines from 27 to 36. It simulates the broadcast of new blocks to the miners. In particular, the module has one queue per miner that stores the messages (the blocks) to be delivered to the corresponding miner. When a node i mines a new block, the block is added to every miner's queue, except the one of miner i (line 32).

Properties. In the remaining part of the section we compute the probability of the Bitcoin system defined in Listing 1 to devolve into inconsistent states, e.g. into a state where at least two nodes have different ledgers. In order to ease our arguments, among the possible states of the stochastic transition system obtained from the model, we select those where the blocks have all been delivered. This scenario is usual in Bitcoin because the rate of block delivery is much higher than the one of mining. For example, the nodes that have not yet received the last block after 40 seconds are less than 5%, whilst blocks are mined every 10 minutes [6].

▶ **Definition 1.** *A state of a Bitcoin system is called* completed *when there is no block to deliver (every* $\mathtt{Q}_i$ *in* `NETWORK` *is empty) and the blocks in the local queues of* $\mathtt{MINER}_i$ *have already been inserted in the corresponding ledgers (every* $\mathtt{QMiner}_i$ *in* $\mathtt{MINER}_i$ *is empty).*

▶ **Proposition 2.** *Let P be a completed state of a Bitcoin system and let $\mathtt{L}_1$ and $\mathtt{L}_2$ be two ledgers in different nodes. Then the trees of $\mathtt{L}_1$ and $\mathtt{L}_2$ are equal. Therefore, if $\mathtt{L}_1 \neq \mathtt{L}_2$ then $\mathtt{handle}(\mathtt{L}_1) \neq \mathtt{handle}(\mathtt{L}_2)$.*

▶ **Definition 3.** *Let $\mathtt{L}_1$ and $\mathtt{L}_2$ be two ledgers and let*
- *m_1 be the length of $\mathtt{L}_1\uparrow$,*
- *m_2 be the length of $\mathtt{L}_2\uparrow$,*
- *h be the length of the maximal common suffix of $\mathtt{L}_1\uparrow$ and $\mathtt{L}_2\uparrow$.*

We say that $\mathtt{L}_1$ and $\mathtt{L}_2$ have a fork of length k, *where $k = \max(m_1 - h, m_2 - h)$.*

For the sake of simplicity, in the following theorem:
- we shorten $\mathtt{mR} \times \mathtt{hR_i}$ into r_{w_i};
- the rates $r_1, \ldots, r_n$ of actions $\mathtt{addBlock}_1, \ldots, \mathtt{addBlock}_n$ are all considered identical by taking the parameter of the exponential distribution mean, which we call $\widehat{r}$ (actually these rates are parameters of an exponential distribution [6]);
- the rate r that corresponds to local management operations by Miners is approximated to 1 because the other rates are very small values less than 1.

▶ **Theorem 4.** *Let P be a completed state of a Bitcoin system consisting of n miners with ledgers $\mathtt{L}_1, \ldots, \mathtt{L}_n$, respectively, such that $\mathtt{L}_1 = \cdots = \mathtt{L}_k$ and $\mathtt{L}_{k+1} = \cdots = \mathtt{L}_n$ and $\mathtt{L}_1 \neq \mathtt{L}_{k+1}$. Let $\mathtt{L}_1$ and $\mathtt{L}_{k+1}$ have fork of length m. Then the probability $\mathsf{Prob}(P_{\leadsto m+1})$ to reach a completed state with fork of length $m+1$ is smaller than ($R = \sum_{j=1}^{n} r_{w_j}$)*

$$\sum_{\substack{1 \leq i \leq n \\ H \subset \{1,\ldots,n\} \setminus i \\ i \leq k \;\Rightarrow\; j \in \{k+1,\ldots,n\} \setminus H \\ i > k \;\Rightarrow\; j \in \{1,\ldots,k\} \setminus H}} \Theta(i, |H|, j)$$

where

$$\Theta(i,\ell,j) = \frac{r_{w_i}\, r_{w_j}}{R\ (R + (n-1-\ell)\widehat{r})} \prod_{1 \leq h \leq \ell} \frac{h\ \widehat{r}}{R + (n-h)\widehat{r}} \prod_{1 \leq a \leq 2n-2-\ell} \frac{a\ \widehat{r}}{R + a\ \widehat{r}}\,.$$

It is worth to notice that $\mathsf{Prob}(P_{\rightsquigarrow m+1})$ of Theorem 4 depends on the number n of nodes, their hashing power r_{w_i} and the latency r_i of the network with respect to the node i. To explain the probability, assume to have a fork of length one due to miners having equal ledgers (since the state is completed) and two different handlers. Let $1, \cdots, k$ be the nodes with one ledger and $k+1, \cdots, n$ be the nodes with the other ledger. Assume that a node $1 \leq i \leq k$ mines a new block; the probability will be $\frac{r_{w_i}}{R}$. The new block is then communicated to a set H of nodes that immediately add it to the local ledger. This operation happens with probability $\left(\prod_{1 \leq h \leq |H|} \frac{h\,\widehat{r}}{R + (n-h)\widehat{r}}\right)$. At this point, in order to obtain a fork of length 2, a node $j \in \{k+1, \ldots, n\} \setminus H$ must mine a block as well. The probability of this operation is $\frac{r_{w_j}}{R + (n-1-\ell)\widehat{r}}$. Finally, every node receives the two mined blocks, which has a probability $\left(\prod_{1 \leq a \leq 2n-2-\ell} \frac{a\,\widehat{r}}{R + a\widehat{r}}\right)$. Obviously, the same result can be obtained if the first node that mines a block belongs to the second partition ($j \in \{k+1, \ldots, n\}$). Henceforth, the probability to reach a completed state with fork of length 2 from the initial state is

$$\sum_{\substack{1 \leq i \leq k \\ H \subset \{1, \cdots, n\} \setminus i \\ j \in \{k+1, \cdots, n\} \setminus H}} \Theta(i, |H|, j) + \sum_{\substack{k+1 \leq j \leq n \\ H \subset \{1, \cdots, n\} \setminus j \\ i \in \{1, \cdots, k\} \setminus H}} \Theta(j, |H|, i)$$

which is exactly what stated in the theorem.

Using a technique similar to Theorem 4 we may compute the probability that a Bitcoin system in a completed consistent state (the nodes have all the same ledger) devolves into an inconsistent state. In this case, the proof is simpler than Theorem 4 because every node may mine after the first one.

▶ **Proposition 5.** *Let P be a completed state of a Bitcoin system consisting of n miners having ledger $\mathtt{L}$. The probability $\mathsf{Prob}(P_{\rightsquigarrow 1})$ to reach a completed state with fork of length 1 is smaller than ($R = \sum_{j=1}^{n} r_{w_j}$)*

$$\sum_{\substack{1 \leq i \leq n \\ H \subset \{1, \ldots, n\} \setminus i \\ j \in \{1, \ldots, n\} \setminus H}} \Theta(i, |H|, j)$$

where

$$\Theta(i, \ell, j) = \frac{r_{w_i}\ r_{w_j}}{R\ (R + (n-1-\ell)\widehat{r})} \prod_{1 \leq h \leq \ell} \frac{h\,\widehat{r}}{R + (n-h)\widehat{r}} \prod_{1 \leq a \leq 2n-2-\ell} \frac{a\,\widehat{r}}{R + a\,\widehat{r}}.$$

In order to bear some numerical results, we instantiate our probability with realistic rates. In [20], the time a miner takes to create a block is exponential with parameter θ, which represents the probability that the miner solves the cryptopuzzle problem in a given time-slot [1]. It follows that $\theta = h/D$, where h is miner's hashing power and D is the cryptopuzzle difficulty set by the protocol in order to set constant to 10 minutes the average duration between two blocks. In our encoding, θ is represented by r_{w_i}, therefore $r_{w_i} = h_i/D$ and, taking the current hashing power distribution of the Bitcoin system illustrated in Figure 1, and letting $D = 600$, we obtain the channel rates of the main pools in the Bitcoin system. As regards the broadcast of messages in the Bitcoin protocol, it is a combination of the transmission time and the local verification of the block. From [6] we know that in a Bitcoin environment, the broadcast can be approximated as an exponential distribution with mean time 12.6 seconds. Therefore we may assume that every r_i is 1/12.6.

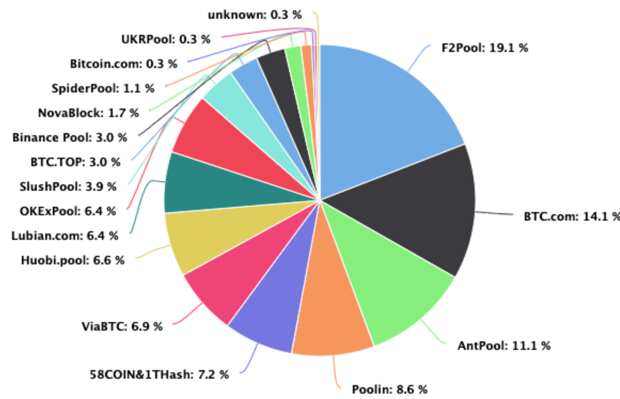


Figure 1 Hashrate distribution of Bitcoin mining pools on May 2020.
Source: `https://www.blockchain.com/`.

In the following Figures, we compare the outputs of our probability formula in Theorem 4 when the nodes are either 1000 (green line) or 17 (blue line). Furthermore, we also highlight the results obtained via the simulation with 17 nodes (red line) from the companion paper [2] because they are compliant with our upper bounds. We did not run simulations on larger sets of nodes because they took too much time (around 48 hours per simulation on a Virtual Machine with 8 VCPU and 64 GB RAM). We have also computed the formula in Theorem 4

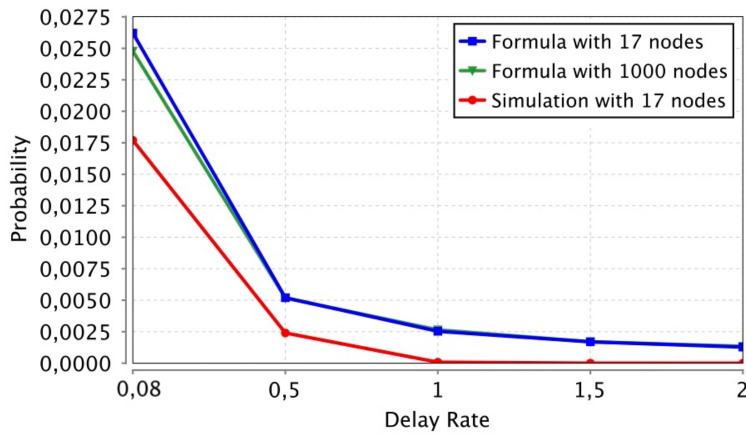


Figure 2 Probability of reaching a fork of length 1 by varying the broadcast delay.

with 10000 nodes: the output has not been displayed because it overlaps with the case of 1000 nodes. The first analysis we present in Figure 2 is the computation of the probability of reaching a state where at least two different blockchains differ for one block (fork of length 1) by varying the broadcast delay. In particular, Figure 2 compares the outputs of our probability formula (both with 17 and 1000 nodes) with results of the simulation we have done with 17 nodes representing the main pools in the Bitcoin system. The reader may notice that the probability decreases with the increase of the communication delay rate. This follows from the remark that the higher is the rate, the smaller is the expected time for the transition to occur. We notice that, with rate $\widehat{r} = 0.08$, we obtain results in line with those of [6]. In particular, for a broadcast delay with mean 2, we obtain that the probability of a

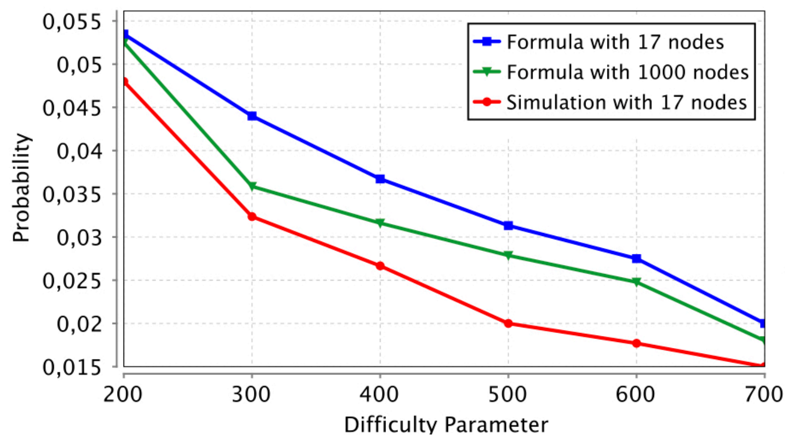


Figure 3 Probability of a fork of length 1 with different difficulty parameter.

fork of length 2 is very low. Figure 3 displays the probability of reaching a fork of length 1 by varying the cryptopuzzle difficulties (parameter D). The reader can observe that the probabilities computed by the formula, also in the case of 1000 nodes, is always an upper bound of the results obtained via simulation. Finally, Figure 4 illustrate the probability of reaching completed states with longer and longer forks. Also in this case, the results of our

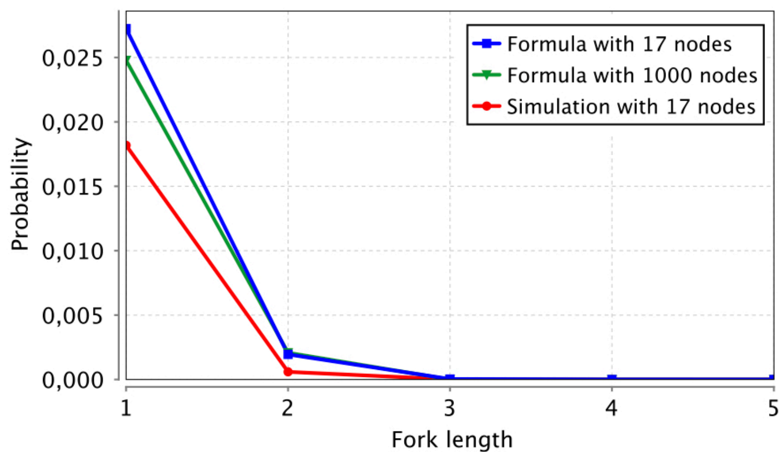


Figure 4 Probability of a fork of increasing length, comparison between formula and simulation results.

simulation are in line with those given by the formula both with 17 and 1000 nodes. In the case of 17 nodes, the probability computed is higher because each miner owns a larger amount of hashing power. Therefore, every miner is more likely to win the cryptopuzzle game. As the reader can observe, the probability to obtain a fork of length 5 is of the order of 10^{-8},

while it is approximately zero when the length of the fork reaches 6. This is a key result, because every block at depth 6 is considered permanent in the Bitcoin blockchain (e.g. the majority of miners have consistent blockchains up to depth 6 with probability almost 1).

5 Analysis of a possible attack

In this section we model and analyze an attack to Bitcoin that has been described in [20], namely a hostile miner tries to create an alternate chain faster than the honest one. This scenario admits that a merchant can be convinced that a transaction has been accepted and then create a new branch of the chain, longer than the valid one, with some other transaction spending the same money (double spending attack).

Let $\mathtt{MINER}_{Hack}$ be the dishonest miner; technically, its behaviour differs from $\mathtt{MINER}_i$ because it mines on a block $\mathtt{b}_{Hack}$ that is not the correct one (e.g. the handle of the ledger). In particular, the operation NewB in $\mathtt{MINER}_{Hack}$ takes an ad-hoc block $\mathtt{b}_{Hack}$ rather than $\mathtt{handle}(\mathtt{L}_{Hack})$. The definition of $\mathtt{MINER}_{Hack}$ is given in Listing 2.

```
37 // states of Miner_Hack: Init = 0, Winner = 1
38 // mR = 1/600 is the Bitcoin mining rate
39 // hR_Hack is the percentage of hashing power of Miner_Hack, 0≤hR_Hack ≤1
40 // r_Hack is the communication delay rate of Miner_Hack
41
42 module Miner_Hack
43   integer Miner_Hack_STATE = Init;
44   block b_Hack = (gen^0,gen^0);
45   ledger L_Hack = ⟨{(gen^0,gen^0)};gen^0⟩;
46   integer c_Hack = 0;
47   queue QMiner_Hack = [];
48
49   [] Miner_Hack_STATE=Init -> mR×hR_Hack : L_Hack' = AddB(L,NewB(Miner_Hack,c,b_Hack)
50                                          & c_Hack' = c_Hack+1
51                                          & b_Hack' = NewB(Miner_Hack,c,b_Hack)
52                                          & Miner_Hack_STATE' = Winner;
53
54   [] Miner_Hack_STATE=Init&canAdd(L_Hack,top(QMiner_Hack)) ->
55                                      r : QMiner_Hack'=dequeue(QMiner_Hack)
56                                          & L_Hack'=addB(L_Hack,top(QMiner_Hack));
57
58   [] Miner_Hack_STATE=Init&!canAdd(L_Hack,top(QMiner_Hack)) ->
59                                      r : QMiner_Hack'= deq_enq(QMiner_Hack);
60
61   [addBlock_Hack] Miner_Hack_STATE=Init -> r_Hack :
62                                  QMiner_Hack' = enqueue(QMiner_Hack,top(Q_Hack))
63
64   [addBlock_Hack] Miner_Hack_STATE=Winner -> r_Hack : Miner_Hack_STATE'=Init;
65 endmodule
```

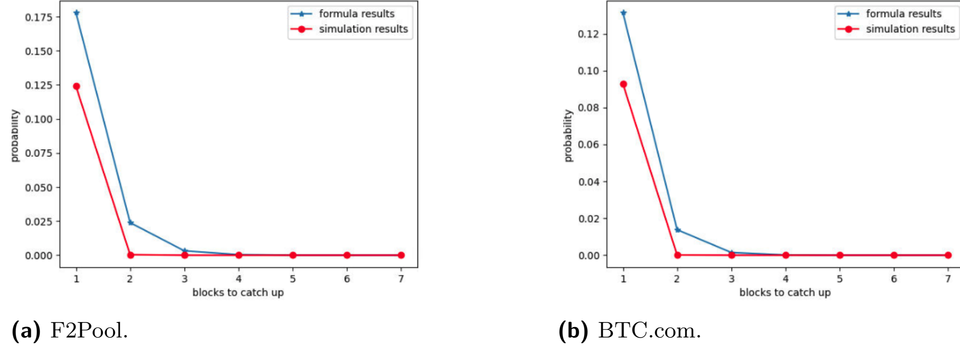
Listing 2 Simplified model of a dishonest Miner.

Following the same pattern of Section 4 and letting $r_{w_{Hack}} = \mathtt{mR} \times \mathtt{hR}_{\mathtt{Hack}}$:

▶ **Theorem 6.** *Let P be a completed state of a Bitcoin system of n miners with exactly one that is hostile and let $r_{w_{Hack}}$ its mining rate. The probability $\mathsf{Prob}(P_m)$ to reach a completed state where the hostile miner has created an alternate chain longer than the honest one from $m, m \geq 1$, blocks behind is smaller than ($R = \sum_{j=1}^{n} r_{w_j}$ and we assume that, for every i, j, $\widehat{r} = r_i = r_j$)*

$$\sum_{k\geq 1}\left[\Phi(r_{w_{Hack}},\widehat{r},R)^k\left(\sum_{1\leq j\leq n-1}\Phi(r_{w_j},\widehat{r},R)\right)^{k-1}\right]^m$$

$$\textit{where}\quad \Phi(r_w,r,R) = \frac{r_w}{R}\prod_{1\leq a\leq n-1}\frac{a\;\widehat{r}}{R+(n-a)\widehat{r}}.$$



(a) F2Pool. **(b)** BTC.com.

Figure 5 Probability of a successful attack for two main pools of Bitcoin system.

As for Theorem 4, the technique used for demonstrating the above statement consists of analyzing the stochastic transition system. To explain the probability, assume to be in a completed state and compute the probability to reach a completed state in which the dishonest node has created an alternate chain from m blocks behind. We start by computing the probability that the dishonest node $\texttt{Miner}_{Hack}$ has caught up by one block. This kind of attack succeeds if $\texttt{Miner}_{Hack}$ mines one block and this happens with probability $\frac{r_{w_{Hack}}}{R}$. It may also happens that honest nodes mine k blocks and $\texttt{Miner}_{Hack}$ mines $k+1$ blocks in the same amount of time. Considering also the probability that the new blocks have been received by the miners, we obtain the formula

$$\sum_{k\geq 1}\Big(\frac{r_{w_{Hack}}}{R}\prod_{1\leq a\leq n-1}\frac{a\,\widehat{r}}{R+(n-a)\widehat{r}}\Big)^{k}\Big(\sum_{1\leq j\leq n-1}\frac{r_{w_j}}{R}\prod_{1\leq a\leq n-1}\frac{a\,\widehat{r}}{R+(n-a)\widehat{r}}\Big)^{k-1}$$

Therefore, the probability $\mathsf{Prob}(P_m)$ that $\texttt{Miner}_{Hack}$ creates an alternative chain faster than the honest nodes from m blocks behind is given by

$$\sum_{k\geq 1}\Big[\Big(\frac{r_{w_{Hack}}}{R}\prod_{1\leq a\leq n-1}\frac{a\,\widehat{r}}{R+(n-a)\widehat{r}}\Big)^{k}\Big(\sum_{1\leq j\leq n-1}\frac{r_{w_j}}{R}\prod_{1\leq a\leq n-1}\frac{a\,\widehat{r}}{R+(n-a)\widehat{r}}\Big)^{k-1}\Big]^{m}$$

which is what what Theorem 6 states. It is worth to notice that this technique is different from the one in [20], where Nakamoto assumed *a priori* that the ratio between the blocks mined by the attacker and those mined by the honest miners is the expected value of a Poisson distribution. In particular, we do not assume that miners' behaviour can be described by a certain statistical model, therefore our context is less restrictive. We also notice that Poisson distribution expresses the probability of a certain event occurs in a time period, independently of the time since the last event. Thus, Nakamoto models the attack counting the number of minings of the attacker in an interval of time, assuming that the probability for success does not change during the experiment. In our case, the probability is computed as the attacker was a standard node and its mining activity was in competition with the same process of the other nodes.

In Figure 5 we illustrate the probability of a successful attack by an hostile node, depending on the number m of blocks to catch up. We analyze two scenarios that highlight the cases when two main Bitcoin pools (see Figure 1) decide to become hostile.

In each image we plot the results given by the formula (blue line) and the results obtained via simulation (red line) for two main miners of the Bitcoin system. As well as for the previous analysis, the probability given by the formula is an upper bound for the results

obtained via simulation. We derive from Figure 5 that the probability a hostile miner catches up from 1 block behind increases with the percentage of the hashing power and drops with the number of blocks to catch up.

6 Related works

The protocol used by Bitcoin was introduced by Haber and Stornetta [14] and only in the last few years, because of Bitcoin, the problem of analyzing the consistency of the ledgers has caught the interest of several researchers.

The formal analysis of the protocol by means of abstract models has been already done in [13, 11, 22, 23]. In [13], the author discusses the blockchain consensus in Bitcoin and Ethereum and compare them with the classic Byzantine consensus. The miners are defined in pseudo-code (without any semantics) and the analysis is probabilistic rather than stochastic. In [11], Garay *et al.* demonstrate the correctness of the protocol when the network communications are synchronous, focusing on its two key security properties: *Common Prefix* and *Chain Quality*. The first property guarantees the existence of a common prefix of blocks among the chain of honest players; Chain Quality constrains the number of blocks mined by hostile players, when the honest players are in the majority and follow the protocol. The extension of this analysis to asynchronous networks with bounded delays of communications and with new nodes joining the network has been undertaken in [22]. In the above contributions, the properties are verified by using oracles that drive the behaviours of actors. Then, combining the probabilistic behaviours and assuming possible distributions, one computes expected values. In [23], Pirlea and Sergey propose a formalization of Bitcoin consensus focusing on the notion of global system safety. They present an operational model that provides an executable semantics of the system where nondeterminism is managed by external schedules and demonstrate the correctness by means of a proof assistant. The main difference between these contributions and our work is that we formalize the Bitcoin protocol as a stochastic system (with exponential distribution of durations) and derive the properties by studying the model. In fact, the probabilities that we compute are, up to our knowledge, original. As regards stochastic models and Bitcoin, few recent researches use them to select optimal strategies for maximizing profit of a player [1] and for formalizing interactions between miners as a game [5, 3].

A number of researches address attacks to the Bitcoin protocol. The works [6, 25, 12] address the delays of communications and [25] also demonstrates that an attacker with more than 51% of the total hashing power could change the past transactions. A larger set of attacks is analyzed in [19, 11, 22], where it is also proved that the Bitcoin protocol is safe as long as honest miners are in the majority. In [21], Ozisik and Levine give a very detailed description of Nakamoto's double spending attack, gathering the mathematics for its modelling. The probability of a successful double spending attack in several scenarios (both fast and slow payments) is analyzed in [16]. Finally, a fully implemented attack against Ethereum blockchain, which covers both a network and a double spending attack, is delivered in [7]. In contrast with these contributions, our results are achieved by analyzing a stochastic transition system, rather than constraining miners' behaviour to adhere to a certain statistical model.

7 Conclusions

We have studied the probability that the blockchain protocol may devolve the ledger into inconsistent copies because of forks. Two cases have been analyzed: the first one is when the system consists of honest miners; the second one is when the system has an hostile node that

mines blocks in wrong positions. The adversary model used in this paper is not the best one an adversary can implement, but the analysis of further strategies is left to future work. Our results are gathered by modelling the Bitcoin system in a stochastic process calculus, `PRISM+`, which has also an automatic tool for analysing systems that exhibit random or probabilistic behaviours. `PRISM+` extends `PRISM` [18] with a library that models Bitcoin datatypes, such as ledgers and blocks.

The main contribution of this paper is the formal demonstration of the probability that Bitcoin ledgers may devolve into inconsistent states, also in presence of attacks. Our probabilities are parametric with respect to the number of nodes, their hashing power and the latency of the network. This work has required a time-consuming analysis of the stochastic transition system. It turns out that our results comply with simulations performed on `PRISM+` systems with at most 17 nodes because of scalability problems (see also [2]).

Our approach is, as far as we know, original and the technique can be applied to analyze other well-known attacks to the Bitcoin protocol, such as failures either of communications or of miners, the inception of new miners that may be hostile, etc. In the future research we also plan to model other blockchain protocols, such as Ethereum or the so-called Proof-of-Stake. The presence of a probabilistic model checker like `PRISM+` will allow us to deliver simulation results without much effort. In this respect, we will try to mitigate the scalability issues we had up to now.

References

1. Bruno Biais, Christophe Bisiere, Matthieu Bouvard, and Catherine Casamatta. The blockchain folk theorem, 2018.
2. S. Bistarelli, R. De Nicola, L. Galletta, C. Laneve, I. Mercanti, and A. Veschetti. Stochastic modelling and analysis of bitcoin. Manuscript submitted for publication, 2020.
3. Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A. Kroll, and Edward W. Felten. Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In *Proc. of SP 2015*, pages 104–121. IEEE Computer Society, 2015.
4. R. Bowden, H. Paul Keeler, Anthony E. Krzesinski, and Peter G. Taylor. Block arrivals in the bitcoin blockchain. *CoRR*, abs/1801.07447, 2018.
5. Miles Carlsten, Harry Kalodner, S. Matthew Weinberg, and Arvind Narayanan. On the instability of bitcoin without the block reward. In *Proc. Computer and Communications Security*, CCS '16, pages 154–167. ACM, 2016.
6. Christian Decker and Roger Wattenhofer. Information propagation in the bitcoin network. In *Proc. 13th IEEE P2P*, pages 1–10. IEEE, 2013.
7. Parinya Ekparinya, Vincent Gramoli, and Guillaume Jourjon. Double-spending risk quantification in private, consortium and public ethereum blockchains. *CoRR*, abs/1805.05004, 2018. `arXiv:1805.05004`.
8. Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. *Commun. ACM*, 61(7):95–102, June 2018. `doi:10.1145/3212998`.
9. Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.
10. A. Fox and E. A. Brewer. Harvest, yield, and scalable tolerant systems. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, pages 174–178, 1999.
11. Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Proc. of EUROCRYPT 2015*, volume 9057 of *Lecture Notes in Computer Science*, pages 281–310. Springer, 2015.
12. Johannes Göbel, Holger Paul Keeler, Anthony E. Krzesinski, and Peter G. Taylor. Bitcoin blockchain dynamics: The selfish-mine strategy in the presence of propagation delay. *Perform. Eval.*, 104:23–41, 2016.

13 Vincent Gramoli. From blockchain consensus back to byzantine consensus. *Future Gener. Comput. Syst.*, 107:760–769, 2020.

14 Stuart Haber and W. Scott Stornetta. How to time-stamp a digital document. *Journal of Cryptology*, 3(2):99–111, January 1991. `doi:10.1007/BF00196791`.

15 Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

16 Ghassan Karame, Elli Androulaki, and Srdjan Capkun. Double-spending fast payments in bitcoin. In *Proc. of CCS'12*, pages 906–917. ACM, 2012.

17 M. Kwiatkowska, G. Norman, and D. Parker. Stochastic model checking. In M. Bernardo and J. Hillston, editors, *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07)*, volume 4486 of *LNCS (Tutorial Volume)*, pages 220–270. Springer, 2007.

18 Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. In *Proc. TACAS 2002*, volume 2280 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 2002.

19 Andrew Miller and Joseph J LaViola Jr. Anonymous byzantine consensus from moderately-hard puzzles: A model for bitcoin. *Available on line: http://nakamotoinstitute.org/research/anonymous-byzantine-consensus*, 2014.

20 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. URL: `http://www.bitcoin.org/bitcoin.pdf`.

21 A. Pinar Ozisik and Brian Neil Levine. An explanation of nakamoto's analysis of double-spend attacks. *CoRR*, abs/1701.03977, 2017. `arXiv:1701.03977`.

22 Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In *Proc. of EUROCRYPT 2017*, volume 10210 of *Lecture Notes in Computer Science*, pages 643–673. Springer, 2017.

23 George Pîrlea and Ilya Sergey. Mechanising blockchain consensus. In *Proc. 7th Certified Programs and Proofs, CPP*, pages 78–90. ACM, 2018.

24 Muhammad Saad, Victor Cook, Lan Nguyen, My T. Thai, and Aziz Mohaisen. Partitioning attacks on bitcoin: Colliding space, time, and logic. In *39th IEEE Conference on Distributed Computing Systems, ICDCS 2019*, pages 1175–1187. IEEE, 2019.

25 Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in bitcoin. In *Proc. of Financial Cryptography and Data Security 2015*, volume 8975 of *Lecture Notes in Computer Science*, pages 507–527. Springer, 2015.

26 Alexei Zamyatin, Nicholas Stifter, Philipp Schindler, Edgar R. Weippl, and William J. Knottenbelt. Flux: Revisiting near blocks for proof-of-work blockchains. *IACR Cryptology ePrint Archive*, 2018:415, 2018.

Deconfined Intersection Types in Java

Dedicated to Maurizio Gabbrielli on the Occasion of His 60th Birthday

Mariangiola Dezani-Ciancaglini
Computer Science Department, University of Torino, Italy
dezani@di.unito.it

Paola Giannini
Department of Advanced Science and Technology, University of Eastern Piedmont, Alessandria, Italy
paola.giannini@uniupo.it

Betti Venneri
Department of Statistics, Computer Science, Applications, University of Firenze, Italy
betti.venneri@unifi.it

Abstract

We show how Java intersection types can be freed from their confinement in type casts, in such a way that the proposed Java extension is safe and fully compatible with the current language. To this aim, we exploit two calculi which formalise the simple Java core and the extended language, respectively. Namely, the second calculus extends the first one by allowing an intersection type to be used anywhere in place of a nominal type. We define a translation algorithm, compiling programs of the extended language into programs of the former calculus. The key point is the interaction between λ-expressions and intersection types, that adds safe expressiveness while being the crucial matter in the translation. We prove that the translation preserves typing and semantics. Thus, typed programs in the proposed extension are translated to typed Java programs. Moreover, semantics of translated programs coincides with the one of the source programs.

2012 ACM Subject Classification Theory of computation → Type structures; Theory of computation → Semantics and reasoning

Keywords and phrases Intersection Types, Featherweight Java, Lambda Expressions

Digital Object Identifier 10.4230/OASIcs.Gabbrielli.2020.3

Acknowledgements We are very pleased to dedicate this work to Maurizio, whose notable contributions to the theory of programming languages are well known and appreciated by the scientific community.
We would like to thank the anonymous referees for their helpful comments.

1 Introduction

Intersection types have been proposed at the beginning of the Eighties [6, 7, 21, 8, 1] for typing all strongly normalising λ-terms and for building models of λ-calculus. Intersection types provide a form of polymorphism that is an attractive suggestion for programming languages. Indeed, they are able to express a huge (potentially infinite) amount of types about a component of a program, simply by listing the ones that matter in each context. On the other hand, intersection type inference turns out to be quite powerful, since it allows the typing of exactly all terminating programs; this power results in many difficult issues for their implementation. John Reynolds designed the first programming language including intersection types, the Algol-like academic language Forsyte [23, 24], in late Nineties. Since then, only recently intersection types have won the attention of language designers and have been successfully included in real programming languages in some restricted forms, e.g., Scala [9, 19, 22]. Concerning Java, in the last years intersection types have gained small spaces, step by step, in the successive releases of the language [14] (Section 4.9).

© Mariangiola Dezani-Ciancaglini, Paola Giannini, and Betti Venneri;
licensed under Creative Commons License CC-BY

Java intersection types take the form $\mathsf{T}\&\mathsf{I}_1\&\ldots\&\mathsf{I}_n$, where T is a class or an interface and $\mathsf{I}_1,\ldots,\mathsf{I}_n$ are interfaces. Thus intersection types break the nominal type system of Java: they allow the user to combine nominal types on demand, and so to express the formal specification of pieces of code as combinations of those interfaces which are exactly needed in each context. This is handy for keeping interfaces very thin (according to the Interface Segregation Principle [18]) and so avoiding interface pollution. However, despite these advantages, intersection types are among the features of Java which have not been adopted by the bulk of the programming community. The main motivation is, in our opinion, the fact that in essence they have never gained full dignity of language types, that is in all sense equivalent to nominal types and therefore as usable as classes and interfaces. We can sum up the boundaries into which Java actually confines their use, as follows:

- as bounds of type variables in generics definitions;
- as target types in explicit type casts;
- as anonymous types that are used only by the type inference system, in particular for typing conditional expressions.

What we immediately notice is that Java lacks support for using intersection types for fields, parameter and return types of methods. This becomes a crucial restriction when considering, in particular, λ-expressions. In Java each λ-expression is always associated with its target type, that is inferred from the enclosing context by the typing system. The target type must be a *functional type*: either a functional interface or an intersection of interfaces, containing exactly one abstract method and any number of *default method* definitions. Thus, the λ-expression provides the implementation for the abstract method, while default methods add new behaviour to the λ-expression. The primary motivation for introducing default methods in Java 8 was interface evolution, i.e., the ability of extending interfaces with new functionalities without breaking down the existing subclasses. But an interesting development arises from the combination of default methods with intersection types and λ-expressions. Assume we have defined several interfaces, containing default methods that can be used and reused in distinct pieces of code. Then each context can choose on the fly the functionalities that are needed and compose them with the abstract method, by casting the λ-expression to the suited intersection type. For instance, in the term $(\mathsf{I}\&\mathsf{I}_1\&\ldots\&\mathsf{I}_n)\,(\mathsf{x}\to\mathsf{t})$, the intersection type becomes the target type of $\mathsf{x}\to\mathsf{t}$. In this case, in addition to implementing the abstract method of I, the λ-expression $\mathsf{x}\to\mathsf{t}$ can be immediately used as the receiver of any default method defined in the other interfaces. This really increases flexibility in using the function $\mathsf{x}\to\mathsf{t}$. The use of cast is needed in Java, where signatures only contain nominal types. Instead, in the proposed extension a λ-expression can get an intersection type as target type even occurring as parameter or return value of a method (see the last example of Section 2).

Concerning generics [4], it is convenient to dispel the false belief that the use of intersection types as bound of generic type variables in some way can make up for missing freely used intersection types. The generic type variable is a placeholder for a type which is unknown to the compiler, until the caller chooses an actual type to replace the type variable. Differently, using intersection types in parameter and return types of a method declaration, we express a precise constraint, that is the type of the actual parameter must implement a given list of types (formal specifications) and so for the return type. For example, the declaration $<\mathsf{X}\ \mathsf{extends}\ \mathsf{I}_1\&\mathsf{I}_2>\mathsf{X}\ \mathsf{mGen}(\mathsf{X}\ \mathsf{x})$ is totally different from $\mathsf{I}_1\&\mathsf{I}_2\ \mathsf{mInt}(\mathsf{I}_1\&\mathsf{I}_2\mathsf{x})$. Let C and D be two unrelated classes implementing both I_1 and I_2. By instantiating X with C (or D) in mGen, we can apply mGen to an object of type C and the result is of type C (or D, respectively). Differently, method mInt can be applied to an object of type C, or to an object of type D; the result can be an object of any type implementing both I_1 and I_2. For instance, the result

will be an object of class C or D, when the body of the method is a conditional expression returning an object C and an object D in the two branches, respectively. Therefore, since the expressive power of intersection types is totally orthogonal to the use of generic types, we leave out generics from the minimal core languages that are exploited in this paper.

In Section 2, we present many examples that show the high degree of boilerplate coding required in Java because of the above restrictions on the usage of intersection types. Our proposal is to desegregate Java intersection types from those restrictions, so that the programmer can use them as field types in class declarations and as parameter and return types in method signatures, as it does with nominal types. This extension of Java is proven to be safe and fully compatible with the current language, that is it does not require any modification of the Java Virtual Machine (JVM), thus keeping the essential property of backward compatibility. The main contribution of this paper is the compilation of the proposed extension into Java core and the proof that compiled programs preserve types and semantics of the source program. To this aim, we exploit two calculi, which formalise the simple Java core and the extended language, respectively. These calculi are minimal core languages in that they omit all the features that are not significant for our purpose.

The first calculus, TJ&, is presented in Section 3. It models how Java 8 deals with λ-expressions and intersection types, that are confined within the above restrictions. TJ& is a lightweight version of FJ&λ, defined in [2], since some features, such as conditional expressions, are avoided to direct attention to the essential points for our issue.

The calculus SJ&+, presented in Section 4, formalises the proposed Java extension. The calculus defined in [10] is a conservative extension of SJ&+, so we inherit from [10] the main properties of type preservation and progress.

In Section 5, we present the compilation algorithm for translating typed SJ&+ programs into TJ& programs. As the first step, we erase all the intersection types appearing in field declarations and in method signatures. Each erased intersection is replaced with its most relevant component, that is either the class or the functional interface (if any). Then the lost type information is recovered by inserting several downcasts into the source code. The intrinsic goal of the added type casts is preserving typing and semantics. As expected, the crucial issue is the translation of λ-expressions with their target types.

Properties of the translation are discussed in Section 6. We prove that translated programs are typed too. Then we show that inserted casts are guaranteed to not fail at run time. Thus source and target programs either produce "indistinguishable" values or both reduce forever. By "indistinguishable" we roughly mean that the difference between the values are type casts which never fail at run-time.

We conclude in Section 7 discussing related and future works.

2 Motivating Examples

This section is split into three parts. The first two parts show the advantages of the proposed extension also in presence of generic types and of the `var` construct. The third part exemplifies the expressivity of deconfined intersection types for typing λ-expressions. The actual Java code is on light-grey background, while the proposal Java code is on light-green background.

Generic Types

We want to implement a game in which players with different moving capabilities explore a world picking up objects as they go along. Object oriented modelling lends itself well to directly translating "real-world" *entities* and their *capabilities* into code, by describing capabilities via *interfaces* and entities via *classes* implementing their capabilities.

In our game some players can fly, some can swim and others can do both. Java interfaces are used to model the two moving capabilities, so we can have players implementing both interfaces as the following code shows.

```
interface Flyable { void fly(); }

interface Swimmable { void swim(); }

public class NaviatorDrone implements Flyable, Swimmable {
 public void fly() { ... }
 public void swim() { ... };
}

public class Pelican implements Flyable, Swimmable {
 public void fly() { ... }
 public void swim() { ... };
}
```

Moreover, we can define *actions*, implemented by *methods*, that may require players with either one of the two capabilities or both. We concentrate on modelling the latter.

We want to write a method, `goAcrossRavine`, that requires to fly over a ravine. If there is an object in the stream at the bottom, the player has to fly down, dive and swim into the water and then fly up, after having picked the object up. So we need `Flyable` and `Swimmable` players.

```
public class Game {
 public static void goAcrossRavine(XXX player, boolean underwaterObj){
  System.out.println("Reached␣the␣ravine");
  if (underwaterObj) {
   player.fly();
   player.swim();
   System.out.println("Picked␣Object");
   player.swim();
   player.fly();
  } else player.fly();
  System.out.println("Crossed␣the␣ravine");
 }
 // Other methods of the game using the capabilities of players
 }
```

We write `XXX` as type of the `player` parameter, since it must implement `Flyable` and `Swimmable`, but in Java we cannot specify `Flyable & Swimmable`. One natural solution is to define a new interface

```
interface FlyableSwimmable extends Flyable, Swimmable {}
```

and use it for `XXX`. Now, if we want to apply the method to our `NaviatorDrone` and `Pelican` we have to change their class definitions and make them implement `FlyableSwimmable`.

```
public class NaviatorDrone implements FlyableSwimmable { ... }

public class Pelican implements FlyableSwimmable { ... }

public class Game {
 public static void
        goAcrossRavine(FlyableSwimmable player, boolean underwaterObj){
    ...
 }
 // Other methods of the game using the capabilities of players
}
```

This change raises some problems, in particular

1. we may not have access to the implementation of `NaviatorDrone` and `Pelican` (that could come from different sources), and
2. we have to anticipate and define all combinations of the capabilities we will need in the evolution of the class `Game`.

The second point is particularly delicate, as it may lead to *interface proliferation*, one of the motivations behind the introduction of intersection types.

To avoid these problems we could use a generic variable with the intersection as a bound. So, instead of defining a new interface, the `player` parameter has a generic variable as type, whose bound is the intersection of `Flyable` and `Swimmable`.

```
public class Game {
 public static void <X extends Swimmable & Flyable>
        goAcrossRavine (X player, boolean underwaterObj){
       ...
 }
 // Other methods of the game using the capabilities of players
 public static void main() {
  // ...
  goAcrossRavine(new Pelican(), true);
  goAcrossRavine(new NaviatorDrone(), false);
 }
}
```

In Java *intersection types cannot be used in the declaration of variables*, i.e., declarations such as

```
Swimmable & Flyable player = new Pelican();
```

are not permitted, even though, as the last two lines of the previous code show, `new Pelican()` or `new NaviatorDrone()` can be arguments of the method `goAcrossRavine`.

We could try to use generic variables with the intersection `Swimmable & Flyable` as bound also for the variable definitions. However we need *type cast* that may cause type errors at run time.

```
public static <X extends Swimmable & Flyable> void main(){
 // ...
 X player = (X) new Pelican();
 goAcrossRavine(player, true);
}
```

Moreover, the interface of the method `main` would espose the type of a local variable!

The `var` Constructor

In Java 10 [16] the `var` construct was introduced having the prominent feature that the type of the declared variable is inferred from the expression assigned to it. The static type of the `var` variable is the type inferred for the expression on the right side. The expression cannot be a λ-expression. The inferred type can be an intersection type, for instance when the expression is a conditional expression. This can be useful to avoid boilerplate code in the body of a method: if the type of the variable is an intersection type you can call on it all methods of this intersection. However, the above benefits are restricted to local variables. In fact `var` variables cannot be used for fields, method parameters and return types, that require explicitly declared types (never inferred types).

For example, with this construct we can declare `player` variables that can be used where we require objects of intersection type, as the following code shows:

```
public static void main() {
 // ...
 var player = new Pelican();
 goAcrossRavine(player, true);
}
```

Allowing *intersections as return types of methods*, we can write the following method, which builds a player for our game:

```
enum Version{HIGHTECH, CLASSICAL}
```

```
public static Swimmable & Flyable makePlayer(Version v){
 return (v==Version.HIGHTECH)? new NaviatorDrone(): new Pelican());
}
```

In Java, we write a corresponding method using both generics and the `var` construct

```
public static<X extends Swimmable & Flyable> X makePlayer(Version v){
  var res=((v==Version.HIGHTECH)? new NaviatorDrone(): new Pelican());
  return (X)res;
}
```

We observe that a cast is needed for a correct compilation and the code is less readable.

λ-expressions

Intersection types and type inference are crucial for typing λ-*expressions*, another feature added to Java 8 [14] (Section 15.27).

Intersections of interfaces may be *target types* of λ-expressions. The *intersection of interfaces must be functional*, i.e., to have exactly one abstract method, the one implemented by the λ-expression. The limitations on the use of intersection types imposed by Java reduce the usability of λ-expressions, as the following example shows.

Assume we want to write a method `finalPrice` that computes the amount to charge for a purchase. This method can vary according to the algorithm for defining the discount and it must choose a policy for charging a delivery cost. We assume that different strategies for the delivery cost are encapsulated in the default methods of several interfaces. The reason motivating the use of interfaces with default methods, instead of classes (as in the Strategy Design Pattern [13]), is to allow these interfaces to appear in the type of a λ-expression, when combined in an intersection type with a functional interface. Thus the behaviour of `finalPrice` can be parametric with respect to one single λ-expression, to which the method delegates the definition of the delivery cost as well as the implementation of the discount algorithm.

For example, we assume the following simple declarations.

```
interface Discount { double discount(int price); }

interface DeliveryPrice {
 default double deliveryPrice(int price) {
  return (price>30)? 0: 5;
 }
}
```

Then the method for the final price would be very compact and clean in our proposed extension of Java, by using intersection types in parameter types. It could be defined as follows:

```
public static double
       finalPrice(Discount & DeliveryPrice funPrice, int price){
 return funPrice.discount(price)+funPrice.deliveryPrice(price);
}
```

For example, the following call of the method applies a 1% discount if the price is more than 100 euros and charges 5 euros for the delivery only if the price is less than or equal to 30 euros (for simplicity, the actual price is denoted by `n`):

```
double computedPrice=finalPrice(x->x-((x>100)? x*0.01: 0),n);
```

Differently, in Java, given the restrictions on the use of intersection types, we have to move the parameter into a local variable inside the method body, in order to obtain the behaviour above. Namely:

```
public static double finalPrice(int price) {
 var funPrice=(Discount & DeliveryPrice)(x->x-((x>100)? x*0.01: 0));
 return funPrice.discount(price)+funPrice.deliveryPrice(price);
}
```

Notice that this code compiles only if the λ-expression `x->x-((x>100)? x*0.01: 0)` is type cast. Most importantly, `finalPrice` is not parametric on the discount policy, i.e., we have to modify the method body for changing the discount algorithm.

Therefore, we can try to use Java generics, where the intersection type can be the bound of the type variable:

```
public static <X extends Discount & DeliveryPrice> double
       finalPrice(X funPrice, int price) {
 return funPrice.discount(price)+funPrice.deliveryPrice(price);
}
```

In this case, the call of `finalPrice` compiles if the passed λ-expression

`x->x-((x>100)? x*0.01: 0)`

is cast to the intersection type, i.e.,

```
double computedPrice=
      finalPrice((Discount & DeliveryPrice)(x->x-((x>100)? x*0.01: 0)),n)
          ;
```

compiles, while the code

```
double computedPrice=finalPrice(x->x-((x>100)? x*0.01: 0),n);
```

gives the error

```
Example.java:20:error:incompatible types: cannot infer type-variable(s) X
double computedPrice = finalPrice(x->x-((x>100)? x*0.01: 0),n);
                                          ^
    X extends Discount,DeliveryPrice declared in
                                      method<X>finalPrice(X,int)
  where INT#1 is an intersection type:
    INT#1 extends Object,Discount,DeliveryPrice
```

The discussion of this example shows the utility of default methods in interfaces, since they can be invoked not only on objects but also on λ-expressions.

3 Java with Confined Intersection Types (TJ&)

In this section we present our *target calculus* TJ& formalising the use of intersection types and λ-expressions in Java 8. A small extension of this calculus has been introduced in [2].

We use $\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}$ to denote classes, I, J to denote interfaces, T, U to denote nominal types, i.e., either classes or interfaces; f, g to denote field names; m, n to denote method names; t to denote terms; x, y to denote variables, including the special variable this. We use $\overrightarrow{\mathsf{I}}$ as a shorthand for the list $\mathsf{I}_1, \dots, \mathsf{I}_n$, $\overline{\mathsf{M}}$ as a shorthand for the sequence $\mathsf{M}_1 \dots \mathsf{M}_n$, and similarly for the other names. The order in lists and sequences is sometimes unimportant, and this is clear from the context. In rules, we write both $\overline{\mathsf{N}}$ as a declaration and $\overrightarrow{\mathsf{N}}$ for some name N: the meaning is that a sequence is declared and the list is obtained from the sequence adding commas. The notation $\overline{\mathsf{T}}\,\overline{\mathsf{f}};$ abbreviates $\mathsf{T}_1\mathsf{f}_1; \dots \mathsf{T}_n\mathsf{f}_n;$ and $\overrightarrow{\mathsf{T}}\,\overrightarrow{\mathsf{f}}$ abbreviates $\mathsf{T}_1\mathsf{f}_1, \dots, \mathsf{T}_n\mathsf{f}_n$ (likewise $\overrightarrow{\mathsf{T}}\,\overrightarrow{\mathsf{x}}$) and $\mathsf{this}.\overline{\mathsf{f}} = \overline{\mathsf{f}};$ abbreviates $\mathsf{this}.\mathsf{f}_1 = \mathsf{f}_1; \dots \mathsf{this}.\mathsf{f}_n = \mathsf{f}_n;$. This convention on $^-$ and $^\rightarrow$ is also used in the reduction and typing rules. Sequences of interfaces, fields, parameters and methods are assumed to contain no duplicate names. The keyword super, used only in constructor's body, refers to the superclass constructor.

Types (ranged over by τ, σ) are generated by the grammar:

$$\tau ::= \mathsf{C} \mid \iota \mid \mathsf{C}\&\iota \quad \text{where} \quad \iota ::= \mathsf{I} \mid \iota\&\mathsf{I}$$

assuming that classes and interfaces in the intersection type have different method names. The notation $\mathsf{C}[\&\iota]$ means either the class C or the type $\mathsf{C}\&\iota$.

The syntax of terms, classes and interfaces of TJ& is defined by:

$$\begin{array}{lll}
\mathsf{t} & ::= & \mathsf{v} \mid \mathsf{x} \mid \mathsf{t.f} \mid \mathsf{t.m}(\overrightarrow{\mathsf{t}}) \mid \mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{t}}) \mid (\tau)\,\mathsf{t} \\
\mathsf{v} & ::= & \mathsf{w} \mid \overrightarrow{\mathsf{x}} \to \mathsf{t} \\
\mathsf{w} & ::= & \mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}}) \mid (\overrightarrow{\mathsf{x}} \to \mathsf{t})^{\varphi} \\
\mathsf{CD}^T & ::= & \mathsf{class}\,\mathsf{C}\,\mathsf{extends}\,\mathsf{D}\,\mathsf{implements}\,\overrightarrow{\mathsf{I}}\,\{\overline{\mathsf{T}}\,\overline{\mathsf{f}}; \mathsf{K}^T\,\overline{\mathsf{M}^T}\} \\
\mathsf{ID}^T & ::= & \mathsf{interface}\,\mathsf{I}\,\mathsf{extends}\,\overrightarrow{\mathsf{I}}\,\{\overline{\mathsf{H}^T}; \overline{\mathsf{M}^T}\} \\
\mathsf{K}^T & ::= & \mathsf{C}(\overrightarrow{\mathsf{T}}\,\overrightarrow{\mathsf{f}})\{\mathsf{super}(\overrightarrow{\mathsf{f}}); \mathsf{this}.\overline{\mathsf{f}} = \overline{\mathsf{f}};\} \\
\mathsf{H}^T & ::= & \mathsf{T}\,\mathsf{m}(\overrightarrow{\mathsf{T}}\,\overrightarrow{\mathsf{x}}) \\
\mathsf{M}^T & ::= & \mathsf{H}^T\,\{\mathsf{return}\ \mathsf{t};\}
\end{array}$$

Terms are values, variables, field accesses, method calls, object creations and casts. Values include λ-expressions. We distinguish between values (ranged over by v, u) and *proper values* (ranged over by w). A pure λ-expression is a value, while a λ-expression decorated by its *target type* φ is a proper value. φ denotes a *functional* type, that is an interface or an intersection of interfaces with *exactly* one abstract method. Decorated λ-expressions are produced at run-time only. We use t_λ to range over pure λ-expressions.

CD^T ranges over class declarations; ID^T ranges over interface declarations; K^T ranges over constructor declarations; H^T ranges over method header (abstract method) declarations; M^T ranges over method declarations. Thus, an interface declaration can contain not only abstract methods but also concrete methods with a default implementation. For simplicity, we omit the keyword *default* and the parentheses around parameters of λ-expressions. Except for these simplifications, every TJ& program is an executable Java program.

In writing examples, we omit implements and extends when the list of interfaces is empty.

A *class table* CT^T is a mapping from nominal types to their declarations. `Object` is a special class without fields and methods and it is not included in the class table.

Lookup functions for a given class table are as follows, where we use inheritance and overriding as expected:

- `A-mtypeT`(φ) gives the parameter and return types of the unique abstract method in φ;
- `A-nameT`(φ) gives the name of the unique abstract method in φ;
- `fieldsT`(C) gives the sequence of fields declarations in class C;
- `mtypeT`$(\mathsf{m}; \tau)$ gives the parameter and return types of method m in τ;
- `mbodyT`$(\mathsf{m}; \tau)$ gives the formal parameters and the body of method m in τ.

$$\frac{\texttt{fieldsT}(\mathsf{C}) = \overrightarrow{\mathsf{T}}\ \overrightarrow{\mathsf{f}}}{\mathsf{new}\ \mathsf{C}(\overrightarrow{\mathsf{v}}).\mathsf{f}_j \longrightarrow_T (\mathsf{v}_j)^{?\mathsf{T}_j}}\ [\text{T-ProjNew}] \qquad \frac{\mathsf{C} <: \tau}{(\tau)\ \mathsf{new}\ \mathsf{C}(\overrightarrow{\mathsf{v}}) \longrightarrow_T \mathsf{new}\ \mathsf{C}(\overrightarrow{\mathsf{v}})}\ [\text{T-CastNew}]$$

$$\frac{\texttt{mbodyT}(\mathsf{m};\mathsf{C}) = (\overrightarrow{\mathsf{x}},\mathsf{t}) \quad \texttt{mtypeT}(\mathsf{m};\mathsf{C}) = \overrightarrow{\mathsf{T}} \to \mathsf{T}}{\mathsf{new}\ \mathsf{C}(\overrightarrow{\mathsf{v}}).\mathsf{m}(\overrightarrow{\mathsf{u}}) \longrightarrow_T [\overrightarrow{\mathsf{x}} \mapsto (\overrightarrow{\mathsf{u}})^{?\overrightarrow{\mathsf{T}}}, \mathsf{this} \mapsto \mathsf{new}\ \mathsf{C}(\overrightarrow{\mathsf{v}})](\mathsf{t})^{?\mathsf{T}}}\ [\text{T-InvkNew}]$$

$$\frac{\texttt{A-nameT}(\varphi) = \mathsf{m} \quad \texttt{A-mtypeT}(\varphi) = \overrightarrow{\mathsf{T}} \to \mathsf{T}}{(\overrightarrow{\mathsf{y}} \to \mathsf{t})^{\varphi}.\mathsf{m}(\overrightarrow{\mathsf{v}}) \longrightarrow_T [\overrightarrow{\mathsf{y}} \mapsto (\overrightarrow{\mathsf{v}})^{?\overrightarrow{\mathsf{T}}}](\mathsf{t})^{?\mathsf{T}}}\ [\text{T-Invk}\lambda\text{-A}]$$

$$\frac{\texttt{mbodyT}(\mathsf{m};\varphi) = (\overrightarrow{\mathsf{x}},\mathsf{t}) \quad \texttt{mtypeT}(\mathsf{m};\varphi) = \overrightarrow{\mathsf{T}} \to \mathsf{T}}{(\mathsf{t}_\lambda)^{\varphi}.\mathsf{m}(\overrightarrow{\mathsf{v}}) \longrightarrow_T [\overrightarrow{\mathsf{x}} \mapsto (\overrightarrow{\mathsf{v}})^{?\overrightarrow{\mathsf{T}}}, \mathsf{this} \mapsto (\mathsf{t}_\lambda)^{\varphi}](\mathsf{t})^{?\mathsf{T}}}\ [\text{T-Invk}\lambda\text{-D}]$$

$$(\varphi)\,\mathsf{t}_\lambda \longrightarrow_T (\mathsf{t}_\lambda)^{\varphi} [\text{T-C}\lambda] \qquad \frac{\varphi <: \varphi'}{(\varphi')\,(\mathsf{t}_\lambda)^{\varphi} \longrightarrow_T (\mathsf{t}_\lambda)^{\varphi}}\ [\text{T-CC}\lambda] \qquad \frac{\mathsf{t} \longrightarrow_T \mathsf{t}'}{\mathcal{E}[\mathsf{t}] \longrightarrow_T \mathcal{E}[\mathsf{t}']}\ [\text{T-Ctx}]$$

Figure 1 Reduction Rules of TJ&.

We assume that there are no cycles in the subclass relation between nominal types induced by the class table. The *subtype relation* $<:$ takes into account both the subclass relation induced by the class table, and the set theoretic properties of intersection, which give the following relations:

$$\frac{\tau <: \mathsf{T}_i \quad \text{for all } 1 \leq i \leq n}{\tau <: \mathsf{T}_1 \& \ldots \& \mathsf{T}_n}\ [<:\&\text{R}] \qquad \frac{\mathsf{T}_i <: \tau \quad \text{for some } 1 \leq i \leq n}{\mathsf{T}_1 \& \ldots \& \mathsf{T}_n <: \tau}\ [<:\&\text{L}]$$

In what follows, to lighten the notation of reduction and typing rules, we assume a fixed class table CT^{T}.

The reduction rules are given in Figure 1, where evaluation contexts $\mathcal{E}$ are defined by:

$$\mathcal{E} ::= [\,] \mid \mathcal{E}.\mathsf{f} \mid \mathcal{E}.\mathsf{m}(\overrightarrow{\mathsf{t}}) \mid \mathsf{w}.\mathsf{m}(\overrightarrow{\mathsf{v}}\mathcal{E}\,\overrightarrow{\mathsf{t}}) \mid \mathsf{new}\ \mathsf{C}(\overrightarrow{\mathsf{v}}\mathcal{E}\,\overrightarrow{\mathsf{t}}) \mid (\tau)\,\mathcal{E}$$

Following [2], reduction rules guarantee that pure λ-expressions are decorated by their target types in the evaluated terms. This is realised by means of the mapping $(\mathsf{t})^{?\tau}$ defined as follows:

$$(\mathsf{t})^{?\tau} = \begin{cases} (\mathsf{t})^{\tau} & \text{if } \mathsf{t} = \mathsf{t}_\lambda, \\ \mathsf{t} & \text{otherwise} \end{cases}$$

Namely, this mapping decorates pure λ-expressions with τ, whereas leaves all the other terms unchanged. It is used in propagating the types expected for λ-expressions in object constructors, method calls and type casts. The typing rules assure that if t is a pure λ-expression, then its target type τ is a functional type. So we only get decorated terms of the shape $(\mathsf{t}_\lambda)^{\varphi}$. The reduction of a method call on a λ-expression distinguishes the case of abstract methods from that of default methods. As usual, $\longrightarrow_T^*$ is the reflexive and transitive closure of $\longrightarrow_T$.

The typing rules for terms are given in Figure 2. These rules are standard, but for [T-λ], [T-DC], [T-UC] and the judgment $\vdash^*$. Rule [T-λ] checks that a λ-expression is typed as required by the only abstract method in φ, and the subject of the conclusion is the decorated

$$\frac{\mathsf{x} : \mathsf{T} \in \Delta}{\Delta \vdash_{\mathcal{T}} \mathsf{x} : \mathsf{T}}\,[\text{T-VAR}] \qquad \frac{\Delta \vdash_{\mathcal{T}} \mathsf{t} : \mathsf{C}[\&\iota] \quad \mathsf{T}\,\mathsf{f} \in \mathtt{fieldsT}(\mathsf{C})}{\Delta \vdash_{\mathcal{T}} \mathsf{t.f} : \mathsf{T}}\,[\text{T-FIELD}]$$

$$\frac{\Delta \vdash_{\mathcal{T}} \mathsf{t} : \tau \quad \mathtt{mtypeT}(\mathsf{m};\tau) = \overrightarrow{\mathsf{T}} \to \mathsf{T} \quad \Delta \vdash^*_{\mathcal{T}} \bar{\mathsf{t}} : \bar{\mathsf{T}}}{\Delta \vdash_{\mathcal{T}} \mathsf{t.m}(\overrightarrow{\mathsf{t}}) : \mathsf{T}}\,[\text{T-INVK}]$$

$$\frac{\mathtt{fieldsT}(\mathsf{C}) = \overrightarrow{\mathsf{T}}\,\overrightarrow{\mathsf{f}} \quad \Delta \vdash^*_{\mathcal{T}} \bar{\mathsf{t}} : \bar{\mathsf{T}}}{\Delta \vdash_{\mathcal{T}} \mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{t}}) : \mathsf{C}}\,[\text{T-NEW}]$$

$$\frac{\mathtt{A\text{-}mtypeT}(\varphi) = \overrightarrow{\mathsf{T}} \to \mathsf{T} \quad \Delta, \overrightarrow{\mathsf{y}} : \overrightarrow{\mathsf{T}} \vdash^*_{\mathcal{T}} \mathsf{t} : \mathsf{T}}{\Delta \vdash_{\mathcal{T}} (\overrightarrow{\mathsf{y}} \to \mathsf{t})^{\varphi} : \varphi}\,[\text{T-}\lambda]$$

$$\frac{\Delta \vdash_{\mathcal{T}} (\mathsf{t}_\lambda)^{\varphi} : \varphi}{\Delta \vdash^*_{\mathcal{T}} \mathsf{t}_\lambda : \varphi}\,[\text{T-*-}\lambda] \qquad \frac{\Delta \vdash_{\mathcal{T}} \mathsf{t} : \sigma \quad \mathsf{t} \neq \mathsf{t}_\lambda \quad \sigma <: \tau}{\Delta \vdash^*_{\mathcal{T}} \mathsf{t} : \tau}\,[\text{T-*-N}]$$

$$\frac{\Delta \vdash^*_{\mathcal{T}} \mathsf{t} : \tau}{\Delta \vdash_{\mathcal{T}} (\tau)\,\mathsf{t} : \tau}\,[\text{T-UC}] \qquad \frac{\Delta \vdash_{\mathcal{T}} \mathsf{t} : \mathsf{T}}{\Delta \vdash_{\mathcal{T}} (\mathsf{T}[\&\iota])\,\mathsf{t} : \mathsf{T}[\&\iota]}\,[\text{T-DC}]$$

Figure 2 Term Typing Rules of TJ&.

λ-expression. Rule [T-DC] is a restricted form of the standard typing rule for downcast. Indeed, it represents exactly the form of the downcasts that we will introduce in a term of SJ&+ when translating it into the target language TJ&. Then an immediate reason for including this rule is that translated terms must be typed. Conversely, by omitting a general downcast rule, we can focus only on the downcasts that are the crucial modification of a term during translation. Concerning the judgment $\vdash^*$, it has a different meaning for decorated λ-expressions and other terms. Rule [T-*-λ] states that we derive a type for a pure λ-expression only by checking that the decorated λ-expression is typed. Instead, for a term which is not a λ-expression, rule [T-*-N] makes subsumption explicit, going from $\vdash$ to $\vdash^*$. The utility of the judgment $\vdash^*$ consists in simplifying the formulation of rules [T-INVK] and [T-NEW]. Rule [T-UC] is shorter than usual, by taking advantage from the judgment $\vdash^*$.

$$\frac{\mathtt{mtypeT}(\mathsf{m};\mathsf{T}) = \overrightarrow{\mathsf{T}} \to \mathsf{T}' \quad \overrightarrow{\mathsf{x}} : \overrightarrow{\mathsf{T}}, \mathsf{this} : \mathsf{T} \vdash^*_{\mathcal{T}} \mathsf{t} : \mathsf{T}'}{\mathsf{T}'\mathsf{m}(\overrightarrow{\mathsf{T}}\,\overrightarrow{\mathsf{x}})\{\mathsf{return}\ \mathsf{t};\}\ \text{T-}\mathit{OK}\ \text{in}\ \mathsf{T}}\,[\mathsf{M}\ \text{T-}\mathit{OK}\ \text{in}\ \mathsf{T}]$$

$$\frac{\begin{array}{c}\mathsf{K}^{\mathcal{T}} = \mathsf{C}(\overrightarrow{\mathsf{U}}\,\overrightarrow{\mathsf{g}}, \overrightarrow{\mathsf{T}}\,\overrightarrow{\mathsf{f}})\{\mathsf{super}(\overrightarrow{\mathsf{g}}); \mathsf{this}.\bar{\mathsf{f}} = \bar{\mathsf{f}};\} \quad \mathtt{fieldsT}(\mathsf{D}) = \overrightarrow{\mathsf{U}}\,\overrightarrow{\mathsf{g}} \quad \overline{\mathsf{M}^{\mathcal{T}}}\ \text{T-}\mathit{OK}\ \text{in}\ \mathsf{C} \\ \mathtt{mtypeT}(\mathsf{m};\mathsf{C})\ \text{defined implies}\ \mathtt{mbodyT}(\mathsf{m};\mathsf{C})\ \text{defined}\end{array}}{\mathsf{class}\ \mathsf{C}\ \mathsf{extends}\ \mathsf{D}\ \mathsf{implements}\ \overrightarrow{\mathsf{I}}\ \{\overline{\mathsf{T}}\,\bar{\mathsf{f}}; \mathsf{K}^{\mathcal{T}}\,\overline{\mathsf{M}^{\mathcal{T}}}\}\ \mathit{OK}}\,[\mathsf{C}\ \text{T-}\mathit{OK}]$$

$$\frac{\overline{\mathsf{M}^{\mathcal{T}}}\ \text{T-}\mathit{OK}\ \text{in}\ \mathsf{I}}{\mathsf{interface}\ \mathsf{I}\ \mathsf{extends}\ \overrightarrow{\mathsf{I}}\ \{\overline{\mathsf{H}^{\mathcal{T}}}; \overline{\mathsf{M}^{\mathcal{T}}}\}\ \mathit{OK}}\,[\mathsf{I}\ \text{T-}\mathit{OK}]$$

Figure 3 Method, Class and Interface Declaration Typing Rules of TJ&.

Moreover, a feature of [T-UC] is the possibility of obtaining a judgment $\vdash$ from a judgment $\vdash^*$. Notice that, if a closed term is typed in $\vdash$, then the type derivation is unique. This is clearly false for $\vdash^*$.

If the body of a typed method contains a λ-expression, then the λ-expression may contain the formal parameters of the enclosing method, which are effectively final variables, as prescribed in [14] (page 607). Moreover, no other final variable from the enclosing environment can occur in this λ-expression, since we are in a purely functional model without assignments.

Typing statements for methods, classes and interfaces are checked by the rules in Figure 3: they say that a method is well formed in a class, a class declaration is well formed and an interface declaration is well formed, respectively. In rule [M T-*OK* in T] we omit the standard condition on soundness for overriding, see [20] (Figure 19-2). A class table is well formed if all class and interface declarations are well formed.

A *program* is a pair $(CT^{\mathcal{T}}, \mathsf{t})$ of a class table $CT^{\mathcal{T}}$ and a closed term t. We say that the program is *typed* if $CT^{\mathcal{T}}$ is well formed and t is typed by using $CT^{\mathcal{T}}$.

Finally, this calculus enjoys Subject Reduction, thanks to the above restriction of the downcast rule. Moreover, a program without downcasts has Progress.

▶ **Theorem 1** (Subject Reduction and Progress)**.**

1. *If* $\Delta \vdash_{\mathcal{T}} \mathsf{t} : \tau$ *and* $\mathsf{t} \longrightarrow_{\mathcal{T}} \mathsf{t}'$*, then* $\Delta \vdash_{\mathcal{T}} \mathsf{t} : \sigma$ *for some* $\sigma <: \tau$*.*
2. *If* $(CT^{\mathcal{T}}, \mathsf{t})$ *is typed without using rule* [T-DC]*, then* t *either is a proper value or reduces.*

Both properties are proved in [2].

4 Java with Deconfined Intersection Types (SJ&+)

In this Sections we extend TJ& with a new crucial feature: intersection types are first class types, that is they can be used everywhere a type is expected. Thus intersection types are allowed to appear as types of fields and as return and parameter types of methods, rather than being confined within a type cast as in TJ&. This extension is formalised by the *source calculus* SJ&+. Terms in SJ&+ are defined as terms in TJ&. Instead, the extended use of intersection types requires the following new definitions:

$$\frac{\texttt{fieldsS}(\mathsf{C}) = \overrightarrow{\tau}\ \overrightarrow{\mathsf{f}}}{\mathsf{new}\ \mathsf{C}(\overrightarrow{\mathsf{v}}).\mathsf{f}_j \longrightarrow_{\mathcal{S}} (\mathsf{v}_j)^{?\tau_j}}\ [\text{S-ProjNew}]$$

$$\frac{\texttt{mbodyS}(\mathsf{m};\mathsf{C}) = (\overrightarrow{\mathsf{x}}, \mathsf{t}) \quad \texttt{mtypeS}(\mathsf{m};\mathsf{C}) = \overrightarrow{\tau} \to \tau}{\mathsf{new}\ \mathsf{C}(\overrightarrow{\mathsf{v}}).\mathsf{m}(\overrightarrow{\mathsf{u}}) \longrightarrow_{\mathcal{S}} [\overrightarrow{\mathsf{x}} \mapsto (\overrightarrow{\mathsf{u}})^{?\overrightarrow{\tau}}, \mathsf{this} \mapsto \mathsf{new}\ \mathsf{C}(\overrightarrow{\mathsf{v}})](\mathsf{t})^{?\tau}}\ [\text{S-InvkNew}]$$

$$\frac{\texttt{A-nameS}(\varphi) = \mathsf{m} \quad \texttt{A-mtypeS}(\varphi) = \overrightarrow{\tau} \to \tau}{(\overrightarrow{\mathsf{y}} \to \mathsf{t})^{\varphi}.\mathsf{m}(\overrightarrow{\mathsf{v}}) \longrightarrow_{\mathcal{S}} [\overrightarrow{\mathsf{y}} \mapsto (\overrightarrow{\mathsf{v}})^{?\overrightarrow{\tau}}](\mathsf{t})^{?\tau}}\ [\text{S-Invk}\lambda\text{-A}]$$

$$\frac{\texttt{mbodyS}(\mathsf{m};\varphi) = (\overrightarrow{\mathsf{x}}, \mathsf{t}) \quad \texttt{mtypeS}(\mathsf{m};\varphi) = \overrightarrow{\tau} \to \tau}{(\mathsf{t}_\lambda)^{\varphi}.\mathsf{m}(\overrightarrow{\mathsf{v}}) \longrightarrow_{\mathcal{S}} [\overrightarrow{\mathsf{x}} \mapsto (\overrightarrow{\mathsf{v}})^{?\overrightarrow{\tau}}, \mathsf{this} \mapsto (\mathsf{t}_\lambda)^{\varphi}](\mathsf{t})^{?\tau}}\ [\text{S-Invk}\lambda\text{-D}]$$

Figure 4 Reduction Rules of SJ&+: rules [S-CastNew], [S-Cλ], [S-CCλ] and [S-Ctx] are omitted.

$$\frac{\mathsf{x} : \tau \in \Gamma}{\Gamma \vdash_S \mathsf{x} : \tau}\ [\text{S-VAR}] \qquad \frac{\Gamma \vdash_S \mathsf{t} : \mathsf{C}[\&\iota] \quad \tau\, \mathsf{f} \in \texttt{fieldsS}(\mathsf{C})}{\Gamma \vdash_S \mathsf{t.f} : \tau}\ [\text{S-FIELD}]$$

$$\frac{\Gamma \vdash_S \mathsf{t} : \tau \quad \texttt{mtypeS}(\mathsf{m}; \tau) = \overrightarrow{\sigma} \to \sigma \quad \Gamma \vdash^*_S \bar{\mathsf{t}} : \bar{\sigma}}{\Gamma \vdash_S \mathsf{t.m}(\overrightarrow{\mathsf{t}}) : \sigma}\ [\text{S-INVK}]$$

$$\frac{\texttt{fieldsS}(\mathsf{C}) = \overrightarrow{\tau}\,\overrightarrow{\mathsf{f}} \quad \Gamma \vdash^*_S \bar{\mathsf{t}} : \bar{\tau}}{\Gamma \vdash_S \mathsf{new}\, \mathsf{C}(\overrightarrow{\mathsf{t}}) : \mathsf{C}}\ [\text{S-NEW}]$$

$$\frac{\texttt{A-mtypeS}(\varphi) = \overrightarrow{\tau} \to \tau \text{ implies } \Gamma, \overrightarrow{\mathsf{y}} : \overrightarrow{\tau} \vdash^*_S \mathsf{t} : \tau}{\Gamma \vdash_S (\overrightarrow{\mathsf{y}} \to \mathsf{t})^{\varphi} : \varphi}\ [\text{S-}\lambda]$$

Figure 5 Term Typing Rules of SJ&+: rules [S-*-λ], [S-*-N] and [S-UC] are omitted.

$$\frac{\texttt{mtypeS}(\mathsf{m}; \mathsf{T}) = \overrightarrow{\tau} \to \tau \quad \overrightarrow{\mathsf{x}} : \overrightarrow{\tau}, \mathsf{this} : \mathsf{T} \vdash^*_S \mathsf{t} : \tau}{\tau\, \mathsf{m}(\overrightarrow{\tau}\,\overrightarrow{\mathsf{x}})\{\mathsf{return}\ \ \mathsf{t};\}\ \text{S-}\mathit{OK} \text{ in } \mathsf{T}}\ [\mathsf{M}\ \text{S-}\mathit{OK} \text{ in } \mathsf{T}]$$

$$\frac{\begin{array}{c}\mathsf{K}^S = \mathsf{C}(\overrightarrow{\sigma}\,\overrightarrow{\mathsf{g}}, \overrightarrow{\tau}\,\overrightarrow{\mathsf{f}})\{\mathsf{super}(\overrightarrow{\mathsf{g}}); \mathsf{this}.\bar{\mathsf{f}} = \bar{\mathsf{f}};\} \quad \texttt{fieldsS}(\mathsf{D}) = \overrightarrow{\sigma}\,\overrightarrow{\mathsf{g}} \quad \overline{\mathsf{M}^S}\ \text{S-}\mathit{OK} \text{ in } \mathsf{C} \\ \texttt{mtypeS}(\mathsf{m}; \mathsf{C}) \text{ defined implies } \texttt{mbodyS}(\mathsf{m}; \mathsf{C}) \text{ defined}\end{array}}{\mathsf{class}\ \mathsf{C}\ \mathsf{extends}\ \mathsf{D}\ \mathsf{implements}\ \overrightarrow{\mathsf{I}}\ \{\bar{\tau}\, \bar{\mathsf{f}}; \mathsf{K}^S\, \overline{\mathsf{M}^S}\}\ \mathit{OK}}\ [\mathsf{C}\ \text{S-}\mathit{OK}]$$

Figure 6 Method and Class Declaration Typing Rules of SJ&+: rule [I S-*OK*] is omitted.

$$\begin{array}{lll}
\mathsf{CD}^S & ::= & \mathsf{class}\ \mathsf{C}\ \mathsf{extends}\ \mathsf{D}\ \mathsf{implements}\ \overrightarrow{\mathsf{I}}\ \{\bar{\tau}\, \bar{\mathsf{f}}; \mathsf{K}^S\, \overline{\mathsf{M}^S}\} \\
\mathsf{ID}^S & ::= & \mathsf{interface}\ \mathsf{I}\ \mathsf{extends}\ \overrightarrow{\mathsf{I}}\ \{\overline{\mathsf{H}^S}; \overline{\mathsf{M}^S}\} \\
\mathsf{K}^S & ::= & \mathsf{C}(\overrightarrow{\tau}\,\overrightarrow{\mathsf{f}})\{\mathsf{super}(\overrightarrow{\mathsf{f}}); \mathsf{this}.\bar{\mathsf{f}} = \bar{\mathsf{f}};\} \\
\mathsf{H}^S & ::= & \tau\, \mathsf{m}(\overrightarrow{\tau}\,\overrightarrow{\mathsf{x}}) \\
\mathsf{M}^S & ::= & \mathsf{H}^S\, \{\mathsf{return}\ \mathsf{t};\}
\end{array}$$

Lookup functions for a given class table in SJ&+ are defined as in TJ&. To distinguish the two calculi, the suffix or prefix `S` replaces `T` in the arrow denoting reduction, in the derivation symbol, and in the labels of the rules. Figures 4, 5 and 6 show the rules which are different from the corresponding rules in Figures 1, 2 and 3, respectively. The other rules are unchanged, namely:

- the reduction rules [S-CastNew], [S-Cλ], [S-CCλ] and [S-Ctx] must be added to Figure 4;
- the typing rules [S-*-λ], [S-*-N] and [S-UC] must be added to Figure 5;
- the rule [I S-*OK*] must be added to Figure 6.

As usual, $\longrightarrow^*_S$ is the reflexive and transitive closure of $\longrightarrow_S$.

We observe that SJ&+ does not have a downcast rule. The reason of this choice is the same which justifies the restricted form of downcast in the typing rule of TJ&. When translating a term of SJ&+ without downcasts, we have the property that the downcasts in the translated term are those and only those introduced by the translation. In this case, the proofs of properties concerning the translation become more concise and compact.

As in TJ&, a *program* of SJ&+ is a pair (CT^S, t) of a class table CT^S and a closed term t. We say that the program is *typed* if CT^S is well formed and t is typed by using CT^S.

Finally, both Subject Reduction and Progress hold. Since SJ&+ does not have any downcast typing rule, Progress requires no conditions.

▶ **Theorem 2** (Subject Reduction and Progress)**.**

1. *If* $\Gamma \vdash_S \mathsf{t} : \tau$ *and* $\mathsf{t} \longrightarrow_S \mathsf{t}'$*, then* $\Gamma \vdash_S \mathsf{t} : \sigma$ *for some* $\sigma <: \tau$*.*
2. *If* (CT^S, t) *is typed, then* t *either is a proper value or reduces.*

Both properties are proved in [10] for a conservative extension of SJ&+.

5 Translation

In this section we want to translate a typed program (CT^S, t) in the source calculus SJ&+ into a program (CT^T, t') in the target calculus TJ&.

To lighten definitions, we adopt the following convention for writing components of intersections in a given order: if ι is a functional type such that $\iota = \mathsf{I}\&\iota'$, then I is a functional interface.

We define the erasure on intersection types as the *erasure* mapping:

$$|\mathsf{T}[\&\iota]| = \mathsf{T}$$

This mapping replaces an intersection type with its first component, a class or an interface. Observe that the above convention on the order in intersection types ensures that the mapping of a functional type is a functional type too.

Now we have to define the translation of a well formed class table CT^S into a corresponding CT^T. We give a preliminary informal discussion about the main underlying ideas of the proposed translation technique.

The initial step must be the erasure of type intersections which are (i) parameter and return types in signatures of methods (ii) types of fields in class declarations. Then the crucial matter becomes the translation of method bodies, which have to recover the type information lost by erasure, bearing in mind two different issues. First, we must ensure that typing is preserved under translation. Second, the behaviour of the translated term must mimic the behaviour of the original term.

To resolve the first issue, we enclose any method call in a type cast to the original return type, since this return type has been erased in the translated signature. Analogously, we insert type casts on field selections and variable occurrences. To address the second issue, we take into account that λ-expressions need to be decorated by their target types to define their behaviour. However, a pure λ-expression can only get its target type from the enclosing context, namely:

- the target type of a λ-expression that occurs as an actual parameter of a constructor call is the type of the field in the class declaration;
- the target type of a λ-expression that occurs as an actual parameter of a method call is the type of the parameter in the method declaration;
- the target type of a λ-expression that occurs as a return term of a method is the result type in the method declaration;
- the target type of a λ-expression that occurs as the argument of a type cast is the type of the cast.

$$\left(\!\left[\frac{\mathsf{x} : \tau \in \Gamma}{\Gamma \vdash_S \mathsf{x} : \tau}\,[\text{S-VAR}]\right]\!\right) = (\tau)\,\mathsf{x}$$

$$\left(\!\left[\frac{\mathcal{D} :: \Gamma \vdash_S \mathsf{t} : \mathsf{C}[\&\iota] \quad \tau\,\mathsf{f} \in \mathtt{fieldsS}(\mathsf{C})}{\Gamma \vdash_S \mathsf{t.f} : \tau}\,[\text{S-FIELD}]\right]\!\right) = (\tau)\,(\llparenthesis\mathcal{D}\rrparenthesis.\mathsf{f})$$

$$\left(\!\left[\frac{\mathcal{D} :: \Gamma \vdash_S \mathsf{t} : \tau \quad \mathtt{mtypeS}(\mathsf{m};\tau) = \overrightarrow{\sigma} \to \sigma \quad \overline{\mathcal{D}} :: \Gamma \vdash^*_S \bar{\mathsf{t}} : \overline{\sigma}}{\Gamma \vdash_S \mathsf{t.m}(\overrightarrow{\mathsf{t}}) : \sigma}\,[\text{S-INVK}]\right]\!\right) = (\sigma)\,(\llparenthesis\mathcal{D}\rrparenthesis.\mathsf{m}(\overrightarrow{\llparenthesis\mathcal{D}\rrparenthesis}))$$

$$\left(\!\left[\frac{\mathtt{fieldsS}(\mathsf{C}) = \overrightarrow{\tau}\,\overrightarrow{\mathsf{f}} \quad \overline{\mathcal{D}} :: \Gamma \vdash^*_S \bar{\mathsf{t}} : \overline{\tau}}{\Gamma \vdash_S \mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{t}}) : \mathsf{C}}\,[\text{S-NEW}]\right]\!\right) = \mathsf{new}\,\mathsf{C}(\overrightarrow{\llparenthesis\mathcal{D}\rrparenthesis})$$

$$\left(\!\left[\frac{\mathtt{A\text{-}mtypeS}(\varphi) = \overrightarrow{\tau} \to \tau \quad \mathcal{D} :: \Gamma, \overrightarrow{\mathsf{y}} : \overrightarrow{\tau} \vdash^*_S \mathsf{t} : \tau}{\Gamma \vdash_S (\overrightarrow{\mathsf{y}} \to \mathsf{t})^{\varphi} : \varphi}\,[\text{S-}\lambda]\right]\!\right) = (\overrightarrow{\mathsf{y}} \to \llparenthesis\mathcal{D}\rrparenthesis)^{\varphi}$$

$$\left(\!\left[\frac{\mathcal{D} :: \Gamma \vdash_S (\mathsf{t}_\lambda)^{\varphi} : \varphi}{\Gamma \vdash^*_S \mathsf{t}_\lambda : \varphi}\,[\text{S-*-}\lambda]\right]\!\right) = (\varphi)\mathsf{t}'_\lambda \qquad \text{if } \llparenthesis\mathcal{D}\rrparenthesis = (\mathsf{t}'_\lambda)^{\varphi}$$

$$\left(\!\left[\frac{\mathcal{D} :: \Gamma \vdash_S \mathsf{t} : \sigma \quad \mathsf{t} \neq \mathsf{t}_\lambda \quad \sigma <: \tau}{\Gamma \vdash^*_S \mathsf{t} : \tau}\,[\text{S-*-N}]\right]\!\right) = \llparenthesis\mathcal{D}\rrparenthesis$$

$$\left(\!\left[\frac{\mathcal{D} :: \Gamma \vdash^*_S \mathsf{t} : \tau}{\Gamma \vdash_S (\tau)\mathsf{t} : \tau}\,[\text{S-UC}]\right]\!\right) = (\tau)\,\llparenthesis\mathcal{D}\rrparenthesis$$

Figure 7 Term Translation.

In the first three cases above, the erasure of method signatures and field types requires to recover the original target types. Thus the translation algorithm can add type casts to pure λ-expressions to preserve their target types. However, a crucial question arises: where do we find the target type of each occurrence of a λ-expression in the original term? The target type is included not in the syntactic structure of the term but in its typing. This motivates our formulation of translation from (typed) terms to terms, that is defined as a mapping from the type derivation of a term in SJ&+ into a term in TJ&.

Going into formalities, the translation of the program is based on three mappings.

The first mapping is the erasure mapping already defined.

The second mapping, dubbed $\llparenthesis\ \rrparenthesis$, applied to a type derivation for a term in SJ&+ gives a term of TJ&. The translation is defined by induction on derivations considering the last rule applied, as shown in Figure 7. The type derivation is used to cast λ-expressions to their target types in the source term. We convene that $\mathcal{D}$ ranges over derivations, i.e. $\mathcal{D} :: \Gamma \vdash_S \mathsf{t} : \tau$ means a derivation with conclusion $\Gamma \vdash_S \mathsf{t} : \tau$, and similarly for $\vdash^*_S$. We denote by $\llparenthesis\mathcal{D}\rrparenthesis$ the result of the translation of $\mathcal{D} :: \Gamma \vdash_S \mathsf{t} : \tau$ or $\mathcal{D} :: \Gamma \vdash^*_S \mathsf{t} : \tau$. With an abuse of language, when t is closed and typed we define $\llparenthesis\mathsf{t}\rrparenthesis = \llparenthesis\mathcal{D}\rrparenthesis$, where $\mathcal{D}$ is the unique derivation in $\vdash_S$ such that t is the subject of the conclusion and the typing context is empty. Notice that the condition for the translation of rule [S-*-λ] is always satisfied, since the judgment $\Gamma \vdash_S (\mathsf{t}_\lambda)^{\varphi} : \varphi$ can only be the conclusion of rule [S-λ]. We point out that decorated λ-expressions are produced only by translating decorated λ-expressions in the source term. Therefore a program of SJ&+ is compiled to a program in TJ& which is Java code.

The third mapping, dubbed $[\![\ ]\!]$, has as arguments and results declarations in class tables of the two calculi, respectively. We start by translating method headers and constructors, which only requires mapping intersections to nominal types.

▶ **Definition 3** (Translation of Headers and Constructors). *We define*

$$[\![\tau\,\mathsf{m}(\overrightarrow{\tau}\,\overrightarrow{\mathsf{x}})]\!] = |\tau|\mathsf{m}(\overrightarrow{|\tau|}\,\overrightarrow{\mathsf{x}})$$
$$[\![\mathsf{C}(\overrightarrow{\sigma}\,\overrightarrow{\mathsf{g}}, \overrightarrow{\tau}\,\overrightarrow{\mathsf{f}})\{\mathsf{super}(\overrightarrow{\mathsf{g}}); \mathsf{this}.\bar{\mathsf{f}} = \bar{\mathsf{f}};\}]\!] = \mathsf{C}(\overrightarrow{|\sigma|}\,\overrightarrow{\mathsf{g}}, \overrightarrow{|\tau|}\,\overrightarrow{\mathsf{f}})\{\mathsf{super}(\overrightarrow{\mathsf{g}}); \mathsf{this}.\bar{\mathsf{f}} = \bar{\mathsf{f}};\}$$

The translation of methods requires the translation of bodies, which being terms need type derivations depending on the class or the interface in which they are defined. For this reason the translation of methods is parametrised on nominal types. We use $[\![\ ]\!]^{\mathsf{T}}$ to denote the dependency on type T.

▶ **Definition 4** (Translation of Methods). *The translation of a method in a class or interface* T *is defined by:*

$$[\![\tau\,\mathsf{m}(\overrightarrow{\tau}\,\overrightarrow{\mathsf{x}})\{\mathsf{return}\ \ \mathsf{t};\}]\!]^{\mathsf{T}} = |\tau|\mathsf{m}(\overrightarrow{|\tau|}\,\overrightarrow{\mathsf{x}})\{\mathsf{return}\ \ (\![\mathcal{D}]\!);\}$$

where $\mathcal{D} :: \mathsf{x} : \overrightarrow{\tau}, \mathsf{this} : \mathsf{T} \vdash^*_S \mathsf{t} : \tau$.

We are now able to define the application of $[\![\ ]\!]$ to class and interface declarations.

▶ **Definition 5** (Translation of Class/Interface Declarations). *We define*

$$[\![\mathsf{class\,C\,extends\,D\,implements}\ \overrightarrow{\mathsf{I}}\ \{\overline{\tau}\,\bar{\mathsf{f}}; \mathsf{K}^S\,\overline{\mathsf{M}^S}\}]\!] = \mathsf{class\,C\,extends\,D\,implements}\ \overrightarrow{\mathsf{I}}\ \{\overline{|\tau|}\,\bar{\mathsf{f}}; [\![\mathsf{K}^S]\!]\ \overline{[\![\mathsf{M}^S]\!]^{\mathsf{C}}}\}$$
$$[\![\mathsf{interface\,I\,extends}\ \overrightarrow{\mathsf{I}}\ \{\overline{\mathsf{H}^S}; \overline{\mathsf{M}^S}\}]\!] = \mathsf{interface\,I\,extends}\ \overrightarrow{\mathsf{I}}\ \{\overline{[\![\mathsf{H}^S]\!]}; \overline{[\![\mathsf{M}^S]\!]^{\mathsf{I}}}\}$$

▶ **Definition 6** (Translation of Typed Programs). *Let* (CT^S, t) *be a typed program. Its translation is the program* $(CT^T, (\![\mathsf{t}]\!))$ *such that:*

$$\textit{if } CT^S(\mathsf{C}) = \mathsf{CD}^S,\ \textit{then } CT^T(\mathsf{C}) = [\![\mathsf{CD}^S]\!]$$
$$\textit{if } CT^S(\mathsf{I}) = \mathsf{ID}^S,\ \textit{then } CT^T(\mathsf{I}) = [\![\mathsf{ID}^S]\!]$$

Notice that $(\![\mathsf{t}]\!)$ is well defined since t is closed and typed.

▶ **Example 7.** Consider the program (CT^S, t) where $\mathsf{t} = \mathsf{new\,C(\ ).m(x' \rightarrow x')}$ and CT^S is the class table on the left-side of Figure 8. Its translation is (CT^T, t'), where

$$\mathsf{t}' = (\mathsf{I}_1\&\mathsf{I}_2)\,(\mathsf{new\,C(\ ).m}((\mathsf{I}_1\&\mathsf{I}_2)\,(\mathsf{x}' \rightarrow (\mathsf{I}_1\&\mathsf{I}_2)\,\mathsf{x}')))$$

and CT^T is the class table on the right-side of Figure 8.

```
interface I1 { I1&I2 m(I1&I2 x); }
interface I2 { I1&I2 n(I1&I2 y){return y; } }
class C extends Object implements I1, I2 {
  C( ) { super( ); }
  I1&I2 m(I1&I2 x){
    return x.n(z → z);
  }
}
```

SJ&+ Class Table

```
interface I1 { I1 m(I1 x); }
interface I2 { I1 n(I1 y){return (I1&I2)y; } }
class C extends Object implements I1, I2 {
  C( ) { super( ); }
  I1 m(I1 x){
    return (I1&I2)((I1&I2)x.n((I1&I2)(z → (I1&I2)z)));
  }
}
```

TJ& Class Table

Figure 8 Class Tables of Example 7.

It is interesting to compare the reduction of t:

$$\begin{array}{lll} \mathsf{t} & \longrightarrow_S & (\mathsf{x'} \to \mathsf{x'})^{\mathsf{I}_1\&\mathsf{I}_2}.\mathsf{n}(\mathsf{z} \to \mathsf{z}) \\ & \longrightarrow_S & (\mathsf{z} \to \mathsf{z})^{\mathsf{I}_1\&\mathsf{I}_2} \end{array}$$

with the reduction of t′:

$$\begin{array}{lll} \mathsf{t}' & \longrightarrow_T & (\mathsf{I}_1\&\mathsf{I}_2)\,(\mathsf{new\,C(\,).m}((\mathsf{x'} \to (\mathsf{I}_1\&\mathsf{I}_2)\,\mathsf{x'})^{\mathsf{I}_1\&\mathsf{I}_2})) \\ & \longrightarrow_T & (\mathsf{I}_1\&\mathsf{I}_2)\,((\mathsf{I}_1\&\mathsf{I}_2)((\mathsf{I}_1\&\mathsf{I}_2)((\mathsf{x'} \to (\mathsf{I}_1\&\mathsf{I}_2)\,\mathsf{x'})^{\mathsf{I}_1\&\mathsf{I}_2}).\mathsf{n}((\mathsf{I}_1\&\mathsf{I}_2)(\mathsf{z} \to (\mathsf{I}_1\&\mathsf{I}_2)\mathsf{z})))) \\ & \longrightarrow_T & (\mathsf{I}_1\&\mathsf{I}_2)\,((\mathsf{I}_1\&\mathsf{I}_2)((\mathsf{I}_1\&\mathsf{I}_2)((\mathsf{x'} \to (\mathsf{I}_1\&\mathsf{I}_2)\,\mathsf{x'})^{\mathsf{I}_1\&\mathsf{I}_2}).\mathsf{n}((\mathsf{z} \to (\mathsf{I}_1\&\mathsf{I}_2)\mathsf{z})^{\mathsf{I}_1\&\mathsf{I}_2}))) \\ & \longrightarrow_T & (\mathsf{I}_1\&\mathsf{I}_2)\,((\mathsf{I}_1\&\mathsf{I}_2)((\mathsf{I}_1\&\mathsf{I}_2)((\mathsf{I}_1\&\mathsf{I}_2)\,(\mathsf{z} \to (\mathsf{I}_1\&\mathsf{I}_2)\mathsf{z})^{\mathsf{I}_1\&\mathsf{I}_2}))) \\ & \longrightarrow_T^* & (\mathsf{z} \to (\mathsf{I}_1\&\mathsf{I}_2)\mathsf{z})^{\mathsf{I}_1\&\mathsf{I}_2} \end{array}$$

where in the last steps we only apply rules [T-CCλ] and [T-Ctx]. The difference between the obtained values $(\mathsf{z} \to \mathsf{z})^{\mathsf{I}_1\&\mathsf{I}_2}$ and $(\mathsf{z} \to (\mathsf{I}_1\&\mathsf{I}_2)\mathsf{z})^{\mathsf{I}_1\&\mathsf{I}_2}$ is the cast of z to $\mathsf{I}_1\&\mathsf{I}_2$. This is due to the difference between the bodies of method m in the class tables CT^S and CT^T. Notice that $(\mathsf{z} \to (\mathsf{I}_1\&\mathsf{I}_2)\mathsf{z})^{\mathsf{I}_1\&\mathsf{I}_2}$ is the translation of $(\mathsf{z} \to \mathsf{z})^{\mathsf{I}_1\&\mathsf{I}_2}$.

6 Translation Correctness

In this section we show that the translation preserves the static and dynamic semantics of programs. To this aim we prove that typed SJ&+ programs are translated into typed TJ& programs (Theorem 11) and that the original and the translated programs either produce values related by the translation or both have infinite computations (Theorem 16 and Corollary 17).

The proof of these results relies on two main features of the translation. The first is that any subterm is cast to the same type τ that was used for typing the subterm in the source code. The second insight is about pure λ-expressions, that become arguments of type casting in translated terms, and so can become decorated λ-expressions in their reducts.

In the following we consider a fixed typed program (CT^S, t) and its translation $(CT^T, (\!|\mathsf{t}|\!))$ as defined in Section 5.

6.1 Typing is preserved under translation

The typability of programs obtained by translation is shown by first proving that the translated terms are typed (Lemma 9) and then that class and interface declarations are well formed (Lemma 10).

We start by stating the relations between the class table CT^S and its translation CT^T. These relations can be easily checked looking at the translations of method headers and constructors (Definition 3).

▶ **Lemma 8.**
1. *The nominal type* T *belongs to* CT^S *if and only if* T *belongs to* CT^T*.*
2. *The subtyping* $\mathsf{T} <: \mathsf{T}'$ *holds in* CT^S *if and only if* $\mathsf{T} <: \mathsf{T}'$ *holds in* CT^T*.*
3. $\mathtt{fieldsS}(\mathsf{C}) = \overrightarrow{\tau}\,\overrightarrow{\mathsf{f}}$ *in* CT^S *if and only if* $\mathtt{fieldsT}(\mathsf{C}) = \overrightarrow{|\tau|}\,\overrightarrow{\mathsf{f}}$ *in* CT^T*.*
4. $\mathtt{mtypeS}(\mathsf{m};\tau) = \overrightarrow{\sigma} \to \sigma$ *in* CT^S *if and only if* $\mathtt{mtypeT}(\mathsf{m};\tau) = \overrightarrow{|\sigma|} \to |\sigma|$ *in* CT^T*.*
5. $\mathtt{A\text{-}mtypeS}(\varphi) = \overrightarrow{\tau} \to \tau$ *in* CT^S *if and only if* $\mathtt{A\text{-}mtypeT}(\varphi) = \overrightarrow{|\tau|} \to |\tau|$ *in* CT^T*.*

Building on previous lemma we can show that the translation of a term t typed by τ in $\vdash_S$ gives a term t′ typed by τ in $\vdash_T$ and similarly for $\vdash_S^*$ and $\vdash_T^*$. Figure 9 shows the mapping $\{\![\;]\!\}$ between type derivations. We also need to translate typing contexts Γ. We

$$\left\{\!\!\left[\frac{\mathsf{x}:\tau\in\Gamma}{\Gamma\vdash_S \mathsf{x}:\tau}\,[\text{S-VAR}]\right]\!\!\right\} = \frac{\dfrac{\mathsf{x}:|\tau|\in|\Gamma|}{|\Gamma|\vdash_T \mathsf{x}:|\tau|}\,[\text{T-VAR}]}{|\Gamma|\vdash_T (\tau)\,\mathsf{x}:\tau}\,[\text{T-DC}]$$

$$\left\{\!\!\left[\frac{\mathcal{D}::\Gamma\vdash_S \mathsf{t}:\mathsf{C}[\&\iota]\quad \tau\,\mathsf{f}\in\mathtt{fieldsS}(\mathsf{C})}{\Gamma\vdash_S \mathsf{t.f}:\tau}\,[\text{S-FIELD}]\right]\!\!\right\} =$$
$$\frac{\dfrac{\{\![\mathcal{D}]\!\}::|\Gamma|\vdash_T \mathsf{t}':\mathsf{C}[\&\iota]\quad |\tau|\,\mathsf{f}\in\mathtt{fieldsT}(\mathsf{C})}{|\Gamma|\vdash_T \mathsf{t}'.\mathsf{f}:|\tau|}\,[\text{T-FIELD}]}{|\Gamma|\vdash_T (\tau)\,(\mathsf{t}'.\mathsf{f}):\tau}\,[\text{T-DC}]$$

$$\left\{\!\!\left[\frac{\mathcal{D}::\Gamma\vdash_S \mathsf{t}:\tau\quad \mathtt{mtypeS}(\mathsf{m};\tau)=\overrightarrow{\sigma}\to\sigma\quad \overline{\mathcal{D}}::\Gamma\vdash^*_S \bar{\mathsf{t}}:\bar{\sigma}}{\Gamma\vdash_S \mathsf{t.m}(\overrightarrow{\mathsf{t}}):\sigma}\,[\text{S-INVK}]\right]\!\!\right\} =$$
$$\frac{\dfrac{\{\![\mathcal{D}]\!\}::|\Gamma|\vdash_T \mathsf{t}':\tau\quad \mathtt{mtypeT}(\mathsf{m};\tau)=\overrightarrow{|\sigma|}\to|\sigma|\quad \dfrac{\overline{\{\![\mathcal{D}]\!\}}:|\Gamma|\vdash^*_T \overline{\mathsf{t}'}:\bar{\sigma}\quad \bar{\sigma}<:|\bar{\sigma}|}{|\Gamma|\vdash^*_T \overline{\mathsf{t}'}:|\bar{\sigma}|}\,[\text{T-**-N}]}{|\Gamma|\vdash_T \mathsf{t}'.\mathsf{m}(\overrightarrow{\mathsf{t}'}):|\sigma|}\,[\text{T-INVK}]}{|\Gamma|\vdash_T (\sigma)\,(\mathsf{t}'.\mathsf{m}(\overrightarrow{\mathsf{t}'})):\sigma}\,[\text{T-DC}]$$

$$\left\{\!\!\left[\frac{\mathtt{fieldsS}(\mathsf{C})=\overrightarrow{\tau}\,\overrightarrow{\mathsf{f}}\quad \overline{\mathcal{D}}::\Gamma\vdash^*_S \bar{\mathsf{t}}:\bar{\tau}}{\Gamma\vdash_S \mathsf{new\,C}(\overrightarrow{\mathsf{t}}):\mathsf{C}}\,[\text{S-NEW}]\right]\!\!\right\} =$$
$$\frac{\mathtt{fieldsT}(\mathsf{C})=\overrightarrow{|\tau|}\,\overrightarrow{\mathsf{f}}\quad \dfrac{\overline{\{\![\mathcal{D}]\!\}}:|\Gamma|\vdash^*_T \overline{\mathsf{t}'}:\bar{\tau}\quad \bar{\tau}<:|\bar{\tau}|}{|\Gamma|\vdash^*_T \overline{\mathsf{t}'}:|\bar{\tau}|}\,[\text{T-**-N}]}{|\Gamma|\vdash_T \mathsf{new\,C}(\overrightarrow{\mathsf{t}'}):\mathsf{C}}\,[\text{T-NEW}]$$

$$\left\{\!\!\left[\frac{\mathtt{A\text{-}mtypeS}(\varphi)=\overrightarrow{\tau}\to\tau\quad \mathcal{D}::\Gamma,\overrightarrow{\mathsf{y}}:\overrightarrow{\tau}\vdash^*_S \mathsf{t}:\tau}{\Gamma\vdash_S (\overrightarrow{\mathsf{y}}\to\mathsf{t})^{\varphi}:\varphi}\,[\text{S-}\lambda]\right]\!\!\right\} =$$
$$\frac{\mathtt{A\text{-}mtypeT}(\varphi)=\overrightarrow{|\tau|}\to|\tau|\quad \dfrac{\{\![\mathcal{D}]\!\}::|\Gamma|,\overrightarrow{\mathsf{y}}:\overrightarrow{|\tau|}\vdash^*_T \mathsf{t}':\tau\quad \tau<:|\tau|}{|\Gamma|,\overrightarrow{\mathsf{y}}:\overrightarrow{|\tau|}\vdash^*_T \mathsf{t}':|\tau|}\,[\text{T-**-N}]}{|\Gamma|\vdash_T (\overrightarrow{\mathsf{y}}\to\mathsf{t}')^{\varphi}:\varphi}\,[\text{T-}\lambda]$$

$$\left\{\!\!\left[\frac{\mathcal{D}::\Gamma\vdash_S (\mathsf{t}_\lambda)^{\varphi}:\varphi}{\Gamma\vdash^*_S \mathsf{t}_\lambda:\varphi}\,[\text{S-*-}\lambda]\right]\!\!\right\} = \frac{\dfrac{\dfrac{\{\![\mathcal{D}]\!\}::|\Gamma|\vdash_T (\mathsf{t}'_\lambda)^{\varphi}:\varphi}{|\Gamma|\vdash^*_T \mathsf{t}'_\lambda:\varphi}\,[\text{T-*-}\lambda]}{|\Gamma|\vdash_T (\varphi)\mathsf{t}'_\lambda:\varphi}\,[\text{T-UC}]\quad \varphi<:\varphi}{|\Gamma|\vdash^*_T (\varphi)\mathsf{t}'_\lambda:\varphi}\,[\text{T-*-N}]$$

$$\left\{\!\!\left[\frac{\mathcal{D}::\Gamma\vdash_S \mathsf{t}:\sigma\quad \mathsf{t}\neq\mathsf{t}_\lambda\quad \sigma<:\tau}{\Gamma\vdash^*_S \mathsf{t}:\tau}\,[\text{S-*-N}]\right]\!\!\right\} = \frac{\{\![\mathcal{D}]\!\}::|\Gamma|\vdash_T \mathsf{t}':\sigma\quad \mathsf{t}'\neq\mathsf{t}_\lambda\quad \sigma<:\tau}{|\Gamma|\vdash^*_T \mathsf{t}':\tau}\,[\text{T-*-N}]$$

$$\left\{\!\!\left[\frac{\mathcal{D}::\Gamma\vdash^*_S \mathsf{t}:\tau}{\Gamma\vdash_S (\tau)\,\mathsf{t}:\tau}\,[\text{S-UC}]\right]\!\!\right\} = \frac{\{\![\mathcal{D}]\!\}::|\Gamma|\vdash^*_T \mathsf{t}':\tau}{|\Gamma|\vdash_T (\tau)\,\mathsf{t}':\tau}\,[\text{T-UC}]$$

Figure 9 Mapping from derivations in SJ&+ to derivations in TJ&.

extend the mapping $|\ |$ to typing contexts in the expected way: $|\Gamma| = \{\mathsf{x} : |\tau| \mid x : \tau \in \Gamma\}$. Each derivation $\mathcal{D} :: \Gamma \vdash_S \mathsf{t} : \tau$ is translated to a derivation $\{[\mathcal{D}]\}$ of $|\Gamma| \vdash_T \mathsf{t}' : \tau$. This modification of the context implies the need to add a downcast of the variable from $|\tau|$ to τ, when translating the axiom. Analogously, we need to add a downcast when the last applied rule is [S-FIELD] or [S-INVK]. This is due to the relations between lookup functions in CT^S and CT^T, see Lemma 8(3) and (4).

In the translations of rules [S-INVK], [S-NEW] and [S-λ] it is handy to consider the following rule:

$$\frac{\Delta \vdash^*_T \mathsf{t} : \sigma \quad \mathsf{t} \neq \mathsf{t}_\lambda \quad \sigma <: \tau}{\Delta \vdash^*_T \mathsf{t} : \tau}\ [\text{T-**-N}]$$

This rule is admissible in the type systems of Figure 2 since, if $\mathsf{t} \neq \mathsf{t}_\lambda$, then the only way to derive $\Delta \vdash^*_T \mathsf{t} : \sigma$ is

$$\frac{\Delta \vdash_T \mathsf{t} : \sigma' \quad \mathsf{t} \neq \mathsf{t}_\lambda \quad \sigma' <: \sigma}{\Delta \vdash^*_T \mathsf{t} : \sigma}\ [\text{T-*-N}]$$

Therefore, being $<:$ transitive

$$\frac{\Delta \vdash_T \mathsf{t} : \sigma' \quad \mathsf{t} \neq \mathsf{t}_\lambda \quad \sigma' <: \tau}{\Delta \vdash^*_T \mathsf{t} : \tau}\ [\text{T-*-N}]$$

This rule does the subsumption needed for the relations between the lookup functions in CT^S and CT^T, see Lemma 8(3), (4) and (5).

Looking at rule [S-λ] we see that the translation of a type derivation for a decorated λ-expression is still a derivation for a decorated λ-expression. This is crucial for the translation of rule [S-*-λ], since $\mathcal{D} :: \Gamma \vdash_S (\mathsf{t}_\lambda)^\varphi : \varphi$ can only be the conclusion of rule [S-λ]. Then $(\![\mathcal{D}]\!) = (\mathsf{t}'_\lambda)^\varphi$ and we can use rule [T-*-λ] before doing the upcast.

The translation of rules [S-*-N] and [S-UC] is the identity. It works since thanks to Lemma 8(2) $\sigma <: \tau$ holds in CT^S if and only if $\sigma <: \tau$ holds in CT^T.

As expected, the subject in the conclusion of $\{[\mathcal{D}]\}$ is $(\![\mathcal{D}]\!)$: it can be easily checked by induction on derivations comparing the definitions of $\{[\ ]\}$ (Figure 9) and $(\![\ ]\!)$ (Figure 7). This implies that no t' in Figure 9 can be a pure λ-expression and so justifies the applications of rule [T-**-N].

To sum up we proved:

▶ **Lemma 9.**
1. *If $\mathcal{D} :: \Gamma \vdash_S \mathsf{t} : \tau$, then $|\Gamma| \vdash_T (\![\mathcal{D}]\!) : \tau$.*
2. *If $\mathcal{D} :: \Gamma \vdash^*_S \mathsf{t} : \tau$, then $|\Gamma| \vdash^*_T (\![\mathcal{D}]\!) : \tau$.*

We end this subsection by showing that the well-formedness of classes and interfaces in CT^S implies the well-formedness of their translations, i.e. the well-formedness of the obtained class table CT^T.

▶ **Lemma 10.**
1. *If the method declaration M^S is S-OK in the class or interface T for the class table CT^S, then its translation $[\![\mathsf{M}^S]\!]^{\mathsf{T}}$ is T-OK in T for the class table CT^T.*
2. *If the class declaration CD^S is S-OK for the class table CT^S, then its translation $[\![\mathsf{CD}^S]\!]$ is T-OK for the class table CT^T.*
3. *If the interface declaration ID^S is S-OK for the class table CT^S, then its translation $[\![\mathsf{ID}^S]\!]$ is T-OK for the class table CT^T.*

Proof. (1). Let $\mathsf{M}^S = \tau \mathsf{m}(\overrightarrow{\tau}\,\overrightarrow{\mathsf{x}})\{\mathtt{return}\ \ \mathsf{t};\}$, then, by rule [M S-*OK* in T] of Figure 6, $\mathtt{mtypeS}(\mathsf{m};\mathsf{T}) = \overrightarrow{\tau} \to \tau$ and $\mathcal{D} :: \overrightarrow{\mathsf{x}} : \overrightarrow{\tau}, \mathsf{this} : \mathsf{T} \vdash^*_S \mathsf{t} : \tau$ for some $\mathcal{D}$. Lemma 8(4) implies $\mathtt{mtypeT}(\mathsf{m};\mathsf{T}) = \overrightarrow{|\tau|} \to |\tau|$. We have $\overrightarrow{\mathsf{x}} : |\overrightarrow{\tau}|, \mathsf{this} : \mathsf{T} \vdash^*_T (\![\mathcal{D}]\!) : \tau$ by Lemma 9(2). By Definition 4 we get $[\![\mathsf{M}^S]\!]^{\mathsf{T}} = |\tau|\mathsf{m}(\overrightarrow{|\tau|}\,\overrightarrow{\mathsf{x}})\{\mathtt{return}\ \ (\![\mathcal{D}]\!);\}$. We can apply rule [M T-*OK* in T] of Figure 3.

(2). Let $\mathsf{CD}^S = \mathsf{class\,C\,extends\,D\,implements}\ \overrightarrow{\mathsf{I}}\ \{\overline{\tau}\,\overline{\mathsf{f}}; \mathsf{K}^S\,\overline{\mathsf{M}^S}\}$, then, by rule [C S-*OK*] of Figure 6, $\mathsf{K}^S = \mathsf{C}(\overrightarrow{\sigma}\,\overrightarrow{\mathsf{g}}, \overrightarrow{\tau}\,\overrightarrow{\mathsf{f}})\{\mathsf{super}(\overrightarrow{\mathsf{g}}); \mathsf{this}.\overline{\mathsf{f}} = \overline{\mathsf{f}};\}$ and $\mathtt{fieldsS}(\mathsf{D}) = \overrightarrow{\sigma}\,\overrightarrow{\mathsf{g}}$ and $\overline{\mathsf{M}^S}$ S-*OK* in C. By Definition 3 $[\![\mathsf{K}^S]\!] = \mathsf{C}(\overrightarrow{|\sigma|}\,\overrightarrow{\mathsf{g}}, \overrightarrow{|\tau|}\,\overrightarrow{\mathsf{f}})\{\mathsf{super}(\overrightarrow{\mathsf{g}}); \mathsf{this}.\overline{\mathsf{f}} = \overline{\mathsf{f}};\}$. Lemma 8(4) implies $\mathtt{fieldsS}(\mathsf{D}) = \overrightarrow{|\sigma|}\,\overrightarrow{\mathsf{g}}$. Point (1) gives $\overline{[\![\mathsf{M}^S]\!]^{\mathsf{C}}}$ T-*OK* in C. By Definition 5 $[\![\mathsf{CD}^S]\!] = \mathsf{class\,C\,extends\,D\,implements}\ \overrightarrow{\mathsf{I}}\ \{\overline{|\tau|}\,\overline{\mathsf{f}}; [\![\mathsf{K}^S]\!]\ \overline{[\![\mathsf{M}^S]\!]^{\mathsf{C}}}\}$. Therefore we can apply rule [C T-*OK*] of Figure 3.

The proof of Point (3) is similar and simpler than the proof of Point (2). ◀

Therefore, the final result is an immediate consequence of Lemmas 10 and 9(1).

▶ **Theorem 11.** *The translation of a typed program in* SJ&+ *is a typed program in* TJ&.

6.2 Semantics is preserved under translation

Given a typed program (CT^S, t) and its translation $(CT^T, (\![\mathsf{t}]\!))$ we want to state the relations between the reduction of t with $\longrightarrow_S$ and the reduction of $(\![\mathsf{t}]\!)$ with $\longrightarrow_T$.

Looking at Figure 7 it is clear that $(\![\mathsf{t}]\!)$ is obtained from t by adding casts. We prove that translated terms can be typed in $\vdash_S$. This implies that translated terms cannot be stuck and that either t and $(\![\mathsf{t}]\!)$ both reduce to values or both have infinite computations.

▶ **Lemma 12.**

1. *If* $\mathcal{D} :: \Gamma \vdash_S \mathsf{t} : \tau$*, then* $\Gamma \vdash_S (\![\mathcal{D}]\!) : \tau$.
2. *If* $\mathcal{D} :: \Gamma \vdash^*_S \mathsf{t} : \tau$*, then* $\Gamma \vdash^*_S (\![\mathcal{D}]\!) : \tau$.

Proof. The proof of (1) and (2) is by simultaneous induction on the construction of $(\![\mathcal{D}]\!)$ done in Figure 7. We only consider some interesting cases.

Rule [S-INVK].

$$\left(\!\!\left[\frac{\mathcal{D} :: \Gamma \vdash_S \mathsf{t} : \tau \quad \mathtt{mtypeS}(\mathsf{m};\tau) = \overrightarrow{\sigma} \to \sigma \quad \overline{\mathcal{D}} :: \Gamma \vdash^*_S \overline{\mathsf{t}} : \overline{\sigma}}{\Gamma \vdash_S \mathsf{t}.\mathsf{m}(\overrightarrow{\mathsf{t}}) : \sigma}\,[\text{S-INVK}]\right]\!\!\right) = (\sigma)\,((\![\mathcal{D}]\!).\mathsf{m}(\overrightarrow{(\![\mathcal{D}]\!)}))$$

By IH $\Gamma \vdash_S (\![\mathcal{D}]\!) : \tau$ and $\Gamma \vdash^*_S \overline{(\![\mathcal{D}]\!)} : \overline{\sigma}$. By applying [S-INVK] we derive $\Gamma \vdash_S (\![\mathcal{D}]\!).\mathsf{m}(\overrightarrow{(\![\mathcal{D}]\!)}) : \sigma$. Rule [S-*-N] gives $\Gamma \vdash^*_S (\![\mathcal{D}]\!).\mathsf{m}(\overrightarrow{(\![\mathcal{D}]\!)}) : \sigma$. We conclude $\Gamma \vdash_S (\sigma)\,((\![\mathcal{D}]\!).\mathsf{m}(\overrightarrow{(\![\mathcal{D}]\!)})) : \sigma$ using [S-UC].

Rule [S-*-λ].

$$\left(\!\!\left[\frac{\mathcal{D} :: \Gamma \vdash_S (\mathsf{t}_\lambda)^\varphi : \varphi}{\Gamma \vdash^*_S \mathsf{t}_\lambda : \varphi}\,[\text{S-*-}\lambda]\right]\!\!\right) = (\varphi)\mathsf{t}'_\lambda \qquad \text{if } (\![\mathcal{D}]\!) = (\mathsf{t}'_\lambda)^\varphi$$

By IH and the condition $(\![\mathcal{D}]\!) = (\mathsf{t}'_\lambda)^\varphi$ we get $\Gamma \vdash_S (\mathsf{t}'_\lambda)^\varphi : \varphi$. By applying [S-*-$\lambda$] we derive $\Gamma \vdash^*_S \mathsf{t}'_\lambda : \varphi$. Rule [S-UC] gives $\Gamma \vdash_S (\varphi)\,\mathsf{t}'_\lambda : \varphi$. We conclude $\Gamma \vdash^*_S (\varphi)\,\mathsf{t}'_\lambda : \varphi$ using [S-*-N]. ◀

In the remaining of this section we want to establish the relations between the reduction of a term in SJ&+, according to a given class table, and the reduction of its translation in TJ&, using the translated class table. This issue deserves a preliminary discussion, because of difficulties arising from the presence of λ-expressions. The value obtained computing a typed program (if any) is always a proper value, i.e., either an object or a λ-expression with its target type.

The first remark is that, in general, *the translation of a proper value is not a proper value.* For example, let us assume a class D that has a field of type I, where I is a functional interface with abstract method of type $\mathsf{C} \to \mathsf{C}$. Then we have $(\![\mathsf{new\,D(x \to x)}]\!) = \mathsf{new\,D((I)\,(x \to (C)\,x))}$, where $\mathsf{new\,D((I)\,(x \to (C)\,x))}$ reduces to $\mathsf{new\,D((x \to (C)\,x)^I)}$. The pure λ-expression $\mathsf{x} \to \mathsf{x}$ has type I in SJ&+, as provided by the class table, namely by the field type. As expected, $\mathsf{x} \to \mathsf{x}$ is explicitly decorated by this target type in $\mathsf{new\,D((x \to (C)\,x)^I)}$ and its body is translated (as well as method bodies are translated in the class table).

The second remark is that $\mathsf{t} \longrightarrow_S \mathsf{t}'$ *does not imply* $(\![\mathsf{t}]\!) \longrightarrow_T^* \mathsf{t}''$ *with* $(\![\mathsf{t}']\!) \longrightarrow_T^* \mathsf{t}''$, for some t''. For example, let us assume a further class E without fields, containing a method m which has return type J, parameter y of type D and body $\mathsf{z} \to \mathsf{y}$. Let t be the term $\mathsf{t} = \mathsf{new\,E().m(new\,D(x \to x))}$. Then t reduces as follows:

$$\mathsf{new\,E().m(new\,D(x \to x))} \longrightarrow_S \mathsf{(z \to new\,D(x \to x))^J}$$

We consider the translation $(\![\mathsf{t}]\!) = \mathsf{(J)\,(new\,E().m(new\,D((I)\,(x \to (C)\,x))))}$ and its reduction:

$$\begin{array}{lll} (\![\mathsf{t}]\!) & \longrightarrow_T & \mathsf{(J)\,(new\,E().m(new\,D((x \to (C)\,x)^I)))} \\ & \longrightarrow_T & \mathsf{(J)\,((J)\,(z \to (D)\,(new\,D((x \to (C)\,x)^I))))} \\ & \longrightarrow_T^* & \mathsf{(z \to (D)\,(new\,D((x \to (C)\,x)^I)))^J} \end{array}$$

It is clear that $(\![\mathsf{(z \to new\,D(x \to x))^J}]\!) = \mathsf{(z \to new\,D((I)\,(x \to (C)\,x)))^J}$ is a value, since the type cast is included in the body of the λ-expression and so no reduction rule applies. Therefore it does not reduce to $\mathsf{(z \to (D)\,(new\,D((x \to (C)\,x)^I)))^J}$.

Finally, we assume the method n in the class E, which has return type J, no parameters and body $\mathsf{z} \to \mathsf{new\,D(x \to x)}$. Then we get

$$\mathsf{new\,E().n()} \longrightarrow_S \mathsf{(z \to new\,D(x \to x))^J}$$

and its translation so reduces

$$\begin{array}{lll} \mathsf{(J)\,(new\,E().n())} & \longrightarrow_T & \mathsf{(J)\,((J)\,(z \to new\,D((I)\,(x \to (C)\,x))))} \\ & \longrightarrow_T^* & \mathsf{(z \to new\,D((I)\,(x \to (C)\,x))^J} \end{array}$$

Comparing methods m and n, we observe that the body of m is a function returning the parameter, while the body of n is a function returning the expression $\mathsf{new\,D(x \to x)}$. Therefore, the evaluations of the two calls, $\mathsf{m(new\,D(x \to x))}$ and $\mathsf{n(\ )}$, return the same value, while their translations reduce to different values. This happens because, when calling m, the translation of the parameter $\mathsf{new\,D(x \to x)}$ is reduced before replacing it to the formal parameter in the λ-expression. Moreover the formal parameter is cast to D.

Notice that, in the above examples, there are no intersection types: this directs our attention to the fact that the casts, that are introduced by translation, are reduced only when they are in evaluation contexts.

This discussion suggests that *the relation between* $(\![\mathsf{t}']\!)$ *and* t'' *when* $\mathsf{t} \longrightarrow_S^* \mathsf{t}'$ *and* $(\![\mathsf{t}]\!) \longrightarrow_T^* \mathsf{t}''$ *cannot be expressed as a function.* Intuitively, this relation is an equivalence that can be realised by:

- ignoring type casts on closed terms different from λ-expressions;
- identifying type casts of λ-expressions with the decorated λ-expressions obtained by reducing them.

▶ **Definition 13.** *The* cast-equivalence *relation* $\approx$ *on closed and typed terms of* SJ&+ *is the smallest congruence which satisfies:*

$$\frac{\vdash_S^* \mathsf{t} : \tau \quad \mathsf{t} \neq \mathsf{t}'_\lambda}{(\tau)\,\mathsf{t} \approx \mathsf{t}} \qquad\qquad (\varphi)\,\mathsf{t}_\lambda \approx (\mathsf{t}_\lambda)^\varphi$$

Notice that cast-equivalence allows to eliminate all casts introduced by the translation of closed and typed source terms. Lemma 12 assures that all these casts are upcasts when typing in SJ&+. It is easy to verify that cast-equivalence preserves typing in SJ&+.

In general $(\!|\mathsf{t}|\!) \approx \mathsf{t}$ does not hold. Let class D be as in previous example, then the translation of $\mathsf{new\,D(x \to x)}$ is $\mathsf{new\,D((I)\,(x \to (C)\,x))}$.

We would like to prove:

$$\text{if } \mathsf{t} \longrightarrow_S \mathsf{t}', \text{ then } (\!|\mathsf{t}|\!) \longrightarrow_T^* \mathsf{t}'' \text{ and } (\!|\mathsf{t}'|\!) \approx \mathsf{t}'' \text{ for some } \mathsf{t}''$$

Lemma 15 shows a more general formulation of this relation, which better fits the proof of Theorem 16.

We first show that terms cast-equivalent to translations of values reduce to cast-equivalent proper values using $\longrightarrow_T$.

▶ **Proposition 14.** *Let* v *be a closed and typed value of* SJ&+ *and* $(\!|\mathsf{v}|\!) \approx \mathsf{t}$*. Then* $\mathsf{t} \longrightarrow_T^* \mathsf{w}$ *and* $(\!|\mathsf{v}|\!) \approx \mathsf{w}$ *for some* w*.*

Proof. The first observation is that in $(\!|\mathsf{v}|\!)$ all λ-expressions are enclosed by a type cast, so reducing $(\!|\mathsf{v}|\!)$ we cannot obtain a pure λ-expression. In general, $(\!|\mathsf{v}|\!)$ is not a value since it contains added casts. If t is a proper value we are done. Otherwise, from t we can get a proper value w by reducing the casts inside evaluation contexts, since by Lemma 12 all casts added by the translation do not fail at run time. By definition of $\approx$ we also get $\mathsf{t} \approx \mathsf{w}$. From the transitivity of $\approx$ we conclude $(\!|\mathsf{v}|\!) \approx \mathsf{w}$. ◀

▶ **Lemma 15.** *Let* t *be a closed and typed term of* SJ&+*. If* $\mathsf{t} \longrightarrow_S \mathsf{t}_1$ *and* $(\!|\mathsf{t}|\!) \approx \mathsf{t}_2$*, then* $\mathsf{t}_2 \longrightarrow_T^* \mathsf{t}'$ *and* $(\!|\mathsf{t}_1|\!) \approx \mathsf{t}'$ *for some* t'*.*

Proof. The proof is by cases and by induction on the reduction rules of Figure 4. We only consider some interesting cases.

Rule [S-InvkNew].

$$\frac{\mathtt{mbodyS}(\mathsf{m};\mathsf{C}) = (\overrightarrow{\mathsf{x}}, \mathsf{t_m}) \quad \mathtt{mtypeS}(\mathsf{m};\mathsf{C}) = \overrightarrow{\tau} \to \tau}{\mathsf{new\,C}(\overrightarrow{\mathsf{v}}).\mathsf{m}(\overrightarrow{\mathsf{u}}) \longrightarrow_S [\overrightarrow{\mathsf{x}} \mapsto (\overrightarrow{\mathsf{u}})^{?\overrightarrow{\tau}}, \mathsf{this} \mapsto \mathsf{new\,C}(\overrightarrow{\mathsf{v}})](\mathsf{t_m})^{?\tau}} \text{[S-InvkNew]}$$

We get $(\!|\mathsf{new\,C}(\overrightarrow{\mathsf{v}}).\mathsf{m}(\overrightarrow{\mathsf{u}})|\!) = (\tau)\,(\mathsf{new\,C}(\overrightarrow{(\!|\mathsf{v}|\!)}).\mathsf{m}(\overrightarrow{(\!|\mathsf{u}|\!)}))$. From $(\tau)\,(\mathsf{new\,C}(\overrightarrow{(\!|\mathsf{v}|\!)}).\mathsf{m}(\overrightarrow{(\!|\mathsf{u}|\!)})) \approx \mathsf{t}_2$ we get $\mathsf{t}_2 = (\overline{\sigma})\,(\mathsf{new\,C}(\overrightarrow{\mathsf{r}}).\mathsf{m}(\overrightarrow{\mathsf{s}}))$, where $\overrightarrow{(\!|\mathsf{v}|\!)} \approx \overrightarrow{\mathsf{r}}$ and $\overrightarrow{(\!|\mathsf{u}|\!)} \approx \overrightarrow{\mathsf{s}}$. This implies $\mathsf{t}_2 \longrightarrow_T^*$ $(\overline{\sigma})\,(\mathsf{new\,C}(\overrightarrow{\mathsf{w}}).\mathsf{m}(\overrightarrow{\mathsf{w}'}))$, where $\overrightarrow{\mathsf{r}} \longrightarrow_T^* \overrightarrow{\mathsf{w}}$ and $\overrightarrow{\mathsf{s}} \longrightarrow_T^* \overrightarrow{\mathsf{w}}'$ by Proposition 14. By rule [T-InvkNew] $(\overline{\sigma})\,(\mathsf{new\,C}(\overrightarrow{\mathsf{w}}).\mathsf{m}(\overrightarrow{\mathsf{w}'})) \longrightarrow_T (\overline{\sigma})\,([\overrightarrow{\mathsf{x}} \mapsto \overrightarrow{\mathsf{w}'}, \mathsf{this} \mapsto \mathsf{new\,C}(\overrightarrow{\mathsf{w}})](\!|\mathsf{t_m}|\!))$, since by construction $(\!|\mathsf{t_m}|\!)$ is the body of method m for class C in the class table CT^T and by definition a translation is never a pure λ-expression. We have

$$\mathsf{t}_1 = [\overrightarrow{\mathsf{x}} \mapsto (\overrightarrow{\mathsf{u}})^{?\overrightarrow{\tau}}, \mathsf{this} \mapsto \mathsf{new\,C}(\overrightarrow{\mathsf{v}})](\mathsf{t_m})^{?\tau}$$

which implies $(\!|\mathsf{t}_1|\!) = [\overrightarrow{\mathsf{x}} \mapsto \overrightarrow{(\!|\mathsf{u}|\!)}, \mathsf{this} \mapsto \mathsf{new\,C}(\overrightarrow{(\!|\mathsf{v}|\!)})](\!|\mathsf{t_m}|\!)$ since the translation of a decorated λ-expression is a λ-expression decorated with the same type, see rules [S-λ] and [S-*-λ]. From Proposition 14 we have $\overrightarrow{(\!|\mathsf{v}|\!)} \approx \overrightarrow{\mathsf{w}}$ and $\overrightarrow{(\!|\mathsf{u}|\!)} \approx \overrightarrow{\mathsf{w}}'$. If $\overline{\sigma} = \overline{\sigma}'\sigma$, since $\mathsf{t}_2 = (\overline{\sigma}')\,(\sigma)\,(\mathsf{new\,C}(\overrightarrow{\mathsf{r}}).\mathsf{m}(\overrightarrow{\mathsf{s}}))$ is typable, we have $\vdash_S^* \mathsf{new\,C}(\overrightarrow{\mathsf{r}}).\mathsf{m}(\overrightarrow{\mathsf{s}}) : \sigma$, which implies $\vdash_S^* [\overrightarrow{\mathsf{x}} \mapsto \overrightarrow{\mathsf{w}'}, \mathsf{this} \mapsto \mathsf{new\,C}(\overrightarrow{\mathsf{w}})](\!|\mathsf{t_m}|\!) : \sigma$ by Subject Reduction (Theorem 2(1)). We conclude

$$(\!|\mathsf{t}_1|\!) = [\overrightarrow{\mathsf{x}} \mapsto \overrightarrow{(\!|\mathsf{u}|\!)}, \mathsf{this} \mapsto \mathsf{new\,C}(\overrightarrow{(\!|\mathsf{v}|\!)})](\!|\mathsf{t_m}|\!) \approx (\overline{\sigma})\,([\overrightarrow{\mathsf{x}} \mapsto \overrightarrow{\mathsf{w}'}, \mathsf{this} \mapsto \mathsf{new\,C}(\overrightarrow{\mathsf{w}})](\!|\mathsf{t_m}|\!))$$

If $\overline{\sigma}$ is empty, then

$$(\![\mathsf{t}_1]\!) = [\overrightarrow{\mathsf{x}} \mapsto \overrightarrow{(\![\mathsf{u}]\!)}, \mathsf{this} \mapsto \mathsf{new}\,\mathsf{C}(\overrightarrow{(\![\mathsf{v}]\!)})](\![\mathsf{t_m}]\!) \approx ([\overrightarrow{\mathsf{x}} \mapsto \overrightarrow{\mathsf{w}}', \mathsf{this} \mapsto \mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{w}})](\![\mathsf{t_m}]\!))$$

Rule [S-Invkλ-A].

$$\frac{\texttt{A-nameS}(\varphi) = \mathsf{m} \quad \texttt{A-mtypeS}(\varphi) = \overrightarrow{\tau} \to \tau}{(\overrightarrow{\mathsf{y}} \to \mathsf{t_m})^{\varphi}.\mathsf{m}(\overrightarrow{\mathsf{v}}) \longrightarrow_S [\overrightarrow{\mathsf{y}} \mapsto (\overrightarrow{\mathsf{v}})^{?\overrightarrow{\tau}}](\mathsf{t_m})^{?\tau}}\ \text{[S-Invk}\lambda\text{-A]}$$

We get $(\![(\overrightarrow{\mathsf{y}} \to \mathsf{t_m})^{\varphi}.\mathsf{m}(\overrightarrow{\mathsf{v}})]\!) = (\tau)\,((\overrightarrow{\mathsf{y}} \to (\![\mathsf{t_m}]\!))^{\varphi}.\mathsf{m}((\![\overrightarrow{\mathsf{v}}]\!)))$.
From $(\tau)\,((\overrightarrow{\mathsf{y}} \to (\![\mathsf{t_m}]\!))^{\varphi}.\mathsf{m}((\![\overrightarrow{\mathsf{v}}]\!))) \approx \mathsf{t}_2$ we get $\mathsf{t}_2 = (\overline{\sigma})\,((\overrightarrow{\mathsf{y}} \to \mathsf{r})^{\varphi}.\mathsf{m}(\overrightarrow{\mathsf{r}}))$, where $(\![\mathsf{t_m}]\!) \approx \mathsf{r}$ and $\overrightarrow{(\![\mathsf{v}]\!)} \approx \overrightarrow{\mathsf{r}}$. This implies $\mathsf{t}_2 \longrightarrow_T (\overline{\sigma})\,((\overrightarrow{\mathsf{y}} \to \mathsf{r})^{\varphi}.\mathsf{m}(\overrightarrow{\mathsf{w}}))$, where $\overrightarrow{\mathsf{r}} \longrightarrow_T^* \overrightarrow{\mathsf{w}}$ by Proposition 14. By rule [T-Invkλ-A]

$$(\overline{\sigma})\,((\overrightarrow{\mathsf{y}} \to \mathsf{r})^{\varphi}.\mathsf{m}(\overrightarrow{\mathsf{w}})) \longrightarrow_T^* (\overline{\sigma})\,([\overrightarrow{\mathsf{y}} \mapsto \overrightarrow{\mathsf{w}}]\mathsf{r})$$

We have $\mathsf{t}_1 = [\overrightarrow{\mathsf{y}} \mapsto (\overrightarrow{\mathsf{v}})^{?\overrightarrow{\tau}}](\mathsf{t_m})^{?\tau}$, which implies $(\![\mathsf{t}_1]\!) = [\overrightarrow{\mathsf{y}} \mapsto \overrightarrow{(\![\mathsf{v}]\!)}](\![\mathsf{t_m}]\!)$. From Proposition 14 we have $\overrightarrow{(\![\mathsf{v}]\!)} \approx \overrightarrow{\mathsf{w}}$. If $\overline{\sigma} = \overline{\sigma}'\sigma$, since $\mathsf{t}_2 = (\overline{\sigma}')\,(\sigma)\,((\overrightarrow{\mathsf{y}} \to \mathsf{r})^{\varphi}.\mathsf{m}(\overrightarrow{\mathsf{r}}))$ is typable, we have $\vdash_S^* (\overrightarrow{\mathsf{y}} \to \mathsf{r})^{\varphi}.\mathsf{m}(\overrightarrow{\mathsf{r}}) : \sigma$, which implies $\vdash_S^* [\overrightarrow{\mathsf{y}} \mapsto \overrightarrow{\mathsf{w}}]\mathsf{r} : \sigma$ by Subject Reduction (Theorem 2(1)). We conclude $(\![\mathsf{t}_1]\!) = [\overrightarrow{\mathsf{y}} \mapsto \overrightarrow{(\![\mathsf{v}]\!)}](\![\mathsf{t_m}]\!) \approx (\overline{\sigma})\,([\overrightarrow{\mathsf{y}} \mapsto \overrightarrow{\mathsf{w}}]\mathsf{r})$. If $\overline{\sigma}$ is empty, then $(\![\mathsf{t}_1]\!) = [\overrightarrow{\mathsf{y}} \mapsto \overrightarrow{(\![\mathsf{v}]\!)}](\![\mathsf{t_m}]\!) \approx ([\overrightarrow{\mathsf{y}} \mapsto \overrightarrow{\mathsf{w}}]\mathsf{r})$. ◀

We now show our main result.

▶ **Theorem 16.** *Let* t_1 *be a closed and typed term of* SJ&+*. If*

$$\mathsf{t}_1 \longrightarrow_S \mathsf{t}_2 \longrightarrow_S \ldots \mathsf{t}_i \longrightarrow_S \ldots$$

is a finite or infinite reduction sequence in SJ&+*, then*

$$(\![\mathsf{t}_1]\!) \longrightarrow_T^* \mathsf{t}'_2 \longrightarrow_T^* \ldots \mathsf{t}'_i \longrightarrow_T^* \ldots$$

where $(\![\mathsf{t}_i]\!) \approx \mathsf{t}'_i$*,* $i > 1$*, is a finite or infinite reduction sequence in* TJ&*.*

Proof. If $i > 1$, then from $\mathsf{t}_i \longrightarrow_S \mathsf{t}_{i+1}$ and $(\![\mathsf{t}_i]\!) \approx \mathsf{t}'_i$ we get $\mathsf{t}'_i \longrightarrow_T^* \mathsf{t}'_{i+1}$ and $(\![\mathsf{t}_{i+1}]\!) \approx \mathsf{t}'_{i+1}$ by Lemma 15. For $i = 1$ we can take $\mathsf{t}'_1 = (\![\mathsf{t}_1]\!)$ since $\approx$ is reflexive. ◀

We end this section by providing the relation between values obtained by reducing a closed and typed source term and its translation. The proof is an easy consequence of previous theorem.

▶ **Corollary 17.** *Let* t *be a closed and typed term of* SJ&+*. If* $\mathsf{t} \longrightarrow_S^* \mathsf{w}$*, then* $(\![\mathsf{t}]\!) \longrightarrow_T^* \mathsf{w}'$ *with* $(\![\mathsf{w}]\!) \approx \mathsf{w}'$*.*

This corollary assures that the behaviour of a translated program in TJ& with value w' exactly reflects the behaviour of the original program in SJ&+ with value w. A simple example is $\mathsf{w} = \mathsf{new}\,\mathsf{C}(\mathsf{x} \to \mathsf{x})$, where the field of C has type $\mathsf{I}\&\mathsf{J}$ and the abstract method of I maps objects of class D to objects of class D. Then $(\![\mathsf{w}]\!) = \mathsf{new}\,\mathsf{C}((\mathsf{I}\&\mathsf{J})\,(\mathsf{x} \to (\mathsf{D})\,\mathsf{x}))$ and $\mathsf{w}' = \mathsf{new}\,\mathsf{C}((\mathsf{x} \to (\mathsf{D})\,\mathsf{x})^{\mathsf{I}\&\mathsf{J}})$. The translation only adds in w redundant type information (with respect to the class table CT^S) in the form of upcasts on subterms. Then, loosely speaking, we can say that w and $(\![\mathsf{w}]\!)$ are contextually equivalent in SJ&+, since their occurrences in a complete program can be interchanged without affecting the result of executing the program in a significant way. Differently, when considering $(\![\mathsf{w}]\!)$ in TJ&, all those type casts which are added by translation are necessary for typing and preserve types. Corollary 17 says that w' is cast-equivalent to $(\![\mathsf{w}]\!)$, which is contextually equivalent to w in SJ&+.

7 Related Work and Conclusion

We have presented a compilation of programs of an extension of Java, where intersection types are completely deconfined from the present boundary within type casts, to the unextended language. Thus the proposed Java extension is guaranteed to be fully backward compatible with the current Java, namely its code can be compiled into JVM.

Formally, we exploited two calculi, TJ& and SJ&+. Since the keystone paper [15], Featherweight Java (FJ) has become the standard calculus for formalising extensions and variants of Java, some of them are listed in the references of [2]. Not surprisingly the two calculi discussed in the present paper are extensions of FJ. Furthermore, they follow the style of FJ, in omitting all the features that do not interact with our issue in a significant way.

TJ& is a core model of Java 8, focusing on the two main novelties: λ-expressions and intersection types in type casts. SJ&+ comes from FJP&λ, presented in [10], the first formal account proposing to extend Java with λ-expressions by the capability of using intersection types as parameter types and return types of methods, as well as field types.

The main contribution of the present paper is a translation of typed programs of SJ&+ into typed programs of TJ&. The translation basically consists in erasing intersection types in method signatures and field types and, consistently, adding type downcasts in terms where needed. Checking these downcasts at run time will always succeed. Namely, we proved that our translation preserves typing and semantics of the source programs.

Intersection types as parameter types and return types of methods were first proposed in [5], which contains interesting examples of structuring and reusing code. Our main inspiration in stating the properties of the translation has been [15], where the safety of GJ (Featherweight Java with generic classes [4]) is shown by compiling GJ into FJ. Also in [15] the translation only adds downcasts, called "synthetic casts", which are proved not to fail at run time.

In [10] we also proposed to overcome the limitation of exactly one abstract method in functional interfaces. This naturally agrees with the standard meaning of intersection types: intersection types express multiple, possibly unrelated, properties of terms. It magnifies also the polyadic nature of λ-expressions, which can match multiple headers of abstract methods, with different signatures.

For future works, we plan to compile functional interfaces with many abstract methods into functional interfaces with exactly one abstract method. The inspiration will be the formulation of intersection types à la Church, i.e. with types decorating terms [17, 3, 11, 12].

Moreover, the relation between traits in Scala [19] (Chapter 12) and Java interfaces with default methods is clearly worth to be deeply investigated. As a further development of the present paper, we would like to study how to implement a form of traits in Java, by exploiting intersection types. The ability of expressing combinations of traits, as intersections types, also in requirements, that is in parameter and return types of methods, seems to be the key tool for using traits as a valuable design concept (see the discussion in [25]).

References

1 Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A Filter Lambda Model and the Completeness of Type Assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983. doi:10.2307/2273659.

2 Lorenzo Bettini, Viviana Bono, Mariangiola Dezani-Ciancaglini, Paola Giannini, and Betti Venneri. Java & Lambda: a Featherweight Story. *Logical Methods in Computer Science*, 14(3):1–24, 2018. doi:10.23638/LMCS-14(3:17)2018.

3 Viviana Bono, Betti Venneri, and Lorenzo Bettini. A Typed Lambda Calculus with Intersection Types. *Theoretical Computer Science*, 398(1-3):95–113, 2008. doi:10.1016/j.tcs.2008.01.046.

4 Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In Bjørn N. Freeman-Benson and Craig Chambers, editors, *OOPSLA*, pages 183–200. ACM Press, 1998. doi:10.1145/286936.286957.

5 Martin Büchi and Wolfgang Weck. Compound Types for Java. In Bjørn N. Freeman-Benson and Craig Chambers, editors, *OOPSLA*, pages 362–373. ACM Press, 1998. doi:10.1145/286936.286975.

6 Mario Coppo and Mariangiola Dezani-Ciancaglini. An Extension of the Basic Functionality Theory for the λ-calculus. *Notre Dame Journal of Formal Logic*, 21(4):685–693, 1980. doi:10.1305/ndjfl/1093883253.

7 Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Principal Type Schemes and λ-calculus Semantics. In *To H.B.Curry: Essays on Combinatory Logic, Lambda-calculus and Formalism*, pages 535–560. Academic Press, 1980. ISBN-13: 9780123490506.

8 Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Functional Characters of Solvable Terms. *Mathematical Logical Quartely*, 27(2-6):45–58, 1981. doi:10.1002/malq.19810270205.

9 Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A Core Calculus for Scala Type Checking. In Rastislav Kralovic and Pawel Urzyczyn, editors, *MFCS*, volume 4162 of *LNCS*, pages 1–23, Berlin, 2006. Springer. doi:10.1007/11821069_1.

10 Mariangiola Dezani-Ciancaglini, Paola Giannini, and Betti Venneri. Intersection Types in Java: Back to the Future. In Tiziana Margaria, Susanne Graf, and Kim G. Larsen, editors, *Models, Mindsets, Meta: The What, the How, and the Why Not?*, volume 11200 of *LNCS*, pages 68–86. Springer, 2018. doi:10.1007/978-3-030-22348-9_6.

11 Andrej Dudenhefner and Jakob Rehof. Intersection Type Calculi of Bounded Dimension. In Giuseppe Castagna and Andrew D. Gordon, editors, *POPL*, pages 653–665. ACM Press, 2017. doi:10.15520/jbme.v6i7.2243.

12 Andrej Dudenhefner and Jakob Rehof. Typability in Bounded Dimension. In Joel Ouaknine, editor, *LICS*, pages 1–12. IEEE Computer Society, 2017. doi:10.1109/LICS.2017.8005127.

13 Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN-13: 0201633612.

14 James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Oracle, 2015. ISBN-13: 9780133900699.

15 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001. doi:10.1145/503502.503505.

16 Java 10, 2018. URL: https://docs.oracle.com/javase/10/language/toc.htm#JSLAN-GUID-7D5FDD65-ACE4-4B3C-80F4-CC01CBD211A4.

17 Luigi Liquori and Simona Ronchi Della Rocca. Intersection-types à la Church. *Information and Computation*, 205(9):1371–1386, 2007. doi:10.1016/j.ic.2007.03.005.

18 Robert C. Martin. *Agile Software Development: Principles, Patterns and Practices*. Pearson Education, 2002. ISBN: 0-13-597444-5.

19 Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: Updated for Scala 2.12*. Artima Incorporation, 3rd edition, 2016. ISBN-13: 9780981531687.

20 Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. ISBN: 0262162091.

21 Garrel Pottinger. A Type Assignment for the Strongly Normalizable λ-terms. In *To H.B.Curry: Essays on Combinatory Logic, Lambda-calculus and Formalism*, pages 561–578. Academic Press, 1980. ISBN-13: 9780123490506.

22 Marianna Rapoport, Ifaz Kabir, Paul He, and Ondrej Lhoták. A Simple Soundness Proof for Dependent Object Types. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):46:1–46:27, 2017. doi:10.1145/3133870.
23 John C. Reynolds. Conjunctive Types and Algol-like Languages. In David Gries, editor, *LICS*, page 119. IEEE Computer Society, 1987.
24 John C. Reynolds. Design of the Programming Language Forsythe. In *Algol-like Languages, Progress in Theoretical Computer Science*, pages 173–233. Birkhäuser, 1997. ISBN-978-1-4612-8661-5.
25 Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable Units of Behaviour. In Luca Cardelli, editor, *ECOOP*, volume 2743 of *LNCS*, pages 248–274. Springer, 2003. doi:10.1007/978-3-540-45070-2_12.

Towards a Unifying Framework for Tuning Analysis Precision by Program Transformation

Mila Dalla Preda
Department of Computer Science, University of Verona, Italy
mila.dallapreda@univr.it

Abstract

Static and dynamic program analyses attempt to extract useful information on program's behaviours. Static analysis uses an abstract model of programs to reason on their runtime behaviour without actually running them, while dynamic analysis reasons on a test set of real program executions. For this reason, the precision of static analysis is limited by the presence of false positives (executions allowed by the abstract model that cannot happen at runtime), while the precision of dynamic analysis is limited by the presence of false negatives (real executions that are not in the test set). Researchers have developed many analysis techniques and tools in the attempt to increase the precision of program verification. Software protection is an interesting scenario where programs need to be protected from adversaries that use program analysis to understand their inner working and then exploit this knowledge to perform some illicit actions. Program analysis plays a dual role in program verification and software protection: in program verification we want the analysis to be as precise as possible, while in software protection we want to degrade the results of the analysis as much as possible. Indeed, in software protection researchers usually recur to a special class of program transformations, called code obfuscation, to modify a program in order to make it more difficult to analyse while preserving its intended functionality. In this setting, it is interesting to study how program transformations that preserve the intended behaviour of programs can affect the precision of both static and dynamic analysis. While some works have been done in order to formalise the efficiency of code obfuscation in degrading static analysis and in the possibility of transforming programs in order to avoid or increase false positives, less attention has been posed to formalise the relation between program transformations and false negatives in dynamic analysis. In this work we are setting the scene for a formal investigation of the syntactic and semantic program features that affect the presence of false negatives in dynamic analysis. We believe that this understanding would be useful for improving the precision of the existing dynamic analysis tools and in the design of program transformations that complicate the dynamic analysis.

To Maurizio on his 60th birthday!

2012 ACM Subject Classification Security and privacy → Software reverse engineering

Keywords and phrases Program analysis, analysis precision, program transformation, software protection, code obfuscation

Digital Object Identifier 10.4230/OASIcs.Gabbrielli.2020.4

Funding The research has been partially supported by the project "Dipartimenti di Eccellenza 2018–2020" funded by the Italian Ministry of Education, Universities and Research (MIUR).

1 Introduction

Program analysis refers, in general, to any examination of programs that attempts to extract useful information on program's behaviours (semantics). As known from the Rice theorem, all nontrivial extensional properties of program's semantics are undecidable in the general case. This means that any automated reasoning on software has to involve some kind of approximation. Programs can be analysed either statically or dynamically. Static program analysis reasons about the behaviour of programs without actually running them. Typically,

© Mila Dalla Preda;
licensed under Creative Commons License CC-BY
Recent Developments in the Design and Implementation of Programming Languages.
Editors: Frank S. de Boer and Jacopo Mauro; Article No. 4; pp. 4:1–4:22

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

static analysis builds an abstract model that over-approximates the possible program's behaviours to examine program properties. This guarantees soundness: what can be derived from the analysis of the abstract model holds also on the concrete execution of the program. The converse does not hold in general due to the presence of *false positives*: spurious behaviours allowed by the abstract model that do not correspond to any real program execution. Static analysis has proved its usefulness in many fields of computer science like in optimising compilers for producing efficient code, for automatic error detection and for the automatic verification of desired program properties (e.g., functional properties and security properties) [21]. Many different static analysis approaches exists, as for example model checking [7], deductive verification [33] and abstract interpretation [12]. In particular, abstract interpretation provides a formal framework for reasoning on behavioural program properties where many static analysis techniques can be formalised. In the rest of this paper we focus on those static analyses that can be formalised in the abstract interpretation framework. Dynamic program analyses, such as program testing [1], runtime monitoring and verification [4], consider an under-approximation of program behaviour as they focus their analysis on a specific subset of possible program executions. In this paper when we speak of dynamic analysis we mainly refer to program testing. Testing techniques start by concretely executing programs on an input set and the so obtained test set of concrete executions is inspected in order to reason on program's behaviour (e.g., reveal failures or vulnerabilities). It is well known that dynamic analysis can precisely detect the presence of failures but cannot guarantee their absence, due to the presence of *false negatives*: concrete program behaviours that do not belong to the test set. There is a famous quote by Dijksta that states that "Program testing can be used to show the presence of bugs, but never to show their absence!". Since it is not possible to guarantee the absence of failures we have to accept the fact that whenever we use software we incur in some risk. Software testing is widely used to reveal possible software failures, to reduce the risk related to the use of software and to increase the quality of software by deciding if the behaviour of software is acceptable in terms of reliability, safety, maintainability, security, and efficiency [1].

Static analysis computes an over-approximation of the program semantics, while dynamic analysis under-approximates program semantics. In both cases, we have a decidable evaluation of the semantic property of interest on an approximation of the program semantics. For this reason what we can automatically conclude regarding the behavioural properties of programs has to take into account false positives for static analysis and false negatives for dynamic analysis. Static analysis is precise when it is *complete* (no false positives) and this relates to the well studied notion of completeness in abstract interpretation [12, 14, 23]. The intuition is that static analysis is complete when the details lost by the abstract model are not relevant for reasoning on the semantic property of interest. Dynamic analysis is precise when it is *sound* (no false negatives) and this happens when the executions in the test set exhibit all the behaviours of the program that are relevant with respect to the semantic property of interest. This means that the under-approximation of the program semantics considered by the dynamic analysis allows us to precisely observe the behavioural property of interest. The essential problem with dynamic analysis is that it is impossible to test with all inputs since the input space is generally infinite. In this context, *coverage criteria* provide structured, practical ways to search the input space and to decide which input set to use. The rationale behind coverage criteria is to partition the input space in order to maximise the executions present in the tests set that are relevant for the analysis of the semantic property of interest. Coverage criteria are useful in supporting the automatic generation of input sets and in providing useful rules for deciding when to terminate the generation of the test set [1].

Program analysis has been originally developed for verifying the correctness of programs and researchers have put a great deal of effort in developing efficient and precise analysis techniques and tools that try to reduce false positives and false negatives as much as possible. Indeed, analysis precision relates to the ability of identifying failures and vulnerabilities that may lead to unexpected behaviours, or that may be exploited by an adversary for malicious purposes. For this reason the main goal of researchers has been to improve the precision and efficiency of both static and dynamic analysis tools.

Software protection is another interesting scenario where program analysis plays a central role but in a dual way. Today, software and the assets embedded in it are constantly under attack. This is particularly critical for those software applications that run in an untrusted environment in a scenario known as MATE (Man-At-The-End) attacks. In this setting, attackers have full control over, and white-box access to, the software and the systems on which the software is running. Attackers can use a wide range of analysis tools such as disassemblers, code browsers, debuggers, emulators, instrumentation tools, fuzzers, symbolic execution engines, customised OS features, pattern matchers, etc. to inspect, analyse and alter software and its assets. In such scenarios, software protection becomes increasingly important to protect the assets, even against MATE attacks. For industry, in many cases the deployment of software-based defense techniques is crucial for the survival of their businesses and eco-systems. In the software protection scenario, program analysis can be used by adversaries to reverse engineer proprietary code and then illicitly reuse portions of the code or tamper with the code in some unauthorised way. Here, in order to protect the intellectual property and integrity of programs we have to force the analysis to be imprecise or so expensive to make it impractical for the adversary to mount an attack.

To address this problem, researchers have developed software-based defense techniques, called *code obfuscations*, that transform programs with the explicit intent of complicating and degrading program analysis [9]. The idea of code obfuscation techniques is to transform a program into a functionally equivalent one that is more difficult (ideally impossible) for an analyst to understand. As well as for program analysis also for code obfuscation we have an important negative result from Barak et al. [3] that proves the impossibility of code obfuscation. Note that, this result states the impossibility of an ideal obfuscator that obfuscates every program by revealing only the properties that can be derived from its I/O semantics. Besides the negative result of Barak et al., in recent decades, we have seen a big effort in developing and implementing new and efficient obfuscation strategies [8]. Of course, these obfuscating techniques introduce a kind of practical obfuscators weakening the ideal obfuscator of Barak et al. in different ways, and which can be effectively used in real application protection in the market. For example, these obfuscators may work only for a certain class of programs, or may be able to hide only certain properties of programs (e.g., control flow). Indeed, the attention on code obfuscation poses the need to deeply understand what we can obfuscate, namely which kind of program properties we can hide by inducing imprecision in their automatic analysis.

A recent survey on the existing code obfuscation techniques shows the efficiency of code obfuscation in degrading the results of static analysis, while existing code obfuscation techniques turn out to be less effective against dynamic analysis [31]. Consider, for example, the well known control flow obfuscation based on the insertion of opaque predicates. An opaque predicate is a predicate whose constant value is known to the obfuscation, while it is difficult for the analyst to recognise such constant value [9]. Consider the program whose control flow graph is depicted on the left of Figure 1 where we have three blocks of sequential instructions A, B and C executed in the order specified by the arrows $A \to B \to C$. Let

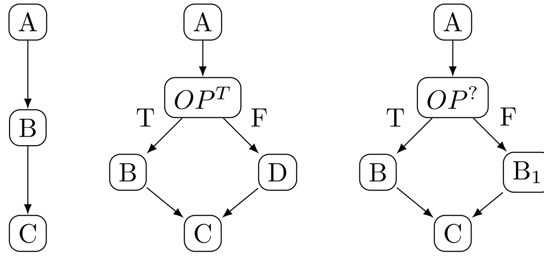


Figure 1 Code obfuscation.

OP^T denote a true opaque predicate, namely a predicate that always evaluates to *true*. In the middle of Figure 1 we can see what happens to the control flow graph when we insert a true opaque predicate: block D has to be considered in the static analysis of the control flow even if it is never executed at runtime. Thus, $A \to OP^T \to D \to C$ is a false positive path added by the obfuscating transformation to the static analysis, while no imprecision is added to dynamic analysis since all real executions follow the path $A \to OP^T \to B \to C$. On the right of Figure 1 we have the control flow graph of the program obtained inserting an unknown opaque predicate. An unknown opaque predicate $OP^?$ is a predicate that sometimes evaluates to *true* and sometimes evaluates to *false*. These predicates are used to diversify program execution by inserting in the true and false branches sequences of instructions that are syntactically different but functionally equivalent (e.g. blocks B and B_1) [9]. Observe that this transformation adds confusion to dynamic analysis: a dynamic analyser has now to consider more execution traces in order to cover all the paths of the control flow graph. Indeed, if the dynamic analysis observes only traces that follow the original path $A \to OP^? \to B \to C$ it may not be sound as it misses the traces that follow $A \to OP^? \to B_1 \to C$ (false negative).

The abstract interpretation framework has been used to formalise, prove and compare the efficiency of code obfuscation techniques in confusing static analysis [17, 25] and to derive strategies for the design of obfuscating techniques that hamper a specific analysis [19]. The general idea is that code obfuscation confuses static analysis by exploiting its conservative nature, and by modifying programs in order to increase its imprecision (adding false positives) while preserving the program intended behaviour. Observe that, in general, the imprecision added by these obfuscating transformations to confuse a static analyser is not able to confuse a dynamic attacker that cannot be deceived by false positives. This is the reason why common deobfuscation approaches often recur to dynamic analysis to reverse engineer obfuscated code [5, 10, 32, 34].

It is clear that to complicate dynamic analysis we need to develop obfuscation techniques that exploit the Achilles heel of dynamic analysis, namely false negatives. In the literature, there are some defense techniques that focus on hampering dynamic analysis [2, 27, 28, 30]. What is still missing is a general framework where it is possible to formalise, prove and discuss the efficiency of these transformations in complicating dynamic analysis in terms of the imprecision (false negatives) that they introduce. As discussed above the main challenge for dynamic analysis is the identification of a suitable input set for testing program's behaviour. In order to automatically build a suitable input set, the analysts either design an input generation tool or an input recogniser tool. In both cases, they need a coverage criterion that defines the inputs to be considered and when to terminate the definition of the input set. Ideally, the coverage criterion is chosen in order to guarantee that the test set precisely reveals the semantic property under analysis (no false negatives). However, to the best of

our knowledge, there is no formal guarantee that a coverage criterion ensures the absence of false negatives with respect to a certain analysis. If hampering static analysis means to increase the presence of false positives, hampering dynamic analysis means to complicate the automatic construction of a suitable input set for a given coverage criterion. In order to formally reason on the effects that code obfuscation has on the precision of dynamic analysis it is important to develop a general framework, analogous to the one based on program semantics and abstract interpretation that formalises the relation between dynamic analysis and code obfuscation. Thus, we need to develop a framework where we can (1) formally specify the relation between the coverage criterion used and the semantic property that we are testing, (2) define when a program transformation complicates the construction of an input set that has to satisfy a given coverage criterion, (3) derive guidelines for the design of obfuscating transformations that hamper the dynamic analysis of a given program property. This formal investigation will allow us to better understand the potential and limits of code obfuscation against dynamic program analysis.

In the following we provide a unifying view of static and dynamic program analysis and of the approaches that researchers use to tune the precision of these analysis. From this unifying overview it turns out that while the relation between the precision of static program analysis and program transformations has been widely studied, both in the software verification and in the software protection scenario, less attention has been posed to the formal investigation of the effects that code transformations have on the precision of program testing. We start to face this problem by showing how it is possible to formally compare and relate coverage criterion, semantic property under testing and false negatives for a specific class of program properties. This discussion leads us to the identification of important and interesting new research directions that would lead to the development of the above mentioned formal framework for reasoning about the effects of program transformations on the precision of dynamic analysis. We believe that this formal reasoning would find interesting applications both in the software verification and in the software protection scenario.

Structure of the paper: In Section 2 we provide some basic notions. In Section 3 we discuss possible techniques for improving the precision of the analysis: Section 3.1 revise the existing and ongoing work in transforming properties and programs toward completeness of static analysis, while Section 3.2 provides the basis for a formal framework for reasoning on possible property and program transformations to achieve soundness in dynamic analysis, these are preliminary results some of which have been recently published in [18]. Section 4 shows how the techniques used to improve analysis precision could be used in the software protection scenario to prove the efficiency of software protection techniques. The use of this formal reasoning for proving the efficiency of software protection techniques against static analysis is known, while it is novel for dynamic analysis. The paper ends with a discussion on the open research challenges that follow from this work.

2 Preliminaries

Given two sets S and T, we denote with $\wp(S)$ the powerset of S, with $S \times T$ the Cartesian product of S and T, with $S \subset T$ strict inclusion, with $S \subseteq T$ inclusion, with $S \subseteq_F T$ the fact that S is a finite subset of T. $\langle C, \leq_C, \vee_C, \wedge_C, \top_C, \bot_C \rangle$ denotes a complete lattice on the set C, with ordering $\leq_C$, least upper bound (*lub*) $\vee_C$, greatest lower bound (*glb*) $\wedge_C$, greatest element (top) $\top_C$, and least element (bottom) $\bot_C$ (the subscript $_C$ is omitted when the domain is clear from the context). Let C and D be complete lattices. Then, $C \xrightarrow{\text{m}} D$ and $C \xrightarrow{\text{c}} D$ denote, respectively, the set and the type of all monotone and (Scott-)continuous

functions from C to D. Recall that $f \in C \xrightarrow{c} D$ if and only if f preserves *lub*'s of (nonempty) chains if and only if f preserves *lub*'s of directed subsets. Let $f : C \to C$ be a function on a complete lattice C, we denote with $lfp(f)$ the least fix-point, when it exists, of function f on C. The well-known Knaster-Tarski's theorem states that any monotone operator $f : C \xrightarrow{m} C$ on a complete lattice C admits a least fix point. It is known that if $f : C \xrightarrow{c} C$ is continuous then $lfp(f) = \vee_{i \in \mathbb{N}} f^i(\bot_C)$, where, for any $i \in \mathbb{N}$ and $x \in C$, the i-th power of f in x is inductively defined as follows: $f^0(x) = x$; $f^{i+1}(x) = f(f^i(x))$.

Program Semantics: Let us consider the set *Prog* of possible programs and the set Σ of possible program states. A program state $s \in \Sigma$ provides a snapshot of the program and memory content during the execution of the program. Given a program P we denote $Init_P$ the set of its initial states. We use Σ^* to denote the set of all finite and infinite sequences or traces of states ranged over by σ. Given a trace $\sigma \in \Sigma^*$ we denote with $\sigma_0 \in \Sigma$ the first element of sequence σ and with σ_f the final state of σ if σ is finite. Let $\tau \subseteq \Sigma \times \Sigma$ denote the transition relation between program states, thus $(s, s') \in \tau$ means that state s' can be obtained from state s in one computational step. The *trace semantics* of a program P is defined, as usual, as the least fix-point computation of function $\mathcal{F}_P : \wp(\Sigma^*) \to \wp(\Sigma^*)$ [11]:

$$\mathcal{F}_P(X) \stackrel{\text{def}}{=} Init_P \cup \left\{ \sigma s_i s_{i+1} \;\middle|\; (s_i, s_{i+1}) \in \tau, \sigma s_i \in X \right\}$$

The trace semantics of P is $[\![P]\!] \stackrel{\text{def}}{=} lfp(\mathcal{F}_P) = \bigcup_{i \in \mathbb{N}} \mathcal{F}_P^i(\bot_C)$. $Den[\![P]\!]$ denotes the denotational (finite) semantics of program P which abstracts away the history of the computation by observing only the input-output relation of finite traces: $Den[\![P]\!] \stackrel{\text{def}}{=} \{\sigma \in \Sigma^+ \mid \exists \eta \in [\![P]\!] : \eta_0 = \sigma_0, \eta_f = \sigma_f\}$.

Concrete domains are collections of computational objects where the concrete semantics is computed, while abstract domains are collections of approximate objects, representing properties of concrete objects in a domain-like structure. It is possible to interpret the semantics of programs on abstract domains thus approximating the computation with respect to the property expressed by the abstract domain. The relation between concrete and abstract domains can be equivalently specified in terms of Galois connections (GC) or upper closure operators in the abstract interpretation framework [12, 13]. The two approaches are equivalent, modulo isomorphic representations of the domain object. A GC is a tuple (C, α, γ, A) where C is the concrete domain, A is the abstract domain and $\alpha : C \to A$ and $\gamma : A \to C$ are respectively the abstraction and concretisation maps that give rise to an adjunction: $\forall a \in A, c \in C : \alpha(c) \leq_A a \Leftrightarrow c \leq_C \gamma(a)$. Abstract domains can be compared with respect to their relative degree of precision: if A_1 and A_2 are abstractions of a common concrete domain C, A_1 is more precise than A_2, denoted $A_1 \sqsubseteq A_2$ when $\forall a_2 \in A_2, \exists a_1 \in A_1 : \gamma_1(a_1) = \gamma_2(a_2)$, namely if $\gamma_2(A) \subseteq \gamma_1(A)$. An upper closure operator on a complete lattice C is an operator $\rho : C \to C$ that is monotone, idempotent, and extensive ($\forall x \in C : x \leq_C \rho(x)$). Closures are uniquely determined by their fix-points $\rho(C)$. If (C, α, γ, A) is a GC then $\rho = \gamma \circ \alpha$ is the closure associated to A, such that $\rho(C)$ is a complete lattice isomorphic to A. The closure $\gamma \circ \alpha$ associated to the abstract domain A can be thought of as the logical meaning of A in C, since this is shared by any other abstract representation for the objects of A. Thus, the closure operator approach is convenient when reasoning about properties of abstract domains independently from the representation of their objects. We denote with $uco(C)$ the set of upper closure operators over C. If C is a complete lattice then $uco(C)$ is a complete lattice where closure are ordered with respect to their relative precision $\rho_1 \sqsubseteq \rho_2 \Leftrightarrow \rho_2(C) \subseteq \rho_1(C)$ which corresponds to the ordering of abstract domains.

The abstract semantics of a program P on the abstract domain $\rho \in uco(\wp(\Sigma^*))$, denoted as $[\![P]\!]^\rho$, is defined as the fix-point computation of function $\mathcal{F}_P^\rho : \rho(\wp(\Sigma^*)) \to \rho(\wp(\Sigma^*))$ where $\mathcal{F}_P^\rho \stackrel{\text{def}}{=} \rho \circ \mathcal{F}_P \circ \rho$ is the best correct approximation of function $\mathcal{F}_P$ on the abstract domain

$\rho(\wp(\Sigma^*))$, namely $[\![P]\!]^\rho \stackrel{\text{def}}{=} lfp(\mathcal{F}^\rho_P) = \bigcup_{i\in\mathbb{N}} \mathcal{F}^\rho_P(\bot_{\rho(C)})$. Given the equivalence between GC and closures, the abstract semantics can be equivalently specified in terms of abstract traces in the corresponding abstract domain and in the following we denote the abstract semantics either with $[\![P]\!]^\rho$ or with $[\![P]\!]^A$ where (C, α, γ, A) is a GC and $\rho = \gamma \circ \alpha$.

Equivalence Relations: Let $\mathcal{R}$ be a binary relation $\mathcal{R} \subseteq C \times C$ on a set C, given $x, y \in C$ we denote with $(x, y) \in \mathcal{R}$ the fact that x is in relation $\mathcal{R}$ with y. $\mathcal{R} \subseteq C \times C$, is an *equivalence relation* if $\mathcal{R}$ is reflexive $\forall x \in C : (x, x) \in \mathcal{R}$, symmetric $\forall x, y \in C : (x, y) \in \mathcal{R} \Rightarrow (y, x) \in \mathcal{R}$ and transitive $\forall x, y, z \in C : (x, y) \in \mathcal{R} \wedge (y, z) \in \mathcal{R} \Rightarrow (x, z) \in \mathcal{R}$. Given a set C equipped with an equivalence relation $\mathcal{R}$, we consider for each element $x \in C$ the subset $[x]_\mathcal{R}$ of C containing all the elements of C in equivalence relation with x, i.e., $[x]_\mathcal{R} = \{y \in C \mid (x, y) \in \mathcal{R}\}$. The sets $[x]_\mathcal{R}$ are called equivalence classes of C wrt relation $\mathcal{R}$ and they induce a partition of the set C, namely $\forall x, y \in C : [x]_\mathcal{R} = [y]_\mathcal{R} \vee [x]_\mathcal{R} \cap [y]_\mathcal{R} = \emptyset$ and $\cup\{[x]_\mathcal{R} \mid x \in C\} = C$. The partition of C induced by the relation $\mathcal{R}$ is denoted by $C/_\mathcal{R}$. Let $Eq(C)$ be the set of equivalence relations on the set C. The set of equivalence relations on C form a lattice $\langle Eq(C), \preceq, \sqcap_{Eq}, \sqcup_{Eq}, id, top\rangle$ where id is the relation that distinguishes all the elements in C, top is the relation that cannot distinguish any element in C, and: $\mathcal{R}_1 \preceq \mathcal{R}_2$ iff $\mathcal{R}_1 \subseteq \mathcal{R}_2$ iff $(x, y) \in \mathcal{R}_1 \Rightarrow (x, y) \in \mathcal{R}_2$, $\mathcal{R}_1 \sqcap_{Eq} \mathcal{R}_2 = \mathcal{R}_1 \cap \mathcal{R}_2$, namely $(x, y) \in \mathcal{R}_1 \sqcap_{Eq} \mathcal{R}_2$ iff $(x, y) \in \mathcal{R}_1 \wedge (x, y) \in \mathcal{R}_2$; $\mathcal{R}_1 \sqcup_{Eq} \mathcal{R}_2$ it is such that $(x, y) \in \mathcal{R}_1 \sqcup_{Eq} \mathcal{R}_2$ iff $(x, y) \in \mathcal{R}_1 \vee (x, y) \in \mathcal{R}_2$. When $\mathcal{R}_1 \preceq \mathcal{R}_2$ we say that $\mathcal{R}_1$ is a refinement of $\mathcal{R}_2$. Given a subset $S \subseteq C$, we denote with $\mathcal{R}_{|_S} \in Eq(S)$ the restriction of relation $\mathcal{R}$ to the domain S.

The relation between closure operators and equivalence relations has been studied in [29]. Each closure operator $\rho \in uco(\wp(C))$ induces an equivalence relation $\mathcal{R}^\rho \in Eq(C)$ where $(x, y) \in \mathcal{R}^\rho y$ iff $\rho(\{x\}) = \rho(\{y\})$ and viceversa, each equivalence relation $\mathcal{R} \in Eq(C)$ induces a closure operator $\rho^\mathcal{R} \in uco(\wp(C))$ where $\rho^\mathcal{R}(\{x\}) = [x]_\mathcal{R}$ and $\rho^\mathcal{R}(X) = \bigcup_{x\in X}[x]_\mathcal{R}$. Of course, there are many closures that induce the same partition on traces and these closures carry additional information other than the underlying state partition, and this additional information that allows us to distinguish them is lost when looking at the induced partition. Indeed, it holds that given $\mathcal{R} \in Eq(C)$ the corresponding closure is such that $\rho^\mathcal{R} = \sqcap\{\rho \mid \mathcal{R}^\rho = \mathcal{R}\}$. The closures in $uco(\wp(C))$ defined form a partition $\mathcal{R} \in Eq(C)$ are called *partitioning* and they identify a subset of $uco(\wp(C))$: $\{\rho^\mathcal{R} \in uco(\wp(C)) \mid \mathcal{R} \in Eq(C)\} \subseteq uco(\wp(C))$ [29].

3 On the precision of program analysis

As argued above program analysis has been originally developed for program verification, namely to ensure that programs will actually behave as expected. Besides the impossibility result of the Rice theorem, a multitude of analysis strategies have been proposed [21]. Indeed, by tuning the precision of the behavioural feature that we want to analyse it is possible to derive an analysable semantic property that, while loosing some details of program's behaviour, may still be of practical interest [12, 14]. We are interested in semantic program properties, namely in properties that deal with the behaviour of programs, but the possibility of precisely analysing such properties depends also on the way in which programs are written. This means that there are programs that are easier to analyse than others with respect to a certain property [6]. Thus, program transformations that preserve the program's intended functionality can affect the precision of the results of the same analysis on the original and transformed program.

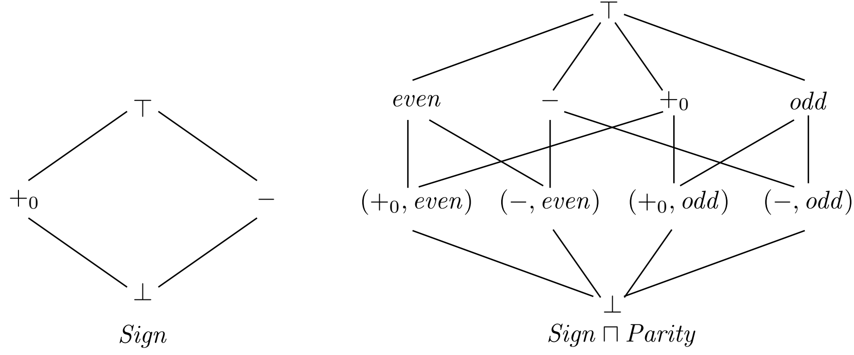


Figure 2 Abstract domain of Sign and Sign ⊓ Parity.

3.1 Static Analysis

Precision in static program analysis means completeness, namely absence of false positives. This means that the noise introduced by the abstract model used for static program analysis does not introduce imprecision with respect to the property under analysis. Consider for example program P on the left of Figure 3 that, given an integer value a, returns its absolute value and it does it by adding some extra controls on the parity of variable a that have no effect on the result of computation[1]. The semantics of program P is:

$$\begin{aligned}\llbracket P \rrbracket &= \{\langle B_1 : \bot\rangle\langle B_2 : v_1\rangle\langle B_4 : 2 * v_1 + 1\rangle\langle B_6 : 2 * v_1 + 1\rangle\langle B_7 : v_1\rangle \mid v_1 \geq 0\} \cup \\ &\quad \{\langle B_1 : \bot\rangle\langle B_3 : v_1\rangle\langle B_4 : 2 * v_1\rangle\langle B_5 : 2 * v_1\rangle\langle B_7 : -v_1\rangle \mid v_1 < 0\}\end{aligned}$$

where $\langle B_i, val\rangle$ denotes the program state specifying the value val of variable a when entering block B_i and $\bot$ denotes the undefined value. Assume that we are interested in the analysis on the abstract domain *Sign* depicted on the left of Figure 2. The $Sign = \{\bot, +_0, -, \top\}$ abstract domain observes the sign of integer values and it is possible to define a GC between $\wp(\mathbb{Z})$ and *Sign* where the abstract element $+_0$ represents all positive values plus 0, the abstract element $-$ represents all negative values, while $\top$ represents all integer values and $\bot$ the emptyset. We denote with $\llbracket P \rrbracket^{Sign} \in \wp(\Sigma^*)$ the abstract interpretation of program P on the domain of *Sign*, where the values of variable a are interpreted on *Sign*.

$$\begin{aligned}\llbracket P \rrbracket^{Sign} &= \{\langle B_1 : \bot\rangle\langle B_2 : +_0\rangle\langle B_4 : +_0\rangle\langle B_6 : +_0\rangle\langle B_7 : +_0\rangle, \\ &\quad \langle B_1 : \bot\rangle\langle B_2 : +_0\rangle\langle B_4 : +_0\rangle\langle B_5 : +_0\rangle\langle B_7, -\rangle[\textit{false positive}] \\ &\quad \langle B_1 : \bot\rangle\langle B_3 : -\rangle\langle B_4 : -\rangle\langle B_5 : -\rangle, \langle B_7, +_0\rangle \\ &\quad \langle B_1 : \bot\rangle\langle B_3 : -\rangle\langle B_4 : -\rangle\langle B_6 : -\rangle\langle B_7, -\rangle[\textit{false positive}]\}\end{aligned}$$

Each abstract trace corresponds to infinitely many concrete traces. So for example the abstract trace $\langle B_1 : \bot\rangle\langle B_2 : +_0\rangle\langle B_4 : +_0\rangle\langle B_6 : +_0\rangle\langle B_7 : +_0\rangle$ corresponds to the infinte set of concrete traces: $\{\langle B_1 : \bot\rangle\langle B_2 : v_1\rangle\langle B_4 : v_2\rangle\langle B_6 : v_3\rangle\langle B_7 : v_4\rangle \mid v_1, v_2, v_3, v_4 \geq 0\}$. Observe that the second and fourth abstract traces are false positives that the abstract analysis has to consider but that cannot happen during computation. This is because the guard at B_4 cannot be precisely evaluated on *Sign* and therefore both branches are seen as possible. This happens because the abstract domain of *Sign* is not complete for the

[1] The notation $\lfloor a/2 \rfloor$ refers to the integer division that rounds the non-integer results towards the lower integer value.

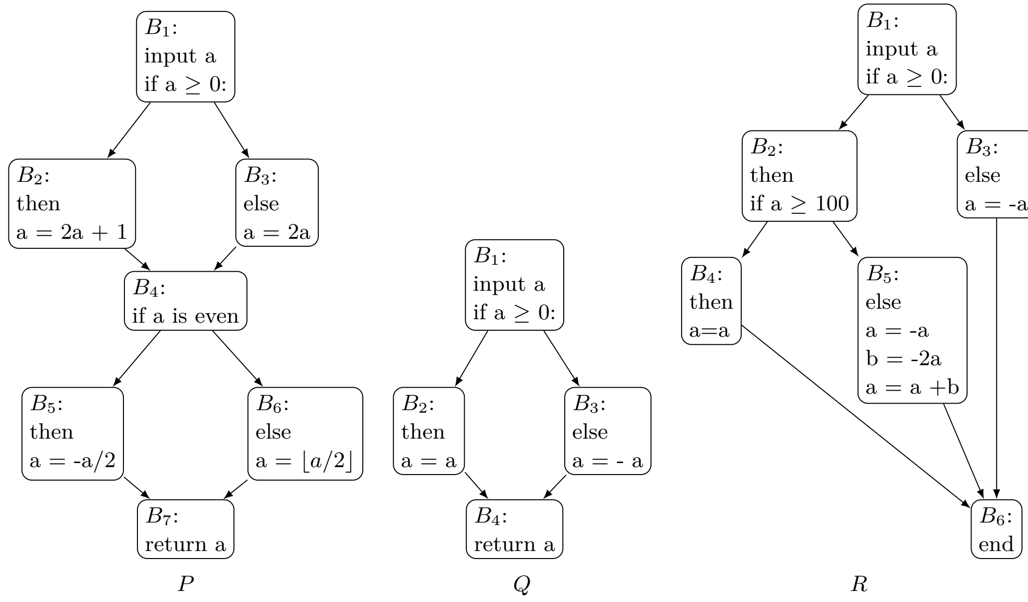


Figure 3 P, Q and R are functionally equivalent programs.

analysis of the program P and we have $[\![P]\!] \subset [\![P]\!]^{Sign}$. This imprecision in the analysis of program P on the abstract domain of *Sign* leads to the approximate conclusion that the value of variable a at the end of execution can be either positive or negative. Let us denote with $[\![P]\!](B_i)$ and with $[\![P]\!]^{Sign}(B_i)$ the possible values that can be assumed by variable a at block B_i when reasoning on the concrete and abstract semantics respectively. In this case we have that $Sign([\![P]\!](B_7)) = Sign(\{v \mid v \geq 0\}) = +_0$ and this is more precise than $[\![P]\!]^{Sign}(B_7) = \sqcup_{Sign}\{+_0, -\} = \top$.

Transforming properties towards completeness

It is well known that completeness is a domain property and that abstract domains can be refined in order to become complete for the analysis of a given program [23]. The idea is that in order to make the analysis complete we need to add to the abstract domain those elements that are necessary to reach completeness. In this case, if we consider the abstract domain that observes the sign and parity of integer values we reach completeness. Thus, let us consider the domain $Sign \sqcap Parity$ depicted on the right of Figure 2, where *even* represents all the even integer values and *odd* represents all the odd integer values. The abstract interpretation of program P on the domain of $Sign \sqcap Parity$ is given by:

$$\begin{aligned} [\![P]\!]^{Sign \sqcap Parity} \quad = \quad & \{\langle B_1 : (+_0, \bot)\rangle\langle B_2 : (+_0, even)\rangle\langle B_4 : (+_0, odd)\rangle\langle B_6 : (+_0, odd)\rangle\langle B_7 : +_0\rangle \\ & \langle B_1 : (+_0, \bot)\rangle\langle B_2 : (+_0, odd)\rangle\langle B_4 : (+_0, odd)\rangle\langle B_6 : (+_0, odd)\rangle\langle B_7 : +_0\rangle \\ & \langle B_1 : (-, \bot)\rangle\langle B_3 : (-, even)\rangle\langle B_4 : (-, even)\rangle\langle B_5 : (-, even)\rangle\langle B_7 : +_0\rangle \\ & \langle B_1 : (-, \bot)\rangle\langle B_3 : (-, odd)\rangle\langle B_4 : (-, even)\rangle\langle B_5 : (-, even)\rangle\langle B_7 : +_0\rangle\} \end{aligned}$$

As we can see all the abstract traces are able to precisely observe that variable a is positive at the end of the execution and that it can be either even or odd. Indeed, we have completeness with respect to the $Sign \sqcap Parity$ property $Sign \sqcap Parity([\![P]\!](B_7)) = [\![P]\!]^{Sign \sqcap Parity}(B_7) = +_0$.

Thus, a possible way for tuning the precision of static analysis is to transform the property that we want to analyse in order to reach completeness, there exists a systematic methodology that allows us to add the minimal amount of elements to the abstract domain in order to make the analysis complete for a given program [23].

Transforming programs towards completeness

The way in which programs are written affects the precision of the analysis. For example we can easily write a program functionally equivalent to P but for which the analysis on *Sign* is complete. Consider, for example, program Q as the one in the middle of Figure 3, we have that:

$$[\![Q]\!] = \{\langle B_1 : \bot\rangle\langle B_2 : v\rangle\langle B_4 : v\rangle \mid v \geq 0\} \cup \{\langle B_1 : \bot\rangle\langle B_3 : v\rangle\langle B_4 : -v\rangle \mid v < 0\}$$

$$[\![Q]\!]^{Sign} = \{\langle B_1 : \bot\rangle\langle B_2 : +_0\rangle\langle B_4 : +_0\rangle, \langle B_1 : \bot\rangle\langle B_3 : -\rangle\langle B_4 : +_0\rangle\}$$

This makes it clear how the abstract computation loses information regarding the modulo of the value of variable a, while it precisely observes the positive value of a at the end of execution. Indeed, in this case we have that: $Sign([\![Q]\!](B_7)) = Sign(\{v \mid v \geq 0\}) = +_0 = [\![Q]\!]^{Sign}(B_7)$. It is worth studying the possibility of transforming programs in order to make a certain analysis complete. In a recent work [6] the authors introduced the notions of complete clique $\mathbb{C}(P, \mathcal{A})$ and incomplete clique $\bar{\mathbb{C}}(P, \mathcal{A})$ that represent the set of all programs that are functionally equivalent to P and for which the analysis on the abstract domain $\mathcal{A}$ is respectively complete and incomplete. They prove that there are infinitely many abstractions for which the systematic removal of false positives for all programs is impossible. Moreover, they observe that false positives are related to the evaluation of boolean predicates that the abstract domain is not able to evaluate precisely (as we have seen in our earlier example). The authors claim that their investigation together with the poof system in [24] should be used as a starting point to reason on a code refactoring strategy that aims at modifying a given program in order to gain precision with respect to a predefined analysis. Given an abstract domain $\mathcal{A}$, the final goal would be to derive a program transformation $\mathcal{T}_A : Prog \rightarrow Prog$ that given a program $P \in \bar{\mathbb{C}}(P, \mathcal{A})$ for which the analysis $\mathcal{A}$ is incomplete, namely $\mathcal{A}([\![P]\!]) \neq [\![P]\!]^{\mathcal{A}}$, transforms it into a program $\mathcal{T}(P) \in \mathbb{C}(P, \mathcal{A})$ for which the analysis is complete, namely $\mathcal{A}([\![P]\!]) = [\![P]\!]^{\mathcal{A}}$.

These recent promising works suggest how to proceed in the investigation of program transformations as a mean for gaining precision in static program analysis.

3.2 Dynamic Analysis

Testing is typically used to discover failures (or bugs), namely an incorrect program behaviour with respect to the requirements or the description of the expected program behaviour. Precision in program testing is expressed in terms of soundness: the ideal situation where no false negatives are present. When speaking of failures, this happens when the executions considered in the test set exhibit at least one behaviour for each one of the failures present in the program. Indeed, when this happens, testing allows us to detect all the failures in the program. It is clear that the choice of the input set to use for testing is fundamental in order to minimise the number of false negatives. What we have just said holds when testing aims at detecting failures as well as for the analysis of any property of traces (as for example the order in which memory cells are accessed, the target of jumps, etc.). Let us denote with $\mathbb{I}_P$ the input space of the possible input values needed to complete an execution of program P under testing[2]. Dynamic analysis considers a finite subset of the input space, called the input set $InSet \subseteq_F \mathbb{I}_P$, that identifies the input values that are used for execution. The

[2] In this work, for simplicity but with no loss of generality, we speak of input values while in the general case we may need collections of values in order to complete an execution of the software under test.

execution traces generated by the input set define the test set, which is the finite set of traces used by dynamic analysis to reason on program behaviour. Given an input value $x \in \mathbb{I}_P$ we denote with $P(x) \in [\![P]\!]$ the execution of program P when fed with input x.

As argued above, the main source of imprecision in testing is that the number of potential inputs for most programs is so large as to be effectively infinite. Since we cannot test with all inputs, researchers typically recur to the use of coverage criteria in order to decide which test inputs to use. A coverage criterion C induces a partition on the input space and in order to minimise the false negatives the input set should contain at least one element for each class of the partition. In the left part of Figure 4 we consider a typical coverage criterion, called path coverage, for the testing of program Q in Figure 3. Path coverage criterion is satisfied when for each path in the control flow graph of the program there exists at least one execution in the test set that follows that path. When considering program Q it is immediate to derive from the coverage criterion the partition of the input space: the class of positive integer values (that follow the path $B_1 \to B_2 \to B_4$) and the class of negatives integer values (that follow the path $B_1 \to B_3 \to B_4$). In this case the coverage criterion is satisfied by every input set that contains at least one positive integer value and one negative integer value.

Since it is the coverage criterion that determines the input set and therefore the executions that are considered by the dynamic analysis, it is very important to select a *good* coverage criterion. However, it is not clearly stated or formally defined what makes a coverage criterion good [1], and this may be one of the reasons why many coverage criteria have been developed by researchers. Generally speaking, there are some features that it is important to consider when speaking of coverage criterion such as:

- the difficulty of deriving the rules to partition the input space with respect to the coverage criterion;
- the difficulty of generating an input set that satisfies the coverage criterion, namely that contains at least one input for each one of the classes in which the input space has been partitioned;
- how well a test set that satisfies the coverage criterion guarantees the absence of false negatives.

To the best of our knowledge there is no general framework that formalises the relation between coverage criterion, partition of the input space and false negatives in the dynamic analysis of a semantic program property. Indeed, while the soundness of dynamic analysis may not be possible in general, we think that it would be interesting to study the soundness of dynamic analysis of a program with respect to a specific semantic property (as usually done when reasoning about completeness in static analysis). We believe that this formal investigation would help in better understanding the cause of false negatives and would be useful in reducing them.

3.2.1 Towards a formal framework for dynamic analysis

We formalise the splitting of the input space induced by a coverage criterion C in terms of an equivalence relation $\mathcal{R}_I^C \in Eq(\mathbb{I}_P)$, and this allows us to formally define when an input set satisfies a coverage criterion.

▶ **Definition 1.** *Given a program P, an input set $InSet \subseteq_F \mathbb{I}_P$ and a coverage criterion C, we say that $InSet$ satisfies C, denoted $InSet \models C$, iff: $\forall [x]_{\mathcal{R}^C} \in \mathbb{I}_P/\mathcal{R}_I^C$ we have that $InSet \cap [x]_{\mathcal{R}_I^C} \neq \emptyset$.*

We have seen this in Figure 4 when considering the partition induced in the input space of program Q and observing that an input set satisfies the path coverage criterion when it contains at least one positive and one negative integer value. When considering coverage

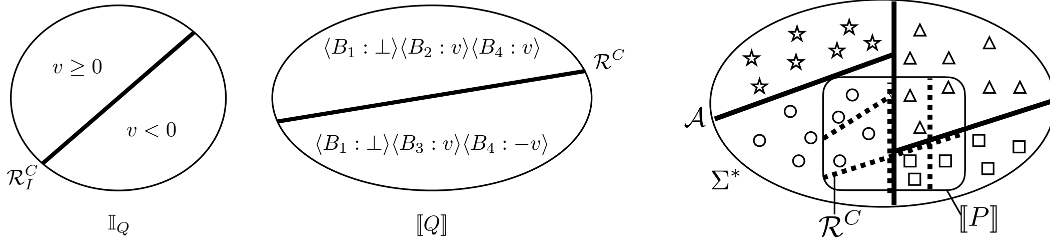


Figure 4 Path coverage criterion on program Q of Figure 3, and soundness conditions.

criteria we need to take into account infeasible requirements: for example when considering coverage criteria related to the paths of the control flow graph we have to handle infeasible paths as it is not possible to define input values that follow these paths (as for example paths $B_1 \to B_2 \to B_4 \to B_5 \to B_7$ and $B_1 \to B_3 \to B_4 \to B_5 \to B_7$ of program P). This is a known challenging problem in dynamic analysis and testing as the detection of infeasible test requirements is undecidable for most coverage criteria [1]. This means that some preliminary analysis is needed in order to ensure the feasibility of the coverage criteria, namely to ensure that it is possible to generate an input set that satisfies a given coverage criterion. Otherwise, we need to somehow quantify how much the input set satisfies the coverage criterion, for example considering the percentage of equivalence classes that are covered by the input set. In this work we do not address this problem and we assume the feasibility of the coverage criteria.

Observe that the equivalence relation $\mathcal{R}_I^C \in Eq(\mathbb{I}_P)$ naturally induces an equivalence relation on traces $\mathcal{R}^C \in Eq([\![P]\!])$ where $(\sigma_1, \sigma_2) \in \mathcal{R}^C$ iff $\exists x_1, x_2 \in \mathbb{I}_P : P(x_1) = \sigma_1$, $P(x_2) = \sigma_2$ and $(x_1, x_2) \in \mathcal{R}_I^C$. Thus, we can say that a given coverage criterion, and therefore any test set that satisfies that coverage criterion, can be associated to a partition of program trace semantics. Our idea is that the partition of the program trace semantics induced by the coverage criterion could be used to reason on the class of semantic program properties for which the coverage criterion can ensure soundness. To this end, we need to represent semantic program properties in a way that can be compared with partitions on traces.

Properties of traces are naturally modelled as abstract domains, namely as closure operators in $uco(\wp(\Sigma^*))$. A semantic property $\rho \in uco(\wp(\Sigma^*))$ maps an execution trace (or a set of execution traces) to the minimal set of traces that cannot be distinguished by the considered property. Each closure operator $\rho \in uco(\wp(\Sigma^*))$ induces an equivalence relation $\mathcal{R}^\rho \in Eq(\Sigma^*)$: $\sigma_1 \mathcal{R}^\rho \sigma_2$ iff $\rho(\{\sigma_1\}) = \rho(\{\sigma_2\})$, where traces are grouped together if they are mapped in the same element by abstraction ρ. In the following, we model the properties of traces as equivalence relations over traces or equivalently as partitioning closures in $uco(\wp(\Sigma^*))$, and we denote these properties as $\mathcal{A} \in Eq(\Sigma^*)$. According to [29] there is more than one closure that maps to the same equivalence relations, thus considering the partitions induced by closure operators corresponds to focusing on the set of partitioning closures (which is a proper subset of closure operators over $\wp(\Sigma^*)$). This allows us to express properties of the single traces but not relational properties that have to take into account more than one trace. This means that we can use equivalence relations in $Eq(\Sigma^*)$ to express properties such as: the order of successive accesses to memory, the order of execution of instructions, the location of the first instruction of a function, the target of jumps, function location, possible data values at certain program points, the presence of a bad states in the trace, and so on. These are properties of practical interest in dynamic program analysis.

What we cannot express are properties on sets of traces, the so called hyper-properties, that express relational properties among traces, like non-interference. The extension of the framework to closures that are not partitioning is left as future work. This allows us to formally model the soundness of dynamic analysis.

▶ **Definition 2.** *Given a program P and a property $\mathcal{A} \in Eq(\Sigma^*)$, the dynamic analysis $\mathcal{A}$ on input set $InSet \subseteq_F \mathbb{I}_P$ is sound, denoted $InSet \overset{s}{\rightsquigarrow} \mathcal{A}(P)$, if $\forall [\sigma]_{\mathcal{A}} \in [\![P]\!]/\mathcal{A}$ we have that $[\sigma]_{\mathcal{A}} \cap InSet \neq \emptyset$.*

This precisely captures the fact that dynamic analysis needs to observe the different behaviours of the program with respect to the property of interest in order to be sound. Indeed, when considering a program P and a property $\mathcal{A}$ it is enough to observe a single trace in an equivalence class $[\sigma]_{\mathcal{A}} \subseteq [\![P]\!]$ in order to observe how property $\mathcal{A}$ behaves in all the traces of program P that belong to that equivalence class. If we consider program Q in Figure 3 we have that in order to precisely observe the evolution of the sign property along the execution we have to consider at least one trace that follows the path $B_1 \to B_2 \to B_4$ and one trace that follows the path $B_1 \to B_3 \to B_4$ as depicted in Figure 4.

Modelling program properties as equivalence relations makes it easy to compare them with the coverage criteria and to reason on soundness.

▶ **Theorem 3.** *Given a program P, a coverage criterion C, an input set $InSet \subseteq_F \mathbb{I}_P$ and a property $\mathcal{A} \in Eq(\Sigma^*)$, we have that if $\mathcal{R}^C \preceq \mathcal{R}^{\mathcal{A}}_{|[\![P]\!]}$ and $InSet \models C$, then $InSet \overset{s}{\rightsquigarrow} \mathcal{A}(P)$.*

Proof. $InSet \models C$ therefore $\forall [x]_{\mathcal{R}^C} \in \mathbb{I}_P/\mathcal{R}^C$ we have that $InSet \cap [x]_{\mathcal{R}^C} \neq \emptyset$. Since $\mathcal{R}^C \preceq \mathcal{A}_{|[\![P]\!]}$ we have that for each equivalence class $[\sigma]_{\mathcal{R}^C}$ there exists an equivalence class $[\sigma]_{\mathcal{A}_{|[\![P]\!]}}$ that $[\sigma]_{\mathcal{R}^C} \subseteq [\sigma]_{\mathcal{A}_{|[\![P]\!]}}$. This implies that for every $[\sigma]_{\mathcal{A}_{|[\![P]\!]}} \in [\![P]\!]/\mathcal{A}$ we have that $[\sigma]_{\mathcal{A}_{|[\![P]\!]}} \cap InSet \neq \emptyset$ and therefore $InSet \overset{s}{\rightsquigarrow} \mathcal{A}(P)$. ◀

In Figure 4 on the right we provide a graphical representation of the above theorem. Traces in Σ^* exhibit different attributes with respect to property $\mathcal{A}$ and this is represented by the different shapes: circle, triangle, square and star. Trace partition is then represented by the thick lines that group together traces that are undistinguishable with respect to property $\mathcal{A}$. Dotted lines are used to represent a trace partition induced by coverage criterion C on the traces of P and that ensures the absence of false negatives in the analysis. Indeed, from the graphical representation it is clear that when $InSet \models C$ then $InSet$ contains at least a trace for each equivalence class of $\mathcal{R}^C$, and this implies that it contains at least a trace for each one of the possible attributes (circle, triangle and square) that traces in $[\![P]\!]$ can exhibit with respect to property $\mathcal{A}$. This allows us to characterise the set of properties for which a given coverage criterion can ensure soundness.

▶ **Definition 4.** *Given a coverage criterion C on a program P, we define the set of properties $\Pi(C) \overset{def}{=} \{\mathcal{A} \in Eq(\Sigma^*) \mid \mathcal{R}^C \preceq \mathcal{A}_{|[\![P]\!]}\}$ that are coarsest than the equivalence relation induced by the coverage criterion.*

It follows that any input set that satisfies a coverage criterion C on a program P would lead to a sound dynamic analysis on any property in $\Pi(C)$.

▶ **Corollary 5.** *Given a coverage criterion C on a program P, and input set $InSet \subseteq_F \mathbb{I}_P$ such that $InSet \models C$, than $\forall \mathcal{A} \in \Pi(C)$ we have that $InSet \overset{s}{\rightsquigarrow} \mathcal{A}(P)$.*

In Figure 5 we summarise the relation between coverage criteria and soundness of a particular program property. Given a program P, Figure 5 depicts the domain of equivalence relations over $[\![P]\!]$ where *id* denotes the most fine equivalence relation that corresponds to the identity

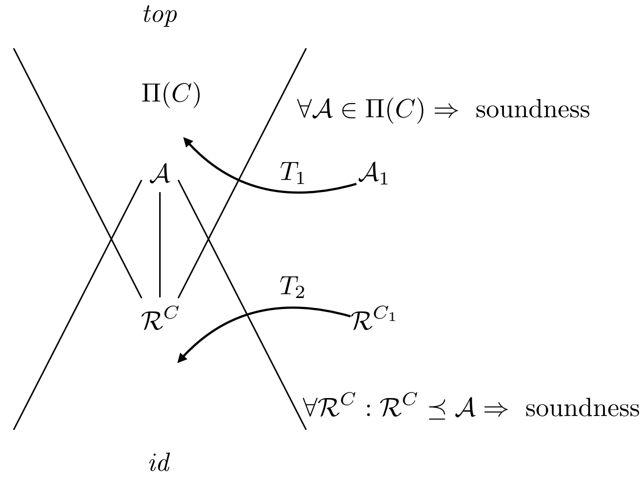


Figure 5 Comparing $\mathcal{R}^C$ and $\mathcal{A}$ for soundness.

relation, $\forall \sigma_1, \sigma_2 \in [\![P]\!] : (\sigma_1, \sigma_2) \in id$ iff $\sigma_1 = \sigma_2$, and *top* denotes the coarser equivalence relation that sees every trace as equivalent $\forall \sigma_1, \sigma_2 \in [\![P]\!]$ it holds that $(\sigma_1, \sigma_2) \in top$. As stated in Theorem 3 whenever $\mathcal{R}^C \preceq \mathcal{A}_{|_{[\![P]\!]}}$ then the coverage criterion C can be used to ensure soundness of the analysis of property $\mathcal{A}$ on program P. As stated by Corollary 5 a coverage criterion C can ensure soundness for all those properties in $\Pi(C)$.

Following our reasoning, the most natural coverage criterion for a given semantic property $\mathcal{A}$ is the one for which $\mathcal{R}^C = \mathcal{A}$, namely the coverage criterion whose partition on states corresponds to the property under analysis. In the literature there exists many different coverage criteria and some of them turn out to be equivalent when compared with respect to the partition that they induce on the input space. It has been observed that all existing test coverage criteria can be formalised in terms of four mathematical structures: input domains, graphs, logic expressions, and syntax descriptions (grammars) [1]. Even if these coverage criteria are not explicitly related to the properties being analysed they have probably been designed while having in mind the kind of properties of interest. For example, some of the most widely known coverage criteria are based on graph features and are typically used for the analysis of properties related to a graphical representation of programs, like control flow or data flow properties of code or variables that can be verified on the control flow graph of a program, or function calls that can be verified on the call graph or a program, and so on. For example code coverage requires the execution of all the basic blocks of a control flow graph and wants to ensure that all the reachable instructions of a program are considered at least in one execution of the test set.

What we have stated so far allows us to begin to answer the question regarding how well the coverage criterion behaves with respect to the analysis of a given semantic property (when this can be modelled as a partitioning closure on the powerset of program traces). The design of an automatic or systematic strategy for the generation of an input set that covers a given coverage criterion remains an open challenge that deserves further investigation.

Transforming properties towards soundness

There are two questions that naturally arise from our reasoning and that would be interesting to investigate regarding the systematic transformation of the property under analysis or the coverage criterion towards soundness.

1. Consider a program P, a coverage criterion C that induces a partition $\mathcal{R}^C \in Eq(\llbracket P \rrbracket)$ on the traces of program P and a trace property $\mathcal{A}_1$ for which the coverage criterion C cannot ensure soundness. We wonder if it is possible to design a systematic transformation of property $\mathcal{A}_1$ that, by grouping some of its equivalence classes, returns a trace property for which we have soundness when C is satisfied by the input set. It would be interesting to understand when this transformation is possible without reaching *top*, i.e., while still being able to distinguish trace properties. This is depicted by the arrow labeled with T_1 in the upper part of Figure 5.
2. Consider a program P, a coverage criterion C_1 that induces a partition $\mathcal{R}^{C_1} \in Eq(\llbracket P \rrbracket)$ on the traces of program P and a trace property $\mathcal{A}$ for which the coverage criterion C_1 cannot ensure soundness. We wonder if it is possible to design a systematic transformation of $\mathcal{R}^{C_1}$ that, by further splitting its equivalence classes, returns a partition of the program traces, and therefore a coverage criterion, that when satisfied by the input set ensures soundness for the analysis of property $\mathcal{A}$. In this case it is interesting to investigate when this refinement is possible without ending up with the identity relation, namely without collapsing to *id* where all program traces needs to be considered for coverage. This is depicted by the arrow labeled with T_2 in the bottom part of Figure 5.

Transforming programs towards soundness

As for static analysis also for dynamic analysis the way in which programs are written influences the precision of the analysis either because they expand the input set that satisfies a given coverage criterion, thus requiring the observation of more program runs, or because they complicate the automatic/systematic extraction of an input set that satisfies a given coverage criterion. We focus on the first case since we still have to formally investigate the extraction of input sets for a given coverage criterion, namely the input generation and input recogniser procedure.

Let us consider program R on the right of Figure 3 that computes the absolute value of an integer value and does it by adding some extra control on the range of the input integer value in order to proceed with the computation of the modulo in some syntactically different, but semantically equivalent ways. Indeed, in this example it is easy to observe that blocks B_4 and B_5 are equivalent, but we can think about more sophisticated ways to write equivalent code in such a way that it would be difficult for the analyst to automatically recognise that they are equivalent. If we consider again the path coverage criterion we can observe that in order to cover the control flow graph of program R we need at least three input values: a negative integer, a positive integer smaller than 100 and a positive integer greater than or equal to 100. Of course what is done in block B_2 can be replicated many times, as far as we are able to write blocks that are syntactically different but semantically equivalent to B_4 or B_3. According to our framework, path coverage is more complicated to reach on program R than on program Q. Indeed, in this case, every input set that satisfies path coverage for program R also satisfies path coverage for program Q while the converse does not hold in general. This reasoning is limited to the amount of traces that we need to satisfy a given coverage criterion and does not take into account the difficulty of generating such traces. Of course both aspects would need to be taken into account by our formal framework.

Moreover, as done for static analysis in [6], it would be interesting to define the notions of sound clique $\mathbb{S}(P, InSet, \mathcal{A})$ and of unsound clique $\bar{\mathbb{S}}(P, InSet, \mathcal{A})$ that represent the sets of all programs that are functionally equivalent to P and for which the dynamic analysis of property $\mathcal{A}$ on input set $InSet \subseteq \mathbb{I}_P$ is respectively sound and not sound:

$$\mathbb{S}(P, InSet, \mathcal{A}) \stackrel{\text{def}}{=} \{Q \in Prog \mid Den(\llbracket P \rrbracket) = Den(\llbracket Q \rrbracket), InSet \stackrel{s}{\rightsquigarrow} \mathcal{A}(P)\}$$

$$\bar{\mathbb{S}}(P, InSet, \mathcal{A}) \stackrel{\text{def}}{=} \{Q \in Prog \mid Den(\llbracket P \rrbracket) = Den(\llbracket Q \rrbracket), Q \notin \mathbb{S}(P, InSet, \mathcal{A})\}.$$

We plan to study the existence of transformations from $\bar{\mathbb{S}}(P, InSet, \mathcal{A})$ to $\mathbb{S}(P, InSet, \mathcal{A})$ in order to rewrite a program toward soundness. It is interesting to identify the properties for which this can be done in a systematic way and the key for reaching soundness. The intuition is that for reaching soundness with respect to a property $\mathcal{A}$ on an input set $InSet$ we should choose programs whose variations of property $\mathcal{A}$ are all considered by the input set as stated in Theorem 3. Thus, in general, if we reduce variations of the considered property by merging traces that are functionally equivalent even if they have diversified $\mathcal{A}$ properties we would probably facilitate soundness. This needs to be formally understood, proved and validated on some existing dynamic analysis.

4 Software protection: a new perspective

In the software protection scenario we are interested in preventing program analysis while preserving the intended behaviour of programs. To face this problem Collberg et al. [9] introduced the notion of code obfuscation: program transformations designed with the explicit intent of complicating and degrading program analysis while preserving program functionality. Few years later Barak et al. [3] proved that it is not possible to obfuscate everything but the input-output behaviour for all programs with limited penalty in performances. However, it is possible to relax some of the requirements of Barak et al. and design obfuscating techniques that are able to complicate certain analysis of programs. This is witnessed by the great amount of obfuscation tools and techniques that researchers, both from academia and industry, have been developing in the last twenty years [8]. What it means for a program transformation to complicate program analysis is something that needs to be formally stated and proved when defining new obfuscating transformations. The extent to which an obfuscating technique complicates, and therefore protects, the analysis of certain program properties is referred to as *potency* of the obfuscation. A formal proof of the quality of obfuscation in terms of its potency is very important in order to compare the efficiency of different obfuscation techniques and in order to understand the degree of protection that they guarantee. Unfortunately, a unifying methodology for the quantitative evaluation of software protection techniques is still an open challenge, as witnessed by the recent Dagstuhl Seminar on this topic [20]. What we have are specific measurements done when new techniques are proposed, or formal proofs that reduce the analysis of obfuscated programs to well known complex analysis tasks (like alias analysis, shape analysis, etc.).

In our framework, complicating program analysis means inducing imprecision in the results of the analysis of the obfuscated program with respect to the results of the analysis of the original program. This means that code obfuscation should induce false positives in static program analysis and false negatives in dynamic program analysis.

4.1 Program transformations against static program analysis

The abstract interpretation framework has been used to reason on the semantic properties that code obfuscation transformations are able to protect and the ones that they can still be analysed on the obfuscated program. It has been observed that a program property expressed by an abstract domain $\mathcal{A}$ is obfuscated (protected) by an obfuscation $\mathcal{O} : Prog \rightarrow Prog$ on a program P whenever $[\![P]\!]^{\mathcal{A}} \leq_{\mathcal{A}} [\![\mathcal{O}(P)]\!]^{\mathcal{A}}$, namely when the analysis $\mathcal{A}$ on the obfuscated program returns a less precise result with respect to the analysis of the same property on the original program P. The spurious information added to the analysis by the obfuscation is the noise that confuses the analyst, thus making the analysis more complicated. The relation between potency of code obfuscation and the notion of (in)completeness in abstract

interpretation has been proven, as obfuscating a property means to induce incompleteness in its analysis [22]. So, for example, the insertion of a true opaque predicate O^T (see the program in the middle of Figure 1) would confuse all those analyses that are not able to precisely evaluate such a predicate and have to consider both branches as possible. No confusion is added for those analyses that are able to precisely evaluate the opaque predicate and consider only the true branch as possible, namely those analyses that are complete for the evaluation of the predicate value. Following this idea, a formal framework based on program semantics and abstract interpretation has been developed, where it is possible to formally prove that a property is obfuscated by a given program transformation, compare the efficiency of different obfuscating techniques in protecting a given property, define a systematic strategy for the design of a code obfuscation technique for protecting a given program property [17, 19, 22, 25]. This semantic understanding of the effects that code obfuscation has on the semantics and semantic properties of programs as shown its usefulness also in the malware detection scenario where malware writers use code obfuscation to evade automatic detection [15, 16].

Thus we can say that the effects of functionality preserving program transformations on program semantics and on the precision of the results of static analysis has been extensively studied and a mature formal framework has been provided [15, 16, 17, 19, 22, 25].

4.2 Program transformations against dynamic program analysis

To the best of our knowledge, the effects of functionality preserving program transformations on the precision of dynamic analysis have not been fully investigated yet. Following our reasoning, the general idea is that dynamic analysis is complicated by program transformations that induce false negatives while preserving program's functionality. Let $\mathcal{A} \in Eq(\Sigma^*)$ denote a property of interest for dynamic analysis. Inducing false negatives for the analysis of a property $\mathcal{A}$ can be done by exploiting the partial observation of program's executions innate in the test set, and thus adding traces that do not belong to the test set and have a different $\mathcal{A}$ property. Thus, the key for software protection against dynamic analysis is software *diversification* with respect to the property under analysis. The ideal obfuscation against the dynamic analysis of property $\mathcal{A}$ should specialise programs with respect to every input in such a way that every input exhibits a different behaviour for property $\mathcal{A}$. Namely, an ideal obfuscation against $\mathcal{A}$ is a program transformation $\mathcal{O} : Prog \to Prog$ such that $\forall \sigma_1, \sigma_2 \in [\![\mathcal{O}(P)]\!]$ we have that $\mathcal{A}(\sigma_1) = \mathcal{A}(\sigma_2) \Leftrightarrow \sigma_1 = \sigma_2$. In this ideal situation in order to avoid false negatives the analyst should consider every possible execution trace of $\mathcal{O}(P)$ since each trace exhibits a different aspects of property $\mathcal{A}$, so missing a trace would mean to miss such an aspect. This intuition is confirmed in a preliminary work in this direction where it is shown how diversification is the basis of existing software protection techniques against dynamic analysis [18]. This work provides a topological characterisation of the soundness of the dynamic analysis of properties expressed as equivalence relations (as we have done in Section 3.2.1). This formal characterisation is then used to define the notion of transformation potency for dynamic analysis.

▶ **Definition 6.** *A functionality preserving program transformation $\mathcal{O} : Prog \to Prog$ is potent for the analysis of $\mathcal{A} \in Eq(\Sigma^*)$ of program P if:*

- $\forall \sigma_1, \sigma_2 \in [\![\mathcal{O}(P)]\!] : [\sigma_1]_{\mathcal{A}} = [\sigma_2]_{\mathcal{A}}$, $\forall \nu_1, \nu_2 \in [\![P]\!] : Den(\nu_1) = Den(\sigma_1), Den(\nu_2) = Den(\sigma_2)$ *then* $[\nu_1]_{\mathcal{A}} = [\nu_2]_{\mathcal{A}}$
- $\exists \nu_1, \nu_2 \in [\![P]\!] : [\nu_1]_{\mathcal{A}} = [\nu_2]_{\mathcal{A}}$ *for which* $\exists \sigma_1, \sigma_2 \in [\![\mathcal{O}(P)]\!] : Den(\nu_1) = Den(\sigma_1), Den(\nu_2) = Den(\sigma_2)$ *such that* $[\sigma_1]_{\mathcal{A}} \neq [\sigma_2]_{\mathcal{A}}$

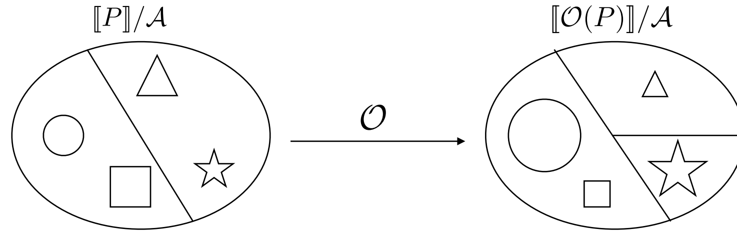


Figure 6 Transformation Potency.

Figure 6 provides a graphical representation of the notion of potency. On the left we have the traces of the original program P partitioned according to the equivalence relation $\mathcal{A}$ induced by the property of interest, while on the right we have the traces of the transformed program $\mathcal{O}(P)$ partitioned according to $\mathcal{A}$. Traces that are denotationally equivalent have the same shape (triangle, square, circle, oval), but different dimension since they are in general different traces. The first condition means that the traces of $\mathcal{O}(P)$ that property $\mathcal{A}$ maps to the same equivalence class (circle and square), are denotationally equivalent to traces of P that property $\mathcal{A}$ maps to the same equivalence class. This means that what is grouped together by $\mathcal{A}$ on $[\![\mathcal{O}(P)]\!]$ was grouped together by $\mathcal{A}$ on $[\![P]\!]$, modulo the denotational equivalence of traces. The second condition requires that there are traces of P (triangle and star) that property $\mathcal{A}$ maps to the same equivalence class and whose denotationally equivalent traces in $\mathcal{O}(P)$ are mapped by $\mathcal{A}$ to different equivalence classes. This means that a defense technique against dynamic analysis with respect to a property $\mathcal{A}$ is successful when it transforms a program into a functionally equivalent one for which property $\mathcal{A}$ is more diversified among execution traces. This implies that it is necessary to collect more execution traces in order for the analysis to be precise. At the limit we have an optimal defense technique when $\mathcal{A}$ varies at every execution trace.

The above definition of transformation potency for dynamic analysis has been validated by modelling in the proposed framework some existing software defence strategies against dynamic analysis for the extraction of the control flow graph of programs like Range Dividers [2] and Gadget diversification [30]. In both cases it is possible to show that the proposed transformations complicate the dynamic extraction of the control flow graph by adding new diversified paths to the control flow graph, as stated in Definition 6. In the following we report a simple example from [18] that shows how the key for obfuscating properties of data values for dynamic analysis is diversification.

▶ **Example 7.** Consider the following programs P and Q that compute the sum of natural numbers from $x \geq 0$ to 49 (we assume that the inputs values for x are natural numbers).

P
```
input x;
sum := 0;
while x < 50
• ≀ X = [0, 49] ≀
   sum := sum + x;
   x := x + 1;
```

Q
```
input x;
n : = select(N,x)
x := x * n;
sum := 0;
while x < 50 * n
• ≀ X = [0, n * 50 − 1] ≀
   sum := sum + x/n;
   x := x + n;
x := x/n;
```

Consider a dynamic analysis that observes the maximal value assumed by x at program point $\bullet$. For every possible execution of program P we have that the maximal value assumed by x at program point $\bullet$ is 49. Consider a state $s \in \Sigma$ as a tuple $\langle pp, [val_x, val_{sum}]\rangle$, where pp denotes the current program point, val_x and val_{sum} denote the current values of variables x and sum respectively. We define a function $\tau : \Sigma \to \mathbb{N}$ that observes the value assumed by x at state s when s refers to program point $\bullet$, and function $max : \Sigma^* \to \mathbb{N}$ that observes the maximal value assumed by x at $\bullet$ along an execution trace:

$$\tau(s) \stackrel{\text{def}}{=} \begin{cases} val_x & \text{if } pp = \bullet \\ \emptyset & \text{otherwise} \end{cases} \qquad max(\sigma) \stackrel{\text{def}}{=} max(\{\tau(s) \mid s \in \sigma\})$$

This allows us to define the equivalence relation $\mathcal{A}_{max} \in Eq(\Sigma^*)$ that observes traces with respect to the maximal value assumed by x at $\bullet$, as $(\sigma, \sigma') \in \mathcal{A}_{max}$ iff $max(\sigma) = max(\sigma')$. We can observe that all the execution traces of P belong to the same equivalence class of $\mathcal{A}_{max}$. In this case, a dynamic analysis of property $\mathcal{A}_{max}$ on P is sound whenever the test set contains at least one execution trace of P. This happens because the property that we are looking for is an invariant property of program executions and it can be observed on any execution trace.

Let us now consider program Q equivalent to P, i.e., $Den[\![P]\!] = Den[\![Q]\!]$, where the value of x is diversified by multiplying it by the parameter n. The guard and the body of the `while` are adjusted in order to preserve the functionality of the program. When observing property $\mathcal{A}_{max}$ on Q, we have that the maximal value assumed by x at program point $\bullet$ is determined by the parameter n generated in the considered trace. The statement `n:=select(N,x)` assigns to n a value in the range $[0, N]$ depending on the input value x. We have that the traces of program Q are grouped by $\mathcal{A}_{max}$ depending on the value assumed by n. Thus, $\mathcal{A}([\![Q]\!])$ contains an equivalence class for every possible value assumed by n during execution. This means that the transformation that rewrites P into Q is potent according to Definition 6. Dynamic analysis of property $\mathcal{A}_{max}$ on program Q is sound if the test set contains at least one execution trace for each of the possible values of n generated during execution.

5 Open research directions

We have provided an unifying view of the relations between properties and program transformations and the precision of static and dynamic analysis in the standard analysis scenario and in the software protection scenario. Researchers have proposed possible ways for tuning the precision of static analysis while less attention has been posed to the formal investigation of dynamic analysis. In this context it is worth to mention the recent work of O'Hearn [26] that defines a formalism called incorrectness logic, which is similar to Hoare's logic, and allows us to prove the presence of bugs but not their absence, thus capturing the essence of program testing. The incorrectness logic is based on a under-approximation triple that plays a dual role when compared to the standard over-approximation triple that we are used to see in Hoare's logic. Indeed, while logic and symbolic reasoning are useful since they can cover many states or program paths at once, they do not allow in general to cover all paths and this makes it difficult to prove the absence of errors. The author claims the necessity and usefulness of incorrectness logic that formalises under-approximate reasoning in order to provide a logical proof of the presence of bugs. Such reasoning should of course be combined with standard correctness proof in order to obtain a global view of program's runtime behaviour. The incorrectness logic of O'Hearn does not try to gain soundness,

namely to avoid or reduce false negatives, but provides formal proofs for what can be derived in an unsound context. Our idea is to investigate the extent to which it is possible to induce or force soundness by modifying either the program, the property to be analysed or the coverage criterion. Once we have understood when and how soundness can be forced we should see how this interacts with incorrectness logic.

The preliminary work done in the investigation of program and properties transformations towards sound dynamic analysis have pointed out many interesting aspects that need to be studied and that we list below as future research directions.

The preliminary results that relate program properties, coverage criteria and the soundness of the analysis should be generalised and extended to properties that cannot be modelled as partitioning closures. Soundness of the analysis and transformation potency should be redefined probably in terms of join-irreducible elements instead of equivalence classes. This further investigation would probably lead to a classification of the properties usually considered by dynamic analysis based on the domain model needed to express them: properties of traces, properties of sets of traces, relational properties, hyper-properties. For each class of properties it would then be interesting to derive a suitable obfuscation strategy. This unifying framework would provide a common ground where to interpret and compare the potency of different software protection techniques in harming dynamic analysis.

As regarding the transformation of properties towards soundness, we plan to verify if and when it is possible to refine the coverage criterion C in order to ensure soundness with respect to a given property $\mathcal{A}$, or when it is possible to further abstract the semantic property $\mathcal{A}$ in order to make it sound for a given coverage criterion C. This should be done starting with properties that can be expressed as partitioning closures and then generalised to the other classes of properties.

As regarding the transformation of programs towards soundness, it is important to investigate when it is possible to transform a program P for which the dynamic analysis of a given property $\mathcal{A}$ is sound (resp. unsound) into a different program P' which is functionally equivalent to P and for which the dynamic analysis of property $\mathcal{A}$ is unsound (resp. sound).

It would also be important to extend the framework in order to take into account the feasibility of the considered coverage criterion, maybe defining some constraints that a program has to satisfy in order to guarantee the feasibility of a given coverage criterion, or by modelling and measuring situations when full coverage is not possible.

References

1. Paul Ammann and Jeff Offutt. *Introduction to software testing.* Cambridge University Press, 2016.
2. Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 189–200, 2016.
3. B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *CRYPTO '01: Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, pages 1–18. Springer-Verlag, 2001.
4. Ezio Bartocci and Yliès Falcone, editors. *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science.* Springer, 2018.
5. Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. Syntia: Synthesizing the semantics of obfuscated code. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 643–659. USENIX Association, 2017.

6 Roberto Bruni, Roberto Giacobazzi, Roberta Gori, Isabel Garcia-Contreras, and Dusko Pavlovic. Abstract extensionality: on the properties of incomplete abstract interpretations. *Proc. ACM Program. Lang.*, 4(POPL):28:1–28:28, 2020.

7 Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking.* Springer, 2018.

8 C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection.* Addison-Wesley Professional, 2009.

9 C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages* (*POPL '98*), pages 184–196. ACM Press, 1998.

10 Kevin Coogan, Gen Lu, and Saumya K. Debray. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 275–284. ACM, 2011.

11 P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theor. Comput. Sci.*, 277(1-2):47–103, 2002.

12 P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages* (*POPL '77*), pages 238–252. ACM Press, 1977.

13 P. Cousot and R. Cousot. A constructive characterization of the lattices of all retractions, preclosure, quasi-closure and closure operators on a complete lattice. *Portug. Math.*, 38(2):185–198, 1979.

14 P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the 6th ACM Symposium on Principles of Programming Languages* (*POPL '79*), pages 269–282. ACM Press, 1979.

15 M. Dalla Preda, M. Christodorescu, S. Jha, and S. Debray. A semantics-based approach to malware detection. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 377–388. ACM Press, 2007. doi:10.1145/1190216.1190270.

16 Mila Dalla Preda, Mihai Christodorescu, Somesh Jha, and Saumya K. Debray. A semantics-based approach to malware detection. *ACM Trans. Program. Lang. Syst.*, 30(5):25:1–25:54, 2008.

17 Mila Dalla Preda and Roberto Giacobazzi. Semantics-based code obfuscation by abstract interpretation. *Journal of Computer Security*, 17(6):855–908, 2009.

18 Mila Dalla Preda, Roberto Giacobazzi, and Niccoló Marastoni. Formal framework for reasoning about the precision of dynamic analysis. In *Static Analysis, 16th International Symposium, SAS 2020*, page to appear, 2020.

19 Mila Dalla Preda and Isabella Mastroeni. Characterizing a property-driven obfuscation strategy. *J. Comput. Secur.*, 26(1):31–69, 2018.

20 Bjorn De Sutter, Christian S. Collberg, Mila Dalla Preda, and Brecht Wyseur. Software protection decision support and evaluation methodologies (dagstuhl seminar 19331). *Dagstuhl Reports*, 9(8):1–25, 2019.

21 Hanne Riis Nielson Flemming Nielson and Chris Hankin. *Principles of Program Analysis.* Springer, 1999.

22 R. Giacobazzi. Hiding information in completeness holes - new perspectives in code obfuscation and watermarking. In *Proc. of The 6th IEEE International Conferences on Software Engineering and Formal Methods (SEFM'08)*, pages 7–20. IEEE Press., 2008.

23 R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretation complete. *Journal of the ACM*, 47(2):361–416, March 2000.

24 Roberto Giacobazzi, Francesco Logozzo, and Francesco Ranzato. Analyzing program analyses. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 261–273, 2015.

25 Roberto Giacobazzi, Isabella Mastroeni, and Mila Dalla Preda. Maximal incompleteness as obfuscation potency. *Formal Asp. Comput.*, 29(1):3–31, 2017.

26 Peter W. O'Hearn. Incorrectness logic. *Proc. ACM Program. Lang.*, 4(POPL):10:1–10:32, 2020.

27 Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. How to kill symbolic deobfuscation for free (or: unleashing the potential of path-oriented protections). In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 177–189, 2019.

28 Andre Pawlowski, Moritz Contag, and Thorsten Holz. Probfuscation: an obfuscation approach using probabilistic control flows. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 165–185. Springer, 2016.

29 Francesco Ranzato and Francesco Tapparo. Strong preservation as completeness in abstract interpretation. In *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2986 of *Lecture Notes in Computer Science*, pages 18–32. Springer, 2004.

30 Sebastian Schrittwieser and Stefan Katzenbeisser. Code obfuscation against static and dynamic reverse engineering. In *International workshop on information hiding*, pages 270–284. Springer, 2011.

31 Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar R. Weippl. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Comput. Surv.*, 49(1):4:1–4:37, 2016.

32 Monirul I. Sharif, Andrea Lanzi, Jonathon T. Giffin, and Wenke Lee. Automatic reverse engineering of malware emulators. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 94–109. IEEE Computer Society, 2009.

33 Bernhard Steffen and Gerhard J. Woeginger, editors. *Computing and Software Science - State of the Art and Perspectives*, volume 10000 of *Lecture Notes in Computer Science*. Springer, 2019.

34 Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. A generic approach to automatic deobfuscation of executable code. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 674–691. IEEE Computer Society, 2015.

The Servers of Serverless Computing: A Formal Revisitation of Functions as a Service

Saverio Giallorenzo
University of Southern Denmark, Odense, Denmark (former)
University of Bologna/INRIA, Italy (current)
saverio.giallorenzo@gmail.com

Ivan Lanese
University of Bologna/INRIA, Italy
ivan.lanese@gmail.com

Fabrizio Montesi
University of Southern Denmark, Odense, Denmark
fmontesi@imada.sdu.dk

Davide Sangiorgi
University of Bologna/INRIA, Italy
davide.sangiorgi@unibo.it

Stefano Pio Zingaro
University of Bologna/INRIA, Italy
stefanopio.zingaro@unibo.it

Abstract

Serverless computing is a paradigm for programming cloud applications in terms of stateless functions, executed and scaled in proportion to inbound requests. Here, we revisit SKC, a calculus capturing the essential features of serverless programming. By exploring the design space of the language, we refined the integration between the fundamental features of the two calculi that inspire SKC: the λ- and the π-calculus. That investigation led us to a revised syntax and semantics, which support an increase in the expressiveness of the language. In particular, now function names are first-class citizens and can be passed around. To illustrate the new features, we present step-by-step examples and two non-trivial use cases from artificial intelligence, which model, respectively, a perceptron and an image tagging system into compositions of serverless functions. We also illustrate how SKC supports reasoning on serverless implementations, i.e., the underlying network of communicating, concurrent, and mobile processes which execute serverless functions in the cloud. To that aim, we show an encoding from SKC to the asynchronous π-calculus and prove it correct in terms of an operational correspondence.

Dedicated to Maurizio Gabbrielli, on his 60th birthday
(… rob da mët !)

2012 ACM Subject Classification Computer systems organization → Cloud computing; Theory of computation → Concurrency

Keywords and phrases Serverless computing, Process calculi, Pi-calculus

Digital Object Identifier 10.4230/OASIcs.Gabbrielli.2020.5

Funding *Fabrizio Montesi*: Partially supported by Villum Fonden (grant no. 29518).
Davide Sangiorgi: Partially supported by MIUR-PRIN project "Analysis of Program Analyses" (ASPRA, ID 201784YSZ5_004).

© Saverio Giallorenzo, Ivan Lanese, Fabrizio Montesi, Davide Sangiorgi, and Stefano Pio Zingaro; licensed under Creative Commons License CC-BY

1 Introduction

Background

Serverless computing, or Functions as a Service (FaaS), is a recent paradigm for programming cloud applications [2]. Programmers code applications in terms of functions. Function definitions can be stored inside of dedicated repositories, and their execution can be triggered by events. An underlying cloud framework is responsible for executing functions whenever triggered, by utilising efficiently a pool of servers. Asynchronous interaction is prominent in this setting, since a function might trigger the execution of another function on another server, and then fetch the result later on to perform further computation.

A framework for serverless computing consists of two layers: a *language layer*, which programmers use to code their applications in terms of (asynchronous) functions; and an *implementation layer*, which executes applications written in the language layer by leveraging a distributed system (the cloud). One can see the first layer as the frontend for developers, and the second layer as the backend that makes serverless systems tick.

Both the language and implementation layers of serverless are complex. Different vendors implement them in different ways with different limitations, in particular regarding how programs can be composed and reasoned about [2, 21, 17]. This motivated recent studies on formal "core" languages for serverless computing, which aim at providing solid foundations for developing and reasoning on serverless systems [13, 20]. We call such core languages *serverless calculi*.

Serverless calculi are based on the λ-calculus: the language layer of serverless frameworks deals with functions, and the λ-calculus is the undisputed reference model for functional programming. The Serverless Kernel Calculus (SKC, or "sketch") is a concurrent λ-calculus enriched with asynchronous function evaluation, futures, and a stateful function repository [13]. SKC leverages the call-by-value evaluation strategy for the λ-calculus to provide a uniform treatment of substitutions for both local and remote serverless computations: evaluating a function synchronously or asynchronously leads to the same substitution carrying the result to the caller.

Motivation

Our work originates from two main motivations.

The first motivation regards the design of the language layer. Both calculi from [13, 20] enhance the λ-calculus with features from process calculi to represent concurrent execution of functions. However, some of these features are slightly different and sometimes more limited. For example, the calculus in [20] uses a "fresh name" condition to transfer the result of function evaluations instead of an explicit operator for name scoping, like name restriction in the π-calculus [27]. SKC adopts a restriction operator to create private names of futures, which are used for the same purpose. Yet, it does not support the creation of new function names that can be stored in the shared function repository, so it is not possible to create private function definitions or private stored data. Furthermore, the expressiveness of SKC is only briefly discussed in [13].

Our second motivation regards the implementation layer. Implementing a serverless framework requires dealing with communication and mobility, in the sense that the connections among the underlying cloud servers and the concurrent processes that run within them change at runtime. The reference theory for mobile processes is the π-calculus, but to the best of our knowledge there is still no exploration of how serverless calculi can be formally linked to it.

This article

In this article, we revisit the theory of SKC and provide a more extensive presentation of its features.

Our new version of SKC (Section 2) provides a few improvements, which are given by a better integration of the essential features of the λ-calculus and the π-calculus. Specifically,

- functions can now be parametric on the names of other functions available in a serverless system, whereas before all references to functions in the repository were statically fixed;
- it is now possible to create new function names for the repository dynamically, so the repository of available functions can now grow freely at runtime.

These new features enhance the expressiveness of the language, which we illustrate through small examples (Section 2) and two use cases (Section 3) from artificial intelligence, one implementing the perceptron algorithm and one for distributed tagging of large images.

We present two semantic interpretations for our version of SKC. The first (Section 2) is a refinement of the original reduction semantics from [13], which supports the aforementioned improvements. This high-level semantics is intended for developers to reason abstractly about SKC programs. The second semantic interpretation is a formalisation of a possible implementation layer for SKC, given in terms of an encoding (Section 5) from SKC to the asynchronous π-calculus [41] (recalled in Section 4). The encoding is inspired by Milner's encoding of the call-by-value λ-calculus [26]. It shows how serverless functions can be implemented by servers (replicated processes in the π-calculus) that can be triggered by messages from clients, and how a serverless implementation layer can be modelled in terms of communications among processes. We prove that the encoding is correct in terms of an operational correspondence result.

Our results show that standard techniques from process calculi can be useful to understand the two layers of serverless calculi. Hopefully, this understanding could also provide foundations for tackling some outstanding questions in serverless computing. For example, predicting resource usage and costs is challenging in general, since it requires knowing how functions are executed by the implementation layer.

2 The Serverless Kernel Calculus, Revised

We now present our refined version of the Serverless Kernel Calculus (SKC).

$$
\begin{array}{rlcl}
\text{Configurations} & C & ::= & \langle S, \mathcal{D} \rangle \mid \boldsymbol{\nu} n\, C \\
\text{Definition repository} & \mathcal{D} & ::= & \{(f_1, M_1), \ldots, (f_k, M_k)\} \quad (k \geq 0) \\
\text{Systems} & S, S' & ::= & c \blacktriangleleft M \mid S \mid S' \mid \boldsymbol{\nu} n\, S \mid 0 \\
\text{Functions} & M, N & ::= & M\ N \mid V \mid \\
 & & & \mathsf{call}\ h \mid \mathsf{store}\ h\ N\ M \mid \mathsf{take}\ h \mid \boldsymbol{\nu} f\ M \mid \mathsf{async}\ M \mid c \\
\text{Values} & V, V' & ::= & x \mid \lambda x.\, M \mid f \\
\text{Restrictable names} & n & ::= & c \mid f \\
 & h & ::= & f \mid x \\
\text{Function names} & f & \in & \mathsf{Fun} \\
\text{Future names} & c & \in & \mathsf{Fut} \\
\text{Variables} & x & \in & \mathsf{Var}
\end{array}
$$

Figure 1 Syntax of SKC.

2.1 Syntax

The syntax of SKC terms is given in Figure 1, and described in the following.

We assume three disjoint enumerable sets of *names*: function names, ranged over by f; future names, ranged over by c; and the usual variables of λ-calculus, ranged over by x.

Configurations and definition repositories

A configuration C represents a running serverless architecture. In a configuration of the form $\langle S, \mathcal{D} \rangle$:

- S is a system composed of functions that are currently being evaluated; and
- $\mathcal{D}$ is a repository of function definitions that can be triggered (and updated) at runtime.

We treat definition repositories $\mathcal{D}$ as partial maps from function names (ranged over by f) to function bodies (ranged over by M). Thus, for example, the writings $\mathcal{D}(f) = M$ and $(f, M) \in \mathcal{D}$ are equivalent. Definition repositories are mutable: their content can change at runtime, as we are going to see when we discuss the syntax of functions.

The configuration term $\boldsymbol{\nu} n\, C$ restricts the scope of a name n to C, binding n in C. A restrictable name n can be either a function name f or a future name c.

Systems

A system S is a composition of functions that are being evaluated in parallel.

Term $c \blacktriangleleft M$ represents a function under evaluation, whose result will be made available under the future name c upon termination.

Systems of running functions can be composed in parallel, written $S \mid S'$. Term 0 is the unit of parallel composition.

Names of futures and functions can be restricted in systems as well, using the term $\boldsymbol{\nu} n\, S$.

The restriction operator ν binds stronger than the parallel operator $\mid$. Thus, for example, $\boldsymbol{\nu} n\, S \mid S'$ is interpreted as $(\boldsymbol{\nu} n\, S) \mid S'$.

Functions

The language of functions includes the usual terms of λ-calculus: the variable term x, the application term MN, and the abstraction term $\lambda x.\, M$. We distinguish the syntactic category of values (V) to make the presentation of our call-by-value semantics easier later on.

We extend the usual syntax of functions with terms for using the definition repository and futures.

A function f in the definition repository can be invoked by term $\mathsf{call}\ f$, which abstractly represents the "triggering" of a function in the definition repository by an event. The syntax is actually more general: in term $\mathsf{call}\ h$, h can be either a function name or a variable. This enables abstracting over function names, as in $\lambda x.\, \mathsf{call}\ x$. Passing a function name as an argument is enabled since term f is a value.

The primitives store and take manipulate the definition repository. Specifically, term $\mathsf{store}\ h\ N\ M$ updates the definition repository with the mapping (h, N) – h is now mapped to N – and then proceeds by evaluating M. Dually, term $\mathsf{take}\ h$ removes the function with name h from the definition repository and then proceeds as its body. For example, assuming that the definition repository contains a mapping (f, M), then $\mathsf{take}\ f$ would erase that mapping and proceed as M.

We allow for function names to be restricted, written $\boldsymbol{\nu} f\, M$, which allows for the definition of functions in the repository that have "private names".

Moving to futures, term $\mathsf{async}\ M$ starts the asynchronous execution of M. The idea is that M is going to run in parallel, and that this execution is going to be connected to the original term $\mathsf{async}\ M$ through a fresh future name that is created automatically. For example, a system running $c' \blacktriangleleft \mathsf{async}\ M$ would reduce to $\boldsymbol{\nu} c\,(c' \blacktriangleleft c \mid c \blacktriangleleft M)$. Term c represents a function waiting for a future to be resolved: it will be replaced with the result of the parallel running function $c \blacktriangleleft M$ once M produces a value that can be returned. Note that futures c are not values, hence they cannot be, e.g., passed to functions. One may increase flexibility e.g., by "thunking" futures such that $\mathsf{async}\ M$ becomes the value $\lambda x.\, c$ instead of the non-value c. We leave the exploration of this direction for future work.

Equality and substitution

The ν and λ operators bind names, giving rise to the expected notions of free names ($\mathtt{fn}(-)$) and bound names ($\mathtt{bn}(-)$). Names ($\mathtt{n}(-)$) are the union of free and bound names. Thus, we obtain the usual notions of α-equivalence, written $=_\alpha$, and capture-avoiding substitution for functions, written $M\{V/x\}$ (read "V replaces x in M").

In the remainder, we equate α-equivalent systems, and the same for functions. Consistently with viewing definition repositories as maps, equality of definition repositories allows for swap – the order in which definitions are given inside of $\mathcal{D}$ does not matter. This is equivalent to extensional equality over finite maps: $\mathcal{D} = \mathcal{D}'$ if and only if $\mathcal{D}$ and $\mathcal{D}'$ have the same domain of definition $F \subset \mathsf{Fun}$ and $\mathcal{D}(f) = \mathcal{D}'(f)$ for all $f \in F$.

2.2 Semantics

We present now the semantics of SKC, given in terms of structural equivalence of terms and a reduction relation that captures term dynamics.

Both structural equivalence and the reduction relation make use of the auxiliary definition of evaluation contexts, given in Figure 2. An evaluation context $\mathcal{E}$ is a running function with a hole $[\cdot]$ that can be replaced with a function term M. We write $\mathcal{E}[M]$ for the term obtained by replacing the hole in $\mathcal{E}$ with M.

$$\begin{array}{lll} \mathcal{E} & ::= & c \blacktriangleleft \mathcal{E}_\lambda \\ \mathcal{E}_\lambda & ::= & [\cdot] \mid \lambda x.\, M\ \mathcal{E}_\lambda \mid \mathcal{E}_\lambda\ M \end{array}$$

Figure 2 SKC, evaluation contexts.

Structural equivalence

Terms in SKC give rise to the expected equivalences regarding name scoping and parallel composition. This is formalised by the *structural equivalence* $\equiv$ between terms, which is the smallest congruence on SKC terms satisfying the rules in Figure 3.

These rules are straightforward adaptations of the typical rules for scoping and the parallel operator found in process calculi. The first row of rules axiomatise that parallel composition ($\mid$) behaves as a commutative monoid with 0 as identity element. The second row deals with extrusion of function names in evaluation contexts. The third row contains rules for garbage collection of restrictions, swapping of restrictions, and name extrusion in systems and configurations.

$$\frac{}{S \mid S' \equiv S' \mid S} \qquad \frac{}{S \mid (S' \mid S'') \equiv (S \mid S') \mid S''} \qquad \frac{}{S \mid 0 \equiv S}$$

$$\frac{f \notin \mathtt{fn}(\mathcal{E})}{\boldsymbol{\nu} f\ \mathcal{E}[M] \equiv \mathcal{E}[\boldsymbol{\nu} f\ M]} \qquad \frac{f \notin \mathtt{fn}(\mathcal{E}_\lambda)}{\boldsymbol{\nu} f\ \mathcal{E}_\lambda[M] \equiv \mathcal{E}_\lambda[\boldsymbol{\nu} f\ M]}$$

$$\frac{}{\boldsymbol{\nu} n\, 0 \equiv 0} \qquad \frac{}{\boldsymbol{\nu} n\, \boldsymbol{\nu} n'\ S \equiv \boldsymbol{\nu} n'\, \boldsymbol{\nu} n\ S} \qquad \frac{n \notin \mathtt{fn}(S')}{\boldsymbol{\nu} n\ (S \mid S') \equiv \boldsymbol{\nu} n\ S \mid S'} \qquad \frac{n \notin \mathtt{fn}(\mathcal{D})}{\boldsymbol{\nu} n\ \langle S, \mathcal{D}\rangle \equiv \langle \boldsymbol{\nu} n\ S, \mathcal{D}\rangle}$$

Figure 3 SKC, rules for structural congruence.

$$\frac{}{\langle \mathcal{E}[(\lambda x.\, M)\ V], \mathcal{D}\rangle \longrightarrow \langle \mathcal{E}[M\{V/x\}], \mathcal{D}\rangle}\ \beta$$

$$\frac{c \notin \mathtt{fn}(M) \quad c \notin \mathtt{n}(\mathcal{E})}{\langle \mathcal{E}[\mathsf{async}\ M], \mathcal{D}\rangle \longrightarrow \langle \boldsymbol{\nu} c\ (\mathcal{E}[c] \mid c \blacktriangleleft M), \mathcal{D}\rangle}\ \textsc{async}$$

$$\frac{}{\langle \mathcal{E}[c] \mid c \blacktriangleleft V, \mathcal{D}\rangle \longrightarrow \langle \mathcal{E}[V] \mid c \blacktriangleleft V, \mathcal{D}\rangle}\ \textsc{push}$$

$$\frac{}{\langle \mathcal{E}[\mathsf{store}\ f\ N\ M], \mathcal{D}\rangle \longrightarrow \langle \mathcal{E}[M], \mathcal{D}[f \mapsto N]\rangle}\ \textsc{store}$$

$$\frac{}{\langle \mathcal{E}[\mathsf{call}\ f], \mathcal{D}\rangle \longrightarrow \langle \mathcal{E}[\mathcal{D}(f)], \mathcal{D}\rangle}\ \textsc{call} \qquad \frac{}{\langle \mathcal{E}[\mathsf{take}\ f], \mathcal{D}\rangle \longrightarrow \langle \mathcal{E}[\mathcal{D}(f)], \mathtt{undef}(\mathcal{D}, f)\rangle}\ \textsc{take}$$

$$\frac{\langle S, \mathcal{D}\rangle \longrightarrow \langle S', \mathcal{D}'\rangle}{\langle \boldsymbol{\nu} n\ S, \mathcal{D}\rangle \longrightarrow \langle \boldsymbol{\nu} n\ S', \mathcal{D}'\rangle}\ \textsc{res-s} \qquad \frac{\langle S_1, \mathcal{D}\rangle \longrightarrow \langle S_1', \mathcal{D}'\rangle}{\langle S_1 \mid S_2, \mathcal{D}\rangle \longrightarrow \langle S_1' \mid S_2, \mathcal{D}'\rangle}\ \textsc{par}$$

$$\frac{C \longrightarrow C'}{\boldsymbol{\nu} n\ C \longrightarrow \boldsymbol{\nu} n\ C'}\ \textsc{res-c} \qquad \frac{C_1 \equiv C_1' \quad C_1' \longrightarrow C_2' \quad C_2' \equiv C_2}{C_1 \longrightarrow C_2}\ \textsc{str}$$

Figure 4 SKC, reduction rules.

Reductions

We can now define the reduction relation $\longrightarrow$, which formalises the execution of terms in SKC. Relation $\longrightarrow$ is the smallest relation closed under the rules displayed in Figure 4.

The semantics of SKC is based on the call-by-value evaluation strategy for λ-calculus. Specifically, rule β allows for an application to reduce only when its argument is a value (V).

Rule ASYNC models the asynchronous execution of a function M: it creates a new future c, a parallel running function to compute the result of M in c, and binds c to the parallel composition of the caller (which is now waiting to receive the result) and the new running function. When the created running function reduces to a result value, the caller can collect this result by rule PUSH.

In rule STORE, a term $\mathsf{store}\ f\ N\ M$ updates the definition repository $\mathcal{D}$ with the mapping (f, N). The notation $\mathcal{D}[f \mapsto N]$ means that $\mathcal{D}$ is updated to contain the mapping (f, N): if f was already mapped to something, the old mapping is discarded.

In rule CALL, a term $\mathsf{call}\ f$ retrieves the body of the function f from the definition repository, if it is defined, and runs it. A term $\mathsf{call}\ f$ is stuck if the definition repository does

not contain any mapping for f (but can become unstuck if a mapping appears later on). Rule TAKE is similar, but the mapping for the called function is removed from the definition repository: we write `undef`$(\mathcal{D}, f)$ for the repository obtained by removing the mapping for f from $\mathcal{D}$.

The other rules are the expected ones for dealing with restriction (RES-S and RES-C), parallel composition (PAR), and structural equivalence (STR): reductions under restriction and parallel composition can be lifted, and the reduction relation $\longrightarrow$ is closed under the structural equivalence $\equiv$.

▶ **Example 1** (Local vs Async execution)**.** As we are going to see in Section 3, the definition repository is useful to store data and functions that are commonly reused. By itself, term call f retrieves the body of function f from the repository and runs it locally. One can combine call with async to execute the retrieved function asynchronously, which gives some control on how functions from the repository should be executed.

The caller of a function does not need to worry about which strategy is used by the callee, since the semantics of SKC makes both to eventually reduce to the same result. For example, assume that $\mathcal{D}(f) = V$. The following reduction chains show the respective behaviours of the two strategies.

$$\langle c \blacktriangleleft \mathsf{call}\ f, \mathcal{D}\rangle \longrightarrow \langle c \blacktriangleleft V, \mathcal{D}\rangle \tag{1}$$

$$\begin{aligned}
&\langle c \blacktriangleleft \mathsf{async\ call}\ f, \mathcal{D}\rangle \\
&\longrightarrow \langle \boldsymbol{\nu} c'\ (c \blacktriangleleft c' \mid c' \blacktriangleleft \mathsf{call}\ f), \mathcal{D}\rangle \\
&\longrightarrow \langle \boldsymbol{\nu} c'\ (c \blacktriangleleft c' \mid c' \blacktriangleleft V), \mathcal{D}\rangle \\
&\longrightarrow \langle \boldsymbol{\nu} c'\ (c \blacktriangleleft V \mid c' \blacktriangleleft V), \mathcal{D}\rangle
\end{aligned} \tag{2}$$

The resulting term has the same behaviour of the one resulting from the local execution. One could make the two terms syntactically equal by implementing garbage collection for unused futures (and the related values).

▶ **Example 2** (Shared state)**.** The definition repository can be used to store and share state. A simple example is keeping a counter of requests. We abuse notation and use arithmetic operators and natural numbers in SKC – as presented in Section 3. The counter can be initialised with

$$(\mathsf{store}\ \mathit{counter}\ V_0\ M)$$

where V_0 is the initial value, and incremented with

$$(\lambda x.\,\mathsf{store}\ \mathit{counter}\ \ (\mathsf{call}\ \mathit{sum}\ 1\ x)\ \ M)\,(\mathsf{take}\ \mathit{counter})$$

In both the cases M is a continuation.

▶ **Example 3** (Libraries)**.** Updating shared state as in the previous example happens often in serverless computing. One could think of offering a replace primitive as syntactic sugar.

$$\mathsf{replace}\ h\ N\ M \triangleq (\lambda x.\,\mathsf{store}\ h\ \ (N\ x)\ \ M)\,(\mathsf{take}\ h)$$

We can then rewrite our previous counter example as follows.

$$\mathsf{replace}\ \mathit{counter}\ \ (\mathsf{call}\ \mathit{sum}\ 1)\ \ M$$

We can actually use the definition repository of SKC to offer these kinds of user-defined functions as libraries. For example, the replace function above can be put in a repository and then invoked by programs as follows.

$$\begin{array}{l}\langle \\ \ c \blacktriangleleft (\mathsf{call}\ \mathit{replace})\ \ \mathit{counter}\ \ (\mathsf{call}\ \mathit{sum}\ 1)\ \ M, \\ \ \{(\mathit{replace}, \lambda \mathit{name}.\, \lambda \mathit{op}.\, \lambda \mathit{cont}.\, (\lambda x.\, \mathsf{store}\ \mathit{name}\ \ (\mathit{op}\ x)\ \ \mathit{cont})\, (\mathsf{take}\ \mathit{name}))\} \\ \rangle\end{array}$$

▶ **Example 4** (Private state). By combining the restriction operator with the primitives for manipulating the definition repository, we obtain private storage. This can be used, for example, to implement a log in a composition of function calls.

The library function *newLog* below creates a private log and initialises it with an empty list (*nil*, cf. Section 3).

$$(\mathit{newLog}, \boldsymbol{\nu} \mathit{log}\, (\mathsf{store}\ \mathit{log}\ \mathit{nil}\ \mathit{log}))$$

A function can now create a log and pass it to other functions that it invokes, as follows (we omit the concrete composed functions M and N). We assume the existence of a library function *pair* that takes two arguments and returns a pair (we show the definition of *pair* in Section 3).

$$(\lambda x.\, (\mathsf{call}\ \mathit{pair}\, ((M\ \ x)(N\ \ x)))\, x)\, (\mathsf{call}\ \mathit{newLog})$$

The functions M and N can internally use the library function *replace* to update the log by appending messages to the list. The idea is that $N\ x$ is evaluated first, the result of which is then taken by M along with x (the name of the private log). The function *pair* retrieved from the repository then takes the final result from M and the x as the elements of the pair.

Notice that if the programmer wishes to run M and/or N asynchronously, they can just prepend the inner calls to these functions with the async keyword.

Using similar devices, one can implement stack traces and other tracing mechanisms with different scoping rules (per session, per client, etc.).

3 Use Cases

In this section, we present a couple of non-trivial SKC programs as use cases to show and comment how developers can use SKC to reason about their serverless implementations.

For the use cases, we take inspiration from the context of artificial intelligence (specifically, machine learning), which can benefit considerably from the dynamic scalability of serverless architectures [17, 21].

First, we present an implementation of the perceptron [34, 12], an algorithm for binary classification which can decide to which of two classes an input, represented by a vector of numbers, belongs. Then, we present a scatter-gather algorithm that uses a neural network trained for image classification to infer the semantic content of an input image.

Before presenting the examples, we assume to extend SKC with the syntax for conditionals `if` M `then` M' `else` M'', with the traditional semantics. Moreover, we assume that the definition repository $\mathcal{D}$ executing our examples includes the definition of the Church encodings for pairs and lists.

$$\begin{array}{lllllll}
\mathcal{D}(pair) & = & \lambda x.\,\lambda y.\,\lambda z.\,z\ x\ y & \mathcal{D}(isNil) & = & \mathsf{call}\ first \\
\mathcal{D}(first) & = & \lambda p.\,p(\lambda x.\,\lambda y.\,x) & \mathcal{D}(cons) & = & \lambda h.\,\lambda t.\,\mathsf{call}\ pair\ \mathtt{false}\ (\mathsf{call}\ pair\ h\ t) \\
\mathcal{D}(second) & = & \lambda p.\,p(\lambda x.\,\lambda y.\,y) & \mathcal{D}(head) & = & \lambda z.\,\mathsf{call}\ first\ (\mathsf{call}\ second\ z) \\
\mathcal{D}(nil) & = & \mathsf{call}\ pair\ \mathtt{true}\ \mathtt{true} & \mathcal{D}(tail) & = & \lambda z.\,\mathsf{call}\ second\ (\mathsf{call}\ second\ z)
\end{array}$$

Hence, for example, we can write the list $1, 2, 3$ as $(\mathsf{call}\ cons\ 1\ (\mathsf{call}\ cons\ 2\ (\mathsf{call}\ cons\ 3\ \mathsf{call}\ nil)))$.

We also assume to extend SKC with the standard arithmetic $(+, -, *)$ and relational $(>, =)$ operators, on which we build the following functions, also assumed present in the definition repository $\mathcal{D}$.

$$\begin{array}{lllllllll}
\mathcal{D}(sum) & = & \lambda x.\,\lambda y.\,x+y & \mathcal{D}(prod) & = & \lambda x.\,\lambda y.\,x*y & \mathcal{D}(sub) & = & \lambda x.\,\lambda y.\,x-y \\
\mathcal{D}(gt) & = & \lambda x.\,\lambda y.\,x>y & \mathcal{D}(eq) & = & \lambda x.\,\lambda y.\,x=y & & &
\end{array}$$

Finally, we extend SKC with *let* expressions: $let\ x \leftarrow M'\, in\ M \triangleq (\lambda x.\,M)\ M'$.

3.1 A Serverless Perceptron Algorithm

The perceptron, first introduced by Rosenblatt [34], is an algorithm for binary classification that, given an input and a set of weights, produces an output representing the predicted class. A simple application of Rosenblatt's perceptron is the prediction of the logical conjunction. Given the list of inputs $(0, 0)$, $(0, 1)$, $(0, 1)$, and $(1, 1)$, we aim at predicting the target values 0, 0, 0, and 1 respectively.

To obtain a trained model, i.e., a list of weights enabling the perceptron to find the targets corresponding to some given inputs, we start from a zero-initialised vector of weights (a list with the same cardinality of the inputs whose elements are all 0). The *train*ing algorithm consists of an iteration over a set of inputs labelled with the correct classification to return a new list of updated weights, aligned with the new provided "knowledge". Concretely, a training set for the logical conjunction contains pairs like $((0, 1), 0)$ and $((1, 1), 1)$, where the left element is the list of input's features and the right one is the classifying label – in this case, the binary representation of the truth value of the conjunction of the two elements in the list. Once trained, the weights can be used to *predict* the classification of further inputs. To complete the scenario, both the *predict* and *train*ing functions take a *bias* parameter that, in the perceptron algorithm, acts as the y-intercept of the line that separates the feature space. Without this added parameter it would not be possible to find a solution for the classification problem.

We start by describing the *predict* function, defined in Figure 5, which the *train*ing function uses to calculate an adjustment gradient and update the weights. Mathematically, the classification is based on the formula $comp(f, w, bias) = bias + \sum_{i=0}^{|f|} w_i * f_i$: the item is classified with label "1" if the result is positive, "0" otherwise.

$$\begin{array}{ll}
\mathcal{D}(predict) = & \lambda\, ft.\,\lambda\, ws.\,\lambda\, bias. \\
& let\ comp\ \leftarrow (\lambda\, f.\,\lambda\, w.\,\lambda\,\rho. \\
& \quad \mathsf{call}\ sum\ (\mathsf{call}\ prod\ (\mathsf{call}\ head\ f)(\mathsf{call}\ head\ w)) \\
& \quad\ \mathtt{if}\ (\mathsf{call}\ isNil\ (\mathsf{call}\ tail\ f))\ \mathtt{then}\ 0\ \mathtt{else}\ \mathsf{async}\ \rho\ (\mathsf{call}\ tail\ f)\ (\mathsf{call}\ tail\ w)\ \rho \\
&)\ in\ \mathtt{if}\ (\mathsf{call}\ gt\ (\mathsf{call}\ sum\ bias\ (\mathsf{async}\ comp\ ft\ ws\ comp))\ 0)\ \mathtt{then}\ 1\ \mathtt{else}\ 0
\end{array}$$

Figure 5 Function *predict*.

```
D(train) =  λ features. λ weights. λ label. λ bias.
  if call eq (async call predict features weights bias) label
  then call pair weights bias
  else
    let routine ← (
      λ f. λ w. λ grad. λ ρ.
        call cons
          call sum (call head w) (call prod grad (call head f))
          if call isNil (call tail f)
            then call nil
            else async ρ (call tail f) (call tail w) grad ρ
      ) in
        let gradient ← (call sub label (async call predict features weights bias))
        in call pair
          async routine features weights gradient routine
          gradient
    )
```

Figure 6 Function *train*.

Given a list of *features* f, a list of *weights* w and a *bias* parameter, the *predict* function calculates the classification of the individual, represented by the list of features, by *sum*ming the *bias* with the recursive *sum* of the pair-wise *prod*ucts of the elements in the two lists. To perform the recursive call, we use the *let*-bound function *comp*. All executions of the *comp* function are asynchronous – called using the async primitive, both inside the definition of *comp* (async ρ …) and at the initial invocation (async *comp* …).

The *train*ing function, defined in Figure 6, takes a list of *features*, a list of *weights*, a *label* classifying the individual represented by the features and the *bias* parameter. First, the function (asynchronously) tests whether the weights are already trained to correctly recognise the individual (i.e., the *predict*ion equates the *label*) and, in that case, it returns a pair containing the current *weights* and the *bias*. Otherwise, we define a *let*-bound *routine* function that performs the training by recursively adjusting the weights (w) with respect to the features (f) and a *grad*ient parameter.

The training corresponds to the application of the following mathematical formula, which returns a list of adjusted weights: $routine(f, w, grad) = [w_0 + grad * f_0, \ldots, w_{|w|} + grad * f_{|f|}]$.

Similarly to the *predict* function described above, we use the async primitive to parallelise the execution of the adjustment of the weights. This is done with the initial asynchronous invocation of the *routine* function (async *routine* …) and the asynchronous recursive call within the definition *routine* (async ρ …).

Finally, we first define (through the *let* construct) the *gradient* parameter, which corresponds to the distance (i.e., the *sub*traction) between the *label* and the *predict*ion (also executed asynchronously) and then we return a new *pair* containing the result of the execution of the *routine* function – which returns the adjusted weights – and the calculated *gradient* – representing the new bias of the adjusted weights.

▶ **Example 5** (Training and Prediction)**.** Now that we have defined both the *predict* and *train* functions, we can use them to "emulate" the logical conjunction (where the weights 0 and 1 represent false and true values). First, we introduce in the definition repository the utility

function *trainAndStore* which, given the weights and bias of the model – stored as a pair by a function *wsName* in the definition repository –, some *features*, and a *label*, performs the training of the model and replaces the "old" weights and bias with the new, trained ones.

$$\mathcal{D}(trainAndStore) = \lambda\, wsName.\, \lambda\, fs.\, \lambda\, label.\, (\\ \lambda\, ws.\, (\mathsf{store}\ wsName\ (\mathsf{call}\ train\ fs\ (\mathsf{call}\ first\ ws)\ label\ (\mathsf{call}\ second\ ws))\ ())\ \mathsf{take}\ wsName)$$

To ensure atomicity, *trainAndStore* uses the take primitive to remove the "weights" from the definition environment while it is computing the new ones. In this way, if other clients are trying to access the same weights (name-wise), they are blocked until it finishes computing and it executes the store instruction to "release" them in the definition repository.

We can now train our model to emulate the logical conjunction (*wa* represents the weights and bias of the model) – below we omit $\mathcal{D}$ for compactness.

$$c_0 \blacktriangleleft \mathsf{store}\ wa\ (\mathsf{call}\ pair\ (\mathsf{call}\ cons\ 0\ \mathsf{call}\ cons\ 0\ \mathsf{call}\ nil)\ 1)\ () \\ \mid c_1 \blacktriangleleft \mathsf{call}\ trainAndStore\ wa\ (\mathsf{call}\ pair\ (\mathsf{call}\ cons\ 0\ \mathsf{call}\ cons\ 0\ \mathsf{call}\ nil)\ 0) \\ \mid c_2 \blacktriangleleft \mathsf{call}\ trainAndStore\ wa\ (\mathsf{call}\ pair\ (\mathsf{call}\ cons\ 0\ \mathsf{call}\ cons\ 1\ \mathsf{call}\ nil)\ 0) \\ \mid c_3 \blacktriangleleft \mathsf{call}\ trainAndStore\ wa\ (\mathsf{call}\ pair\ (\mathsf{call}\ cons\ 1\ \mathsf{call}\ cons\ 0\ \mathsf{call}\ nil)\ 0) \\ \mid c_4 \blacktriangleleft \mathsf{call}\ trainAndStore\ wa\ (\mathsf{call}\ pair\ (\mathsf{call}\ cons\ 1\ \mathsf{call}\ cons\ 1\ \mathsf{call}\ nil)\ 1) \\ \mid c_5 \blacktriangleleft \lambda\, w.\, (\mathsf{call}\ predict\ (\mathsf{call}\ cons\ 0\ \mathsf{call}\ cons\ 1\ \mathsf{call}\ nil)\ (\mathsf{call}\ first\ w)\ (\mathsf{call}\ second\ w))\ \mathsf{call}\ wa$$

Above, the running function at the bottom (with future c_5) uses the trained weights – at any possible stage of the training, due to the interleaving of the execution – to *predict* the result of the conjunction of the Boolean values 0 and 1.

In the example above, we showed an initial configuration already featuring some running functions. Indeed, since we consider a reduction semantics, we do not model the invocation of functions from outside the system. To consider also the point of view of the user of the serverless system, one could equip SKC with a labelled semantics supporting both the invocation of functions and the retrieval of the results of the evaluation from outside the system. We leave this direction for future work.

3.2 A Serverless Large Image Tagger

In the following, we illustrate the use of the proposed language abstractions in order to model a simple system for tagging large images. The example takes advantage of an Artificial Intelligence (AI) algorithm to extract semantic content from an image. In computer vision it is common practice to segment the content of a photo to assign a label indicating the nature of the object(s) represented in each segment, for example a person, an animal, or a thing. Although modern AI techniques and in particular deep convolutional neural networks are able to predict the semantic content of an image with extreme accuracy, these algorithms normally take small inputs and are not adequate to classify images at ultra-high resolutions, e.g., the recent 4K format corresponding to 3840×2160 pixels. As a reference, MobileNetV2 [35] is a well-known neural network architecture able to achieve fast object classification with accuracy of around 90% over 3-channel colour image inputs of 224×224 pixels.

With the purpose to build a system for annotating ultra-high resolution images, we want to exploit the parallel execution of inference processes to find the labels associated with each image portion and aggregate them at the end of the single computations. The scenario lends itself well to a serverless deployment strategy and can benefit from the language constructs provided by the SKC language. Summarising, our strategy is to split the image into portions that can be quickly annotated by the AI and to aggregate the results computed for each of these parts. This can be simply rendered in SKC by the *tag* function below:

$$\mathcal{D}(tag) = \lambda\, image.\, (\textsf{call}\ aggregate\ (\textsf{call}\ split\ image))$$

where function *split* splits the image and function *aggregate* classifies each portion and aggregates the results.

We discuss below in detail function *aggregate*, defined in Figure 7, which scatters the asynchronous computations over the fixed-side portions of the original image (as obtained from the *split* function, omitted here) and assembles their results.

```
D(aggregate) =
  λ splits. (call cons
    call pair
      call first (call head splits)
      async (call infer (call second (call head splits)))
    if (call isNil (call tail splits))
    then nil
    else (async (call aggregate (call tail splits)))
  )
```

Figure 7 Function *aggregate*.

In the snippet in Figure 7, we use the functions to manage pairs and lists already existing in the $\mathcal{D}$ repository, defined at the beginning of this section. A function *infer* is also left undefined in the example. In principle, the *infer* function could be any machine learning algorithm that performs the actual recognition task for the image content given its feature set. Notably, we use function *isNil* from the previous section to calculate the condition of the block if and check whether the result list contains further elements. If not, the algorithm terminates and returns the generated list.

The aggregation function takes a vector of splits, parts of the image, and returns a list containing for each portion its identifier and the list of associated labels. An example of the output for an image representing a seascape in the vicinity of a harbour will take the form ((((0, 0), (223, 223)), (house, sea)), (((0, 224), (224, 447)), (house, boat, sea)), ...). Each identifier contains the coordinates of the portion of the image, represented as two pairs of pixel coordinates. For simplicity, we assume that the result of the *split* function is a list of pairs, the first element of each pair contains the coordinates of the portion while the second element contains the actual portion features.

The *infer* function, that computes the single-portion prediction, retrieves a list of strings, which represents the image tags. The *aggregate* function concatenates the result of the prediction of the first portion of the image with the result of the recursive call on the rest of the split list. Note that both the calls of *infer* and of *aggregate* are made using the async construct. This allows for parallel execution. The result is nevertheless deterministic since the asynchronous functions are independent.

The example presented is intuitive but complex enough to show a highly distributed behaviour. The serverless deployment scenario takes advantage of the constructs for asynchronous execution and helps the programmer to think about the system in a compositional and holistic way.

4 Background: the π-calculus

In this section we introduce the syntax and semantics of π-calculus, and some elements of its theory, to be used in Section 5 to define and prove the correctness of the encoding of SKC into the π-calculus itself. In the encoding, we actually use the *asynchronous* π-calculus, as normal when encoding functional languages, though we usually still call it π-calculus. The asynchronous π-calculus has some striking algebraic properties that fail in the ordinary π-calculus [3], and that will be important in our study. The encoding of a λ-term will be parametric on a name. Such parametric processes are called *abstractions*. The actual instantiation of the parameters of an abstraction F is done via the *application* construct $F\langle\tilde{a}\rangle$. Processes and abstractions form the set of π*-agents* (or simply *agents*). Small letters $a, b, \ldots, x, y, \ldots$ range over the infinite set of names and $\tilde{a}$ denotes a tuple of such names. The grammar of the π-calculus is thus:

$$
\begin{array}{llll}
\text{Agents} & A & := & P \;\big|\; F \\[1em]
\text{Processes} & P & := & 0 \;\big|\; a(\tilde{b}).\,P \;\big|\; \overline{a}\langle\tilde{b}\rangle.\,P \;\big|\; \boldsymbol{\nu} a\, P \\
 & & & \;\big|\; P_1 \mid P_2 \;\big|\; !a(\tilde{b}).\,P \;\big|\; F\langle\tilde{a}\rangle \\[1em]
\text{Abstractions} & F & := & (\tilde{a})\; P
\end{array}
$$

In the grammar for processes, 0 is the inactive process. An input-prefixed process $a(\tilde{b}).\,P$, where $\tilde{b}$ has pairwise distinct components, waits for a tuple of names $\tilde{c}$ to be sent along a and then behaves like $P\{\tilde{c}/\tilde{b}\}$, where $\{\tilde{c}/\tilde{b}\}$ is the simultaneous substitution of names $\tilde{b}$ with names $\tilde{c}$. An output particle $\overline{a}\langle\tilde{b}\rangle$ emits names $\tilde{b}$ at a. Parallel composition allows one to run two processes in parallel. The restriction $\boldsymbol{\nu} a\, P$ makes name a local, or private, to P. A replication $!a(\tilde{x}).\,P$ stands for a countable infinite amount of copies of $a(\tilde{x}).\,P$ in parallel.

When the tuple $\tilde{b}$ is empty, the surrounding brackets in prefixes will be omitted. We abbreviate $\boldsymbol{\nu} a \boldsymbol{\nu} b\, P$ as $(\boldsymbol{\nu} a, b)P$. An input prefix $a(\tilde{b}).\,P$, a restriction $\boldsymbol{\nu} b\, P$, and an abstraction $(\tilde{b})\; P$ are binders for names $\tilde{b}$ and b, respectively, and give rise in the expected way to the definition of *free names* (`fn(-)`) and *bound names* (`bn(-)`) of a term or a prefix, and α-conversion.

Since the calculus is polyadic, a type system is needed to avoid disagreements in the arities of the tuples of names carried by a given name and in applications of abstractions. We will not present the typing system, however, because not really essential. A *context* E of π is a π-agent in which some subterms have been replaced by the hole $[\cdot]$; then $E[A]$ is the agent resulting from replacing the hole with the term A. Of course it is assumed that, under a type system, we only relate (e.g., using barbed congruence, described later on) agents obeying the same typing, and then we insert them only in contexts that respect such a typing. We assign parallel composition the lowest precedence among the operators.

The operational semantics of the π-calculus is standard [41] and given in Figure 8. Transitions are of the form $P \xrightarrow{a(\tilde{b})} P$ (an input, $\tilde{b}$ are the bound names of the input prefix that has been fired), $P \xrightarrow{\boldsymbol{\nu}\tilde{d}\,\overline{a}\langle\tilde{b}\rangle} P'$ (an output, where $\tilde{d} \subseteq \tilde{b}$ are private names extruded in the output), and $P \xrightarrow{\tau} P'$ (an internal action). As usual, $\Longrightarrow$ is the reflexive and transitive closure of $\xrightarrow{\tau}$, and $\stackrel{\mu}{\Longrightarrow}$ is $\Longrightarrow\xrightarrow{\mu}\Longrightarrow$.

The reference behavioural equivalence for π-calculus will be the usual *barbed congruence* [28]. We recall its definition. We write $P \Downarrow_a$ if $P \stackrel{\mu}{\Longrightarrow} P'$, for some P' and μ is an output action at a. (We make only output observable because this is standard in asynchronous calculi.)

$$\frac{}{a(\tilde{b}).\, P \xrightarrow{a(\tilde{b})} P} \qquad \frac{}{!a(\tilde{b}).\, P \xrightarrow{a(\tilde{b})} !a(\tilde{b}).\, P \mid P} \qquad \frac{}{\overline{a}\langle\tilde{b}\rangle.\, P \xrightarrow{\overline{a}\langle\tilde{b}\rangle} P}$$

$$\frac{P \xrightarrow{\boldsymbol{\nu}\tilde{d}\,\overline{a}\langle\tilde{b}\rangle} P' \quad n \in \tilde{b} - \tilde{d}}{\boldsymbol{\nu}n\, P \xrightarrow{(\boldsymbol{\nu}n,\tilde{d})\overline{a}\langle\tilde{b}\rangle} P} \qquad \frac{P \xrightarrow{\mu} P' \quad n \notin \mu}{\boldsymbol{\nu}n\, P \xrightarrow{\mu} \boldsymbol{\nu}n\, P'} \qquad \frac{P \xrightarrow{a(\tilde{b})} P' \qquad Q \xrightarrow{\boldsymbol{\nu}\tilde{d}\,\overline{a}\langle\tilde{b'}\rangle} Q'}{P \mid Q \xrightarrow{\tau} \boldsymbol{\nu}\tilde{d}\,(P'\{\tilde{b'}/\tilde{b}\} \mid Q')}$$

$$\frac{P \xrightarrow{\mu} P' \quad \mathtt{bn}(\mu) \cap \mathtt{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{\mu} P' \mid Q} \qquad \frac{P\{\tilde{b}/\tilde{a}\} \xrightarrow{\mu} P'}{((\tilde{a})\ P)\langle\tilde{b}\rangle \xrightarrow{\mu} P'}$$

Figure 8 Labelled Transition Semantics for the π-calculus.

▶ **Definition 6** (Barbed congruence). Barbed bisimilarity *is the largest symmetric relation* $\dot{\simeq}$ *on* π*-calculus processes such that* $P \;\dot{\simeq}\; Q$ *implies:*

1. *If* $P \Longrightarrow P'$ *then there is* Q' *such that* $Q \Longrightarrow Q'$ *and* $P' \;\dot{\simeq}\; Q'$.
2. $P \Downarrow_a$ *iff* $Q \Downarrow_a$.

Processes P *and* Q *are* barbed congruent*, written* $P \approx Q$*, if for each context* E*, it holds that* $E[P] \;\dot{\simeq}\; E[Q]$.

As explained in the description of the encoding, we will need capability types [33] for the π-calculus (or their ω-receptive refinement [38]), so to allow us to use the following replication theorem in the correctness proof of the encoding.

▶ **Theorem 7** (Replication theorem). *Suppose only the output capability of* x *can be communicated (or that* x *is* ω*-receptive); then we have:*

- $\boldsymbol{\nu}x\,(P_1 \mid P_2 \mid !x(\tilde{u}).\, R) \approx \boldsymbol{\nu}x\,(P_1 \mid !x(\tilde{u}).\, R) \mid \boldsymbol{\nu}x\,(P_2 \mid !x(\tilde{u}).\, R)$;
- $\boldsymbol{\nu}x\,(!P \mid !x(\tilde{u}).\, R) \approx !\boldsymbol{\nu}x\,(P \mid !x(\tilde{u}).\, R)$.

5 Encoding SKC into π-calculus

In this section, based on Milner's encoding of call-by-value λ-calculus [26], we study an encoding of SKC into π-calculus. We follow the investigation of the correctness of the representation of pure functions in π-calculus, in particular [26, 37].

We assume a base type system in SKC that at least distinguishes *function values* such as $\lambda x.\, M$ from *nominal values* such as f (of course, depending on how the type system for SKC is defined, function values and nominal values could be collections of types; we do not go into the details of the types as they would obscure the readability of the encoding and its correctness proofs). Moreover, the type system ensures us that whenever a nominal value f is used with the store construct then the repository is not defined on f (thus the replacement of an element of the repository needs a take and then a store).

The encoding is presented in Figure 9. We have to distinguish the encoding of functions and values, that returns an abstraction, from the encoding of the other syntactic categories, that returns a process. We use $[\![-]\!]$ for the former, and $[\![-]\!]^*$ for the latter.

In the encoding of functions and values, the parameter may be thought of as the *location* of that term. A term that becomes a value signals so at its location name and provides access to the body of the value. Such body is replicated and thus may be copied several times. When the value is a λ-function, it receives two names: (the access to) its value-argument, and the location for the resulting term.

A repository of terms is modelled as the parallel composition ($\prod$ denotes indeed n-ary parallel composition) of the encodings of the individual terms, in which a function name f is used to obtain access to the location of the λ-term referred to by f. The imperative nature of the repository is reflected in the reference-like usage of function names. A repository in which f is assigned to M becomes, in the π-calculus, an output $\overline{f}\langle a\rangle$ where a is a fresh name that gives access to (the location of) M. This also explains the encoding of the constructs $\mathsf{take}\ f$ and $\mathsf{call}\ f$, in which the current content of f is read (as an input). Only in $\mathsf{call}\ f$ such content is then re-emitted; $\mathsf{take}\ f$ does not, as it is supposed to remove f from the repository. Construct $\mathsf{store}\ f\ N\ M$ introduces the appropriate output at f, so to add f to the repository.

The encodings use different kinds of names: *location* names, ranged over by p, q; *value* names, ranged over by x, y, for accessing the body of a value; *function* names, ranged over by f, for accessing functions in the repository; *future* names, ranged over by c; *support* names, ranged over by a, b, used to access terms in the repository. In the encoding, as well as in the syntax of SKC, h ranges over value and function names. We assume a type system in the π-calculus in which these kinds of names are separated. Depending on the types of SKC, these kinds may correspond to collections of π-calculus types (we recall that the encoding of pure untyped λ-calculus can be refined to an encoding of typed λ-calculus [41]). A sorting system, à la Milner [25], may be used to keep track of the arities of the names. However we need at least to impose I/O capabilities [33] to make sure that only the output capability of the value and support names is communicated. This is essential for the correctness results below. The typing could however be more precise, by stipulating that function names should follow the discipline of *reference* names of the π-calculus [18] (precisely, a destructive variant in which at any time at most one output at one of these names is available). Similarly, a more precise typing would set value and support names as ω*-receptive* names [38]. For proving more refined correctness properties on the encoding, or using the encoding to validate program transformations or optimisations (e.g., [39, 40, 9]), such refined types would have to be taken into account.

In the encodings of store and async, a τ prefix is added so to make sure that a reduction in SKC corresponds to at least one reduction in π-calculus. This is also needed for Lemma 8. The encodings are extended to contexts and evaluation contexts, in the expected manner, exploiting the compositionality of the encoding.

Lemma 8 shows that only the translation of evaluation contexts yields evaluation contexts in the π-calculus. An occurrence of a term in a π-calculus expression is *unguarded* if that occurrence is not underneath a prefix.

▶ **Lemma 8.** *For any function context E, the hole of the π-calculus context $[\![E]\!]$ is unguarded iff E is an evaluation context.*

▶ **Lemma 9.** *For all M, V, we have: $[\![(\lambda x.\, M)V]\!] \approx [\![M\{^V\!/\!_x\}]\!]$.*

Proof. We follow a case analysis on V. The case when $V = x$ or $V = f$ is easy. When V is an abstraction, we proceed by induction on M, exploiting the Replication Theorem 7 (here, the output capability on names x, y is essential).

The induction is similar to that in the proof of validity of β-reduction for the pure call-by-value λ-calculus [36], the main difference is that there are more cases to consider, however the reasoning is analogous. ◀

▶ **Theorem 10** (Operational correspondence, from SKC to π). *If $C \longrightarrow C'$ then $[\![C]\!]^* \longrightarrow\approx [\![C']\!]^*$.*

Encoding of definition repositories

$$[\![D]\!]^* \stackrel{\text{def}}{=} \Pi_{(f,M)\in D} \boldsymbol{\nu} a\, (\overline{f}\langle a\rangle \mid\, !a(q).\, [\![M]\!]\langle q\rangle)$$

Encoding of configurations

$$\begin{aligned} [\![\langle S, D\rangle]\!]^* &\stackrel{\text{def}}{=} [\![S]\!]^* \mid [\![D]\!]^* \\ [\![\boldsymbol{\nu} n\, C]\!]^* &\stackrel{\text{def}}{=} \boldsymbol{\nu} n\, [\![C]\!]^* \end{aligned}$$

Encoding of systems

$$\begin{aligned} [\![c \blacktriangleleft M]\!]^* &\stackrel{\text{def}}{=} \boldsymbol{\nu} p\, ([\![M]\!]\langle p\rangle \mid p(y).\, !c(z).\, \overline{z}\langle y\rangle) \\ [\![S \mid S']\!]^* &\stackrel{\text{def}}{=} [\![S]\!]^* \mid [\![S']\!]^* \\ [\![\boldsymbol{\nu} c\, S]\!]^* &\stackrel{\text{def}}{=} \boldsymbol{\nu} c\, [\![S]\!]^* \\ [\![0]\!]^* &\stackrel{\text{def}}{=} 0 \end{aligned}$$

Encoding of functions

$$\begin{aligned} [\![M N]\!] &\stackrel{\text{def}}{=} (p)\ \boldsymbol{\nu} q\, \big([\![M]\!]\langle q\rangle \mid q(y).\, \boldsymbol{\nu} r\, ([\![N]\!]\langle r\rangle \mid r(w).\, \overline{y}\langle w, p\rangle)\big) \\ [\![\mathsf{call}\ h]\!] &\stackrel{\text{def}}{=} (p)\ (h(a).\, (\overline{h}\langle a\rangle \mid \overline{a}\langle p\rangle)) \\ [\![\mathsf{async}\ M]\!] &\stackrel{\text{def}}{=} (p)\ \tau.\, \boldsymbol{\nu} c\, ([\![c]\!]\langle p\rangle \mid [\![c \blacktriangleleft M]\!]^*) \\ [\![\mathsf{store}\ h\ N\ M]\!] &\stackrel{\text{def}}{=} (p)\ \tau.\, ([\![M]\!]\langle p\rangle \mid \boldsymbol{\nu} a\, (\overline{h}\langle a\rangle \mid\, !a(q).\, [\![N]\!]\langle q\rangle)) \\ [\![\mathsf{take}\ h]\!] &\stackrel{\text{def}}{=} (p)\ h(y).\, \overline{y}\langle p\rangle \\ [\![\boldsymbol{\nu} f\ M]\!] &\stackrel{\text{def}}{=} (p)\ \boldsymbol{\nu} f\, [\![M]\!]\langle p\rangle \\ [\![c]\!] &\stackrel{\text{def}}{=} (p)\ \overline{c}\langle p\rangle \end{aligned}$$

Encoding of values

$$\begin{aligned} [\![\lambda x.\, M]\!] &\stackrel{\text{def}}{=} (p)\ \boldsymbol{\nu} y\, \overline{p}\langle y\rangle.\, !y(x, q).\, [\![M]\!]\langle q\rangle \\ [\![h]\!] &\stackrel{\text{def}}{=} (p)\ \overline{p}\langle h\rangle \end{aligned}$$

Figure 9 The encoding of SKC into the π-calculus.

Proof. We proceed by rule induction. The inductive part follows from the compositionality of the encoding. For the base case, we make a case analysis on the rule applied.

- Rule β. We use Lemma 9.
- Rule ASYNC. We use the definition of the encoding and Lemma 8.
- Rule PUSH. We have (omitting the store, which does not contribute)

$$\mathcal{E}[c] \mid c \blacktriangleleft V \longrightarrow \mathcal{E}[V] \mid c \blacktriangleleft V$$

We assume V is an abstraction (the case when it is a function name is simpler); we then abbreviate the encoding of an abstraction $\lambda x.\, M$ as

$$[\![\lambda x.\, M]\!]\langle p\rangle = \boldsymbol{\nu} y\,(\overline{p}\langle y\rangle \mid [\![M]\!]^y_{\mathrm{V}})$$

In the encoding, we have, for some p::

$$\begin{array}{rcl} [\![\mathcal{E}[c] \mid c \blacktriangleleft V]\!]^* & = & [\![\mathcal{E}]\!][\overline{c}\langle p\rangle] \mid (\boldsymbol{\nu} q, y\,)(\overline{q}\langle y\rangle \mid [\![V]\!]^y_{\mathrm{V}} \mid q(y).\, !c(z).\, \overline{z}\langle y\rangle) \\ & \longrightarrow\equiv & [\![\mathcal{E}]\!][\overline{c}\langle p\rangle] \mid \boldsymbol{\nu} y\,([\![V]\!]^y_{\mathrm{V}} \mid !c(z).\, \overline{z}\langle y\rangle) \\ & \approx & \boldsymbol{\nu} y\,([\![\mathcal{E}]\!][\overline{p}\langle y\rangle] \mid [\![V]\!]^y_{\mathrm{V}} \mid !c(z).\, \overline{z}\langle y\rangle) \\ & \approx & [\![\mathcal{E}]\!][\boldsymbol{\nu} y\,(\overline{p}\langle y\rangle \mid [\![V]\!]^y_{\mathrm{V}})] \mid \boldsymbol{\nu} y\,([\![V]\!]^y_{\mathrm{V}} \mid !c(z).\, \overline{z}\langle y\rangle) \\ & \approx & [\![\mathcal{E}[V]]\!]^* \mid [\![c \blacktriangleleft V]\!]^* \end{array}$$

where in the second use of $\approx$ we exploit the Replication Theorem 7 (i.e., the output capability constraint on y).

- Rule STORE. The rule is

$$\langle \mathcal{E}[\mathsf{store}\ f\ N\ M], \mathcal{D}\rangle \longrightarrow \langle \mathcal{E}[M], \mathcal{D}[f \mapsto N]\rangle$$

We have, for some Q that does not contain f, and some p:

$$\begin{array}{rcl} [\![\langle \mathcal{E}[\mathsf{store}\ f\ N\ M], \mathcal{D}\rangle]\!]^* & = & [\![\mathcal{E}]\!]^*[\tau.\, ([\![M]\!]\langle p\rangle \mid \boldsymbol{\nu} a\,(\overline{f}\langle a\rangle \mid !a(q).\, [\![N]\!]\langle q\rangle))] \mid Q \\ & \longrightarrow & [\![\mathcal{E}]\!]^*[([\![M]\!]\langle p\rangle \mid \boldsymbol{\nu} a\,(\overline{f}\langle a\rangle \mid !a(q).\, [\![N]\!]\langle q\rangle))] \mid Q \\ & \approx & [\![\mathcal{E}]\!]^*[([\![M]\!]\langle p\rangle] \mid \boldsymbol{\nu} a\,(\overline{f}\langle a\rangle \mid !a(q).\, [\![N]\!]\langle q\rangle)) \mid Q \\ & = & [\![\langle \mathcal{E}[M], \mathcal{D}[f \mapsto N]\rangle]\!]^* \end{array}$$

- Rule CALL. The rule is

$$\langle \mathcal{E}[\mathsf{call}\ f], \mathcal{D}\rangle \longrightarrow \langle \mathcal{E}[M], \mathcal{D}\rangle$$

for $M = \mathcal{D}(f)$. We have, for some Q that does not contain f, and some p:

$$\begin{array}{rcl} [\![\langle \mathcal{E}[\mathsf{call}\ f], \mathcal{D}\rangle]\!] & = & [\![\mathcal{E}]\!]^*[f(a').\, (\overline{f}\langle a'\rangle \mid \overline{a'}\langle p\rangle)] \mid \boldsymbol{\nu} a\,(\overline{f}\langle a\rangle \mid !a(q).\, [\![M]\!]\langle q\rangle \mid Q \\ & \longrightarrow & \boldsymbol{\nu} a\,([\![\mathcal{E}]\!]^*[\overline{f}\langle a\rangle \mid \overline{a}\langle p\rangle] \mid !a(q).\, [\![M]\!]\langle q\rangle \mid Q) \\ & \approx & \boldsymbol{\nu} a\,([\![\mathcal{E}]\!]^*[\overline{f}\langle a\rangle] \mid [\![M]\!]\langle p\rangle \mid !a(q).\, [\![M]\!]\langle q\rangle \mid Q) \\ & \approx & [\![\mathcal{E}]\!]^*[[\![M]\!]\langle p\rangle] \mid \boldsymbol{\nu} a\,(\overline{f}\langle a\rangle \mid !a(q).\, [\![M]\!]\langle q\rangle \mid Q) \\ & = & [\![\langle \mathcal{E}[M], \mathcal{D}\rangle]\!]^* \end{array}$$

where in the first application of $\approx$ we exploit the τ-insensitiveness property of the name a (equivalently, the Replication Theorem 7 on a).

- The case of rule TAKE is easy. ◀

The converse direction is more delicate because the π-calculus encoding presents a number of administrative reductions, which do not correspond to actual reductions in the source calculus. We overcome the problem following [11], which presents an optimisation of Milner's encoding of application (and hence of the encoding of SKC). The optimised encoding, indicated as $\{\![-]\!\}$ below, is obtained from the initial one by performing a few (deterministic) reductions, at the price of a more complex definition. These reductions are performed in the clause for application, when at least one of the two involved terms is a value. We only show the definition for the different clauses. In the last clause below it is intended that N is not a value. The three clauses are applied only on applications MN in which both M and N are closed (i.e., they have no free λ-variables); if they are applicable they have priority over the clauses of the initial encoding $[\![-]\!]$. Moreover the first clause has priority over the last one.

$$\begin{array}{rcl} \{\![(\lambda x.\, M)(\lambda z.\, N)]\!\} & \stackrel{\text{def}}{=} & (p)\ \boldsymbol{\nu} y\,(!y(x,q).\, \{\![M]\!\}\langle q\rangle \mid !y'(z,r).\, \{\![N]\!\}\langle r\rangle \mid \overline{y}\langle y', p\rangle) \\ \{\![(\lambda x.\, M)h]\!\} & \stackrel{\text{def}}{=} & (p)\ \boldsymbol{\nu} y\,(!y(x,q).\, \{\![M]\!\}\langle q\rangle \mid \overline{y}\langle h, p\rangle) \\ \{\![(\lambda x.\, M)N]\!\} & \stackrel{\text{def}}{=} & (p)\ \boldsymbol{\nu} y\,(!y(x,q).\, \{\![M]\!\}\langle q\rangle \mid \{\![N]\!\}\langle r\rangle \mid r(w).\, \overline{y}\langle w, p\rangle) \end{array}$$

The following lemma establishes the correctness of the optimised encoding.

▶ **Lemma 11.** *For any C, we have $[\![C]\!]^* \approx \{\![C]\!\}^*$.*

We state the converse of Theorem 10 on the optimised encoding $\{\![-]\!\}$.

▶ **Theorem 12** (Operational correspondence, from π to SKC)**.** *If $\{\![C]\!\}^* \longrightarrow P$ then there is C' with $C \longrightarrow C'$ and $P \approx [\![C']\!]^*$.*

Proof. Using a case analysis similar to that in the proof of Theorem 10, taking on applications, when applicable, the clauses of the optimised encoding. ◀

Both Theorems 10 and 12 can be extended to multi-step reductions (i.e., relation $\Longrightarrow$ in place of the one-step reduction $\longrightarrow$).

6 Discussion and Conclusion

In this work, we explore the SKC serverless calculus under two aspects: its design space (the language layer) and its usefulness in reasoning on serverless implementations (the implementation layer).

Towards the first aim, we consider features from process calculi to represent the concurrent execution of functions and to better integrate the essential features of the two calculi that inspire SKC: the λ- and the π-calculus. Part of that redesign consists in having functions parametric on the names of other functions. That inclusion allows us to create new function names for the function-definition repository dynamically, so the repository of available functions can grow freely at runtime, as well as holding private function definitions and private stored data. As part of that exploration, we present a refined syntax and semantics for SKC, supporting the aforementioned improvements, and we illustrate how those new features enhance the expressiveness of the language through small examples and two use cases from artificial intelligence.

Then, to illustrate how SKC supports reasoning on the implementation layer, we focus on how to translate SKC-defined architectures (of functions) into a network of communicating, concurrent, and mobile processes, representing the actual network of concurrent processes that run those serverless functions in the cloud. We tackle this second task by presenting an encoding from SKC to the asynchronous π-calculus and proving it correct in terms of an operational correspondence result, using standard techniques – whose side-effect is to cast a good outlook on the affordability of extending results from the literature to SKC.

We remark that the choice of using futures as one of the main building blocks of SKC comes from a careful consideration on the minimality of the language. Indeed, if one might consider named channels (as in CCS/π-calculus [24, 41]) a standard choice of communicating systems, in the case of SKC they would increase the distance between the language and concrete serverless implementations. For example, with channels one could have re-usable, bi-directional communication between functions, which is a feature no serverless implementation currently provides. Moreover, our encoding shows a way to define futures using channels while the opposite is also possible [31].

Regarding related work, the proposal closest to SKC is [20], where a calculus more involved than SKC is presented. It captures the low-level details of current serverless implementations (e.g., cold/warm components, storage, and transactions are primitive features of their model), essentially mixing the language and implementation layers. Contrarily, SKC strives to be a kernel model of serverless computing, with the suggested strategy to reason on implementations via encodings. Another work close to SKC is [31], where the authors introduce a λ-calculus with futures. Since the aim of [31] is to formalise and reason on

a concurrent extension of Standard ML, their calculus is more involved than SKC, as it contains primitive operators (handlers and cells) to capture safe non-deterministic concurrent operations, which we can encode as macros in SKC. An interesting future work is to investigate which results from [20, 31] we can adapt to SKC.

Other future directions of research on SKC include the exploration of guarantees on sequential execution across functions, which compels the investigation of new tools to enforce sequential consistency [23] or serialisability [32] of the transformations of the global state [17]. That challenge can be tackled developing static analysis techniques and type disciplines [19, 1] for SKC. Another direction concerns programming models, which should give to programmers an overview of the overall logic of the distributed functions and capture the loosely-consistent execution model of serverless [17]. Choreographic Programming [29, 7] is a promising candidate for that task, as choreographies are designed to capture the global interactions in distributed systems [22], and recent results [6, 8, 15, 16] confirmed their applicability to microservices [10], a neighbouring domain to that of serverless architectures. Such an approach can also cover Edge and Internet-of-Things scenarios (as targeted by projects like AWS Greengrass[1]), using language abstractions borrowed from the world of microservice systems [14]. The language extensions and results on the encoding presented in this paper are stepping stones for a transformation framework between serverless and microservice architectures. There, the final goal would be to provide an infrastructure where, depending on the application context (e.g., the amount of stateful interactions) and inbound load (steadier traffic benefit the always-on microservices deployment, while serverless is more efficient when considering traffic bursts), users (or automatic optimisation systems) can decide whether to deploy a given architecture (or part of it) as a network of serverless functions or microservices. Indeed, if SKC is the model for serverless and π-calculus-inspired process calculi [30] represent microservices, our encoding is a first result towards a framework for the semantic-preserving transition between the two implementation/deployment paradigms.

A clearer understanding of the implementation layer also provides foundations for tackling some outstanding questions in serverless computing. For example, predicting resource usage and costs is challenging in general, since it requires knowing how functions are executed by the implementation layer. This last one is particularly relevant in the per-usage model of serverless architectures, yet it requires to extend SKC with an explicit notion of time to support quantitative behavioural reasoning for timed systems [5, 4]. A starting point could be to use the encoding into π-calculus to prove termination properties of the source SKC language, combining type techniques from functional and concurrent languages [9].

References

1 Davide Ancona et al. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3(2-3):95–230, 2016.

2 Ioana Baldini, Paul C. Castro, Kerry Shih-Ping Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. Serverless computing: Current trends and open problems. In Sanjay Chaudhary, Gaurav Somani, and Rajkumar Buyya, editors, *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017. doi:10.1007/978-981-10-5026-8_1.

3 Michele Boreale and Davide Sangiorgi. Some congruence properties for π-calculus bisimilarities. *Theor. Computer Science*, 198:159–176, 1998.

[1] https://aws.amazon.com/greengrass/

4 Tomasz Brengos and Marco Peressotti. A uniform framework for timed automata. In *CONCUR*, volume 59 of *LIPIcs*, pages 26:1–26:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

5 Tomasz Brengos and Marco Peressotti. Behavioural equivalences for timed systems. *Logical Methods in Computer Science*, 15(1), 2019.

6 Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *POPL*, pages 263–274. ACM, 2013.

7 Luís Cruz-Filipe and Fabrizio Montesi. A core model for choreographic programming. In *FACS*, Lecture Notes in Computer Science, pages 17–35. Springer, 2016.

8 Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies: Theory and implementation. *Logical Methods in Computer Science*, 13(2), 2017.

9 Romain Demangeon, Daniel Hirschkoff, and Davide Sangiorgi. Termination in impure concurrent languages. In *Proc. 21th Conf. on Concurrency Theory*, volume 6269 of *Lecture Notes in Computer Science*, pages 328–342. Springer, 2010.

10 Nicola Dragoni et al. Microservices: Yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.

11 Adrien Durier, Daniel Hirschkoff, and Davide Sangiorgi. Eager functions as processes. In *33nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018*. IEEE Computer Society, 2018.

12 Yoav Freund and Robert E Schapire. Large margin classification using the perceptron algorithm. *Machine learning*, 37(3):277–296, 1999.

13 Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, Fabrizio Montesi, Marco Peressotti, and Stefano Pio Zingaro. No more, no less - A formal model for serverless computing. In *Coordination Models and Languages - 21st International Conference, COORDINATION 2019*, volume 11533 of *Lecture Notes in Computer Science*, pages 148–157. Springer, 2019.

14 Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Stefano Pio Zingaro. A language-based approach for interoperability of IoT platforms. In Tung Bui, editor, *51st Hawaii International Conference on System Sciences, HICSS 2018, Hilton Waikoloa Village, Hawaii, USA, January 3-6, 2018*, pages 1–10. ScholarSpace / AIS Electronic Library (AISeL), 2018. URL: `http://hdl.handle.net/10125/50603`.

15 Saverio Giallorenzo, Fabrizio Montesi, and Maurizio Gabbrielli. Applied choreographies. In *FORTE*, Lecture Notes in Computer Science, pages 21–40. Springer, 2018.

16 Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. Choreographies as objects. *CoRR*, abs/2005.09520, 2020. `arXiv:2005.09520`.

17 Joseph M. Hellerstein et al. Serverless computing: One step forward, two steps back. In *CIDR*. www.cidrdb.org, 2019.

18 Daniel Hirschkoff, Enguerrand Prebet, and Davide Sangiorgi. On the representation of references in the pi-calculus. In *CONCUR'20*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2020.

19 Hans Hüttel et al. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016.

20 Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. Formal foundations of serverless computing. *Proc. ACM Program. Lang.*, 3(OOPSLA):149:1–149:26, 2019. `doi:10.1145/3360575`.

21 Eric Jonas et al. Cloud programming simplified: A Berkeley view on serverless computing. Technical report, EECS Department, University of California, Berkeley, February 2019.

22 Nickolas Kavantzas, David Burdett, Gregory Ritzinger, Tony Fletcher, Charlton Barreto, and Yves Lafon. Web services choreography description language version 1.0, W3C candidate recommendation. Technical report, W3C, 2005. URL: `http://www.w3.org/TR/ws-cdl-10`.

23 Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

24 Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

25 Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, LFCS, Dept. of Comp. Sci., Edinburgh Univ., October 1991. Also in *Logic and Algebra of Specification*, ed. F.L. Bauer, W. Brauer and H. Schwichtenberg, Springer Verlag, 1993.

26 Robin Milner. Functions as processes. *Math. Struct. in Computer Science*, 2(2):119–141, 1992.

27 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992. doi:10.1016/0890-5401(92)90008-4.

28 Robin Milner and Davide Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer, 1992.

29 Fabrizio Montesi. *Choreographic Programming*. Ph.D. Thesis, IT University of Copenhagen, 2013. http://www.fabriziomontesi.com/files/choreographic_programming.pdf.

30 Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-oriented programming with Jolie. In Athman Bouguettaya, Quan Z. Sheng, and Florian Daniel, editors, *Web Services Foundations*, pages 81–107. Springer, 2014. doi:10.1007/978-1-4614-7518-7_4.

31 Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. *Theor. Comput. Sci.*, 364(3):338–356, 2006.

32 Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

33 Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Math. Struct. in Computer Science*, 6(5):409–454, 1996.

34 Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

35 Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4510–4520. IEEE, 2018. doi:10.1109/CVPR.2018.00474.

36 Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis CST–99–93, Department of Computer Science, University of Edinburgh, 1992.

37 Davide Sangiorgi. An investigation into functions as processes. In *Proc. Math. Foundations of Programming Semantics*, volume 802 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 1993.

38 Davide Sangiorgi. The name discipline of uniform receptiveness. *Theoretical Computer Science*, 221:457–493, 1999.

39 Davide Sangiorgi. Typed π-calculus at work: a correctness proof of Jones's parallelisation transformation on concurrent objects. *Theory and Practice of Object Systems*, 5(1):25–34, 1999.

40 Davide Sangiorgi. Termination of processes. *Math. Struct. in Computer Science*, 16(1):1–39, 2006.

41 Davide Sangiorgi and David Walker. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press, 2001.

A Logic Programming Approach to Reaction Systems

Moreno Falaschi

Department of Information Engineering and Mathematics, University of Siena, Italy
https://www3.diism.unisi.it/people/person.php?id=485
moreno.falaschi@unisi.it

Giulia Palma

Department of Computer Science, University of Pisa, Italy
giuliapalma29@gmail.com

Abstract

Reaction systems (RS) are a computational framework inspired by the functioning of living cells, suitable to model the main mechanisms of biochemical reactions. RS have shown to be useful also for computer science applications, e.g. to model circuits or transition systems. Since their introduction about 10 years ago, RS matured into a fruitful and dynamically evolving research area. They have become a popular novel model of interactive computation. RS can be seen as a rewriting system interacting with the environment represented by the context. RS pose some problems of implementation, as it is a relatively recent computation model, and several extensions of the basic model have been designed. In this paper we present some preliminary work on how to implement this formalism in a logic programming language (Prolog). To the best of our knowledge this is the first approach to RS in logic programming. Our prototypical implementation does not aim to be highly performing, but has the advantage of being high level and easily modifiable. So it is suitable as a rapid prototyping tool for implementing several extensions of reaction systems in the literature as well as new ones. We also make a preliminary implementation of a kind of memoization mechanism for stopping potentially infinite and repetitive computations. Then we show how to implement in our interpreter an extension of RS for modeling a nondeterministic context and interaction between components of a (biological) system. We then present an extension of the interpreter for implementing the recently introduced networks of RS.

2012 ACM Subject Classification Theory of computation → Semantics and reasoning; Computing methodologies → Symbolic calculus algorithms; Software and its engineering → Interpreters

Keywords and phrases reaction systems, logic programming, non deterministic context

Digital Object Identifier 10.4230/OASIcs.Gabbrielli.2020.6

Acknowledgements We thank the anonymous reviewers for their detailed and very useful criticisms and recommendations that helped us to improve our paper.

1 Introduction

Natural Computing is an area of research which has two main aspects: human designed computing (models and computational techniques) inspired by nature and computation taking place in nature (i.e. it also investigates processes taking place in nature in terms of information processing). The first strand of research is quite well-established. This paper falls into this second strand of research, since it discusses reaction systems which are a formal model for the investigation of the functioning of the living cell introduced by A. Ehrenfeucht and G. Rozenberg [16, 17]. The functioning is viewed in terms of formal processes resulting from interactions between biochemical reactions taking place in the living cell. The basic model of reaction systems abstracts from various (technical) features of biochemical reactions to such an extent that it becomes a qualitative rather than quantitative model [7, 15]. However, it takes into account the basic bioenergetics (flow of energy) of the

© Moreno Falaschi and Giulia Palma;
licensed under Creative Commons License CC-BY
Recent Developments in the Design and Implementation of Programming Languages.
Editors: Frank S. de Boer and Jacopo Mauro; Article No. 6; pp. 6:1–6:15

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

living cell, and it also takes into account that the living cell is an open system (in the sense that it interacts with its environment) and its behavior is influenced by its environment. The main focus of research is on understanding processes that take place in these models. The Reaction Systems model has already been applied and extended successfully to various areas of research, since it is relevant in several different fields, such as computer science, biology, molecular chemistry [20, 21, 9, 3].

In this paper we present our preliminary work on how to implement the framework of RS in a logic programming language (Prolog). To the best of our knowledge this is the first approach to RS in logic programming. We illustrate our program by means of some simple basic examples throughout the paper, and then we consider a more complex example, by modeling a reaction system representing a regulatory network for *lac* operon, presented in [12]. We have also implemented a kind of memoization mechanism for stopping potentially infinite and repetitive computations. We discuss also some extensions of the basic framework and their implementation. First, we discuss how to implement an extension of RS for modeling nondeterministic contexts. Then, we show how to implement two RS which interact between them. Finally we present a prototypical implementation of the recently introduced networks of reaction systems [6]. Our interpreter is freely available online.

Structure of the paper. In Section 2, we present the basic framework of Reaction Systems. Then Section 3 is devoted to describing the implementation of the basic framework. Section 4 presents the implementation of a biological example. In Section 5 we discuss some extensions of RS and the corresponding implementation. We draw some conclusions and discuss future work in Section 6.

1.1 Related work

Reaction systems pose some interesting problems of implementation, as it is a recent computation model, and several extensions of the basic model have been designed. In particular, RS are amenable to both theoretical studies and as a modeling tool for biological processes. Their dynamics occupy an intermediate position between Cellular Automata [19] and Boolean Automata Networks [13]. However, since reaction systems have also been employed to model real-world systems, the availability of fast and efficient simulators is essential for a more widespread use of them as a modeling tool. The first available simulator was *brsim* [1], written in Haskell and it has been the fastest CPU-based simulator available for a relatively long time. Later, a GPU-based approach to the simulation of reaction systems has been explored with *HERESY* in [23], written using CUDA. It has been shown to be the fastest simulator for large-scale systems, due to its ability to exploit the large number of computational units inside GPUs. It also provides a CPU-based simulator written in Python 2, however it is more a "fallback" simulator when GPUs are not available, and is slower than *brsim*. Both simulators employ the same direct simulation method, which is based on the set-theoretic definition of the reaction systems' dynamics. Recently, in [18] the authors provide an optimized Common Lisp simulator, employing the direct simulation method, which is able to offer performances comparable with the GPU-based simulator on a large-scale real-world model, the *ErbB* model. It has been shown to be the fastest CPU-based simulator currently available. They also explore other ways of performing the simulation, in particular, by looking at the graph of dependencies between reactions, it is possible to avoid performing the simulation of parts of the reactions that cannot produce any effect on its dynamics; and by rewriting the dynamical evolution of a reaction system in terms of matrix-vector multiplications, vector additions, and clipping operations, they exploit the existing high-performance linear algebra libraries to

perform the simulation and therefore they use a proof-of-concept implementation employing Python 3 and Numpy. Regarding the non-determinism of the context, some results are illustrated in [9], in which the authors consider the *link*-calculus [8, 11], which allows to model multiparty interaction in concurrent systems, and show that it allows to embed reaction systems, by representing the behaviour of each entity and preserving faithfully their features. Such a framework contribute to increase the expressiveness of reaction systems, indeed it exploits the interaction among different reaction systems. In [10] the authors show how to define a context which can be really expressive, by adding to it a non deterministic and a recursive operator.

In this paper we present some preliminary work on how to implement the formalism of RS in a logic programming language: Prolog. Although this prototypical implementation is not highly performing and competitive compared to the above mentioned implementation, it has the considerable advantage of being very high level and easily modifiable. Therefore, it is suitable as a working rapid prototyping tool for implementing extensions of reaction systems, as we show in this paper.

2 Reaction Systems

Natural Computing is concerned with human-designed computing inspired by nature as well as with computation taking place in nature. The theory of Reaction Systems [16, 7] was born in the field of Natural Computing to model the behavior of biochemical reactions taking place in living cells. The original motivation was to understand interactions of biochemical reactions in the living cell from the natural computing point of view. These interactions are based on mechanisms of *facilitation* and *inhibition*, which underlie the definition of *reaction system*. A *reaction* is a chemical process in which substances act mutually on each other and are changed into different substances, or one substance changes into other substances. A reaction takes place if all its *reactants* are present and none of its *inhibitors* is present. If a reaction takes place, then it creates its *products*. Therefore to specify a reaction one needs to specify its set of *reactants*, its set of *inhibitors* and its set of *products*.

▶ **Definition 1** (Reaction). *A reaction is a triplet $a = (R, I, P)$, where R, I, P are finite sets. If S is a set such that $R, I, P \subseteq S$, then a is a reaction in S.*

The sets R, I, P are also written R_a, I_a, P_a and called the reactant set of a, the inhibitor set of a, and the product set of a, respectively. Also, $R_a \cup I_a$ is the set of the resources of a and $rac(S)$ denotes the set of all reactions in S. Because R and I are non empty, all products are produced from at least one reactant and every reaction can be inhibited in some way. Sometimes artificial inhibitors are used that are never produced by any reaction. For the sake of simplicity, in some examples, we will allow I to be empty.
The effect of a reaction a is conditional: if R_a is present and no element of I_a is present, then P_a is produced. Otherwise, the reaction does not take place, and "nothing" is produced.

▶ **Definition 2** (Result of Reaction). *Let a be a reaction, A a finite set of reactions and T a finite set.*

- *a is enabled by T if $R_a \subseteq T$ and $I_a \cap T = \emptyset$ (indicated by $en_a(T)$);*
- *The result of a on T is defined by:*

$$res_a(T) = \begin{cases} P_a & if \quad en_a(T) \\ \emptyset & otherwise \end{cases}$$

- *The results of A on T is defined by $res_A(T) = \bigcup_{a \in A} res_a(T)$.*

Now that the formal notion of a reaction and its effect on states have been established, we can proceed to define reaction systems, which are an abstract model of the functioning of the living cell. A *reaction system* is essentially a set of reactions. We also specify the background set, which consists of entities needed for defining the reactions and for reasoning about the system.

▶ **Definition 3** (Reaction Systems). *A reaction system, abbreviated rs, is an ordered pair $\mathcal{A} = (S, A)$ such that S is a finite set, and $A \subseteq rac(S)$.*

The set S is called the *background set* of $\mathcal{A}$. Its elements are called *entities*, they represent molecular entities (e. g. atoms, ions, molecules) that may be present in the state of a biochemical system modeled by $\mathcal{A}$. The set A is the set of *reactions* of $\mathcal{A}$. Since S is finite, so is A. All the notations introduced for sets of reactions carry over to reaction systems: $T \subseteq S, en_{\mathcal{A}}(T) = en_A(T); res_{\mathcal{A}}(T) = res_A(T)$; T is active in $\mathcal{A}$, if $en_{\mathcal{A}}(T) \neq \emptyset$. The theory of Reaction Systems is based on the following assumptions:

1. **No permanency.** An entity of a set T vanishes unless it is sustained by a reaction. This reflects the fact that a living cell would die for lack of energy, without chemical reactions.
2. **No counting.** The basic model of reaction systems is very abstract and qualitative, i.e. the quantity of entities that are present in a cell is not taken into account. In fact the number of reagents does not count in the reaction systems model, unlike the stoichiometric equations, in which the quantities are fundamental.
3. **Threshold nature of resources.** From the previous item, we assume that either an entity is available and there is enough of it (i.e. there are no conflicts), or it is not available at all.

The dynamic behavior of reaction systems is captured through the notion of interactive process:

▶ **Definition 4** (Interactive Process). *Let $\mathcal{A} = (S, A)$ be a reaction system. An interactive process in $\mathcal{A}$ is a pair $\pi = (\gamma, \delta)$ of finite sequences such that: $\gamma = C_0, C_1, \ldots, C_n$, $\delta = D_1, \ldots, D_n$, $n \geq 1$, where $C_0, \ldots, C_n, D_1, \ldots, D_n \subseteq S, D_1 = res_{\mathcal{A}}(C_0)$, and $D_i = res_{\mathcal{A}}(D_{i-1} \cup C_{i-1})$ for $2 \leq i \leq n$.*

Living cells are seen as open systems that continuously react with the external environment, in discrete steps. The sequence γ is the *context sequence* of π and represents the influence of the environment on the Reaction System. The sequence δ is the *result sequence* of π and it is entirely determined by γ and A. Note that C_i and D_i do not have to be disjoint. Let $W_0 = C_0$ and $W_i = C_i \cup D_i$ for all $1 \leq i \leq n$. The sequence $W_0, \ldots, W_n$ is the *state sequence of* π, $sts(\pi)$. W_0 is the initial state of π. For each $0 \leq j \leq n$, C_j is the *context* of W_j.

Let us consider a clarifying example that illustrates the concepts introduced in Section 2.

▶ **Example 5.** Let us consider a reaction system $\mathcal{A} = (\{e_1, e_2, e_3, e_4\}, A)$, where A is the set of the two reactions:

$$a_1 = (\underbrace{\{e_1, e_2\}}_{R_1 = \text{ Reactants of } a_1}, \underbrace{\{e_3\}}_{I_1 = \text{ Inhibitors of } a_1}, \underbrace{\{e_2, e_3\}}_{P_1 = \text{ Products of } a_1})$$

$$a_2 = (\underbrace{\{e_1\}}_{R_2 = \text{ Reactants of } a_2}, \underbrace{\{e_4\}}_{I_2 = \text{ Inhibitors of } a_2}, \underbrace{\{e_1, e_4\}}_{P_2 = \text{ Products of } a_2})$$

The sequence $\tau = \{e_2, e_3, e_1, e_4\}, \varnothing$ is a context-indipendent state sequence of $\mathcal{A}$, assuming that the initial state is $T_0 = \{e_1, e_2\}$. Indeed: $R_1 = T_0$ and $T_0 \cap I_1 = \varnothing$, then reaction a_1 takes place, producing P_1; also $R_2 \subset T_0$ and $I_2 \cap T_0 = \varnothing$, then reaction a_2 takes place producing P_2.

Therefore, we get $T_1 = P_2 \cup P_3 = \{e_2, e_3, e_1, e_4\}$. Now, since $I_1 \cap T_1 = \{e_3\} \neq \varnothing$, then reaction a_1 does not take place. Reaction a_2 does not take place either, in fact $I_2 \cap T_1 = \{e_4\} = \varnothing$. Finally, we get $T_2 = \varnothing$.
Let us now assume that the computation is not context-independent. If the context sequence is $\gamma = \{e_1, e_4\}, \{e_4\}$, then the corresponding state sequence is $\tau = \{e_4\}, \varnothing$. Indeed: $G_0 = T_0 \cup C_0 = \{e_1, e_2, e_4\}$. From the fact that $G_0 \cap I_2 = \{e_4\} \neq \varnothing$, we get the reaction a_2 does not take place. Instead reaction a_1 occurs in fact $R_1 \subseteq G_0$ and $I_1 \cap G_0 = \varnothing$. Then we get $T_1 = \{e_2, e_3\}$. Now, we have $G_1 = T_1 \cup \{e_4\} = \{e_2, e_3, e_4\}$. Since $G_1 \cap I_1 = \varnothing$ and $G_1 \cap I_2 = \varnothing$, neither reaction a_1 nor reaction a_2 take place. Therefore, we get $T_2 = \varnothing$.

3 A logic programming approach to Reaction Systems

In this Section we briefly describe a prototypical implementation of the Reaction Systems framework in a logic programming language (Prolog), which is available on-line[1], together with a small manual to use it.

3.1 An Interpreter of Reaction Systems in logic programming

Sets are represented by corresponding lists of values. The background set `S` of a reaction system is represented by a list of distinct constant symbols. A reaction `(R, I, P)` is represented by a triple of lists, where `R` is the list of the reactants, `I` is the list of the inhibitors and `P` is the list of products. The set of reactions in a reaction system is defined by a list of reactions and is introduced by using the predicate `reactionSet/1`. So, this predicate is fundamental and one fact for this predicate must be included. If the computation is context independent `reactionSet/1` is the only predicate for which we have to add a unit fact in the program. If we want to perform a computation context dependent, then we have to add also a unit fact for the other fundamental predicate `context/1`. The predicate `context/1` takes as input a list of context lists. Hence a user has to modify only one, or at most two unit facts to be able to run her reaction system.

3.2 A computation with the interpreter of Reaction Systems

Now we briefly describe some of the main predicates which are part of the interpreter of Reaction Systems.

When evaluating a query to our interpreter, the predicate which needs to be called is `computation(InitialState, ListOfStates)`. The first input argument `InitialState` is the list of the reagents to be put in the initial state. The second argument `ListOfStates` is the list of states which is computed in the reaction system by our interpreter. So, a query to our interpreter consists of a call to `computation/2`.

The execution of the predicate `computation/2` starts by making some preliminary checks (predicate `preliminaryCheck/1`) to verify that the basic assumptions on reaction systems are respected. Namely, for each reaction `(R,I,P)` the set of reagents `R` and the sets of inhibitors `I` are non empty, and they don't share elements. Then, the interpreter will give the user some choices:

1) whether she wants to make a context independent computation or a computation which interacts with the context.

[1] `https://www3.diism.unisi.it/~falaschi/ReactionSystems`

2) whether she wants to make a computation with a limited maximal number of steps, or if the computation should be of a possibly unlimited length.

Then the appropriate predicate corresponding to the choice of the user is selected, between the following four ones:

```
computationLimitedToKStepsContextIndependent/2
computationLimitedToKSteps/2
unlimitedComputationContextIndependent/2
unlimitedComputation/2
```

Notice that `unlimitedComputation/2` and `unlimitedComputationContextIndependent/2` may enter in a loop if there is a reaction `(R,I,P)` in which the same reactant in `R` appears in `P`, or more in general when there are dependencies between the reactants in R and the ones computed by some other reaction. A loop can be stopped by using the inhibition mechanism, or by a memoization mechanism, as explained at the end of this section.

A single step of the computation is performed as follows. The result of a single reaction (without the context) is computed by the predicate `result (T, R, I, P, P1)`. Given the state `T` and the reaction `(R, I, P)`, the result `P1` will be `P` if the reaction is enabled in `T` (that is, if the predicate `enable` is true), otherwise it will return the empty set. The predicate `enable (R, I, T)` checks if the reaction with reactants `R` and inhibitors `I` is enabled in the state `T`. We recall that in a reaction system for a reaction to occur it must hold: $\mathtt{R} \subseteq \mathtt{T}$ and $\mathtt{I} \cap \mathtt{T} = \emptyset$. Then the result of all reactions on T is computed by the predicate `resultallreactions (T, ReactionSet, T1)`, which recursively calls `result/5` and collects the union of its outcomes.

Let us see a trivial example.

▶ **Example 6.** Let us define the predicate `reactionset` as follows:

```
reactionset ([([ e1, e2 ], [ e3 ], [ e2, e3 ]),  ([e1], [ e4 ], [ e1, e4 ])]).
```

This means that there are two reactions in the system. We execute the following query:

```
? - computation ( [ e1, e2], L ).
```

Then, by selecting the modality "computation context independent" we get:

```
L = [ [ e2, e3, e1, e4 ], [ ] ]
```

that is the next state `[ e2, e3, e1, e4 ]` and the final state `[ ]`. The computation in this case uses an empty context represented internally by an empty list.

We now modify the example in order to show the interactive influence of the context. To consider the effect of the context, we need to add a unit fact for the predicate `context/1`. The input argument of this predicate must be a list of context reagent lists. The list in position k corresponds to the context to be added at step k of the computation. For instance: `context([[e1, e2], [e3, e2, e5]])`.

If the context list has length m, and the computation is longer, it continues from step $m+1$ as context independent. We define the predicate that calculates a computation starting from an initial state, returning a list of states. The context is taken into account now. We report here a small fragment of the interpreter. The predicate
`computeWithContext(ComputState,Context,Reactions,ComputStateSequence)` takes in input the current `ComputState`, the `Context` sequence, the list of `Reactions` in the Reaction System, and returns the computed `ComputStateSequence`.

```
unlimitedComputation(InitialState,L):-
     reactionSet(R),context([C0|Cs]),
     union(InitialState,C0,SC),computeWithContext(SC,Cs,R,L).

computeWithContext ([  ], C, R, [ ]).
computeWithContext ([ X | L ], [  ], R, [ S1 | S ]) : -
     resultallreactions ([ X | L ], R, S1),
     computeWithContext (S1,  [  ], R, S).
computeWithContext ([ X | L ], [C | Cs], R, [S1 | S ]) : -
     resultallreactions ([ X | L ], R, S1),
     union (S1, C, S2), computeWithContext (S2, Cs, R, S).
```

Let us see a simple example.

▶ **Example 7.** Let us define the context and the reactions of a reaction system as follows:

```
context([ [e1, e2], [e3, e2, e5] ]).
reactionSet([([e1,e2],[e3],[e2,e3]),  ([e5], [e4], [e1,e4])]).
```

We can execute the following query (starting from an empty initial state):

```
? - computation ( [ ], L ).
```

We get:

```
L = [ [e2, e3], [e1, e4], [ ] ].
```

that is the next state is `[e2, e3]`, then `[e1, e4]` and the final state `[ ]`.

3.3 Stopping unlimited computations with memoization

A problem which may arise during a computation in a reaction system is that a loop can be created easily either directly in one reaction or with dependencies between different reactions. For instance consider the reaction `[[a],[b],[a]]`. If we start with the initial state `[a]`, then an infinite sequence `[a]`, `[a]`, `[a]`, ... will be generated, unless the context introduces the inhibitor `b` at some stage of the computation.

We have extended our interpreter by using a technique in the style of "memoization". So we have defined a predicate `unlimitedComputationContextIndependentMemoized` which keeps track of the states of the computation generated, and as soon as a state of the computation is repeated (i.e. it appears identical in a previous step), the computation is stopped and the finite sequence of states until the current one is returned.

4 Reaction systems: a biological example

In this section we present the encoding of a reaction system example taken from [12], that regards the *lac* operon mechanism in the reaction system formalism. Therefore, we preliminary introduce the most essential notions about the *lac* operon.

4.1 The *lac* operon

An *operon* is a functioning unit of DNA containing a cluster of genes under the control of a single promoter (i.e. a sequence of DNA to which proteins bind in order to initiate transcription). The *lac* operon is involved in the metabolism of lactose in *Escherichia coli* cells (i.e. a bacteria which lives in the intestines of mammals and birds and which is needed

to digest food). It is composed by three adjacent structural genes (plus some regulatory components): $lacZ$, $lacY$ and $lacA$ encoding for two enzymes Z and A, and a transporter Y, involved in the digestion of the lactose. The main regulations are:

- The DNA sequence, called *promoter*, is recognized by a RNA polymerase (i.e. an enzyme that synthesizes RNA from a DNA template) to iniziate the transcription of the genes $lacZ$, $lacY$ and $lacA$;
- The gene $lacI$ encodes for a repressor protein I;
- A DNA segment, called the *operator* (OP), obstructs the RNA polymerase functionality when the repressor protein I is bound to it forming I-OP;
- A short DNA sequence, called the CAP-binding site, when it is bound to the complex composed by the protein CAP and the signal molecule $cAMP$, acts as a promoter for the interaction between the RNA polymerase and the promoter.

The functionality of the *lac* operon is based on the integration of two control mechanisms of which, one is mediated by lactose, while the other one is mediated by glucose.

1. In the first control mechanism, an effect of the absence of the lactose is that I can bind the operator sequence preventing in this way the lac operon expression. If lactose is available, I is unable to bind the operator sequence, and then the *lac* operon can be potentially expressed.
2. In the second control mechanism, in the absence of glucose, the molecule $cAMP$ and the protein CAP increase the lac operon expression, thanks to the fact that the binding between the molecular complex $cAMP$-CAP and the CAP-binding site increases.

Therefore, to sum up, the condition that promotes the operon gene expression is the presence of lactose and the absence of glucose.

4.2 The Reaction System formalization

The reaction system for the *lac* operon is defined as $A_{lac} = (S, A)$, where the set S represents the main biochemical components involved in the considered genetic system and the reaction set A contains the biochemical reactions involved in the regulation of the *lac* operon expression. $S = \{lac, Z, Y, A, lacI, I, I\text{-}OP, cya, cAMP, crp, CAP, cAMP\text{-}CAP, lactose, glucose\}$ and A consists of the following 10 reactions:

$$
\begin{array}{lllllll}
a_1 & = & (\{lac\}, \{...\}, \{lac\}), & & a_6 & = & (\{cya\}, \{...\}, \{cAMP\}),\\
a_2 & = & (\{lacI\}, \{...\}, \{lacI\}), & & a_7 & = & (\{crp\}, \{...\}, \{crp\}),\\
a_3 & = & (\{lacI\}, \{...\}, \{I\}), & & a_8 & = & (\{crp\}, \{...\}, \{CAP\}),\\
a_4 & = & (\{I\}, \{lactose\}, \{I\text{-}OP\}), & & a_9 & = & (\{cAMP, CAP\}, \{glucose\}, \{cAMP\text{-}CAP\}),\\
a_5 & = & (\{cya\}, \{...\}, \{cya\}), & & a_{10} & = & (\{lac, cAMP\text{-}CAP\}, \{I\text{-}OP\}, \{Z, Y, A\}).
\end{array}
$$

where $\{...\}$ stands for any dummy inhibitor. Observe that reactions a_1, a_2, a_5, a_7 are necessary to grant the permanency of the genes in the system; while reactions a_4, a_9, a_{10} can only be enabled if the current state of the system does not include the inhibitor elements specified in each reaction. In more details, reaction a_4 can be applied only in the absence of lactose, reaction a_9 in the absence of glucose, and reaction a_{10} when repressor I is not bound to the operator OP.

The *lac* operon expression is based on which substrates the environment provides. In order to translate this situation in the *lac* operon reaction system, we need to evaluate what happens to the system when the context provides both glucose and lactose, only glucose, only lactose, or none of them. To do this, we define a default context (DC) that mimics the real biological system in which the genomic elements plus their encoded proteins are

normally present; hence DC is composed by those entities that are always present in the system $DC = \{lac, lacI, I, cya, cAMP, crp, CAP\}$, whereas the lactose and the glucose are given non-deterministically by the context.

4.3 The Reaction System encoding in Prolog

We now show the encoding of the considered reaction systems in Prolog. We note that, by definition, the set of inhibitors should not be empty, but in this example most of the triples have empty inhibitors. Thus, we add a dummy inhibitor *gp*, i.e. a new constant that does not appear in any reaction, if the set of inhibitors is empty. If we add the new inhibitor *gp* in all sets of inhibitors in the rules, then we can use it to force the termination of a computation, when the context introduces it. This is useful, as reactions such as "$([a], [\ ], [a])$" may cause an infinite loop. Notice that since a reaction system has a finite background set, we can prove the following property:

▶ **Proposition 8.** *If a reaction system enters in an infinite loop then the infinite computation has the form* $W_0, W_1, \ldots\ W_m \ldots W_k \ldots$, *where* $m < k$, *and* $W_m = W_k$.

This means that the subsequence $W_m, \ldots, W_k$ will then be repeated iteratively.

Let us now consider the program presented in Section 3, where we replace the rules for predicates `reactionset` and `context` by the following ones:

```
reactionset([([lac], [gp], [lac]), ([laci],[gp],[laci]),
      ([laci],[gp],[ig]), ([ig],[lactose, gp],[iop]),
      ([cya],[gp],[cya]), ([cya],[gp], [camp]),
      ([crp],[gp],[crp]), ([crp],[gp],[cap]),
      ([camp, cap],[glucose, gp],[campcap]),
      ([lac, campcap], [iop, gp],[z,y,a])]).

context([[lac,laci,ig,cya,camp,crp,cap],
        [glucose], [glucose], [glucose],
        [glucose], [glucose], [glucose, lactose],
        [glucose, lactose], [glucose, lactose],
        [glucose, lactose], [glucose, lactose],[gp]]).
```

By executing the following query:

```
? - computation ( [  ], L ).
```

We get:

```
L =  [[lac, laci, ig, iop, cya, camp, crp, cap, campcap],
      [[lac, laci, ig, iop, cya, camp, crp, cap],
      [lac, laci, ig, iop, cya, camp, crp, cap],
      [lac, laci, ig, iop, cya, camp, crp, cap],
      [lac, laci, ig, iop, cya, camp, crp, cap],
      [lac, laci, ig, iop, cya, camp, crp, cap],
      [lac, laci, ig, cya, camp, crp, cap],
      [lac, laci, ig, cya, camp, crp, cap],
      [lac, laci, ig, cya, camp, crp, cap],
      [lac, laci, ig, cya, camp, crp, cap],
      [lac, laci, ig, cya, camp, crp, cap], [  ] ]
```

In the following section, we will present some extensions of the basic framework and a possible implementation in our interpreter.

5 Extensions of the basic framework: modifications to the context

In this section we want to show that our interpreter is flexible and can be exploited as a rapid prototyping tool for implementing prototypes of extensions of the basic framework of reaction systems. We will illustrate this characteristic by defining first a very simple nondeterministic extension of reaction systems, and then showing how to implement a recent extension of reaction systems to a network of them [6].

5.1 Non-deterministic context

Reaction systems are deterministic. However, the evolution of a computation interacting with a context depends on such interaction. So, recently some work has focused on extending the context behaviour to make it more expressive. For instance [10] has designed an extension of the context based on process algebras which allows for non deterministic and even recursive contexts. Here we propose a much simpler extension, by adding a nondeterministic operator to the context.

The implementation of non-deterministic finite transition systems provides an instructive insight into the role of context in interactive processes. Let's modify our program and add a non-determinism operator in context. In this way, the context instead of being made from a sequence of lists $S_1, S_2, \ldots,$ will be a list in which each element of the list is a list of lists from which it can be chosen not deterministically. For example $(S_{11} + S_{12} + \cdots + S_{1k_1})(S_{21} + S_{22} + S_{23} + \cdots + S_{k_2})\ldots$ and the system chooses one of these lists at each step in a completely non-deterministic way. If our context sequence was $(S_{11})(S_{21} + S_{22})(S_{31} + S_{32})$, then the possible (context) sequences generated in a non-deterministic way would be S_{11}-S_{21}-S_{31} or S_{11}-S_{21}-S_{32} or S_{11}-S_{22}-S_{31} or S_{11}-S_{22}-S_{32}.

The nondeterministic choice on each step of the context is performed by the predicate `chooseContext/2`, which chooses randomly one of the contexts in the list.

We modify our program in the following way:

```
contextND([[[a1,a2],[a3,a2,a5]],[[a2,a3,a,4],[a1,a2,a5],[a3]]]).

chooseContext(PossibleContext, ChoosenContext):-
        length(PossibleContext, Length),
        random(0,Lenght,Index),
        nth0(Index,PossibleContext,ChoosenContext).

context([],[]).
context([L|OtherList],[Cc|Cot]):- chooseContext(L,Cc),
context(OtherList,Cot).
computation(InitialState,L):- reactionSet(R), contexND(C),
         context(C,[C0|Cs]), union(InitialState,C0,SC),
         computeWithContext(SC,Cs,R,L).
```

We have defined the predicate *chooseContext* that selects a random element from a list of lists. The new context is a list whose elements are lists in which to choose a list. The new context is given by *contextND*. To create the context we defined the *context* predicate. Finally, we modified the *computation* predicate.

We notice that we will not add these modifications to our interpreter. These modifications could be useful to model easily non deterministic systems, with a *don't care* kind of non-determinism which is typical of concurrent systems [24]. Don't care nondeterminism means that only one of the possible choices is chosen, and the other alternatives are discarded. For an

extension of our interpreter which exploits a non deterministic context with the typical *don't know* non determinism of logic programming please see [10]. Don't know nondeterminism means that all possible choice alternatives are tried.

In the following section we show that our interpreter can be extended to model two interacting reaction systems.

5.2 Interaction of two Reaction Systems

We start by discussing first a simple extension to a network made by two reaction systems which "cooperate". We illustrate how it is possible to program two reaction systems encodings, in such a way that the entities that usually come from the context of one reaction system will be provided instead from the other reaction system.
A slight modification of our program allows us to consider two reaction systems in which the output of the first reaction system becomes the context of the other.

We can define two separate unit predicates for the reactions in the two RS, `reactionsetF/1` and `reactionsetS/1`. Then, we have to modify the `computation/2` predicate so that in the case of the first reaction system we provide the context via the context predicate; while for the second reaction system we use the list obtained from the computation of the first reaction system.

```
reactionsetF([([a1,a2],[a3],[a2,a3]),([a5],[a4],[a1,a4])]).
reactionsetS([([a1,a3],[a4],[a1,a3]),([a3,a5],[a4],[a2,a4])]).

computationF(InitialState,L):-reactionsetF(R),context([C0|Cs]),
       union(InitialState,C0,SC),computeWithContext(SC,Cs,R,L).

/* new predicate to calculate the second reaction system */
computation(InitialStateF,InitialStateS, L):- reactionsetS(R),
       computationF(InitialStateF,[C0|L1]), context([C0|L1]),
       union(InitialState,C0,SC),computeWithContext(SC,Cs,R,L).
```

In the following section we show that our interpreter can be extended to model networks of reaction systems. We have enclosed this extension in the interpreter available online.

5.3 Networks of Reaction Systems

Here we illustrate how to extend our interpreter for modeling networks of reaction systems as introduced in [6]. In [6] the context has its own structure: the context for a reaction system originates from a network of reaction systems. Such a network is formalized as a graph where nodes represent reaction systems, and where each reaction system contributes to defining the context of all its neighbours. Thus, as the context for a reaction system is given by a network of reaction systems communicating with it, the interaction between two reaction systems that we have introduced in Section 5.2 can be seen as a special case of the definition of a network of reaction systems. In the basic model of [6] reported here, all edges function as communication channels and states of reaction systems residing at nodes are synchronized according to a global clock.
We start by introducing the general notions of *centralized network* of reaction systems and *interactive network process*. In the network of RS that we will define, the j-th RS will be denoted by $\mathcal{A}^j = (S^j, A^j)$. $\mu : V \to \mathcal{F}$ is a *location function*, which assigns RS to nodes. So, for $v_j \in V$, $\mu(v_j) = \mathcal{A}^j$. The set of incoming neighbours of a node v in a graph, namely those nodes for which there is an edge connecting them to v, is denoted by $in(v)$. The following two definitions are from paper [6].

▶ **Definition 9.** *A centralized network of reaction systems is a tuple $\mathcal{N} = (G, \mathcal{F}, \mu)$, where $G = (V, E, v_0)$ is a finite centralized graph such that $in(v_0) \neq \emptyset$, $\mathcal{F}$ is a nonempty finite set of reaction systems, and $\mu : V \to \mathcal{F}$ is a location function, assigning reaction systems to nodes.*

The following definition formalises the notion of a computation of length n for an *interactive network process*, which is given by a vector of individual interactive processes of the reaction systems in the network nodes. The computation starts from an initial given distribution (C_0^j, D_0^j). Roughly speaking C_k^j represents the context for RS j at step k of computation, and D_k^j represents the state of RS j at step k of computation. Thus, for any node v_j, for each subsequent step i of the process associated with such a node π^j, the component D_i^j is obtained by applying enabled reactions from A^j to the current state, while the component C_i^j is given by the union of the results produced, at the previous step, by the incoming neighbours of v_j. It is finally made an intersection with S^j to filter out entities which are not in the background set of $\mathcal{A}^j$.

▶ **Definition 10.** *Let $\mathcal{N} = (V, E, v_0, \mathcal{F}, \mu)$ be a centralized network of reaction systems with $|V| = m + 1$ for some $m \geq 0$. For $n \in \mathbb{N}^+$, an interactive (n-step) network process is a tuple $\Pi = (\pi^0, \ldots, \pi^m)$, where, for $j \in \{0, \ldots, m\}$, $\pi^j = (\gamma^j, \delta^j)$ and $\gamma^j = (C_0^j, \ldots, C_n^j)$, $\delta^j = (D_0^j, \ldots, D_n^j)$, are such that:*

1. $C_k^j = S^j \cap \bigcup\{D_{k-1}^i | v_i \in in(v_j)\}$*, for* $k \in \{1, \ldots, n\}$*,*
2. $D_k^j = res_{A^j}(D_{k-1}^j \cup C_{k-1}^j)$ *for* $k \in \{1, \ldots, n\}$*.*
3. *If* $in(v_j) = \emptyset$*, then* $C_0^j = \emptyset$*.*

We now illustrate the implementation of the network of Reaction Systems. The new implementation proceeds for a limited number of steps K, where K is a number given at the beginning when it is requested by the program, or until an empty state is encountered. The complete derivation of the Reaction System 1 is calculated (it was called 0 in the previous definition). The reaction systems of the network corresponds to the nodes of the network and are numbered by positive integers 1, 2, 3, and so on. At the beginning, the overall number of Reaction Systems in the network must be given as input. In the following we present an example consisting of two reaction systems, but the program is valid for an arbitrary finite number of nodes. As output we obtain the final state of all the Reaction Systems in the network, and the complete computation of the reaction system 1. It is sufficient to invoke *main(F, D)*, so that the program calculates F and D, i.e. the overall final state F of the network and the complete derivation D for reaction system 1. We do not restrict the set of computed values to the background of the node. It would be easy to add such a restriction. For the sake of simplicity we assume that all RSs in the network have the same background set.

The edges of the network are represented by a predicate `network/1` introducing a list of pairs of the form `[m,n]` meaning that there is an arc from node `m` to node `n`. The predicate `search(N,Net,S0,S1)` looks for all pairs `[N1,N]` in `Net` and returns `S1` = union of the states in position `N1` of `S0`, thus computing the context for `N` in the network of RS.

The predicate `initialStates/1` is defined by a unit fact which introduces a list of list defining the initial states of the nodes in the network. So list in position k corresponds to the initial state of node k.

Let us see a fragment of one example. For more details please refer to the interpreter online.

```
reaction(1,[([lac],[ a],[cya]),([lacI],[a ],[lac2]),
       ...([lac2],[a ],[lac3]),([cya],[ a],[cya3])]).
reaction(2,[([lac],[ a],[cya1,cya]),([lacI],[a ],[lac2]),
       ...([lac2],[a ],[lac3]),([cya],[ a],[cya,cya2])]).

network([[2,1]]).

computeOneStep(N,S,S2):- computeState(S,SO,1),
  network(Net), computeContext(SO,Net,N,S1),
  unionList(SO,S1,S2).

computeState([],[],K).
computeState([S|Ss],[S1|S1s],K):-reaction(K,R),
  resultallreactions(S,R,S1), K1 is K+1,
  computeState(Ss,S1s,K1).

computeContext(S,Net,N,SO):- computeContext1(S,Net,N,SO,1).

computeContext1(S,Net,N,[],N1):-N<N1.
computeContext1(S,Net,N,[S1|SO],N1):-N1=<N,
  search(N1,Net,S,S1),N2 is N1+1,
  computeContext1(S,Net,N,SO,N2).

initialStates([[lac],[lac]]).
```

An example of execution follows:

```
| ?- main(F,D).
Give me the number of Reaction Systems in the Network
(a positive integer, followed by a dot) 2.
Give me the maximun number of computation steps
(a positive integer, followed by a dot) 5.

D = [[cya1,cya],[cya3,cya,cya2],[cya3,cya,cya2], [cya3,cya,cya2],[cya3,cya,cya2]]
F = [[cya3,cya,cya2],[cya,cya2]]
```

This model of communicating reaction systems can enable the study of the behaviour of one reaction system in relation to other ones. This way, the lac operon system can be connected with the two systems producing the lactose and the glucose, and therefore the presence of these two entities in the lac operon system can be regulated by realistic mechanisms.

6 Conclusions and future work

In this paper we have recalled the framework of Reaction Systems introduced by A. Ehrenfeucht and G. Rozenberg [16]. Then we have described our preliminary implementation of this framework in Prolog. We have then shown that our interpreter is flexible and suitable for rapid prototyping and implementing extensions of the basic framework. It allows to make indefinitely long computations, computations limited to a maximum of k steps, and we have also introduced a kind of memoization mechanism based on accumulators for stopping a computation when a state gets repeated. The user can choose her preferences. Thus, we have shown how to implement an extension of RS for modeling nondeterministic contexts with don't care non determinism, and two interacting RS, and then we have implemented the recently introduced networks of reaction systems [6]. Our interpreter is freely available

online. As a future work we plan to improve the implementation to make it more efficient by using constraint logic programs, by exploiting finite domains, and CLP(SET) [14], and more user friendly, also by interfacing it to graphical tools for showing the computations in our framework. We also plan as a future work to study how to exploit the structures which have been defined for representing efficiently enormous numbers of states in model checking, in order to improve the evaluation of reaction systems. Some work has already been done in [22]. We also want to study the application of static analysis techniques [2, 5, 4] to RS.

References

1 S. Azimi, C. Gratie, S. Ivanov, and I. Petre. Dependency graphs and mass conservation in reaction systems. *Theoretical Computer Science*, 598:23—-39, 2015. doi:10.1016/j.tcs.2015.02.014.

2 R. Barbuti, R. Gori, F. Levi, and P. Milazzo. Investigating dynamic causalities in reaction systems. *Theor. Comput. Sci.*, 623:114–145, 2016. doi:10.1016/j.tcs.2015.11.041.

3 A. Bernini, L. Brodo, P. Degano, M. Falaschi, and D. Hermith. Process calculi for biological processes. *Natural Computing*, 17(2):345–373, 2018. doi:10.1007/s11047-018-9673-2.

4 C. Bodei, L. Brodo, and R. Focardi. Static evidences for attack reconstruction. In *Proc. of Programming Languages with Applications to Biology and Security*, volume 9465 of *Lecture Notes in Computer Science*, pages 162–182. Springer, 2015. doi:10.1007/978-3-319-25527-9_12.

5 C. Bodei, L. Brodo, R. Gori, F. Levi, A. Bernini, and D. Hermith. A static analysis for brane calculi providing global occurrence counting information. *Theor. Comput. Sci*, 696:11–51, 2017. doi:10.1016/j.tcs.2017.07.008.

6 P. Bottoni, A. Labella, and G. Rozenberg. Networks of reaction systems. *International Journal of Foundations of Computer Science*, 31:53–71, 2020. doi:10.1142/S0129054120400043.

7 R. Brijder, A. Ehrenfeucht, M. Main, and G. Rozenberg. A tour of reaction systems. *International Journal of Foundations of Computer Science*, 22(07):1499–1517, 2011. doi:10.1142/S0129054111008842.

8 L. Brodo. On the expressiveness of pi-calculus for encoding mobile ambients. *Mathematical Structures in Computer Science*, 28(2):202–240, 2018. doi:10.1017/S0960129516000256.

9 L. Brodo, R. Bruni, and M. Falaschi. Enhancing reaction systems: a process algebraic approach. In M. Alvim, K. Chatzikokolakis, C. Olarte, and F. Valencia, editors, *The Art of Modelling Computational Systems: A Journey from Logic and Concurrency to Security and Privacy*, volume 11760 of *Lecture Notes in Computer Science*, pages 68–85. Springer Berlin, 2019. doi:10.1007/978-3-030-31175-9_5.

10 L. Brodo, R. Bruni, and M. Falaschi. SOS rules for equivalences of reaction systems. In *Pre-proceedings of the 28th Int. workshop on Functional and Logic Programming (WFLP 2020)*, 2020. arXiv:2009.01001.

11 L. Brodo and C. Olarte. Symbolic semantics for multiparty interactions in the link-calculus. In *Proc. of SOFSEM'17*, volume 10139 of *Lecture Notes in Computer Science*, pages 62–75. Springer, 2017. doi:10.1007/978-3-319-51963-0_6.

12 L. Corolli, C. Maj, F. Marinia, D. Besozzi, and G. Mauri. An excursion in reaction systems: From computer science to biology. *Theoretical Computer Science*, 454:95–108, 2012. doi:10.1016/j.tcs.2012.04.003.

13 J. Demongeot, M. Noual, and S. Sené. On the number of attractors of positive and negative boolean automata circuits. In *2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops*, pages 782–789, 2010. doi:10.1109/WAINA.2010.141.

14 A. Dovier, C. Piazza, E. Pontelli, and G. Rossi. Sets and constraint logic programming. *ACM Transactions on Programming Languages and Systems*, 22(5):861–931, 2000. doi:10.1145/365151.365169.

15 A. Ehrenfeucht, J. Kleijn, M. Koutny, and G. Rozenberg. Qualitative and quantitative aspects of a model for processes inspired by the functioning of the living cell. In Evgeny Katz, editor, *Biomolecular Information Processing: From Logic Systems to Smart Sensors and Actuators*, pages 323–331. Wiley, 2012. doi:10.1002/9783527645480.ch16.

16 A. Ehrenfeucht and G. Rozenberg. Reaction systems. *Fundamenta Informaticae*, 76:1–18, 2006. URL: https://content.iospress.com/articles/fundamenta-informaticae/fi75-1-4-15.

17 A. Ehrenfeucht and G. Rozenberg. Reaction systems: a formal framework for processes based on biochemical interactions. *Electronic Communications of the EASST*, 26:1–10, 2010. doi:10.1007/978-3-642-02424-5_3.

18 C. Ferretti, A. Leporati, and L. Manzoni. The many roads to the simulation of reaction systems. *Fundamenta Informaticae*, 171(1-4):175–188, 2020. doi:10.3233/FI-2020-1878.

19 J. Kari. Theory of cellular automata: A survey. *Theoretical Computer Science*, 334(1-3):3–33, 2005. doi:10.1016/j.tcs.2004.11.021.

20 H-J. Kreowski and G. Rozenberg. Graph surfing by reaction systems. In Lambers L. and Weber J., editors, *Graph Transformation. ICGT 2018.*, volume 10887 of *Lecture Notes in Computer Science*, pages 45–62. Springer Berlin, 2018. doi:10.1007/978-3-319-92991-0_4.

21 H-J. Kreowski and G. Rozenberg. Graph transformation through graph surfing in reaction systems. *Journal of Logical and Algebraic Methods in Programming*, 109, 2019. doi:10.1016/j.jlamp.2019.100481.

22 A. Męski, W. Penczek, and G. Rozenberg. Model checking temporal properties of reaction systems. *Information Sciences*, 313:22–42, 2015. doi:10.1016/j.ins.2015.03.048.

23 M.S. Nobile, A.E. Porreca, S. Spolaor, L. Manzoni, P. Cazzaniga, G. Mauri, and D. Besozzi. Efficient simulation of reaction systems on graphics processing units. *Fundamenta Informaticae*, 154(1-4):307—321, 2017. doi:10.3233/FI-2017-1568.

24 E. Shapiro. The family of concurrent logic languages. *ACM Computing Surveys*, 21(3):412–510, September 1989. doi:10.1145/72551.72555.

Abstract Interpretation, Symbolic Execution and Constraints

Roberto Amadini
University of Bologna, Italy
https://www.unibo.it/sitoweb/roberto.amadini/en
roberto.amadini@unibo.it

Graeme Gange
Monash University, Clayton, Australia
https://research.monash.edu/en/persons/graeme-gange
graeme.gange@monash.edu

Peter Schachte
The University of Melbourne, Australia
https://people.eng.unimelb.edu.au/schachte/
schachte@unimelb.edu.au

Harald Søndergaard
The University of Melbourne, Australia
https://people.eng.unimelb.edu.au/harald/
harald@unimelb.edu.au

Peter J. Stuckey
Monash University, Clayton, Australia
https://research.monash.edu/en/persons/peter-stuckey
peter.stuckey@monash.edu

Abstract

Abstract interpretation is a static analysis framework for sound over-approximation of all possible runtime states of a program. Symbolic execution is a framework for reachability analysis which tries to explore all possible execution paths of a program. A shared feature between abstract interpretation and symbolic execution is that each – implicitly or explicitly – maintains constraints during execution, in the form of invariants or path conditions. We investigate the relations between the worlds of abstract interpretation, symbolic execution and constraint solving, to expose potential synergies.

2012 ACM Subject Classification Theory of computation → Program analysis; Theory of computation → Invariants; Software and its engineering → Software maintenance tools; Software and its engineering → Software testing and debugging

Keywords and phrases Abstract interpretation, symbolic execution, constraint solving, dynamic analysis, static analysis

Digital Object Identifier 10.4230/OASIcs.Gabbrielli.2020.7

Acknowledgements We wish to thank the anonymous reviewers for their detailed and constructive suggestions.

1 Introduction

The *abstract interpretation* framework proposed by Cousot and Cousot in the 1970s [14, 15] provides an elegant and generic approach for static analysis. Under certain reasonable assumptions, this method is guaranteed to terminate with a sound abstraction of *all* the possible program traces.

© Roberto Amadini, Graeme Gange, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey;
licensed under Creative Commons License CC-BY

Recent Developments in the Design and Implementation of Programming Languages.
Editors: Frank S. de Boer and Jacopo Mauro; Article No. 7; pp. 7:1–7:19

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Since different information is pertinent in different contexts and applications, each analysis specifies the abstraction of the computation state to use: the *abstract domain* of the analysis. Each element of the abstract domain is an abstract value that approximates a set of "concrete" values, i.e., values that a variable can take during the program execution.

The most common form of abstract interpretation mimics forward program execution in such a manner that it eventually *over-approximates* the set of possible run-time states, for all possible input values, with corresponding abstract values. Abstract interpretation thus provides a method for *invariant generation* – it produces, in finite time, a valid invariant for each program point, including the start and the end of loop bodies. Whether these invariants are useful for whatever task is at hand may depend on the granularity of the chosen abstract domain.

The first *symbolic execution* tools [8, 25] were developed around the same time as the abstract interpretation framework. The main idea behind symbolic execution is to use symbolic expressions instead of concrete values to explore the possible program paths and reasoning about the conditions under which the program execution will branch this way or that. The constraints leading to a particular path being taken are called *path conditions*, so that a given path is feasible if and only if the corresponding path condition is satisfiable. This enables symbolic execution to perform *reachability* analysis.

The early work on symbolic execution saw the technique as an important aid in the systematic testing and debugging [8, 25] and those applications remain the most common. However, it has been noted that Burstall's technique [9] for proving total correctness of programs also involves the use of symbolic execution. (Burstall used the term "hand simulation", and his technique has since been referred to as the "sometime" method, and also as the "intermittent assertion" method.) The method associates, with a program point p, assertions of the form "control will, sometime during execution, reach p with the program state satisfying φ" (note that there is no claim that this will be the case *every* time control reaches p). Showing that a program terminates and satisfies a specification φ thus boils down to being able to associate φ (or something that entails it) with each exit point of the program. As with the more commonly known invariant assertion method, the intermittent assertion methods relies on the discovery of suitable lemmas and their proof by induction. Unlike the invariant assertion method, it can establish *total* correctness.

In this paper we mainly have the less ambitious "debugging" use of symbolic execution in mind. We note, however, that there is continued development of deductive verification systems that utilise the idea behind Burstall's method and its application of symbolic execution. We discuss such systems in Section 6. One, the KeY project [1], is of particular interest, as it extends the symbolic execution mechanism with an invariant generation ability, using ideas from abstract interpretation.

At first glance, symbolic execution may seem very similar to abstract interpretation. Both are methods for abstracting the runtime behaviour of a program across all its possible input values. However, while abstract interpretation is naturally considered a *static analysis*, we contend that symbolic execution is for all intents a *dynamic analysis* for a number of reasons. First, symbolic execution always executes the target program, even if symbolically, in a forward manner. Second, symbolic execution cannot by itself guarantee that *all* the possible paths are covered. Unless aided by some external oracle, symbolic execution *under-approximates* the set of possible runtime states with a number of path conditions. As a dynamic analysis, symbolic execution can produce *witnesses* of fault, but it offers weak guarantees for coverage and termination. In particular symbolic execution runs the risk of "getting caught" in loops. Verification tools based on symbolic execution make use of a variety of techniques to remedy the situation, such as requiring users to suggest invariants.

In contrast, static analyses produce *alarms*, including false alarms, they tend to rely on *intrinsic* conditions and, in principle at least, provide strong guarantees for termination and coverage. Because dynamic analysis focuses on (possibly long) program paths, it is able to find relations among program entities that may be textually far apart. Static analysis instead usually performs detailed, but *local*, analysis, detailed only within basic blocks or functions.

A connecting point between abstract interpretation and symbolic execution is that they both – implicitly or explicitly – collect and solve a number of *constraints* along their execution.

For abstract interpretation, the constraints are implicitly collected during the abstract execution in the form of invariants, which are relations over the program variables. The constraint perspective becomes more evident when we have *relational* abstract domains involving different variables. For example, if we use the polyhedra [16] or the octagon [27] abstract domain we explicitly collect and update linear constraints over the program variables.

The relation between symbolic execution and constraint solving is more straightforward. The path conditions collected by the symbolic engine are constraints over the variables occurring at each branching point of the program. The test for satisfiability is delegated to a *constraint solver*. Its role is crucial for symbolic execution, because its efficiency and expressiveness can strongly affect the performance of the program analysis. As we shall see in Section 3, this is even more true for *concolic testing*, a hybrid technique based on symbolic execution where the constraint solver is used to generate the next input to test according to the last path condition explored.

In this paper we study the relationship between abstract interpretation, symbolic execution and constraint solving. Based on a small Turing complete language, $\mathcal{L}$, we first define the semantics of symbolic execution over $\mathcal{L}$ by specifying how constraints are collected, updated and solved during the execution. Then, after describing the abstract interpretation of $\mathcal{L}$, we show how these techniques are complementary and can help each other to get an overall better program analysis. In particular, we focus on how abstract interpretation may help symbolic execution escape loops through simple program transformations.

Paper structure: Section 2 gives some technical background notions. In Section 3 we explain symbolic (and concolic) execution, and in Section 4 we cover abstract interpretation. Section 5 discusses the relationships between the above techniques, before reporting the related literature in Section 6 and concluding in Section 7.

2 Preliminaries

2.1 Constraint solving

As with abstract interpretation and symbolic execution, the theory of constraint solving dates back to the 1970s [28, 26]. Although there is not a univocal definition, we can informally refer to constraint solving – or constraint satisfaction – as the process of finding a solution to a problem whose variables are subject to a number of constraints restricting their domains. More formally, we can define constraint solving as the process of finding a solution to a *constraint satisfaction problem* (CSP), which is a triple $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ where: $\mathcal{X} = \{x_1, \dots, x_n\}$ is a set of variables; $\mathcal{D} = \{D_1, \dots, D_n\}$ is a set of domains, where domain D_i contains the possible values that x_i can take for $i = 1, \dots, n$; $\mathcal{C} = \{C_1, \dots, C_m\}$ is a set of constraints over the variables of $\mathcal{X}$.

Finding a solution of $\mathcal{P}$ means finding a consistent assignment of domain values to variables. Formally, a solution is a map $\xi : \mathcal{X} \to \bigcup \mathcal{D}$ such that $\xi(x_i) \in D_i$ for $i = 1, \dots, n$ and $C(\xi(x_{i_1}), \dots, \xi(x_{i_k}))$ holds for each constraint $C \in \mathcal{C}$ over variables $x_{i_1}, \dots, x_{i_k}$. If $\mathcal{P}$ admits a solution ξ, then it is called *satisfiable* and we will write $\xi \models \mathcal{P}$. If no solution exists,

then $\mathcal{P}$ is *unsatisfiable* and we will write $\mathcal{P} \models \bot$. Note that $\mathcal{P} \not\models \bot$ denotes a satisfiable problem $\mathcal{P}$ without specifying a solution for it (i.e., $\mathcal{P} \not\models \bot$ iff there exists $\xi : \mathcal{X} \to \bigcup \mathcal{D}$ such that $\xi \models \mathcal{P}$). As a small abuse of notation, we extend this notation to constraints: if $C \in \mathcal{C}$ is defined over variables $x_{i_1}, \dots, x_{i_k}$, then $\xi \models C$ means that $C(\xi(x_{i_1}), \dots, \xi(x_{i_k}))$ holds, while $C \not\models \bot$ (resp., $C \models \bot$) means that C is satisfiable (resp., unsatisfiable).

The definition of CSP is very general, because no limits are posed on the type of the domains (e.g., they can be integers, reals, rationals, strings, arrays, and so on), or on the type of the constraints to be solved. According to the type of domain and constraints, and the way constraints are solved, different paradigms have been proposed, e.g., Boolean satisfiability/satisfiability modulo theory [7], constraint programming [31], linear programming [17], and so on.

2.2 Abstract interpretation

Abstract interpretation is a framework for the sound over-approximation of program computations. Let $\mathbb{S}$ be the set of concrete values that a program variable can take (e.g., integers, floating point, strings, ...) in any possible concrete execution. The *concrete domain* $\mathcal{C} = \mathcal{P}(\mathbb{S})$ is defined as the powerset of $\mathbb{S}$, and a sound abstraction of $\mathcal{C}$ is given by an *abstract domain* $\mathcal{A}$ for $\mathcal{C}$ and a concretization function $\gamma : \mathcal{A} \to \mathcal{C}$ inducing a partial order $\sqsubseteq$ over $\mathcal{A}$ such that $a \sqsubseteq a' \iff \gamma(a) \subseteq \gamma(a')$ for each $a, a' \in \mathcal{A}$. Typically, a domain $\mathfrak{A}$ is equipped with an order that makes it a *lattice* $\mathfrak{A} = \langle \mathcal{A}, \sqsubseteq, \sqcap, \sqcup, \bot, \top \rangle$, where $\sqcup$ and $\sqcap$ are the meet and join operations, respectively, according to $\sqsubseteq$, and $\bot$ and $\top$ are unique least and greatest elements of $\mathcal{A}$, respectively[1]. Choosing an abstract domain is a compromise between its *precision* (how faithfully it can approximate a concrete domain) and the computational cost of conducting the analysis with it.

The "abstract lattice" $\mathfrak{A}$ is typically connected to the "concrete lattice" $\mathfrak{C} = \langle \mathcal{C}, \subseteq, \cap, \cup, \emptyset, \mathbb{S} \rangle$ via an abstraction function $\alpha : \mathcal{C} \to \mathcal{A}$ mapping concrete elements to corresponding abstract elements. The pair (α, γ) often forms a *Galois connection*, i.e., $\alpha(C) \sqsubseteq a \iff C \subseteq \gamma(a)$ for each $a \in \mathcal{A}$, $C \in \mathcal{C}$. Having a Galois connection corresponds to the existence of a unique *best abstraction* for each $C \in \mathcal{C}$.

Termination of abstract execution can be guaranteed, even in the presence of loops. In some cases, so-called *widening* operators [12] are required to achieve this (or sometimes just to accelerate convergence). A widening operator ∇ for abstract domain $\mathcal{A}$ satisfies two conditions: *(i)* $a, a' \sqsubseteq a \nabla a'$ for any $a, a' \in \mathcal{A}$, and *(ii)* for any sequence $a_0, a_1, a_2, \dots \in \mathcal{A}$ the sequence $b_0, b_1, b_2, \dots$ with $b_0 = a_0$ and $b_i = b_{i-1} \nabla a_i$ for $i > 0$ is ultimately stationary, i.e., there does exist $k \in \mathbb{N}$ such that $b_i = b_k$ for each $i \geq k$. In practice, widening allows us to "short-cut" infinite ascending chains, guaranteeing that a (post-) *fixpoint* is eventually reached.

Abstract domains can also be *combined*. Given $n > 1$ abstract domains $\langle \mathcal{A}_i, \sqsubseteq_i, \sqcap_i, \sqcup_i, \bot_i, \top_i \rangle$ abstracting a concrete domain $\mathcal{C}$ with abstraction functions $\alpha_i : \mathcal{C} \to \mathcal{A}_i$ and concretization functions $\gamma_i : \mathcal{A}_i \to \mathcal{C}$ for $i = 1, \dots, n$, their *direct product* is the structure $\langle \mathcal{A}, \sqsubseteq, \sqcap, \sqcup, \bot, \top \rangle$ where:

- $\mathcal{A} = \mathcal{A}_1 \times \cdots \times \mathcal{A}_n$
- $(a_1, \dots, a_n) \sqsubseteq (a'_1, \dots, a'_n) \iff a_1 \sqsubseteq_1 a'_1 \wedge \cdots \wedge a_n \sqsubseteq_n a'_n$
- $(a_1, \dots, a_n) \sqcap (a'_1, \dots, a'_n) = (a_1 \sqcap_1 a'_1, \dots, a_n \sqcap_n a'_n)$
- $(a_1, \dots, a_n) \sqcup (a'_1, \dots, a'_n) = (a_1 \sqcup_1 a'_1, \dots, a_n \sqcup_n a'_n)$
- $\bot = (\bot_1, \dots, \bot_n)$ and $\top = (\top_1, \dots, \top_n)$

[1] One can also find examples of non-lattice abstract domains [19].

A drawback of the direct product is that γ may not be injective, even when all of the γ_i are injective. Its use may give rise to sub-optimal precision, although it does not threaten soundness of the analysis. For optimal precision, the *reduced product* $\mathcal{A}' = \mathcal{A}_1 \otimes \cdots \otimes \mathcal{A}_n$ is required [15]. Informally, $\mathcal{A}'$ removes redundant tuples from $\mathcal{A}$. More formally, $\mathcal{A}'$ is the quotient set of the equivalence relation $\equiv$ on $\mathcal{A}$ such that $(a_1, \dots, a_n) \equiv (a'_1, \dots, a'_n) \Leftrightarrow \gamma(a_1, \dots, a_n) = \gamma(a'_1, \dots, a'_n)$. This ensures that the resulting γ is injective. For example, if $\mathcal{A}_1$ is the parity domain and $\mathcal{A}_2$ is the interval domain, then $(\mathit{Odd}, [0, 2])$ and $(\mathit{Odd}, [1, 1])$ are different elements of $\mathcal{A}_1 \times \mathcal{A}_2$, yet they have the same meaning. In $\mathcal{A}_1 \otimes \mathcal{A}_2$ they are considered one and the same.

While the reduced product has a simple mathematical definition, it can be difficult or cumbersome to implement. A common alternative is to use *ad hoc* channelling functions to refine pairs of abstract elements, with no guarantee of optimal reduction. Care is needed when widening is defined on reduced products: The distributed operation $(a_1 \nabla_1 a'_1, \dots, a_n \nabla_n a'_n)$ combining the widening operators of the individual domains is not necessarily a valid widening operation.

2.3 Language $\mathcal{L}$

We now define a simple language, $\mathcal{L}$, that we shall use in Sections 3 and 4 to illustrate how symbolic execution and abstract interpretation work.

We denote by *Var* the set of all the variables that can occur in an $\mathcal{L}$ program, and with *Val* the set of values that a variable of *Var* can take (e.g., *Val* may contain integers, floating point numbers, string literals, and so on). We assume that values 0 and 1 (which may represent falsehood and truth) always belong to *Val*. We denote by *Loc* the set of all the locations, or program points, for a program written in $\mathcal{L}$.

A *concrete state*, or runtime state, is a map $\mathit{Var} \to \mathit{Val}$ from variables to values. A *concrete trace* $\tau : \mathit{Var} \times \mathit{Loc} \to \mathit{Val} \cup \{\bot, \top\}$ is a function returning the concrete state $\tau(x, \ell)$ of variable x at program point ℓ in a given execution of the program. If $\tau(x, \ell) = \bot$, then either ℓ is unreachable or x is not defined at ℓ. If $\tau(x, \ell) = \top$, then the value of x at ℓ is unknown.

Let *Fun* be the set of all the possible functions of $\mathcal{L}$, i.e., the allowed operations over the variables and values of $\mathcal{L}$. We denote by $\mathit{Fun}^k \subseteq \mathit{Fun}$ the set of functions having arity k, and with *BFun* the set of the Boolean-valued functions (predicates), that is, the functions that yield values in $\{0, 1\}$ only. The set of expressions *Exp* over $\mathcal{L}$ is recursively defined by:

$$\mathit{Val}, \mathit{Var} \subseteq \mathit{Exp}$$

$$f \in \mathit{Fun}^k,\ e_1, \dots, e_k \in \mathit{Exp} \implies f(e_1, \dots, e_k) \in \mathit{Exp}$$

We define the set $\mathit{Bool} \subseteq \mathit{Expr}$ of the Boolean expressions as follows:

$$0, 1 \in \mathit{Bool}$$

$$b(e_1, \dots, e_k) \in \mathit{Exp},\ b \in \mathit{BFun} \implies b(e_1, \dots, e_k) \in \mathit{Bool}$$

A Boolean expression $b \in \mathit{Bool}$ defines a constraint: if b can evaluate to 1, the corresponding constraint is satisfiable; otherwise, its *negation* $\neg b = 1 - b$ is satisfiable (this however does not imply that b is unsatisfiable).

Now we can define the BNF syntax of the statements of $\mathcal{L}$ as:

$$S ::= \textbf{skip} \mid x \leftarrow e \mid x \leftarrow \top \mid S_1; S_2 \mid \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} \mid \textbf{while } b \textbf{ do } S \textbf{ od}$$

where $x \in \mathit{Var}$, $e \in \mathit{Exp}$, $b \in \mathit{Bool}$, and S, S_1, S_2 are statements.

We are deliberately vague about the values in *Val* and the functions in *Fun* because we aim to provide a high-level view of the semantics of the symbolic execution and abstract interpretation of $\mathcal{L}$ without going into too much detail.

3 Symbolic Execution

Symbolic execution was introduced with the aim of describing in a compact way the inputs causing each part of a program to be executed or “covered”. In this section we shall see how symbolic (and concolic) execution works for $\mathcal{L}$ from Section 2.

A peculiarity of $\mathcal{L}$ is that it allows the definition of assignments of the form $x \leftarrow \top$. We use this to indicate that the value of x is unknown. It corresponds to an annotation required for symbolic execution, to mark a program variable x as “symbolic”. After $x \leftarrow \top$ a Boolean expression containing x will be evaluated in terms of *all* the possible values that x can take in a *particular* concrete trace. In the interest of generality we make no assumption about the domain of a symbolic variable.

Rather than maintaining concrete traces, the symbolic execution of an $\mathcal{L}$ program defines *symbolic traces* keeping track of: 1) the *path* π describing the evolution of the program execution; 2) a *symbolic state* σ mapping variables to (symbolic) expressions; 3) a constraint ϕ denoting a *path condition* for π, i.e., a necessary and sufficient condition, on symbolic values, for execution to follow path π. Formally, a symbolic trace is a triple (π, σ, ϕ) where:

1. $\pi \in (Loc \times \{0,1\})^*$ is a tuple of branch points of the form $\langle \ell_1^{b_1}, \dots, \ell_k^{b_k} \rangle$ where ℓ_i is the location of the i-th branch point encountered along the execution path, and b_i is either 0 or 1 depending on whether the corresponding condition evaluated to false or true respectively.
2. $\sigma : Var \to Exp$ maps variables to expressions. We extend σ to expressions in the natural way by defining $\overline{\sigma} : Exp \to Exp$ as:

$$\begin{aligned} \overline{\sigma}(c) &= c && \text{for each } c \in Val \\ \overline{\sigma}(x) &= \sigma(x) && \text{for each } x \in Var \\ \overline{\sigma}(f(x_1, \dots, x_k)) &= f(\overline{\sigma}(x_1), \dots, \overline{\sigma}(x_k)) && \text{for each } k \in \mathbb{N}, f \in Fun^k \end{aligned}$$

3. $\phi = C_1 \wedge \dots \wedge C_k$ is a conjunction of constraints denoting the path condition of π; each C_i is either a Boolean expression (if $b_i = 1$) or a negated Boolean expression (if $b_i = 0$) describing the direction taken at each branch point.

A natural way to capture the semantics of a symbolic execution is with a *structural operational semantics* (SOS) representing how the symbolic trace evolves. Fig. 1 shows the SOS definition of $\mathcal{L}$'s semantics. The initial symbolic state is always $(\langle\,\rangle, \emptyset, \top)$, where $\langle\,\rangle$ is the empty path and $\top$ indicates a constraint that is always true. We identify σ with the set of pairs $\{x \mapsto e \mid \sigma(x) = e\}$.

The rules in line 1 of Fig. 1 are the usual rules for the skip statement and statement sequencing.

The rules in line 2 handle variable assignment. The assignment $x \leftarrow e$ where $e \in Expr$ simply replaces the entry for x in σ with the expression resulting from the evaluation of $\overline{\sigma}(e)$. The assignment $x \leftarrow \top$ enables variable x to be treated as symbolic: in this case a fresh symbolic value $\widetilde{x}$ is assigned to x. As we shall see, the constraints of ϕ are actually constraints over these symbolic values. Because no branching point is encountered, π and ϕ remain unchanged.

$$\frac{}{\langle \textbf{skip}, (\pi,\sigma,\phi)\rangle \rightarrow (\pi,\sigma,\phi)} \qquad \frac{\langle S_1, (\pi,\sigma,\phi)\rangle \rightarrow (\pi',\sigma',\phi')}{\langle S_1; S_2, (\pi,\sigma,\phi)\rangle \rightarrow \langle S_2, (\pi',\sigma',\phi')\rangle} \tag{1}$$

$$\frac{\sigma'(v) = \begin{cases}\sigma(x) & \text{if } v \neq x\\ \overline{\sigma}(e) & \text{if } v = x\end{cases}}{\langle x \leftarrow e, (\pi,\sigma,\phi)\rangle \rightarrow (\pi,\sigma',\phi)} \qquad \frac{\sigma'(v) = \begin{cases}\sigma(x) & \text{if } v \neq x\\ \widetilde{x} & \text{if } v = x\end{cases}}{\langle x \leftarrow \top, (\pi,\sigma,\phi)\rangle \rightarrow (\pi,\sigma',\phi)} \tag{2}$$

$$\frac{\phi \wedge \overline{\sigma}(b) \not\models \bot \quad \phi' = \phi \wedge \overline{\sigma}(b) \quad \pi' = \pi \oplus \ell^1}{\langle (\ell)\ \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}, (\pi,\sigma,\phi)\rangle \rightarrow \langle S_1, (\pi',\sigma,\phi'\rangle} \tag{3}$$

$$\frac{\phi \wedge \neg\overline{\sigma}(b) \not\models \bot \quad \phi' = \phi \wedge \neg\overline{\sigma}(b) \quad \pi' = \pi \oplus \ell^0}{\langle (\ell)\ \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}, (\pi,\sigma,\phi)\rangle \rightarrow \langle S_2, (\pi',\sigma,\phi')\rangle} \tag{4}$$

$$\frac{\phi \wedge \neg\overline{\sigma}(b) \not\models \bot \quad \phi' = \phi \wedge \neg\overline{\sigma}(b) \quad \pi' = \pi \oplus \ell^0}{\langle (\ell)\ \textbf{while}\ b\ \textbf{do}\ S\ \textbf{od}, (\pi,\sigma,\phi)\rangle \rightarrow (\pi',\sigma,\phi')} \tag{5}$$

$$\frac{\phi \wedge \overline{\sigma}(b) \not\models \bot \quad \phi' = \phi \wedge \overline{\sigma}(b) \quad \pi' = \pi \oplus \ell^1}{\langle (\ell)\ \textbf{while}\ b\ \textbf{do}\ S\ \textbf{od}, (\pi,\sigma,\phi)\rangle \rightarrow \langle S; (\ell)\ \textbf{while}\ b\ \textbf{do}\ S\ \textbf{od}, (\pi',\sigma,\phi')\rangle} \tag{6}$$

Figure 1 Semantics of symbolic execution.

The rules in lines 3–4 show the semantics of the "if-then-else" statement. In line 3, we first check if the constraint $\phi \wedge \overline{\sigma}(b)$ is satisfiable with a suitable *constraint solver*. If so, the "then" branch is feasible and the constraint $\overline{\sigma}(b)$ is added to ϕ. In this case, given that ℓ is the location of the "if-then-else" statement, we also update π with the path π' obtained by appending ℓ^1 to π (we use $\oplus$ to denote the append operation). The map σ remains unchanged, because no symbolic variable is updated.

The rule in line 4 for the "else" branch is totally symmetric: we just consider $\neg\overline{\sigma}(b)$ instead of $\overline{\sigma}(b)$ and append ℓ^0 instead of ℓ^1. Note that symbolic execution is *nondeterministic* in the sense that at each branch point we can follow both branches. Indeed, in general, given constraint C we might find both an assignment ξ satisfying C and one ξ' satisfying $\neg C$.

The rules in lines 5–6 give the semantics of "while" loops. Rule 5 is basically the same as rule 4, while rule 6 is similar to rule 3: the difference is that here we can execute the loop an arbitrary number of times. This is one reason to consider symbolic execution *dynamic* (unless it is somehow enriched with an oracle or some mechanism for inductive reasoning): unlike static analysis, it may get stuck in loops. The applications to debugging and test generation therefore make use of termination criteria based on resource consumption.

▶ **Example 1.** Fig. 2 shows a simple $\mathcal{L}$ program, where $\langle \mathit{StmtA} \rangle$ and $\langle \mathit{StmtB} \rangle$ are unspecified statements. Variable y is symbolic, so before the while loop the symbolic trace is $(\langle\,\rangle, \{x \leftarrow 0, y \leftarrow \widetilde{y}\}, \emptyset)$. The $y > 0$ condition of the while loop at location ℓ_1 (line 3) is then evaluated.

On the one hand, we invoke a constraint solver to check if $\neg\overline{\sigma}(y > 0) = \neg(\sigma(y) > \sigma(0)) = \neg(\widetilde{y} > 0) = \widetilde{y} \leq 0$ is satisfiable (see rule 5 of Fig. 1). Assuming that $\widetilde{y}$ has a numeric domain with a lower bound smaller than or equal to 0, this constraint is clearly satisfiable (e.g., $\widetilde{y} = 0$ is a solution) so ℓ_1^0 is added to π, $\widetilde{y} \leq 0$ is added to ϕ and we skip the body of the loop.

$x \leftarrow 0$;
$y \leftarrow \top$;
while $y > 0$ **do** ▷ Location ℓ_1
  $x \leftarrow x + 1$;
  $y \leftarrow y - 2$;
od;
if $x > 1117$ **then** ▷ Location ℓ_2
  $\langle \mathit{StmtA} \rangle$;
else
  $\langle \mathit{StmtB} \rangle$;
fi

Figure 2 A simple $\mathcal{L}$ program.

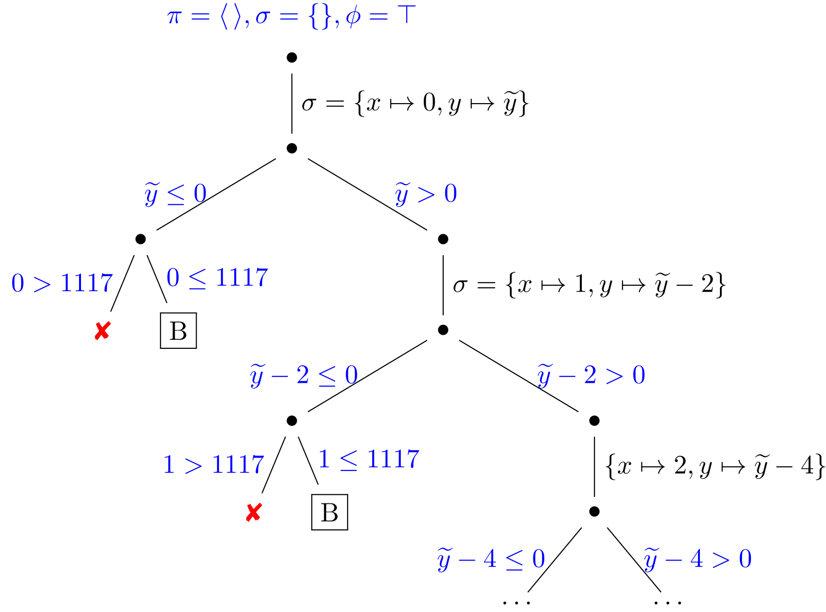


Figure 3 Top part of the symbolic execution tree for the program in Fig. 2.

On the other hand, we also consider the case where the $y > 0$ condition holds: we invoke the solver to check if $\overline{\sigma}(y > 0) = \widetilde{y} > 0$ is satisfiable and we keep on iterating the loop (see rule 6 of Fig. 1). Clearly, if the domain of $\widetilde{y}$ has no upper bound and no halting criterion is set (e.g., a timeout or a maximum number of loops iterations) the computation will not converge. This can be seen from the rightmost branch of the symbolic execution tree in Fig. 3, where the constraint $\widetilde{y} - 2k > 0$ is repeatedly added to ϕ after the k-th evaluation of the while loop condition, corresponding to a k-length path $\langle \ell_1^1, \ell_1^1, \ldots, \ell_1^1 \rangle$.

The if-then-else statement at location ℓ_2 is then processed using rules 3 and 4 of Fig. 1. In the first case, we try to solve the constraint $\phi \wedge \overline{\sigma}(x > 1117)$, where ϕ is the path condition at location ℓ_2. This constraint can only be satisfied if the while loop is executed $k > 1117$ times and $\widetilde{y} - 2k \leq 0$. This invariant cannot be inferred by symbolic execution, hence to find that $\langle \mathit{StmtA} \rangle$ is reachable the symbolic interpreter has to go through the while loop at least 1118 times, hoping that the resource limit is not reached earlier.

Similarly, we apply rule 4 and solve $\phi \wedge \neg\overline{\sigma}(x > 1117)$ to reach $\langle \mathit{StmtB} \rangle$. ⌟

As mentioned, the constraints of ϕ form a path condition because they refer to the conditions under which a path π is feasible, i.e., π is feasible if and only if ϕ is satisfiable. It is important to note that symbolic execution welcomes the definition of branch points and program paths, but in general has no notion of program points. If we define $loc(\pi)$ as the set of all the program points traversed along the execution of path π, a path condition ϕ for π is a *sufficient* condition to cover all the locations of $loc(\pi)$. However, ϕ is *not necessary*: different path conditions $\phi', \phi'', \phi''', \ldots$ not entailing ϕ can equally cover $loc(\pi)$. Think for example of Fig. 2: statement $\langle StmtB \rangle$ can be reached via $\pi = \langle \ell_1^1, \ell_1^0, \ell_2^0 \rangle$ (the while loop is executed once) with associated path condition $\phi = \widetilde{y} > 0 \wedge \widetilde{y} - 2 \leq 0$, and $\pi' = \langle \ell_1^1, \ell_1^1, \ell_1^0, \ell_2^0 \rangle$ (the while loop is executed twice) with associated path condition $\phi' = \widetilde{y} > 0 \wedge \widetilde{y} - 2 > 0 \wedge \widetilde{y} - 4 \leq 0$. Both these paths cover the same program points: $loc(\pi) = loc(\pi')$, even if $\pi \neq \pi'$ and $\phi \wedge \phi' \models \bot$.

The output of symbolic execution for a given program is a set Θ of symbolic traces, each of which corresponds to a path from the entry point of the program to its end point (including truncated paths if a termination criterion, e.g., a timeout, is met). We can define the set $\mathbb{T}(\Theta)$ of all the *concrete traces* corresponding to Θ as:

$$\mathbb{T}(\Theta) = \bigcup_{(\pi,\sigma,\phi) \in \Theta} \left\{ (x, \ell) \mapsto \text{ if } \ell \in loc(\pi) \text{ then } [\![\overline{\sigma}(x)]\!]_\xi \text{ else } \top \;\middle|\; \xi \models \phi \right\}$$

where $[\![\cdot]\!]_\xi : Exp \to Val$ is recursively defined by:

$$\begin{aligned} [\![c]\!]_\xi &= c && \text{for each } c \in Val \\ [\![x]\!]_\xi &= \xi(x) && \text{for each } x \in Var \\ [\![f(e_1, \ldots, e_k)]\!]_\xi &= f([\![e_1]\!]_\xi, \ldots, [\![e_k]\!]_\xi) && \text{for each } k \in \mathbb{N}, f \in Fun^k \end{aligned}$$

Each symbolic trace $(\pi, \sigma, \phi) \in \Theta$ can be unfolded into k concrete traces following π, where k is the (possibly infinite) number of solutions of path condition ϕ. Note that if a location ℓ is uncovered by a symbolic trace, we are not able to say that ℓ is unreachable in general because, as seen above, symbolic execution may miss concrete states. The set $\mathbb{T}(\Theta)$ is therefore an *under-approximation* of the set of all the feasible concrete states for a given program, i.e., there may exist a feasible concrete trace $\tau \notin \mathbb{T}(\Theta)$.

3.1 Concolic Testing

Historically, symbolic execution was suggested as a way of generating compact suites of test inputs, i.e., small sets producing large coverage. However, a number of issues have hindered its spread. Among these, we mention:

1. the source program may call library functions or make system calls
2. the underlying constraint solver may not be efficient or expressive enough to solve a given path condition
3. even simple programs tend to generate huge numbers of paths.

Concolic testing (or dynamic symbolic execution) was proposed [20] to overcome the first issue. Concolic is a portmanteau for concrete/symbolic, as concolic testing is a hybrid, maintaining both concrete and symbolic states, while executing a program. Thus, it has to be *seeded* with concrete values for symbolic variables. As it executes, concolic testing records alternative path constraints that can lead to new execution paths. A constraint solver is used to decide which ones are feasible and to provide new concrete inputs for the next path to explore.

Formally, we can define a *concolic trace* as a quadruple $(\rho, \pi, \sigma, \phi)$ where $\rho : \mathit{Var} \to \mathit{Val}$ is a *concrete state* and (π, σ, ϕ) is a *symbolic trace*. The concrete state ρ defines assignments of concrete values to symbolic variables. Each $\mathcal{L}$ assignment $x \leftarrow \top$ denoting a symbolic variable x is repeatedly replaced with a concrete assignment $x \leftarrow \rho(x)$, where the value $\rho(x)$ is decided at each iteration of the concolic testing according to the path condition found in the previous concolic iteration.

The semantics of concolic testing is very similar to that of symbolic execution. The main difference is that we actually execute the program, because symbolic variables now also take concrete values. So, the concolic execution is purely *deterministic*. The constraint solver is not used to evaluate the conditions at each branch point, because they are evaluated at runtime while executing the program. Instead, it is used to *generate* the concrete values to be assigned to the corresponding symbolic variables in the next concolic iteration.

Path conditions ϕ are recorded as for the symbolic execution. At the end of the concrete execution, a constraint C is removed from ϕ (typically the last). Then, a constraint solver is used to solve $\phi \wedge \neg C$. If there is a solution, we have new input values that will be used to feed the symbolic variables at the next iteration. Otherwise, the concolic process backtracks and a new constraint, not already negated, will be flipped. This process is repeated until all the constraints of ϕ are negated (or a termination criterion is met).

▶ **Example 2.** Let us see how concolic testing works on the $\mathcal{L}$ program in Fig. 2. Let us suppose that initially the value 0 is assigned to symbolic variable y (i.e., $\rho(y) = 0$). Both the conditions of the while loop and the "if" statement evaluate to false, so the concrete execution reaches statement $\langle \mathit{StmtB} \rangle$ with path condition $\phi = \neg(y > 0)$ (condition $x > 1117$ is not considered because x is not symbolic).

Then, $\neg(y > 0)$ is negated and a constraint solver will be used to solve $y > 0$. Assuming it returns the solution $y = 1$, we have a new concrete state (i.e., $\rho(y) = 1$) and hence we repeat the concrete execution by setting $y \leftarrow 1$. In this case, the loop is executed exactly once and $\langle \mathit{StmtB} \rangle$ is reached again, but this time with path condition $\phi = y > 0 \wedge \neg(y - 2 > 0)$. So we flip $\neg(y - 2 > 0)$ (we cannot flip $y > 0$, because it was already negated) and we solve $y > 0 \wedge y - 2 > 0$.

Assuming the solver returns the solution $y = 3$, we repeat the concrete execution by setting $y \leftarrow 3$ and so on, until all the constraints have been negated or the termination criterion is met. Note that the convergence of concolic testing also depends on the generated solutions. For example, if the solution of $y > 0$ was $y = 5000$ we would have reached $\langle \mathit{StmtA} \rangle$ right after the first iteration (by executing, however, the while loop 2500 times). ⌟

Concolic testing yields a set of concolic traces Γ containing both the concrete states and the symbolic traces for each explored path. From this point of view, concolic testing generalizes symbolic execution and we may define the concrete traces corresponding to Γ as $\mathbb{T}(\{(\pi, \sigma, \phi) \mid (\rho, \pi, \sigma, \phi) \in \Gamma\})$. However, often the symbolic computation is only used to generate the next input, so the path conditions can be overwritten at each concolic iteration – there is no need to keep track of them. We can therefore consider the output of concolic testing simply as $\{\rho \mid (\rho, \pi, \sigma, \phi) \in \Gamma\}$.

Concolic testing can be seen as an under-approximation of symbolic execution because it only considers a *witness* for each explored path, i.e., a solution for each path condition. Instead, symbolic execution defines *invariants* over program paths, i.e., necessary and sufficient conditions for the paths to be feasible. The ultimate goal of concolic testing is not to characterize the paths explored, but to maximize the overall *coverage* of a program, which we can define as the set $\bigcup_{\pi} \mathit{loc}(\pi)$ of all the locations traversed by any explored path π of Γ.

Concolic testing alleviates some of the issues of symbolic execution. Indeed, we can perform concolic testing without modelling library functions, system calls or third-party software by *directly invoking* all the functions. Constraint solving is more lightweight because constraints are simplified by the replacement of symbolic variables with concrete values. Importantly, we can simply *ignore* or approximate unsupported constraints. In this way, completeness is sacrificed but what is gained is that symbolic execution can proceed more smoothly. In many practical applications this is totally acceptable, provided that "good enough" coverage is achieved in reasonable time.

Thanks to a greater simplicity and efficiency w.r.t. symbolic execution, as well as the progress made by constraint solvers over the last years, concolic testing as become increasingly popular, and several concolic testing frameworks have been developed (e.g., DART [20], CUTE [34], jCUTE [33], KLEE [10]). The most recent tools for symbolic execution of C achieve considerable performance gains, for example by utilising tools that can perform *runtime instrumentation of binary code* (QSYM [38]), albeit at the cost of architecture dependence, or by *compiling* symbolic execution directly into LLVM bitcode (SymCC [29]), rather than interpreting bitcode, as done by KLEE.

4 Abstract Interpretation

Abstract interpretation is a well-established framework for static analysis. It is commonly used for *invariant generation*, i.e., for inferring program properties that hold for each possible program execution. To do so, it makes use of *abstract domains* for approximating sets of concrete runtime states.

Abstract interpretation is in a sense dual to symbolic execution. Indeed, plain symbolic execution under-approximates the set of possible concrete traces and can prove *reachability* (a semi-decidable problem in general). Analyses based on abstract interpretation usually over-approximate the set of concrete states, which means they can sometimes prove *unreachability*. However, one thing in common between these techniques is that they both, implicitly or explicitly, collect constraints along their execution. For symbolic (and concolic) execution, this aspect is evident. For abstract interpretation, we just have to make a little effort to see that abstract domains actually represent constraints defining invariant properties over the program variables.

For example, if the abstract value for an integer variable x is $Even$, the associated constraint is $x = 2k$ with $k \in \mathbb{Z}$. The direct product implicitly defines constraint *conjunctions*, e.g., $(Even, Pos, [-3, 8])$ corresponds to the constraint $x = 2k \wedge x > 0 \wedge -3 \leq x \leq 8$, whose feasible solutions are $\{2, 4, 6, 8\}$. A reduced product would refine these constraints into $x = 2k \wedge x > 0 \wedge 2 \leq x \leq 8$. In the constraint solving world, we are typically only interested in finding a feasible solution. Abstract interpretation aims instead to find a minimal set of constraints wrapping *all* the feasible solutions. The abstract domains define what *type* of constraints we are allowed to use to describe the invariants.

The link between constraints and abstract interpretation is clearer when we consider *relational* abstract domains. The simple non-relational domains seen above are efficient and easy to implement, but do not take into account the relations between variables, and thus tend to be imprecise. Relational domains instead are actually constraints capturing the relations between different variables. Examples of well known relational domains for numeric abstractions include linear congruence [22], octagons [27], and convex polyhedra [16]. For instance, the domain of convex polyhedra uses linear constraints of the form $a_1x_1 + \cdots + a_nx_n \leq b$, where the x_i are variables and the a_i, b are constants, to model the invariants. This domain offers considerable precision but also has high computational complexity (exponential in the worst case).

Formally defining a semantics as done in Section 3 capturing the abstract execution is not easy because there can be different ways of conducting the analysis and computing the invariants. For example, abstract domains can change dynamically (e.g., the CEGAR approach guides abstraction refinement via counter-example generation [11]). Moreover, unlike symbolic execution, abstract interpretation is not necessarily executed in a forward way. Instead of following the rules of an operational semantics, it effectively reasons *about* the execution flow of the program.

If *Abs* is a collection of abstract domains, we can define the *abstract state* for a given program as a map $\mathit{Var} \to \bigcup \mathit{Abs}$ from program variables to abstract elements. We denote the *abstract trace* with a pair (δ, λ) where $\delta : \mathit{Var} \to \mathit{Abs}$ maps program variable x to the chosen abstract *domain* $\delta(x)$ for it (including domain products and relational domains) and $\lambda : \mathit{Var} \times \mathit{Loc} \to \bigcup \mathit{Abs}$ returns the abstract *element* $\lambda(x, \ell) \in \delta(x)$ for x at program point ℓ (for simplicity, we assume that δ never varies during the program analysis). If variables $x_1, \ldots, x_k$ are abstracted with the same relational domain, then $\delta(x_1) = \cdots = \delta(x_k)$ and $\lambda(x_1) = \cdots = \lambda(x_k)$.

For example, given variable x and location ℓ, if $\delta(x)$ is the domain of intervals we may have $\lambda(x, \ell) = [-2, 7]$ but not $\lambda(x, \ell) = \mathit{Even}$. If $\delta(x) = \mathit{Parity} \otimes \mathit{Sign}$, a valid abstract element is $\lambda(x, \ell) = (\mathit{Odd}, NotPos)$. If both $\delta(x)$ and $\delta(y)$ are the octagon domain, we can have $\lambda(x, \ell) = \lambda(y, \ell) = \{x + y \leq 3, -x + y \leq 0, -x \leq 5, y \leq -1\}$ while, e.g., $\{x^2 + y^2 \leq 1\}$ is not a valid octagon.

We can define the *abstract execution* as the process of deriving *the* abstract trace (δ, λ) for a given program. In fact, here we can have at most one abstract trace per program, and not a collection of traces as happens for symbolic and concolic execution. This holds because the abstract trace over-approximates *all* the feasible concrete traces of the program. At a high level, abstract execution for $\mathcal{L}$ follows the control flow graph and updates (δ, λ) according to the statement encountered, e.g.:

- Initially $\lambda(x, \ell) = \bot_{\delta(x)}$ for each program variable x and location ℓ, except for the location ℓ_0 of the entry point of the program, for which $\lambda(x, \ell_0) = \top_{\delta(x)}$.
- For $(\ell)\ x \leftarrow e\ (\ell')$, the abstract value $\lambda(x, \ell')$ is defined according to the transfer function for the concrete expression e. For relational domains, we may also update $\lambda(y, \ell')$ for each variable y occurring in e.
- For (ℓ) **if** b **then** $(\ell_1)\ S_1\ (\ell'_1)$ **else** $(\ell_2)\ S_2\ (\ell'_2)$ **fi** (ℓ_3), condition b is used to refine the abstract elements $\lambda(x, \ell_1)$ and condition $\neg b$ is used to refine the abstract elements $\lambda(x, \ell_2)$ for each variable x involved in b. Once S_1 and S_2 are processed, the control flow merges and so we join the abstract value of each variable occurring in S_1 or S_2: $\lambda(x, \ell_3) = \lambda(x, \ell'_1) \sqcup_{\delta(x)} \lambda(x, \ell'_2)$.
- For (ℓ) **while** b **do** $(\ell_1)\ S\ (\ell'_1)$ **od** (ℓ_2), condition b is used to refine the abstract elements $\lambda(x, \ell_1)$ and then the widening operation $\lambda(x, \ell_1) \nabla_{\delta(x)} \lambda(x, \ell'_1)$ is repeatedly applied until a fixpoint is reached. Condition $\neg b$ is used to refine $\lambda(x, \ell)$, and then the join operation is applied between such refined abstract element and the "stationary' element computed by ∇.

▶ **Example 3.** Consider again the $\mathcal{L}$ program in Fig. 2. Suppose that for variables x and y we chose the sign domain and the interval domain respectively. At location ℓ_1, just before the while loop, we have $\lambda(x) = \mathit{Zero}$ and $\lambda(y) = \top$. At location ℓ_2, just before the if-then-else statement, the abstract execution is able to determine that we have $\lambda(x) = NotNeg$ (i.e., $x \geq 0$) and $\lambda(y) = (-\infty, 0]$. However, a more precise analysis would be able to infer that $y \in [-1, 0]$. Even more precisely, a relational analysis may detect the invariant $2x + y - y_0 = 0$ where y_0 is the value of y before entering the while loop. ◢

Each invariant $\lambda(x, \ell)$ is to all effects a constraint over the possible values that x can take at location ℓ. The type of this constraint is defined by $\delta(x)$: we have unary constraints for non-relational domains, conjunctions of constraints for domain products, and (conjunctions of) k-ary constraints for relational domains involving k variables. If we denote with $[\![a]\!]_{\mathcal{A}}$ the constraint corresponding to an abstract element $a \in \mathcal{A}$, we can define the set of the concrete traces corresponding to an abstract trace (δ, λ) as:

$$\mathbb{T}(\delta, \lambda) = \{(x, \ell) \mapsto \xi(x) \mid \xi \models [\![\lambda(x, \ell)]\!]_{\delta(x)}\}.$$

In practice, for each $x \in \mathit{Var}$ and $\ell \in \mathit{Loc}$, we associate to the pair (x, ℓ) the concrete value $\xi(x)$, where ξ is a solution for the constraint corresponding to the abstract element $\lambda(x, \ell)$.

As mentioned, abstract execution returns a single abstract trace for each program. In general, if $\mathbb{T}^*$ is the set of all the feasible concrete traces for a given program, Θ is the set of the symbolic traces resulting from a symbolic execution of that program, and (δ, λ) is the abstract trace resulting from its abstract execution, we have that:

$$\mathbb{T}(\Theta) \subseteq \mathbb{T}^* \subseteq \mathbb{T}(\delta, \lambda).$$

In other words, symbolic execution under-approximates the feasible concrete states, while abstract interpretation over-approximates them. Again, the assumption made here is that symbolic execution operates without assistance from oracles or added induction tools. Moreover, we assume that the underlying abstract execution is *sound*.

5 Synergy

In this section we bring things together by discussing some synergies between the worlds of symbolic execution, abstract interpretation, and constraint solving. In particular, we show how abstract interpretation can enrich symbolic execution through program transformation.

5.1 Abstract interpretation and constraint solving

As seen in Section 4, there is an implicit bond between abstract interpretation and constraint solving. Because the invariants computed by abstract interpretation are actually constraints over-approximating concrete states, constraint solvers can be used to improve the precision of the analysis by *refining* the abstract domains. Constraint solving can soundly rule out infeasible configurations and possibly detect unsatisfiability. Moreover, it can be used to generate counterexamples. For example, Ponsini, Michel and Rueher [30] present a hybrid approach for the abstract interpretation of floating-point programs using constraint programming to tighten the abstract domains and therefore reduce the number of false alarms.

What constraint solving can learn from abstract interpretation is the use of abstract domains to represent the domain of the decision variables and the relations between them – especially for non-trivial, structured types. For example, the dashed string abstraction introduced in [3] for string constraint solving is based on the Bricks abstract domain introduced in [13] for the analysis of strings.

5.2 Symbolic execution and constraint solving

Constraint solving and symbolic (and concolic) execution are strongly coupled because path conditions are iteratively solved with an underlying constraint solver. The expressiveness and the efficiency of the solver have huge impact on the performance of the symbolic execution: the

better the solver, the better the symbolic execution. Arguably, the remarkable improvements of constraint solvers over the last decades positively affected the development of symbolic and concolic execution frameworks.

Symbolic execution can also help the development of new and better constraint solvers. The path conditions arising from program analysis can suggest the development of new types of variables, constraints and search heuristics (e.g., aggregate types and complex operations) and be used as benchmarks – generated for free from the program source – to validate and evaluate constraint solvers.

5.3 Abstract interpretation and symbolic execution

Somehow, abstract interpretation is performed by "symbolically" executing the input program, not necessarily in a forward way, using abstract values instead of the concrete ones. In principle, because it works by over-approximations, it can only generate false positives. In practice, unsound domains are sometimes used: in this way, we can also have false negatives. Symbolic (or concolic) execution might be used to *post-process* the abstract execution via counterexample generation w.r.t. a property ϕ of interest, provided that we can encode that ϕ with a corresponding Boolean expression b in the input language. In this way, we can check if ϕ (or $\neg\phi$) holds at program point ℓ by inserting at that point an "if-then-else" construct having guard condition b.

What is probably more interesting is instead the other way round: how abstract interpretation can help symbolic execution.

For example, a common problem for "classical" symbolic execution, as we defined it in the previous sections, is that it often gets stuck exploring loops for an intolerably long time. This happens because usually symbolic execution makes no attempt to reason about a program – all it does is following the rules defined by the transition system of its operational semantics. Abstract interpretation, on the other hand, steps outside of the operational semantics of symbolic execution by reasoning *about* loops rather than simply executing them.

Abstract interpretation can be used *transform* the source program in a target program where symbolic execution can escape loops. For example, *loop counters* can sometimes aid the abstraction enhanced symbolic execution [2]. The addition of counters does not interfere with the semantics of the program – it is essentially a semantics-preserving transformation. The kind of source-to-source transformations that aim to preserve some testing metric (but possibly not the semantics of the program) are called *testability transformations* [23]. Examples include the merging or splitting of loops, induction variable substitution, changing the type of a variable from float to int, and others [23].

A possible way of enriching the symbolic execution of the $\mathcal{L}$ language is to augment it with *assumptions*, whose purpose is to actually *replace* the while loops with the invariants that abstract interpretation yields at the end of each loop. In practice, for transforming while loops into corresponding "assume" statements, we replace rule 6 of Fig. 1 with:

$$\frac{\phi \wedge \overline{\sigma}(b) \not\models \bot \quad b' = \bigwedge_{x \in S} [\![\lambda(x,\ell)]\!]_{\delta(x)} \quad \sigma'(x) = \begin{cases} \widetilde{x} & \text{if } x \in S \\ \sigma(x) & \text{if } x \notin S \end{cases}}{\langle \textbf{while } b \textbf{ do } S \textbf{ od } (\ell), (\pi, \sigma, \phi)\rangle \to \langle (\ell) \textbf{ assume } b', (\pi, \sigma', \phi)\rangle}$$

where $b' = \bigwedge_{x \in S} [\![\lambda(x,\ell)]\!]_{\delta(x)}$ is the conjunction of all the constraints corresponding to the invariant $\lambda(x,\ell)$, for each variable x occurring in statement S. Note that this rule does not modify either the path π or the path condition ϕ. However, map σ is updated by assigning a new, fresh variable $\widetilde{x}$ to each variable x occurring in statement S. This somehow has the effect of forgetting the history of these variables, in order to avoid conflicts with the introduced invariant b'.

The rule for the assume statement is simply defined as:

$$\frac{\phi \wedge \overline{\sigma}(b) \not\models \bot \quad \phi' = \phi \wedge \overline{\sigma}(b) \quad \pi' = \pi \oplus \ell^1}{\langle (\ell) \textbf{ assume } b, (\pi, \sigma, \phi) \rangle \to (\pi', \sigma, \phi')}$$

Clearly this kind of loop elimination can introduce false positives, but it allows symbolic execution to escape loops (the more precise the abstract execution is, the less likely this is to generate false positives). This transformation relies on the fact that test data generation is a *forgiving* application: the possibility of *path divergence* [4] is already a reality in concolic testing, and the damage risked is simply the generation of sub-optimal test sets. One can also consider a parametric approach where the while loop is executed at most k times: if after k iterations the loop condition still holds, then the while loop is transformed.

▶ **Example 4.** Consider once again the $\mathcal{L}$ program in Fig. 2. Let us suppose that the abstract interpretation is conducted with the interval domain, so after the while loop we have the following invariants: $\lambda(x, \ell) = [0, +\infty)$ and $\lambda(y, \ell) = (-\infty, 0]$ where ℓ is the location corresponding to the end of the loop (line 6). If we apply the loop elimination described above, we set $\sigma = \{x \leftarrow \widetilde{x}, y \leftarrow \widetilde{y}\}$ and we replace the while statement with: (ℓ) **assume** $x \geq 0 \wedge y \leq 0$.

Then, the assume statement is evaluated and the symbolic execution can proceed with the evaluation of the “if-then-else” statement. Clearly, this approach ensures the termination of the symbolic execution but it does not guarantee its soundness. For example, the evaluation of the “if” condition $x > 1117$ can succeed with solution $\widetilde{x} = 1118$, $\widetilde{y} = 0$ not corresponding to any feasible concrete state. However, with more precise (and relational) domains many of these spurious configurations can be ruled out (e.g., by adding the invariant $2\widetilde{x} + \widetilde{y} - y_0 = 0$ where y_0 is the value of y before executing the while loop) ⌟

6 Related Work

A number of systems for program verification have been based on symbolic execution. VeriFast [24] uses separation logic to verify various properties of (subsets of) C and Java, including properties of functions that manipulate inhabitants of recursively defined data types such as lists and trees.

The KeY project [6] (`https://www.key-project.org/`) is an active Java program verification project. KeY uses a sequent-based dynamic logic. Users can provide method contracts and loop invariants, with system support for checking the validity of invariants. The system can then use the invariants in place of the loops that are thus abstracted. This may simplify the verification task, and solve the problem of symbolic execution getting caught in loops (naturally, if the abstraction is too imprecise, it may also fail to enable a desired proof). Abstraction can also be used to model components for which the source code is unavailable. For consistency, such models must over-approximate the possible runtime states (in contrast to the “concretizing” approach used by concolic testing tools).

Of particular interest in our context is the integration of abstract interpretation with KeY [37]. The symbolic execution based reasoning about the values that a variable can take is interleaved with reasoning about value sets (which are elements of abstract domains), and as a result, some invariants can be generated automatically. The abstract domains may be refined in the process. From available descriptions, it appears that *values*, rather than program states, are abstracted (so that relational analysis is excluded).

It is interesting to compare the KeY project’s use of abstraction in symbolic execution based verification with the use we suggest for concolic testing. A static analysis that provides “attribute independent” (or non-relational) abstraction can be of great value in the verification

context, but it is of little use to concolic testing. To be of any value, an oracle for a concolic testing tool has to be able to provide non-trivial information about how the values of different variables are related. Fortunately, the availability of sophisticated tools for abstract interpretation allows a clean separation of concern, effectively providing us with a highly parametric oracle.

Another application with a very focused aim is the static analysis used by Feist et al. [18] to identify "use after free" vulnerabilities in a binary-code. Static analysis is used to compute "weighted slices" which are then used to guide DSE. The weight attached to a slice reflects the degree to which the slice is able to include an "allocate then free then use" pattern for some pointer variable.

Shastry et al. [35] apply static analysis to improve non-grammar based fuzzing. A simple static analysis is used to construct better fuzzing input dictionaries. Normally such dictionaries are created based on string literals found in the program under analysis. Shastry et al. show how a kind of taint analysis combined with backward slicing can help generate more effective dictionaries in the context of network applications.

Adding loop counters as a benign transformation to facilitate better analysis is also seen in work on abstract domains [36] and in symbolic execution. For example, the "loop-extended" grammar-based symbolic execution proposed by Saxena et al. [32] involves adding a new symbolic variable (or *trip count*) per loop. The trip count is related to the program's input format through further auxiliary variables, to capture how variables assigned in loops depend on the lengths and counts of elements in the program input. Godefroid and Luchaup [21] identify unbounded loops that use induction variables (in a linear manner). These loops are summarized by pre- and post-conditions, derived from inferred partial loop invariants, relating program inputs to the values of induction variables. No static analysis is involved in this.

7 Discussion

This paper has explored interactions between abstract interpretation, symbolic execution and constraint solving. We exposed the – sometimes implicit – bond that constraints have with both symbolic execution and abstract interpretation, and we put forward a view that symbolic execution is best considered a dynamic analysis.

We discussed a way of helping concolic testing escape loops via abstract interpretation and program transformation. In a context of under-approximating dynamic analysis it may, at first, seem surprising that an over-approximating static analysis can be of help. Indeed, it can be of help only because it can approximate sets of program states, rather than simply sets of values. That is, unlike the dynamic analysis, the static analysis can contribute *relational* information. We believe that this tells us something new and important about the nature of dynamic vs static analysis.

A variety of characterizations and definitions can be found in the literature, the most common one being that static analysis is an analysis that is valid for any execution of the program. In a similar vein, Tom Ball [5] characterises static analysis as "*program centric*" as opposed "*input centric*" dynamic analysis. He also sees the higher precision of information gained through dynamic analysis as an important characteristic.

We suggest that another important distinction is between what may be termed *value oriented* and *state oriented* analysis. Dynamic analysis, symbolic execution included, is usually value oriented: a "symbolic" runtime state is a mapping from variables to symbolic expressions, the latter describing sets of concrete *values*. Abstract interpretation, in contrast,

bases itself on a "collecting semantics", associating possible runtime states with each program point. An "abstract" runtime state is sometimes designed as a mapping from variables to abstract values (describing sets of concrete values), but it does not have to be a mapping. It could be a more fine-grained description in the form of a *relation* that describes a set of concrete *runtime states*. We have exemplified how this can be used to improve applications of symbolic execution, such as concolic testing. The only relational information considered by a concolic testing tool is how program variables may depend on input, that is, how they are related to symbolic variables (as expressed through path constraints). A static analysis can expose more complex relations between variables, thus providing additional information about runtime states – information that, for example, a test-data generating constraint solver can utilise.

In future work we would like to revisit the points made here, and develop a proper framework that can serve as a formal foundation for this discussion.

References

1. W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, editors. *Dedecutive Software Verification – The KeY Book*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016.
2. Eman Alatawi, Harald Søndergaard, and Tim Miller. Leveraging abstract interpretation for efficient dynamic symbolic execution. In G. Rosu, M. Di Penta, and T. N. Nguyen, editors, *Proc. 32nd IEEE/ACM Int. Conf. Automated Software Engineering*, pages 619–624. IEEE Comp. Soc., 2017.
3. Roberto Amadini, Graeme Gange, Peter J. Stuckey, and Guido Tack. A novel approach to string constraint solving. In J. C. Beck, editor, *Proc. 23rd Int. Conf. Principles and Practice of Constraint Programming*, volume 10416 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2017. doi:10.1007/978-3-319-66158-2_1.
4. Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys*, 51(3):50:1–50:39, 2018.
5. Thomas Ball. The concept of dynamic analysis. In O. Nierstrasz and M. Lemoine, editors, *Software Engineering – ESEC/FSE'99*, volume 1687 of *Lecture Notes in Computer Science*, pages 216–234. Springer, 1999.
6. Bernhard Beckert, Vladimir Klebanov, and Benjamin Weiß. Dynamic logic for Java. In *Deductive Software Verification – The KeY Book*, volume 10001 of *Lecture Notes in Computer Science*, pages 49–106. Springer, 2016.
7. Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.
8. Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT: A formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices*, 10(6):234–245, 1975.
9. R. M. Burstall. Program proving as hand simulation with a little induction. In *Information Processing: Proc. IFIP Congress 1974*, pages 308–314. North-Holland, 1974.
10. Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. 8th USENIX Conf. Operating Systems Design and Implementation*, volume 8, pages 209–224, 2008.
11. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification: Proc. 12th Int. Conf.*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.
12. Agostino Cortesi and Matteo Zanioli. Widening and narrowing operators for abstract interpretation. *Computer Languages, Systems and Structures*, 37(1):24–42, 2011.

13 Giulia Costantini, Pietro Ferrara, and Agostino Cortesi. A suite of abstract domains for static analysis of string values. *Software Practice and Experience*, 45(2):245–287, 2015.

14 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symp. Principles of Programming Languages (POPL'77)*, pages 238–252. ACM, 1977.

15 Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proc. 6th ACM SIGACT-SIGPLAN Symp. Principles of Programming Languages (POPL'79)*, pages 269–282. ACM, 1979.

16 Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear constraints among variables of a program. In *Proc. Fifth ACM Symp. Principles of Programming Languages (POPL'78)*, pages 84–97. ACM, 1978.

17 George B. Dantzig. Linear programming. *Operations Research*, 50(1):42–47, 2002.

18 Josselin Feist, Laurent Mounier, Marie-Laure Potet, Sébastien Bardin, and Robin David. Finding the needle in the heap: Combining static analysis and dynamic symbolic execution to trigger use-after-free. In *Proceedings of the 6th ACM Workshop on Software Security, Protection, and Reverse Engineering*, pages 2:1–2:12. ACM, 2016.

19 Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. Abstract interpretation over non-lattice abstract domains. In F. Logozzo and M. Fähndrich, editors, *Static Analysis*, volume 7935 of *Lecture Notes in Computer Science*, pages 6–24. Springer, 2013.

20 Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI'05)*, pages 213–223. ACM, 2005. doi:10.1145/1065010.1065036.

21 Patrice Godefroid and Daniel Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proc. 2011 Int. Symp. Software Testing and Analysis (ISSTA'11)*, pages 23–33. ACM, 2011. doi:10.1145/2001420.2001424.

22 Philippe Granger. Static analysis of linear congruence equalities among variables of a program. In Samson Abramsky and T. S. E. Maibaum, editors, *TAPSOFT'91: Proc. Int. Joint Conf. Theory and Practice of Software Development, Volume 1: Colloquium on Trees in Algebra and Programming (CAAP'91)*, volume 493 of *Lecture Notes in Computer Science*, pages 169–192. Springer, 1991.

23 Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.

24 Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the VeriFast program verifier. In K. Ueda, editor, *Programming Languages and Systems: Proc. 8th Asian Symp.*, volume 6461 of *Lecture Notes in Computer Science*, pages 304–311. Springer, 2010.

25 James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

26 Jean-Louis Laurière. A language and a program for stating and solving combinatorial problems. *Artificial Intellgence*, 10(1):29–127, 1978.

27 Antoine Miné. The Octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

28 Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 1974.

29 Sebastian Poeplau and Aurélien Francillon. Symbolic execution with SymCC: Don't interpret, compile! In *Proc. 2020 USENIX Security Symp.* USENIX, 2020. Could not find this on the USENIX Security 20 web site.

30 Olivier Ponsini, Claude Michel, and Michel Rueher. Verifying floating-point programs with constraint programming and abstract interpretation techniques. *Automated Software Engineering*, 23(2):191–217, 2016.

31 F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

32 Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. In *Proc. 18th Int. Symp. Software Testing and Analysis (ISSTA'09)*, pages 225–236. ACM, 2009. doi:10.1145/1572272.1572299.

33 Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In T. Ball and R. B. Jones, editors, *Computer Aided Verification: Proc. 18th Int. Conf.*, volume 4144 of *Lecture Notes in Computer Science*, pages 419–423. Springer, 2006.

34 Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *Proc. 10th European Software Engineering Conf.*, pages 263–272. ACM, 2005. doi:10.1145/1081706.1081750.

35 Bhargava Shastry, Markus Leutner, Tobias Fiebig, Kashyap Thimmaraju, Fabian Yamaguchi, Konrad Rieck, Stefan Schmid, Jean-Pierre Seifert, and Anja Feldmann. Static program analysis as a fuzzing aid. In M. Dacier, M. Bailey, M. Polychronakis, and M. Antonakakis, editors, *Research in Attacks, Intrusions, and Defenses: Proc. 20th Int. Symp. (RAID'17)*, volume 10453 of *Lecture Notes in Computer Science*, pages 26–47. Springer, 2017.

36 Arnaud J. Venet. The Gauge domain: Scalable analysis of linear inequality invariants. In P. Madushan and S. A. Seshia, editors, *Computer Aided Verification*, volume 7358 of *Lecture Notes in Computer Science*, pages 139–154. Springer, 2012.

37 Nathan Wasser, Reiner Hähnle, and Richard Bubel. Abstract interpretation. In *Deductive Software Verification – The KeY Book*, volume 10001 of *Lecture Notes in Computer Science*, pages 167–189. Springer, 2016.

38 Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *Proc. 27th USENIX Security Symp.*, pages 745–761. USENIX, 2018.

The Standard Model for Programming Languages: The Birth of a Mathematical Theory of Computation

Simone Martini
Department of Computer Science and Engineering, University of Bologna, Italy
INRIA, Sophia-Antipolis, Valbonne, France
http://www.cs.unibo.it/~martini
simone.martini@unibo.it

Abstract

Despite the insight of some of the pioneers (Turing, von Neumann, Curry, Böhm), programming the early computers was a matter of fiddling with small architecture-dependent details. Only in the sixties some form of "mathematical program development" will be in the agenda of some of the most influential players of that time. A "Mathematical Theory of Computation" is the name chosen by John McCarthy for his approach, which uses a class of recursively computable functions as an (extensional) model of a class of programs. It is the beginning of that grand endeavour to present programming as a mathematical activity, and reasoning on programs as a form of mathematical logic. An important part of this process is the *standard model* of programming languages – the informal assumption that the meaning of programs should be understood on an abstract machine with unbounded resources, and with true arithmetic. We present some crucial moments of this story, concluding with the emergence, in the seventies, of the need of more "intensional" semantics, like the sequential algorithms on concrete data structures.

The paper is a small step of a larger project – reflecting and tracing the interaction between mathematical logic and programming (languages), identifying some of the driving forces of this process.

to Maurizio Gabbrielli, on his 60th birthday

2012 ACM Subject Classification Social and professional topics → History of programming languages; Software and its engineering → General programming languages

Keywords and phrases Semantics of programming languages, history of programming languages, mathematical theory of computation

Digital Object Identifier 10.4230/OASIcs.Gabbrielli.2020.8

Funding *Simone Martini*: Research partially conducted while on sabbatical leave at the Collegium – Lyon Institute for Advanced Studies, 2018–2019. Partial support from French ANR project PROGRAMme.

Acknowledgements I am happy to thank Edgar Daylight for mentioning me Strachey's letter to the Computer Journal, and for the many critical reactions to the main thesis of this paper.

1 Introduction

Statements like "the Java programming language is Turing-complete", or "the C program '`int x=0; while 1 {x++;}`' is divergent", or even "the halting problem is undecidable for C programs", are common in most textbooks and in the practice of many computer scientists, despite the fact that they are *false*, for any actual implementation. The finiteness of any such implementation implies that the halting problem is decidable, being the system only a finite-state automaton; that the increment `x++` will produce after a finite time an overflow which will be caught by some hardware interrupt; and that any Java implementation cannot compute

© Simone Martini;
licensed under Creative Commons License CC-BY

functions requiring more resources than those of the universe. Of course, the statements become true when they are applied to a *model* of C or Java programs allowing for true unbounded arithmetic and unbounded (finite) resources. We propose to call *standard model* of programming languages this (informal) assumption, where programs are interpreted on an abstract machine with unbounded resources, and with true arithmetic. It is so "standard" that its use is almost never esplicitated in the literature (scientific, or in textbooks) – on the contrary, when *another* model is implied (for instance, one with finite arithmetic), that use is instead acknowledged and remarked.

We are so used to the fact that a finite computer program could be regarded as the infinite function it computes, that we tend to forget that this "double view" was not born together with the general purpose electronic computer – it is rather the result of a deliberate research agenda of the end of fifties and the early sixties of the last century, responding both to internal and external forces of the discipline that was going to be called, also thanks to that agenda, Computer *Science*. It is the beginning of that grand endeavour to present programming as a mathematical activity, and reasoning on programs as a form of mathematical logic. The paper will present some of the moments of this process, where the components of a mathematical theory of computations are introduced as the technical, formal elements of the informal standard model. Before delving into these developments, however, we cannot forget the founding fathers of the discipline, who already realised this double nature of computer programs, but whose insight did not pass, at that time, into the practice of computing.

2 The Pioneers

The two years from 1947 to 1949 are of particular interest for our story. First, it is when Herman Goldstine and John von Neumann wrote the second part [20] of a series of reports on the "mathematical and logical aspects of an electronic computing instrument". Its aim is to give an account of the "methods of coding and of the philosophy which governs it". A major methodological tool is the use of *flow diagrams* for expressing the dynamics of a computation. Flow diagrams are made of four distinct classes of boxes: operation, alternative, substitution, and assertion boxes. While boxes of the first two kinds contain operations that the machine will ultimately perform, the contents of an *assertion* box "are one or more relations." A box of this kind "never requires that any specific calculation be made, it indicates only that certain relations are automatically fulfilled whenever" the control reaches that point[1]. Assertion boxes are "logical expressions" that remain invariant during the computation – their only reason for being present in a diagram is that they are needed (or useful) in establishing that the diagram correctly renders "the numerical procedure by which the planner has decided to solve the problem", which is expressed in the "language of mathematics"[2]. For this reason, coding "has to be viewed as a logical problem and one that represents a new branch of formal logics." While the word "logic" (or "logical") does not necessarily refer to mathematical (formal, or symbolic) logic in the literature of that period[3], the reference to a "new branch

[1] Also substitution boxes do not specify any computation; they represent a change in *notation*, more specifically in the relation between the internal and the external notation, a significantly different perspective from the modern use of flow charts, see [29].

[2] We will come back to this duality between the *specification* and the *implementation* (in the today's terminology, of course) at the end of Section 4.

[3] "Logical" is usually opposed to "physical", or "electronical", like in "logical design of digital circuits"; or in the very title of the series in which [20] appears: "mathematical and logical aspects of an electronic computing instrument."

of formal logics" is explicit. Assertions bring mathematical logic into the very notation for writing programs; moreover, references to other notions of formal languages are present in other places of [20], like the distinction between free and bound variables. "Formal logic" is for Goldstine and von Neumann at the core of programming, as the discipline where assertions may be written and proven.

An analogous use of assertions for "checking [the correctness of] a large routine" will be proposed by Alan Turing two years later. In [54][4] he lucidly describes a proof of correctness as consisting of three different steps. First, "the programmer should make assertions about the various states that the machine can reach." Then, "the checker [i.e., the one doing the proof] has to verify that [these assertions] agree with the claims that are made for the routine as a whole." "Finally the checker has to verify that the process comes to an end." Observe the clarity by which full correctness (in today's terms) is spelled out, the distinction between specification ("claims that are made for the routine as a whole") and implementation, and the interplay between these two, where assertions play their role.

Turing brings us back to year 1947, when he also writes a paper with some remarks on mathematical logic in computer programming. In the "Lecture on Automatic Computing Engine" [53] he has already clear that programming a computing machine could be done, in principle, with languages much more sophisticated than the machine instructions that were available at the time. Indeed, "in principle one should be able to communicate [with these machines] in any symbolic logic, provided that the machine were given instruction tables which would allow it to interpret that logical system." Like Goldstine and von Neumann, also Turing sees a bright future for mathematical logic (to be understood, in this case, as the discipline of artificial languages): "there will be much more practical scope for logical systems than there has been in the past."

Haskell B. Curry in those same years uses his logical apparatus in order to program the new computing machines. Discussing an inverse interpolation routine and its correctness, in [10] he introduces a notion of *type* for memory words: those containing instructions (*orders*), and those containing data (*quantities*). Starting from this, as reconstructed by [15], he builds a surprising, non-trivial mathematical theory of programs, containing theorems analogous to the "well-typed expressions do not go wrong"[5] of [38], and he uses them to define classes of program transformations and compositions which are "safe" for the intended operational semantics. The presence of mathematical logic is so explicit that Curry's reports will get a review on the Journal of Symbolic Logic[6].

A lesser-known contribution is the early work of Corrado Böhm, a few years later. His thesis at ETH Zurich[7] esplicitely connects the new computing machines to the mathematical, abstract analysis of Turing, up to the claim that "les calculatrices les plus évoluées sont universelles, au sens spécifié par M. Turing." Under this assumption, Böhm may assume that all the general purpose, stored-program computers "sont, au point de vue logico-mathématique, équivalentes entre elles," so that he may choose a specific type of computer (a three-address machine) without any loss of generality. These remarks, and the explicit

[4] See also [39], for a reprint of the original paper, and a commentary.

[5] "Suppose we have an initial type determination for the initial program", that is an assignment of types to words which assign type "order" to any instruction, and type "quantity" to any data, then the memory "word at the control location is always [*scil.*, at any time during execution] an order" [10], number 26.

[6] By G.W. Patterson, JSL 22(01), 1957, 102-103.

[7] Written under the direction of E. Stiefel and P. Bernays, submitted in 1952, and published in 1954 [6]; see also Knuth's [29].

connection to Turing's theory, that to most of us seem obvious, were not (yet) part of the standard background of the people working in the field of automatic computing machines. Of course, they were known to some of the other giants (von Neumann and his peers *in primis*; Böhm's advisor Paul Bernays is the probable source who mentioned Turing's work to him), but it will be only much later that Turing will become the iconic figure of father for computer science [11][8]. Another striking observation in Böhm's thesis is that a program is seen under a double interpretation: as a description of the behaviour of a computer, and as the description of a "méthode numérique de calcul." Without forcing the interpretation, we may read this as one of the first explicit references to the duality between an operational description of the behaviour of an (abstract) machine, and the numerical function that results from that sequence of operations.

The far-sight of these pioneers should not obfuscate the fact that the role of logic in the early days of the digital computing machines (both their design and their programming) was modest, if not absent (e.g., [11, 14]). Programming those machines was more a technological affair, strictly coupled to the technology of the different computers. Despite the genial recognition by Turing that "any symbolic logic" could be used as a programming language, it is only during the fifties, and with graduality, that programming started to be perceived as a "linguistic activity" [42]. As for the correctness of programs, Knuth's recollection [28] is that at the end of the fifties "the accepted methodology for program construction was [...]: People would write code and make test runs, then find bugs and make patches, then find more bugs and make more patches, and so on. We never realized that there might be a way to construct a rigorous proof of validity." "The early treatises of Goldstine and von Neumann," and of Turing, and Curry, "which provided a glimpse of mathematical program development, had long been forgotten." In summary, while there are important relations with other parts of mathematics, like numerical analysis (e.g., Jim Wilkinson's backward error analysis), or automata theory (from von Neumann, to Kleene's regular events, to Rabin and Scott), or cybernetics, at the macro scale little happens on the explicit border between logic and the new field of computing, which at that same time struggled to be recognised as an autonomous scientific discipline[9]. In this process, the availability of computer-independent ("universal," in the terminology of the time) programming languages allowed the expression of *algorithms*

[8] Describing the relations between Turing's work – and especially the notion of Turing machine, – the modern digital computer, and computer programming, is well outside the scope of this paper. For some of the relations between Turing and von Neumann, see Stanley Frankel's letter to Brian Randell, quoted in [45]. An argument on the equivalence of Turing machines and McCulloch and Pitts's neuron nets "supplied with an infinite blank tape," can be found in von Neumann's [58]. A balanced review of the actual impact of Turing on computer science may found in Section 5 of Liesbeth De Mol's entry on Turing machines for the Stanford Encyclopedia of Philosophy [13], from which we quote the following. "Recent historical research shows also that one should treat the impact of Turing machines with great care and that one should be careful in retrofitting the past into the present." Only "in the 1950s then the (universal) Turing machine starts to become an accepted model in relation to actual computers and is used as a tool to reflect on the limits and potentials of general-purpose computers by both engineers, mathematicians and logicians." See also [22, 11].

[9] A struggle that was going to be long. The first *Computer Science* department of the US was established in 1962 at Purdue University; Samuel D. Conte, first Head of that department, will recall in a 1999 Computerworld magazine interview: "Most scientists thought that using a computer was simply programming – that it didn't involve any deep scientific thought and that anyone could learn to program. So why have a degree? They thought computers were vocational vs. scientific in nature" (quoted in Conte's obituary at Purdue University, 2002). Next computer science departments to be established would be those at the University of North Carolina at Chapel Hill, in 1964, and at Stanford in 1965. Still in 1967, Perlis, Newell and Simon (all of them will receive the Turing award; Simon will also be a Nobel laureate in Economics) feel the need of a letter to Science [41] to argue "why there is such a thing like computer science".

in a machine neutral way, thus making algorithms and their properties amenable to a formal study. Programming languages themselves were treated as an object of study, starting from the formal definition of Algol's syntax [1, 2]. For an entrance ticket to "science", however, a complete "theory" was needed, encompassing also semantics of programming languages and their use in program development.

3 Towards a General theory

The construction of a "mathematical theory of computation," (or a "mathematical science of computation") is the explicit goal of John McCarthy, starting from 1961 (when he was still at MIT) and covering his first period at Stanford, where he moved in 1963. In [35][10] he sketches an ambitious plan for a general theory, based on mathematical logic, that could serve as a foundation for computation, in such a way that "it is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last." After having dismissed numerical analysis (for being too narrow for a general theory), computability theory (for focussing on undecidability, instead of positive results, and for being uninterested on properties of algorithms), and finite automata theory (for being of no use in the treatment of computers, because they have too many states), it proceeds to list the "practical" (*sic*) results of the theory. The first is "to develop a universal programming language": Algol is a good step in the right direction, but it has several "weaknesses", among which the impossibility to describe different kinds of data[11]. Second, "to define a theory of the equivalence of computation processes," to be used to define and study equality preserving transformation. Let us esplicitate what McCarthy leaves unsaid: once we have an accepted *model* for the behaviour of a program, we may study under which transformations of (the syntactical presentation of) the program the behaviour remains invariant (in the model). A third goal goes in the same direction: "To represent computers as well as computations in a formalism that permits a treatment of the relation between a computation and the computer that carries out the computation." The models for both the program and its executing agent should be expressed in the same conceptual framework, in the same theory, so that one may express relations among the two, like the fact that (the model of) the computer that carries out the computation is "sound" with respect to (the model of) the program. Contrasting the lack of interest of recursive function theory for the positive results and for the properties of algorithms, a fourth goal is "to give a quantitative theory of computation [...] analogous to Shannon's measure of information." The paper does not elaborate further on this point; we may probably ascribe to this goal the development of computational complexity. Finally, a last purpose of a general theory would be "to represent algorithms by symbolic expressions in such a way that significant changes in the behavior represented by the algorithms are represented by simple changes in the symbolic expressions." Once again the point will not be taken up again in the rest of the paper, besides explaining that it is relevant for programs that "learn from experience". For our purposes, it suffices to stress the reference to the availability of (a model of) the behaviour of a program and to the interplay between the syntactic representation of the algorithm and that behaviour.

[10] The preliminary version is from 1961; the final 1963 version contains a new section on the "relations between mathematical logic and computation."
[11] Note, en passant, that the word "type" is still not used in this context, for a collection of homogeneous values; see [32].

The technical contents of the paper, of course, cannot match the *grandeur* of these goals. The paper introduces a theory of higher order, partial computable functions on arbitrary domains, defined by composition, conditional statements, and general recursion. Moreover, a uniform way to define new data spaces is presented – data space constructors are the Cartesian product, the disjoint union, and the power set, each of them equipped with its canonical maps, which are used to define functions on the new spaces from functions on the base spaces. Base data spaces could be taken as frugal as the single "null set", since natural numbers could be defined from it. The goal is to provide general, abstract mechanisms, instead of choosing an arbitrary new palette of primitive types.

The picture sketched in [35] is further developed in [34] (to "be considered together with my earlier paper"), which contains a first attempt for an epistemology for (the still unborn) computer science[12], and introduces again, after the pioneers, the problem of program correctness: "Instead of debugging a program, one should prove that it meets its specifications, and this proof should be checked by a computer program. For this to be possible, formal systems are required in which it is easy to write proofs." Of course, one needs first a semantics for the behaviour of a program. This is an important contribution of the paper: programs expressed as sequences of assignments and conditional go to's are given meaning as recursive functions acting on the set of current values of the variables (the "state vector"). Each statement of the program corresponds to such a recursive function, the meaning of a program being obtained by a suitable composition of these functions, in a *compositional* approach which will be the cornerstone of formal program semantics. Moreover, the proof technique of "recursion induction," already introduced in [35], is extended, so that it could be applied directly to programs without first transforming them to recursive functions. This last contribution is particularly relevant for our story: the program is implicitly understood as a representative for its meaning, so that one may argue on the program (a finite, textual object) for obtaining results on its model (an infinite function over the set of possible data).

McCarthy will not develop the formal semantics introduced in his two papers, and the creation of a mathematical semantics for programs is usually credited to Robert Floyd [18][13] and Tony Hoare [23]. Both are lucid on the need of a machine independent meaning of programs, and the need of formal theories for reasoning on programs. For Hoare, "computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning." *Sola scripta* are normative for the behaviour of a program: a model for the semantics is implicit and implied.

In those same years, the construction of an *explicit* semantic model of a programming language is the goal of Christopher Strachey, after ideas of Peter Landin [31]. Starting with [50], presented at a working conference in 1964, and especially with the (then unpublished) notes of a course at Copenhagen in 1967 [51], Strachey presents a full-blown account[14] of a mathematical semantics of a programming language, introducing the notions of "abstract store" and "environment," for modelling assignments and side-effects. Commands are interpreted

[12] "What are the entities with which the science of computation deals? What kinds of facts about these entities would we like to derive? What are the basic assumptions from which we should start?"

[13] "An adequate basis for formal definitions of the meanings of programs [...] in such a way that a rigorous standard is established for proofs about computer programs." "Based on ideas of Perlis and Gorn." "That semantics of a programming language may be defined independently of all processors [...] appear[s] to be new, although McCarthy has done similar work for programming languages based on evaluation of recursive functions."

[14] See, for instance, Figure 1 of [51], page 17 of the reprinted version.

as functions from stores to stores. The basic semantical domains for values are not discussed, but in a later monograph Strachey will esplicitely refer to "abstract mathematical objects, such as integers," in such a way that "when we write a statement such as `x := Sin(y + 3)` in ALGOL 60, what we have in mind is the mathematical functions sine and addition. It is true that our machines can only provide an approximation to these functions but the discrepancies are generally small and we usually start by ignoring them. It is only after we have devised a program which would be correct if the functions used were the exact mathematical ones that we start investigating the errors caused by the finite nature of our computer."

We cannot treat in this paper how mathematical logic and programming languages interacted from the end of sixties – a process much less linear than we may think from today's perspective. Even the relations between the notion of "type" in the two fields are not as straightforward as they may seem [32, 33]. An important chapter of that story would be that of *logic programming languages* (starting from Colmerauer in 1970 and Kowalski in 1974), where logic enters as a first-class actor (see [36]).

4 The Standard Model

The previous section sketched the gradual proposal of (several) semantics for programming languages that could abstract from specific processors and their limitations. While general semantical theories of programming languages had limited impact outside the research communities, the natural approach to a program as the description of a computation happening on an abstract (and largely unspecified) computational device had a major impact. True arithmetic and (in principle) unbounded resources is all that is required of such an abstract processor, that in the Introduction we identified as the *standard model*: the naturalness of assuming these simple hypotheses, gave the standard model the momentum it needed to establish itself as a permanent fixture in programming language theory *and* education[15]. Several reasons cooperate for the establishment of such standard model. One is certainly the need to validate computing as a science – -a mathematical theory is always the entrance ticket to science. Several successes of the use of mathematics into computing were already present at the end of the sixties, like the theory of deterministic parsing (LL and LR), the application of formal language theory, complexity theory, and of course numerical analysis. Despite the large body of results (and relevant problems) in these areas, it is mathematical logic that, as we have seen, dominate the field of programming languages, up to the view (or dream) of McCarthy that logic will be for computation what analysis has been for physics. But a theory of program correctness (which, as we have seen, is deeply connected to the emergence of the standard model) is needed also by other reasons, internal to the

[15] This is not to say that other perspectives were absent. In a 1965 letter to the Computer Journal [49] Strachey proves the undecidability of the halting problem for his Algol-like language CPL, assuming (without saying it!) the standard model. Among the reactions to the letter, the one by W.D. Maurer (27 August 1965) clearly doesn't share the "common ground": Strachey's "letter was of particular interest to me because I had, several months ago, proved that it is indeed possible to write such a program," evidently by assuming a finite automaton as a processor. We find different views also among the founders of the mathematical theory of programs. In the same year of Strachey's [52], Edsger W. Dijkstra writes: "We are considering finite computations only; therefore we may restrict ourselves to computational processes taking place in a finite state machine – although the possible number of states may be very, very large – and take the point of view that the net effect of the computation can be described by the transition from initial to final state." "One often encounters the tacit assumption that [. . .] the inclusion of infinite computations leads to the most appropriate model." "However, we know that the inclusion of the infinite computation is not a logically painless affair." "It seems more effective to restrict oneself to finite computations." [16].

discipline, and probably more important. As an applied discipline – at the end, programs will be used in the real world by real people – computing needs a way to ensure that what it delivers satisfies the requirements on its use. McCarthy's remark on the relations between physics and mathematics seems to suggest that the model here is structural engineering, where mathematical physics laws and empirical knowledge are used together to understand, predict, and calculate the stability, strength and rigidity of structures for buildings. The mathematical theory of computation and its standard model are instrumental for reaching an analogous standard of rigor, so that "when the correctness of a program, its compiler, and the hardware of the computer have all been established with mathematical certainty, it will be possible to place great reliance on the results of the program, and predict their properties with a confidence limited only by the reliability of the electronics," as Hoare writes in his landmark paper on program correctness [23][16]. In this, computing has a big advantage over structural engineering – only the very last layer of the deployment of a system ("the reliability of the electronics") is left out of reach of the formal approach. Since all levels in the hierarchy of a computing system are of the same, *abstract* nature, all levels could be subject (at least conceptually) to the same analysis. When a formally proved chain of compilers will be availabile, a proof that a model of the higher level program satisfies a certain condition, transfers automatically to a proof that a model of the low level program satisfies some other condition, also obtained automatically from the higher level one. No concrete, no iron, no workmanship is involved.

In this context, the standard model is to programming languages what movement without friction is to mechanics. And this is why it is so important. It is not that it implies Turing-completeness that matters, but its simplicity and the fact that, indeed, one may do some mathematics with it. The analogy with the Galilean effort for physics is illuminating – no bodies of different masses reach the ground at the same time when they actually fall from the leaning tower of Pisa, like there is no true unbounded arithmetic inside any laptop; or there is no true isochronous pendulum in nature. Yet, you do not understand a single bit of mechanics if you don't abstract away friction, and don't approximate to small oscillations.

The standard model comes with three features, deeply connected among them. First, *compositionality* – the meaning of a complex construct is obtained from the meaning of its constituents, by composing them in a way that only depends on the construct under consideration. Second, *extensionality* – two constructs with the same input-output behaviour (on their intended domains) have also the same meaning. Finally, *referential transparency* – "if we wish to find the value of an expression which contains a sub-expression, the only thing we need to know about the sub-expression is its value. Any other features of the sub-expression, such as its internal structure [...] are irrelevant to the value of the main expression" [51][17].

Other subtle ingredients are present, but mostly hidden, the most important one being *continuity*, which is the abstract characterization of the *finitary character* of the operations which occur during a computation[18]. In recursive function theory, continuity was implicit in Kleene's first recursion theorem [27], and then exploited in a beautiful set of results, whose apex are the Rice-Shapiro and Myhill-Shepherdson theorems (see [46]), of the second half

[16] Verification of a concrete system, of course, is not the same as verifying its (mathematical) model. A full literature exists on the limitations of formal verification, e.g. [21, 17, 12, 56].

[17] For this use of the expression "referential transparency", Strachey quotes Quine [44] (it is in §30), who in turn refers to Whitehead and Russell's *Principia Mathematica.*

[18] See [8] for a historical reconstruction of the relevance of continuity in early programming language semantics.

of of the fifties. Programming language semantics has a problem analogous to those solved by Kleene's recursion theorem – how to give meaning to (multiple) recursive definitions. In his PhD thesis at MIT, Morris [40] uses the fixed-point combinator of the λ-calculus to link recursion in programming languages to recursive function theory. He shows that the fixed-point operator, applied to a recursive definition, gives the *least* solution with respect to a certain operational order. It was then Dana Scott to put continuity under spotlight [47], and to make it one of the cornerstone of the denotational semantics approach (also to solve those recursive equations *between domains* which are needed in Strachey's approach.)

One of the most important characteristics of a good semantic definition is that it allows for multiple concrete realisations ("implementations") – it must not anticipate those choices that should only be made when the language is implemented. One possible approach to the definition of a language, in fact, could be to define the meaning through a particular interpreter: "This used to be quite common: languages would be "as defined by" some particular compiler" [48]. In this way, however, all the details of that "particular compiler" are needed in order to understand a program. Even more important, to what level of detail is this canonical implementation normative? Is the computation time of a program part of its definition? Is the reporting of errors? The difference between definition and implementation is crucial in the literature we have cited in the previous section. The proposed models are a possible result of the difficult quest for the happy medium between exactness and flexibility, in such a way as to remove ambiguity, still leaving room for implementation (see [19]).

Finally, the explicit availability of models (and especially of the standard model) allows for a clear separation between specification (normative, expressed in the explicit or implicit model), and implementation (that it be abstract or concrete has little importance in this context). Programs are not only abstract mathematical objects living in the theory of computation; nor are only textual, concrete objects embodied in a processor, and thus living in the physical world. The specification defines their *function*[19], while implementation realises that function – it is only in the interplay among these two aspects that programs get their ontology as technical artifacts [55].

5 More Intensional Models

What we have called the standard model is in reality a plurality of abstractions, depending on the language which is modelled. They all share the fact that the numerical functions on the integers are the true arithmetical ones, and that computation happens on an (abstract) processor with unlimited resources (in storage, and time). Coping with real languages required, since the beginning, the introduction of several complications, for instance to deal with side effects (which needed environments and stores), or unrestricted jumps (which required continuations to be used). Still, much effort was in ensuring that the "internal structure" of a program didn't influence its meaning: two different algorithms for the same functions (with the same side-effects, if any), should give rise to the same semantics. This is, after all, what extensionality is about. But this is also an important simplification, or abstraction, that at some point one needs to overcome – real movement happens with friction. Moreover, the fact that the domains involved in the semantics should have "the usual mathematical properties" is something that called for a subtler investigation.

One of the first questions to be tackled, was how to characterise (express, study) *sequentiality*, an important, intensional aspect of certain computations. For this, some notion of "event" seemed essential, to be the elementary building block to serialize. In one of

[19] See, for instance, [30, 57]

the first papers attempting to model parallel programming, Gilles Kahn [24] described a network of sequential processes, communicating through unbounded queues; together with Dave MacQueen, he constructed an operational model for such processes, which, in response to a request, consume data to produce output [25]. Is there a reasonable mathematical (denotational) model for such processes?

Almost at the same time, Gérard Berry studied bottom-up computations, where recursion is understood as a production of "events" [3]; he soon discovered [4] that computation in λ-calculus is intrinsically sequential[20]. Are there models built only with sequential functions, thus "closer" to the operational behaviour[21]?

The answer to these questions came in a series of contributions by Kahn, Berry, Plotkin, and Pierre-Louis Curien (in several configurations as co-authors). The outcome were the notions of *concrete data structure*, *concrete domain*, and *sequential algorithms*. Sequential algorithms [5] are (Kahn-Plotkin [26]) sequential functions, equipped with a strategy for their computation, expressed in a demand-driven way, as in [25][22]. The model of sequential algorithms is "intensional": there are programs with the same input-output behaviour which are separated in the model. It is not a surprise that a similar notion of intensionality is found in Glynn Winskel's *event structures* [59], developed almost at the same time than sequential algorithms, and which may be seen as a generalization of concrete domains (which already contains a notion of incompatibility between elements)[23].

From there, semantics of programming languages strived to cope *also* with intensional phenomena, trying, at the same time, to not abandon the power and simplicity of extensional reasoning.

But telling that story would require another paper. Which I shall write for Maurizio's seventieth birthday.

References

1 John W. Backus. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. In *Proceedings Int. Conf. on Information Processing, UNESCO*, pages 125–132, 1959.

2 John W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Report on the algorithmic language ALGOL 60. *Commun. ACM*, 3(5):299–314, 1960.

---

[20] The function "parallel or" (or "parallel if"):

$$por(x,y) = \begin{cases} tt & \text{if } x = tt \text{ or } y = tt \\ \mathit{ff} & \text{if } x = y = \mathit{ff} \\ \bot & \text{otherwise} \end{cases}$$

is not λ-definable, although it is clearly computable: simultaneously evaluate x and y (e.g., by dovetailing) until one of the two terminates. The parallel or is also a continuous function present in the Scott models for the λ-calculus, and hence it is a "spurious" element of these models, being undefinable by syntactic means. Gordon Plotkin re-discovered the same result for Milner's PCF [43].

[21] And are there models where any element is definable, thus excluding the parallel or? Definability, as it was soon discovered, is related (indeed, for PCF it is equivalent) to full-abstraction [37] (which we do not treat here; see [9].)

[22] The detailed story of the discovery of sequential algorithms is told by Stephen Brookes [7] in the introduction to the journal version (1993) of the technical report (1978) by Kahn and Plotkin [26] on concrete domains and sequential functions.

[23] See Cardone [8] for the relevance of the notion of continuity in this context, and for some of the relations of event structures to Scott's theory and to Carl Adam Petri's analysis of concurrency.

3 Gérard Berry. Séquentialité de l'évaluation formelle des lambda-expressions. In B. Robinet, editor, *Program Transformations*, 3eme Colloque International sur la Programmation, pages 67–80, Paris, 1978. Dunod.

4 Gérard Berry. Modèles complètement adéquats et stables des lambda-calculs typés. Thèse de Doctorat d'État, Université Paris VII, 1979.

5 Gérard Berry and Pierre-Louis Curien. Sequential algorithms on concrete data structures. *Theor. Comput. Sci.*, 20:265–321, 1982.

6 Corrado Böhm. Calculatrices digitales. Du déchiffrage des formules logico-mathématiques par la machine même dans la conception du programme. *Annali di matematica pura e applicata*, IV-37(1):1–51, 1954.

7 Stephen D. Brookes. Historical introduction to "Concrete Domains" by G. Kahn and Gordon D. Plotkin. *Theor. Comput. Sci.*, 121(1&2):179–186, 1993.

8 Felice Cardone. Continuity in semantic theories of programming. *History and Philosophy of Logic*, 26(3):242–261, 2015.

9 Felice Cardone. Games, full abstraction and full completeness. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2017 edition, 2017.

10 Haskell B. Curry. On the composition of programs for automatic computing. Technical Report Memorandum 10337, Naval Ordnance Laboratory, 1949.

11 Edgar Daylight. Towards a historical notion of 'Turing — the father of computer science'. *History and Philosophy of Logic*, 36(3):205–228, 2015.

12 Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Commun. ACM*, 22(5):271–280, 1979.

13 Liesbeth De Mol. Turing Machines. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2019 edition, 2019.

14 Liesbeth De Mol, Maarten Bullynck, and Edgar G. Daylight. Less is more in the fifties. Encounters between logical minimalism and computer design during the 1950s. *IEEE Annals of the History of Computing*, 2018.

15 Liesbeth De Mol, Martin Carlé, and Maarten Bullyinck. Haskell before Haskell: an alternative lesson in practical logics of the ENIAC. *Journal of Logic and Computation*, 25(4):1011–1046, 2015.

16 Edsger W. Dijkstra. A simple axiomatic basis for programming language constructs. *Indagationes Mathematicae*, 36:1–15, 1974.

17 James H. Fetzer. Program verification: The very idea. *Commun. ACM*, 31(9):1048–1063, 1988.

18 Robert W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19:19–32, 1967.

19 Maurizio Gabbrielli and Simone Martini. *Programming Languages: Principles and Paradigms*. Undergraduate Topics in Computer Science. Springer, 2010.

20 Hermann Goldstine and John von Neumann. Planning and coding of problems for an electronic computing instrument. Technical Report Part II, Volume 1-3, Institute of Advanced Studies, 1947.

21 Solomon W. Golomb. Mathematical models: Uses and limitations. *IEEE Transactions on Reliability*, R-20(3):130–131, 1971.

22 Thomas Haigh. Actually, turing did not invent the computer. *CACM*, 57(1):36–41, 2014.

23 C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

24 Gilles Kahn. The semantics of a simple language for parallel processing. In Jack L. Rosenfeld, editor, *Information Processing 74, Proceedings of IFIP Congress*, pages 471–475. North-Holland, 1974.

25 Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In *Information Processing 77, Proceedings of IFIP Congress*, pages 993–998. North Holland, 1977.

26 Gilles Kahn and Gordon D. Plotkin. Concrete domains. *Theor. Comput. Sci.*, 121(1&2):187–277, 1993. Reprint of the IRIA-LABORIA rapport 336 (1978).

27 Stephen C. Kleene. *Introduction to Metamathematics.* Van Nostrand, New York, 1959.

28 Donald E. Knuth. Robert W Floyd, *in memoriam*. *SIGACT News*, 34(4):3–13, December 2003.

29 Donald E. Knuth and Luis T. Pardo. The early development of programming languages. In N. Metropolis, J. Howlett, and Gian-Carlo Rota, editors, *A History of Computing in the Twentieth Century*, pages 197–273. Academic Press, New York, NY, USA, 1980.

30 Peter Kroes. Engineering and the dual nature of technical artefacts. *Cambridge Journal of Economics*, 34(1):51–62, 2010.

31 Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6:308–320, 1964.

32 Simone Martini. Several types of types in programming languages. In Fabio Gadducci and Mirko Tavosanis, editors, *HAPOC 2015*, number 487 in IFIP Advances in Information and Communication Technology, pages 216–227. Springer, 2016.

33 Simone Martini. Types in programming languages, between modelling, abstraction, and correctness. In Arnold Beckmann, Laurent Bienvenu, and Nataša Jonoska, editors, *CiE 2016: Pursuit of the Universal*, volume 9709 of *LNCS*, pages 164–169. Springer, 2016.

34 John McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.

35 John McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, volume 35 of *Studies in Logic and the Foundations of Mathematics*, pages 33–70. Elsevier, 1963. A preliminary version presented at the Western Joint IRE-AIEE-ACM 1961 Computer Conference, pp. 225–238. ACM, New York, NY, USA (1961).

36 Dale Miller. Reciprocal influences between proof theory and logic programming. *Philosophy & Technology*, 2019. `doi:10.1007/s13347-019-00370-x`.

37 Robin Milner. Processes: a mathematical model of computing agents. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73*, number 80 in Studies in the Logic and the Foundations of Mathematics, pages 157–174, Amsterdam, 1975. North-Holland.

38 Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

39 Francis Lockwood Morris and Cliff B. Jones. An early program proof by Alan Turing. *Annals of the History of Computing*, 6:139–143, 1984.

40 James H. Morris. *Lambda-calculus models of programming languages*. PhD thesis, MIT, 1968.

41 Allen Newell, Alan J. Perlis, and Herbert A. Simon. Computer science. *Science*, 157(3795):1373–1374, 1967.

42 David Nofre, Mark Priestley, and Gerard Alberts. When technology became language: The origins of the linguistic conception of computer programming, 1950–1960. *Technology and Culture*, 55:40–75, 2014.

43 Gordon D. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5(3):223–255, 1977.

44 Willard Van Orman Quine. *Word and Object.* MIT Press, 1960.

45 B. Randell. On Alan Turing and the origins of digital computers. *Machine Intelligence*, pages 3–20, 1972.

46 Hartley Rogers. *Theory of Recursive Functions and Effective Computability.* McGraw Hill, New York, 1967.

47 Dana Scott. Outline of a mathematical theory of computation. In *Proc. Fourth Annual Princeton Conference on Information Sciences and Systems*, pages 169–76, Also Tech. Mono. PRG-2, Programming Research Group, University of Oxford., 1970.

48 Joseph E. Stoy. *Denotational semantics: The Scott-Strachey approach to programming language theory.* MIT Press, 1977.

49 Christopher Strachey. An impossible program. *The Computer Journal*, 7(4):313, 1965.

50 Christopher Strachey. Towards a formal semantics. In T.B. Jr. Steel, editor, *Formal Language Description Languages for Computer Programming*, pages 198–220. North-Holland, 1966.

51 Christopher Strachey. Fundamental concepts in programming languages. International Summer School in Computer Programming; Copenhagen. Reprinted in Higher-Order and Symbolic Computation, 13, 11–49, 2000, August 1967.

52 Christopher Strachey. The varieties of programming language. Technical Report PRG-10, Oxford University Computing Laboratory, 1973.

53 Alan M. Turing. Lecture to L.M.S. Feb. 20 1947. In Turing archive, AMT/B/1, 1947.

54 Alan M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 70–72. University Mathematical Laboratory, Cambridge, 1949.

55 Raymond Turner. Programming languages as technical artefacts. *Philosophy and Technology*, 27(3):377–397, 2014.

56 Raymond Turner. *Computational Artifacts*. Springer, 2018.

57 Raymond Turner and Nicola Angius. The philosophy of computer science. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy (Winter 2014 Edition)*. Stanford University, 2017. URL: `http://plato.stanford.edu/archives/win2014/entries/computer-science`.

58 John von Neumann. Rigorous theories of control and information. Published in *Theory of Self-Reproducing Automata*, A. W. Burks (Ed.), University of Illinois Press, 1966, pages 42–56, 1949.

59 Glyn Winskell. *Events in Computation*. PhD thesis, University of Edinburgh, 1981.

A Concurrent Language for Argumentation: Preliminary Notes

Stefano Bistarelli
University of Perugia, Italy
http://www.dmi.unipg.it/bista/
stefano.bistarelli@unipg.it

Carlo Taticchi
Gran Sasso Science Institute, L'Aquila, Italy
carlo.taticchi@gssi.it

Abstract

While agent-based modelling languages naturally implement concurrency, the currently available languages for argumentation do not allow to explicitly model this type of interaction. In this paper we introduce a concurrent language for handling process arguing and communicating using a shared argumentation framework (reminding shared constraint store as in concurrent constraint). We introduce also basic expansions, contraction and revision procedures as main bricks for enforcement, debate, negotiation and persuasion.

2012 ACM Subject Classification Computing methodologies → Knowledge representation and reasoning; Theory of computation → Concurrency; Computing methodologies → Concurrent programming languages

Keywords and phrases Argumentation, Concurrent Language, Debating, Negotiation, Belief Revision

Digital Object Identifier 10.4230/OASIcs.Gabbrielli.2020.9

Funding This work was partially supported by "Argumentation 360" (Ricerca di Base 2017–2019), "RACRA" (Ricerca di Base 2018–2020) and "ASIA" (Social Interaction with Argumentation – GNCS-INDAM).

1 Introduction

Many applications in the field of artificial intelligence aim to reproduce the human behaviour and reasoning in order to allow machines to think and act accordingly. One of the main challenges in this sense is to provide tools for expressing a certain kind of knowledge in a formal way so that the machines can use it for reasoning and infer new information. Argumentation Theory provides formal models for representing and evaluating arguments that interact with each other. Consider, for example, two people arguing about whether lowering taxes is good or not. The first person says that a) lowering taxes would increase productivity; the second person replies with b) a study showed that productivity decrease when taxes are lowered; then, the first person adds c) the study is not reliable since it uses data from unverified sources. The dialogue between the two people is conducted through three main arguments (a,b and c) whose internal structure can be represented through different formalisms [26, 30], and for which we can identify the relations b attacks a and c attacks b. In this paper, we use the representation for Argumentation Frameworks introduced by Dung [18], in which arguments are abstract, that is their internal structure, as well as their origin, is left unspecified. Abstract Argumentation Frameworks (AFs), have been widely studied from the point of view of the acceptability of arguments and, recently, several authors

© Stefano Bistarelli and Carlo Taticchi;
licensed under Creative Commons License CC-BY

Recent Developments in the Design and Implementation of Programming Languages.
Editors: Frank S. de Boer and Jacopo Mauro; Article No. 9; pp. 9:1–9:22

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

have investigated the dynamics of AFs, taking into account both theoretical [28, 4, 10] and computational aspects (for example, a special track on dynamics [7] appeared in the Third International Competition on Computational Models of Argumentation[1]).

Logical frameworks for argumentation, like the ones presented in [17, 19], have been introduced to fulfil the operational tasks related to the study of dynamics in AFs, such as the description of AFs, the specification of modifications, and the search for sets of "good" arguments. Although some of these languages could be exploited to implement applications based on argumentation, for instance to model debates among political opponents, none of them consider the possibility of having concurrent interactions or agents arguing with each other. This lack represents a significant gap between the reasoning capacities of AFs and their possible use in real-life tools. As an example, consider the situation in which two debating agents share a knowledge base, represented by an AF, and both of them want to update it with new information, in such a way that the new beliefs are consistent with the previous ones. The agents can act independently and simultaneously. Similarly to what happens in concurrent programming, if no synchronization mechanism is taken into account, the result of update or revision can be unpredictable and can also lead to the introduction of inconsistencies.

Motivated by the above considerations, we introduce a concurrent language for argumentation (CA) that aims to be used also for modelling different types of interaction between agents (as negotiations, persuasion, deliberation and dialogues). In particular, our language allows for modelling concurrent processes, inspired by notions such as the *Ask-and-Tell constraint system* [29], and using AFs as centralised store. The language is thus endowed with primitives for the specification of interaction between agents through the fundamental operations of adding (or removing) and checking arguments and attacks. Besides specifying a logic for argument interaction, our language can model debating agents (e.g., chatbots) that take part in a conversation and provide arguments.

Alchourrón, Gärdenfors, and Makinson (AGM) theory [1] gives operations (like expansion, contraction, revision) for updating and revising beliefs on a knowledge base. We propose a set of AGM-style operations that allow for modifying an AF (which constitutes the shared memory our agents access to communicate) and changing the status of its arguments so as to allow the implementation of more complex operations, like negotiation and the other forms of dialogues.

The rest of this paper is structured as follows: in Section 2 we recall some notions from Argumentation Theory; in Section 3 we define a labelling semantics for AFs upon which the agents build their beliefs; in Section 4 we present the syntax and the operational semantics of our concurrent language, together with some high level operations that realize the interaction between agents; in Section 5 we discuss existing formalisms from the literature that bring together argumentation and multiagent systems, highlighting the contact points and the differences with our work; Section 6 concludes the paper with final remarks and perspectives on future work.

2 Abstract Argumentation Frameworks

In this section, we briefly recall the basic concepts we refer to in our proposal. In particular, we give the fundamental definition for Abstract Argumentation Frameworks, together with the notions of acceptable argument and argumentation semantics.

---

[1] ICCMA2019 website: `http://iccma2019.dmi.unipg.it`.

Argumentation is an interdisciplinary field that aims to understand and model the human natural fashion of reasoning. In Artificial Intelligence, argumentation theory allows one to deal with uncertainty in non-monotonic (defeasible) reasoning, and it is used to give a qualitative, logical evaluation to sets of interacting arguments, called extensions. In his seminal paper [18], Dung defines the building blocks of abstract argumentation.

▶ **Definition 1** (AFs). *Let U be the set of all possible arguments, which we refer to as the "universe". An Abstract Argumentation Framework is a pair $\langle Arg, R\rangle$ where $Arg \subseteq U$ is a set of arguments and R is a binary relation on Arg representing attacks*[2].

AFs can be represented through directed graphs, that we depict using the standard conventions. For two arguments $a, b \in Arg$, $(a, b) \in R$ represents an attack directed from a against b. Moreover, we say that an argument b is *defended* by a set $B \subseteq Arg$ if and only if, for every argument $a \in Arg$, if $R(a, b)$ then there is some $c \in B$ such that $R(c, a)$.

The goal is to establish which are the acceptable arguments according to a certain semantics, namely a selection criterion. Non-accepted arguments are rejected. Different kinds of semantics have been introduced [18, 2] that reflect qualities which are likely to be desirable for "good" subsets of arguments. We first give the definition for the extension-based semantics (also referred to as Dung semantics), namely admissible, complete, stable, preferred, and grounded semantics (denoted with *adm*, *com*, *stb*, *prf* and *gde*, respectively, and generically with σ).

▶ **Definition 2** (Extension-based semantics). *Let $F = \langle Arg, R\rangle$ be an AF. A set $E \subseteq Arg$ is conflict-free in F, denoted $E \in S_{cf}(F)$, if and only if there are no $a, b \in E$ such that $(a, b) \in R$. For $E \in S_{cf}(F)$ we have that:*

- $E \in S_{adm}(F)$ *if each* $a \in E$ *is defended by* E*;*
- $E \in S_{com}(F)$ *if* $E \in S_{adm}(F)$ *and* $\forall a \in Arg$ *defended by* E, $a \in E$*;*
- $E \in S_{stb}(F)$ *if* $\forall a \in Arg \setminus E$, $\exists b \in E$ *such that* $(b, a) \in R$*;*
- $E \in S_{prf}(F)$ *if* $E \in S_{adm}(F)$ *and* $\nexists E' \in S_{adm}(F)$ *such that* $E \subset E'$*;*
- $E \in S_{gde}(F)$ *if* $E \in S_{com}(F)$ *and* $\nexists E' \in S_{com}(F)$ *such that* $E' \subset E$*.*

Moreover, if E satisfies one of the above semantics, we say that E is an extension for that semantics (for example, if $E \in S_{adm}(F)$ we say that E is an admissible extension).

The different semantics described in Definition 2 corresponds to different styles of reasoning, each of which may be more appropriate for being applied to a particular application domain. The characterisation of the reasoning requirements for the various domains is still a largely open research problem [3] and can only be based on general criteria rather than on specific cases. The stable semantics can be considered the strongest one: the accepted arguments attack all the others in the framework. Since a stable extension may not exist, the preferred semantics can be used as a valid alternative. The preferred semantics, in turn, does not have a unique extension, making the grounded semantics (that always exists and admits exactly one solution) an overall good option for establishing which arguments have to be accepted.

A partial order can be defined among the set of extensions for the different semantics. In detail, we know that $S_{stb}(F) \subseteq S_{prf}(F) \subseteq S_{com}(F) \subseteq S_{adm}(F) \subseteq S_{cf}(F)$ and $S_{gde}(F) \subseteq S_{com}(F)$. Besides enumerating the extensions for a certain semantics σ, one of the most

[2] We introduce both U and $Arg \subseteq U$ (not present in the original definition by Dung) for our convenience, since in the concurrent language that we will define in Section 4 we use an operator to dynamically add arguments from U to Arg.

common tasks performed on AFs is to decide whether an argument a is accepted in some extension of $S_\sigma(F)$ or in all extensions of $S_\sigma(F)$. In the former case, we say that a is *credulously* accepted with respect to σ; in the latter, a is instead *sceptically* accepted with respect to σ. The grounded semantics, in particular, coincides with the set of arguments sceptically accepted by the complete ones.

▶ **Example 3.** In Figure 1 we provide an example of AF where sets of extensions are given for all the mentioned semantics[3]. We discuss some details: the singleton $\{e\}$ is not conflict-free because e attacks itself. The argument b is not contained in any admissible extension because no other argument (included itself) defends b from the attack of a. The empty set $\{\}$, and the singletons $\{c\}$ and $\{d\}$ are not complete extensions because a, which is not attacked by any other argument, has to be contained in all complete extensions. Only the maximal (with respect to set inclusion) admissible extensions $\{a, c\}$ and $\{a, d\}$ are preferred, while the minimal complete $\{a\}$ is the (unique) grounded extension. Then, the arguments in the subset $\{a, d\}$, that conduct attacks against all the other arguments (namely b, d and e), represent a stable extension. To conclude the example, we want to point out that argument a is sceptically accepted with respect to the complete semantics, since it appears in all three subsets of $S_{com}(F)$. On the other hand, argument c, that is in just one complete extension, is credulously accepted with respect to the complete semantics.

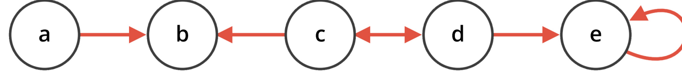


Figure 1 An argumentation framework F for which we compute the following sets of extensions: $S_{cf}(F) = \{\{\},\{a\},\{b\},\{c\},\{d\},\{a,c\},\{a,d\},\{b,d\}\}$, $S_{adm}(F) = \{\{\},\{a\},\{c\},\{d\},\{a,c\},\{a,d\}\}$, $S_{com}(F) = \{\{a\},\{a,c\},\{a,d\}\}$, $S_{prf}(F) = \{\{a,c\},\{a,d\}\}$, $S_{stb}(F) = \{\{a,d\}\}$, and $S_{gde}(F) = \{\{a\}\}$.

Many of the above-mentioned semantics (such as the admissible and the complete ones) exploit the notion of defence in order to decide whether an argument is part of an extension or not. The phenomenon for which an argument is accepted in some extension because it is defended by another argument belonging to that extension is known as *reinstatement* [11]. In that paper, Caminada also gives a definition for a reinstatement labelling.

▶ **Definition 4** (Reinstatement labelling). *Let $F = \langle Arg, R\rangle$ be an AF and $\mathbb{L} = \{in, out, undec\}$. A labelling of F is a total function $L : Arg \to \mathbb{L}$. We define $in(L) = \{a \in Arg \mid L(a) = in\}$, $out(L) = \{a \in Arg \mid L(a) = out\}$ and $undec(L) = \{a \in Arg \mid L(a) = undec\}$. We say that L is a reinstatement labelling if and only if it satisfies the following:*

- *$\forall a, b \in Arg$, if $a \in in(L)$ and $(b, a) \in R$ then $b \in out(L)$;*
- *$\forall a \in Arg$, if $a \in out(L)$ then $\exists b \in Arg$ such that $b \in in(L)$ and $(b, a) \in R$.*

In other words, an argument is labelled *in* if all its attackers are labelled *out*, and it is labelled *out* if at least an *in* node attacks it. In all other cases, the argument is labelled *undec*. A labelling-based semantics [2] associates with an AF a subset of all the possible labellings. In Figure 2 we show an example of reinstatement labelling on an AF. Moreover, there exists a connection between reinstatement labellings and the Dung-style semantics.

[3] The examples are made using the ConArg suite [8]. Web interface: `http://www.dmi.unipg.it/conarg`.

This connection is summarised in Table 1: the set of *in* arguments in any reinstatement labelling constitutes a complete extension; then, if no argument is *undec*, the reinstatement labelling provides a stable extension; if the set of *in* arguments (or the set of *out* arguments) is maximal with respect to all the possible labellings, we obtain a preferred extension; finally the grounded extension is identified by labellings where either the set of *undec* arguments is maximal, or the set of *in* (respectively *out*) arguments is maximal.

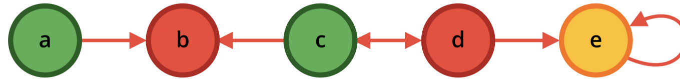


Figure 2 an example of AF in which reinstatement labelling is showed by using colours. Arguments a and c highlighted in green are *in*, red ones (b and d) are *out*, and the the yellow argument e (that attacks itself) is *undec*.

Table 1 Reinstatement labelling vs semantics.

Labelling restrictions		Semantics
no restrictions		complete
empty	*undec*	stable
maximal	*in*	preferred
maximal	*out*	preferred
maximal	*undec*	grounded
minimal	*in*	grounded
minimal	*out*	grounded

Reinstatement labelling allows to inspect AFs on a finer grain than Dung's extensions, since the *undec* label identifies arguments that are not acceptable, but still not directly defeated by accepted arguments. However, the information brought by the *undec* label can be misleading. Consider for example an AF in which two arguments a and b are attacking each other (Figure 3, left). A possible labelling for such a framework would label both arguments as *undec*. Indeed, we cannot decide whether, in general, it is worth accepting a (or b). Consider now a second AF composed of two arguments c and d where only c attacks d and both arguments are labelled as *undec* (Figure 3, right). At this point, one could conclude that it is not possible to univocally establish whether c is a good argument or not, similarly to what happens in the previous example. However, in this case the fact of c being *undec* does not depend on the structure of the framework, but rather on the choice of just ignoring it.

Figure 3 Two AFs where all arguments are labelled *undec*. The one on the left has two undistinguishable arguments a and b, while argument c of the AF on the right is arguably better than d, from the point of view of acceptability.

Ambiguity of the *undec* label is solved in the four-state labelling introduced by [22], where arguments that are assigned the label *in* are accepted, those that are assigned the label *out* are rejected, those that are assigned both *in* and *out* (which we denote as *undec*) are neither fully accepted nor fully rejected, and those that are not considered at all are assigned the empty set $\emptyset$. The labelling of [22] is defined as follow.

▶ **Definition 5** (Four-state labelling). *A four-state labelling consists of a total mapping* $L : Arg \rightarrow 2^{\{in,out\}}$ *that satisfies the following conditions:*

- $\forall a \in Arg$*, if* $out \in L(a)$*, then* $\exists b \in Arg$ *such that* $(b, a) \in R$ *and* $in \in L(b)$*;*
- $\forall a \in Arg$*, if* $in \in L(a)$*, then* $\forall b \in Arg$ *such that* $(b, a) \in R$*,* $out \in L(b)$*;*
- $\forall a \in Arg$*, if* $in \in L(a)$*, then* $\forall c$ *such that* $(a, c) \in R$*,* $out \in L(c)$*.*

A four-state labelling is said to be total[4] *if and only if* $\forall a \in Arg$*,* $L(a) \neq \emptyset$*. A labelling which is not total is called partial. Moreover, the four labels form the lattice of Figure 4, in which undec (that is the set* $\{in, out\}$*) is the top element and* $\emptyset$ *is the bottom.*

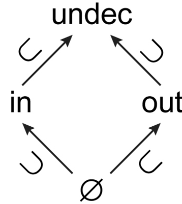


Figure 4 Lattice of labels in the four-state labelling.

We show an example of labelling in Figure 5, where all four labels are used. Note that the arguments labelled *in* and *out* in the figure do not satisfy the condition of the reinstatement labelling. Even though the labelling of Definition 5 is more informative than

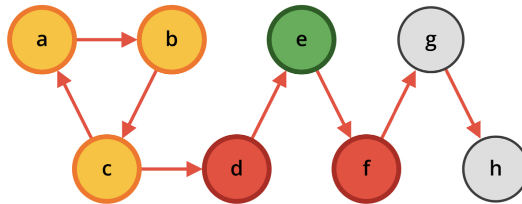


Figure 5 Labelling of an AF showed through colours. Argument e, highlighted in green, is the only *in*; red arguments d and f are *out*; those in yellow, i.e., a, b and c, are *undec*; and the grey arguments g and h are left with an empty label $\emptyset$.

the reinstatement labelling of Definition 4 (that does not comprehend an empty label), there is no direct connection between labellings and extensions of a certain semantics, as it happens for the reinstatement labelling.

3 A Four-state Labelling Semantics

We showed in the previous section that both reinstatement and four-state labellings have both pros and cons. The labelling by Caminada does not allow to leave unlabelled arguments that we do not want to consider in computing acceptability and forces all arguments that are

[4] The total labelling is called "complete" in the original definition [22]. We changed it to avoid ambiguity with the complete semantics.

neither *in* nor *out* to be labelled *undec*. On the other hand, the set of arguments labelled *in* by the reinstatement labelling showed in Definition 4 always correspond to a complete extension and some other semantics can be obtained by applying restrictions on the labelling itself (see Table 1), while four-state labelling does not necessarily correspond to any particular extension. To overcome this problem, in the following we establish a mapping between a modified four-state labelling and the classical semantics of Definition 2.

The labelling of an AF gives information about the acceptability of the arguments in the framework (according to the various Dung's semantics) and can be used by intelligent agents to represent the state of their beliefs. Each different label can be traced to a particular meaning. $\emptyset$ stands for "don't care" [22] and identifies arguments that are not considered by the agents. For instance, arguments in $U \setminus Arg$, that are only part of the universe, but not of the shared AF, are labelled with $\emptyset$ since they are outside the interest of the agents. Accepted and rejected arguments (labelled as *in* and *out*, respectively), allow agents to discern true beliefs from the false ones. At last, *undec* arguments possess both *in* and *out* labels, meaning that agents cannot decide about the acceptability of a belief ("don't know", indeed).

▶ **Definition 6** (Four-state labelling semantics). *Let U be a universe of arguments, $F = \langle Arg, R\rangle$ an AF with $Arg \subseteq U$ and $R \subseteq Arg \times Arg$ the arguments and attacks. L is a four-state labelling on F if and only if*

- $\forall a \in U \setminus Arg. L(a) = \emptyset$;
- $\forall a \in Arg$, *if* $out \in L(a)$, *then* $\exists b \in Arg$ *such that* $(b, a) \in R$ *and* $in \in L(b)$;
- $\forall a \in Arg$, *if* $in \in L(a)$, *then* $\forall b \in Arg$ *such that* $(b, a) \in R$, $out \in L(b)$;
- $\forall a \in Arg$, *if* $in \in L(a)$, *then* $\forall c$ *such that* $(a, c) \in R$, $out \in L(c)$.

Moreover,

- L *is a conflict-free labelling if and only if:*
  - $L(a) = \{in\} \implies \forall b \in Arg \mid (b, a) \in R. L(b) \neq \{in\}$ *and*
  - $L(a) = \{out\} \implies \exists b \in Arg \mid (b, a) \in R \wedge L(b) = \{in\}$
- L *is an admissible labelling if and only if:*
  - $L(a) = \{in\} \implies \forall b \in Arg \mid (b, a) \in R. L(b) = \{out\}$ *and*
  - $L(a) = \{out\} \implies \exists b \in Arg \mid (b, a) \in R \wedge L(b) = \{in\}$
- L *is a complete labelling if and only if:*
  - $L(a) = \{in\} \iff \forall b \in Arg \mid (b, a) \in R. L(b) = \{out\}$ *and*
  - $L(a) = \{out\} \iff \exists b \in Arg \mid (b, a) \in R \wedge L(b) = \{in\}$
- L *is a stable labelling if and only if:*
  - L *is a complete labelling and*
  - $\nexists a \in Arg \mid L(a) = \{in, out\}$
- L *is a preferred labelling if and only if:*
  - L *is an admissible labelling and*
  - $\{a \mid L(a) = \{in\}\}$ *is maximal among all the admissible labellings*
- L *is a grounded labelling if and only if:*
  - L *is a complete labelling and*
  - $\{a \mid L(a) = \{in\}\}$ *is minimal among all the complete labellings*

We can show there is a correspondence between labellings satisfying the restrictions given in the definition above and the extensions of a certain semantics. We use the notation $L \in S_\sigma(F)$ to identify a labelling L corresponding to an extension of the semantics σ with respect to the AF F.

▶ **Theorem 7.** *A four-state labelling L of an AF $F = \langle Arg, R\rangle$ is a conflict-free (respectively admissible, complete, stable, preferred, grounded) labelling as in Definition 6 if and only if the set I of arguments labelled in by L is a conflict-free (respectively admissible, complete, stable, preferred, grounded) extension of F.*

Proof. We sketch the proof for the admissible labelling. The conflict-free case is obtained through a similar reasoning and the remaining can be constructed as in [12].

$\Rightarrow$) Consider an admissible labelling L on $F = \langle Arg, R\rangle$. We have to show that there are no $a, b \in I$ such that $(a, b) \in R$ and that each $a \in I$ is defended by I. First of all, arguments labelled *in* by L can only be attacked by *out* arguments, so for all $a, b \in I$ we have $(a, b) \notin R$. Then, if a is attacked by an argument b (which we know must be *out*) that argument is necessarily in turn attacked by at least one *in*. We conclude that I defends all its elements and therefore it is an admissible extension.

$\Leftarrow$) We have an admissible extension E composed of arguments labelled *in* by L, and we know that all arguments in E does not attack each other and are defended by E. Hence, *in* arguments of L cannot be attacked by other arguments with the label *in*. Finally, arguments that are attacked from E are *out*. ◀

In the next session, where we present our concurrent language for argumentation, the labelling of Definition 6 is used to implement both primitives and high level operations that rely on the acceptability state of agent's belief and are able to change the underlying knowledge base accordingly.

4 The Language

Agents/processes in a distributed/concurrent system can perform operations that affect the behaviour of other components. The indeterminacy in the execution order of the processes may lead to inconsistent results for the computation or even cause errors that prevent particular tasks from being completed. We refer to this kind of situation as a *race condition*. If not properly handled, race conditions can cause loss of information, resource starvation and deadlock. In order to understand the behaviour of agents and devise solutions that guarantee correct executions, many formalisms have been proposed for modelling concurrent systems. Concurrent Constraint Programming (CC) [29], in particular, relies on a constraint store of shared variables in which agents can read and write in accordance with some properties posed on the variables. The basic operations that can be executed by agents in the CC framework are a blocking *Ask* and an atomic *Tell*. These operations realise the interaction with the store and also allow one to deal with partial information.

Starting from the CC syntax, we enrich the ask and tell operators in order to handle the interaction with an AF used as knowledge base for the agents. We replace the ask with three decisional operations: a syntactic *check* that verifies if a given set of arguments and attacks is contained in the knowledge base, and two semantic *test* operations that we use to retrieve information about the acceptability of arguments in an AF. The tell operation (that we call *add*) augments the store with additional arguments and attack relations. We can also remove parts of the knowledge base through a specifically designed removal operation. Finally, a guarded parallel composition $\|_G$ allows for executing all the operations that satisfy some given conditions, and a prioritised operator $+_P$ is used to implement if-then-else constructs. The syntax of our concurrent language for argumentation is presented in Table 2, while in Table 3 we give the definitions for the transition rules.

Suppose to have an agent A whose knowledge base is represented by an AF $F = \langle Arg, R\rangle$. An $add(Arg', R')$ action performed by the agent results in the addition of a set of arguments $Arg' \subseteq U$ (where U is the universe) and a set of relations R' to the AF F. When performing

Table 2 CA syntax.

$$
\begin{aligned}
P &::= C.A \\
C &::= p(a,l,\sigma) :: A \mid C.C \\
A &::= \mathit{success} \mid \mathit{add}(Arg,R) \rightarrow A \mid \mathit{rmv}(Arg,R) \rightarrow A \mid E \mid A \| A \mid \exists_x A \mid p(a,l,\sigma) \\
E &::= \mathit{test}_c(a,l,\sigma) \rightarrow A \mid \mathit{test}_s(a,l,\sigma) \rightarrow A \mid \mathit{check}(Arg,R) \rightarrow A \\
&\quad \mid E + E \mid E +_P E \mid E \|_G E
\end{aligned}
$$

an Addition, (possibly) new arguments are taken from $U \setminus Arg$. We want to make clear that the tuple (Arg', R') is not an AF, indeed it is possible to have $Arg' = \emptyset$ and $R' \neq \emptyset$, which allows to perform an addition of only attack relations to the considered AF. It is as well possible to add only arguments to F, or both arguments and attacks. Intuitively, $rmv(Arg, R)$ allows to specify arguments and/or attacks to remove from the knowledge base. Removing an argument from an AF requires to also remove the attack relations involving that argument and trying to remove an argument (or an attack) which does not exist in F will have no consequences. The operation $check(Arg', R')$ is used to verify whether the specified arguments and attack relations are contained in the set of arguments and attacks of the knowledge base, without introducing any further change. If the check is positive, the operation succeeds, otherwise it suspends. We have two distinct test operations, both requiring the specification of an argument $a \in A$, a label $l \in \{in, out, undec, \emptyset\}$ and a semantics $\sigma \in \{adm, com, stb, prf, gde\}$. The credulous $test_c(a, l, \sigma)$ succeeds if there exists at least an extension of $S_\sigma(F)$ whose corresponding labelling L is such that $L(a) = l$; otherwise (in the case $L(a) \neq l$ in all labellings) it suspends. The sceptical $test_s(a, l, \sigma)$ succeeds[5] if a is labelled l in all possible labellings $L \in S_\sigma(F)$; otherwise (in the case $L(a) \neq L$ in some labellings) it suspends. The guarded parallelism $\|_G$ is designed to execute all the operations for which the guard in the inner expression is satisfied. More in detail, $E_1 \|_G E_2$ is successful when either E_1, E_2 or both are successful and all the operations that can be executed are executed. This behaviour is different both from classical parallelism (for which all the agents have to terminate in order for the procedure to succeed) and from nondeterminism (that only selects one branch). The operator $+_P$ is left-associative and realises an if-then-else construct: if we have $E_1 +_P E_2$ and E_1 is successful, than E_1 will be always chosen over E_2, even if also E_2 is successful, so in order for E_2 to be selected, it has to be the only one that succeeds. Differently from nondeterminism, $+_P$ prioritises the execution of a branch when both E_1 and E_2 can be executed. Moreover, an if-then-else construct cannot be obtained starting from nondeterminism since of our language is not expressive enough to capture success or failure conditions of each branch.

The remaining operators are classical concurrency compositions: an agent in a parallel composition obtained through $\|$ succeeds if all the agents succeed; any agent composed through $+$ is chosen if its guards succeeds; the existential quantifier $\exists_x A$ behaves like agent A where variables in x are local to A[6]. The parallel composition operator enables the specification of complex concurrent argumentation processes. For example, a debate

[5] The set of extensions $S_\sigma(F)$ is finite, thus both $test_c(a, l, \sigma)$ and $test_s(a, l, \sigma)$ are decidable.
[6] We plan to use existential quantifiers to extend our work by allowing our agents to have local stores.

Table 3 CA operational semantics.

Rule	Name
$\langle add(Arg', R') \rightarrow A, \langle Arg, R\rangle\rangle \longrightarrow \langle A, \langle Arg \cup Arg', R \cup R'\rangle\rangle$	Addition
$\langle rmv(Arg', R') \rightarrow A, \langle Arg, R\rangle\rangle \longrightarrow \langle A, \langle Arg \setminus Arg', R \setminus \{R' \cup R''\}\rangle\rangle$ where $R'' = \{(a,b) \in R \mid a \in Arg' \vee b \in Arg'\}$	Removal
$\dfrac{Arg' \subseteq Arg \wedge R' \subseteq R}{\langle check(Arg', R') \rightarrow A, \langle Arg, R\rangle\rangle \longrightarrow \langle A, \langle Arg, R\rangle\rangle}$	Check
$\dfrac{\exists L \in S_\sigma(F) \mid l \in L(a)}{\langle test_c(a, l, \sigma) \rightarrow A, F\rangle \longrightarrow \langle A, F\rangle}$	Credulous Test
$\dfrac{\forall L \in S_\sigma(F).l \in L(a)}{\langle test_s(a, l, \sigma) \rightarrow A, F\rangle \longrightarrow \langle A, F\rangle}$	Sceptical Test
$\dfrac{\langle A_1, F\rangle \longrightarrow \langle A_1', F'\rangle}{\begin{array}{c}\langle A_1 \Vert A_2, F\rangle \longrightarrow \langle A_1' \Vert A_2, F'\rangle \\ \langle A_2 \Vert A_1, F\rangle \longrightarrow \langle A_2 \Vert A_1', F'\rangle\end{array}}$ $\quad$ $\dfrac{\langle A_1, F\rangle \longrightarrow \langle success, F'\rangle}{\begin{array}{c}\langle A_1 \Vert A_2, F\rangle \longrightarrow \langle A_2, F'\rangle \\ \langle A_2 \Vert A_1, F\rangle \longrightarrow \langle A_2, F'\rangle\end{array}}$	Parallelism
$\dfrac{\langle E_1, F\rangle \longrightarrow \langle A_1, F\rangle, \langle E_2, F\rangle \not\longrightarrow}{\begin{array}{c}\langle E_1 \Vert_G E_2, F\rangle \longrightarrow \langle A_1, F\rangle \\ \langle E_2 \Vert_G E_1, F\rangle \longrightarrow \langle A_1, F\rangle\end{array}}$	Guarded Parallelism (1)
$\dfrac{\langle E_1, F\rangle \longrightarrow \langle A_1, F\rangle, \langle E_2, F\rangle \longrightarrow \langle A_2, F\rangle}{\langle E_1 \Vert_G E_2, F\rangle \longrightarrow \langle A_1 \Vert A_2, F\rangle}$	Guarded Parallelism (2)
$\dfrac{\langle E_1, F\rangle \longrightarrow \langle A_1, F\rangle}{\begin{array}{c}\langle E_1 + E_2, F\rangle \longrightarrow \langle A_1, F\rangle \\ \langle E_2 + E_1, F\rangle \longrightarrow \langle A_1, F\rangle\end{array}}$	Nondeterminism
$\dfrac{\langle E_1, F\rangle \longrightarrow \langle A_1, F\rangle}{\langle E_1 +_P E_2, F\rangle \longrightarrow \langle E_1, F\rangle}$	If Then Else (1)
$\dfrac{\langle E_1, F\rangle \not\longrightarrow, \langle E_2, F\rangle \longrightarrow \langle A_2, F\rangle}{\langle E_1 +_P E_2, F\rangle \longrightarrow \langle E_2, F\rangle}$	If Then Else (2)
$\dfrac{\langle A[y/x], F\rangle \longrightarrow \langle A', F'\rangle}{\langle \exists_x A, F\rangle \longrightarrow \langle A', F'\rangle}$ with y fresh	Hidden Variables
$\langle p(b, m, \gamma), F\rangle \longrightarrow \langle A[b/a, m/l, \gamma/\sigma], F\rangle$ when $p(a, l, \sigma) :: A$	Procedure Call

involving many agents that asynchronously provide arguments can be modelled as a parallel composition of add operations performed on the knowledge base. Concluding, P is the class of programs, and the procedure call C has three parameters that allow the implementation of operators which takes into account an argument, a label and a semantics. Below, we give an example of a CA program.

▶ **Example 8.** Consider the AF in Figure 6 (left), where the complete semantics is the set $\{\{a\}, \{a, e\}, \{a, d\}\}$ and the preferred coincides with $\{\{a, d\}, \{a, e\}\}$. An agent A wants to perform the following operation: if argument d is labelled *out* in all complete extensions, then remove the argument c from the knowledge base. At the same time, an agent B wants to add an argument f attacking d only if e is labelled *in* in some preferred extension. If A is the first agent to be executed, the sceptical test on argument d will suspend, since d belongs to the complete extension $\{a, d\}$. The credulous test performed by agent B, instead, is successful and so it can proceed to add an argument f that defeats d. Now d is sceptically rejected by the complete semantics and agent A can finally remove the argument c. After the execution of the program below, we obtain the AF of Figure 6 (right).

$$A : test_s(d, out, com) \rightarrow rmv(\{c\}, \{(a, c)\}) \rightarrow success$$
$$B : test_c(e, in, prf) \rightarrow add(\{f\}, \{(f, d)\}) \rightarrow success$$
$$P : A \| B$$

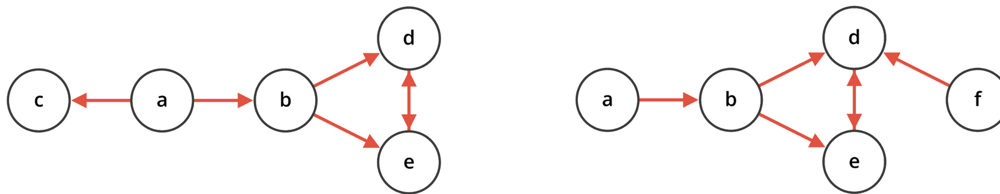


Figure 6 The AF on the right is obtained starting from the one on the left trough the addition of an argument f attacking d and the removal of c together with the attack (a, c).

As we will see in the next session, we aim to use the operators of our language to model the behaviour of agents involved in particular argumentative processes (such as persuasion and negotiation). Note that the language is very permissive: there are no constraints on which arguments or attacks an agent can add/remove. Future work include the partition of arguments and attack with respect to the owner's capabilities and restrict permissions on legal moves.

4.1 Belief Revision and the AGM Framework

Interaction between agents can be modelled in different ways, according to the purposes of the communication. Negotiating agents need to find a common agreement that is beneficial to all, while, for instance, an agent with the goal of persuading its opponents has to both defend its position from the attacks of the other agents and defeat all the arguments against its proposal. The operations needed for the implementation of such kind of interactions must be able to modify the knowledge base shared between the communicating parts so as to model the behaviour of the agents. In particular, usually agents interact modifying part of the shared AF, trying to change the state of acceptance of an argument, often alternating with other agents or concurrently performing syntactic changes to the AF.

The AGM framework [1] provides an approach to the problem of revising knowledge basis by using theories (deductively closed sets of formulae) to represent the beliefs of the agents. A formula α in a given theory can have different statuses for an agent, according to its knowledge base K. If the agent can deduce α from its beliefs, then we say that α is *accepted* ($K \vdash \alpha$). Such a deduction corresponds with the entailment of α by the knowledge base. If the agent can deduce the negation of α, then we say that α is *rejected* ($K \vdash \neg\alpha$). Otherwise, the agent cannot deduce anything and α is *undetermined*. The correspondence between accepted/rejected beliefs and *in*/*out* arguments in a labelling is straightforward. Since the undetermined status represents the absence of a piece of information (nothing can be deduced in favour of either accepting or rejecting a belief) it can be mapped into the empty label $\emptyset$. Finally, the *undec* label is assigned to arguments that are both *in* and *out*, boiling down to the notion of inconsistency in AGM. The empty label, in particular, plays a fundamental role in identifying new arguments that agents can bring to the debate to defend (or strengthen) their position. The status of a belief can be changed through some operations (namely expansion $\oplus$, contraction $\oslash$ and revision $\circledast$) on the knowledge base, as depicted in Figure 7 (notice the similarity with the lattice in Figure 4).

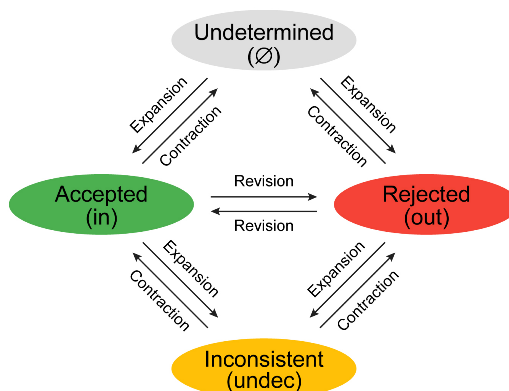


Figure 7 Transitions between AGM beliefs states.

An expansion basically brings new pieces of information to the base, allowing for undetermined belief to become either accepted or refused. A contraction, on the contrary, reduces the information an agent can rely on in making its deduction, and an accepted (or refused) belief can become undetermined. A revision introduces conflicting information, making acceptable belief refused and vice-versa. The AGM framework also defines three sets of rationality postulates (one for each operation) that any good operator should satisfy. To give an example, if we want to add a new belief on a knowledge base, then we expect that no other information in the base is removed. AGM operators provide building blocks for realizing complex interaction processes between agents. Below, we provide some examples:

- **Negotiation** is a process that aims to solve conflicts arising from the interaction between two or more parties that have different individual goals (for instance, a request of computational resources in a distributed network), and its outcome is an agreement that translates in common benefits for all participants. Expansion, here, can be used to model the behaviour of an agent presenting claims towards its counterparts, while contraction represents the act of retracting a condition to successfully conclude the negotiation.

- Contrary to negotiation, a **debate** takes place when the goal of the agents in the system is to promote their own point of view and thus "convince" the others about a conclusion or a statement. A debate [21] can be considered as a mechanism through which a decision maker extracts information from two (or more) counterparts, each of them holding different positions with respect to the right choice. In a multi-agent system, a debate is a process carried out as the interaction between more parties, each of them trying to provide arguments strong enough to support their own conclusion. In this case, agents can make their beliefs accepted in different ways, exploiting AGM operators: inconsistent beliefs can be made accepted through a contraction, while expansion can make beliefs which state is undetermined acceptable.
- The notion of **persuasion** in dialogue games [25] aims to solve conflicts of points of view between two counterparts. In order to persuade the opponent, an agent has to defend its position by replying to every attack against its initial claim. If it fails, the opponent wins the game. Agents involved in this kind of persuasive dialogue games have to elaborate strategies [23], for supporting their beliefs and defeating the adversaries, that consist in a sequence of actions to perform in the system. Again, revision operations on the knowledge base are responsible for changing the status of the beliefs of a persuaded agent.

As for knowledge basis in belief revision, AFs can undergo changes that modify the structure of the framework itself, either integrating new information (and so increasing the arguments and the attacks in the AF) or discarding previously available knowledge. Agents using AFs as the mean for exchanging and inferring information has to rely on operations able to modify such AFs. Besides considering the mere structural changes, also modifications on the semantics level need to be addressed by the operations performed by the agents. In the following, we define three operators for AFs, namely *argument expansion*, *contraction* and *revision*, that comply with classical operators of AGM and that can be built as procedures in our language.

The argumentation frameworks $\langle Arg, R\rangle$ we use as the knowledge base for our concurrent agents are endowed with a universe of arguments U that are used to bring new information. Since arguments in $U \setminus Arg$ do not constitute an actual part of the knowledge base, they are always labelled $\emptyset$, until they are added into the framework and acquire an *in* and/or an *out* label. Notice also that changes to the knowledge base we are interested in modelling are restricted to a single argument at a time, miming the typical argument interaction in dynamic AF.

▶ **Definition 9** (Argument extension expansion, contraction, revision). *Let $F = \langle Arg, R\rangle$ be an AF on the universe U, $Arg \subseteq U$, $R \subseteq Arg \times Arg$, σ a semantics, $L \in S_\sigma(F)$ a given labelling, and $a \in U$ an argument.*

- *An argument extension expansion $\oplus^\sigma_{a,L} : AF \rightarrow AF$ computes a new AF $F' = \oplus^\sigma_{a,L}(F)$ with semantics $S_\sigma(F')$ for which $\exists L' \in S_\sigma(F')$ such that $L'(a) \supseteq L(a)$ (if $L'(a) \supset L(a)$ the expansion is strict).*
- *An argument extension contraction $\oslash^\sigma_{a,L} : AF \rightarrow AF$ computes a new AF $F' = \oslash^\sigma_{a,L}(F)$ with semantics $S_\sigma(F')$ for which $\exists L' \in S_\sigma(F')$ such that $L(a) \supseteq L'(a)$ (if $L(a) \supset L'(a)$ the expansion is strict).*
- *An argument extension revision $\circledast^\sigma_{a,L} : AF \rightarrow AF$ computes a new AF $F' = \circledast^\sigma_{a,L}(F)$ with semantics $S_\sigma(F')$ for which $\exists L' \in S_\sigma(F')$ such that if $L(a) = in/out$, then $L'(a) = out/in$ and $\forall b \in Arg$ with $b \neq a$, $L'(b) = L(b) \vee L'(b) \neq undec$ (that is no inconsistencies are introduced).*

Moreover, we denote with $\oplus^{\sigma,l}_{a,L}(F)$, $\oslash^{\sigma,l}_{a,L}(F)$ and $\circledast^{\sigma,l}_{a,L}(F)$ an argument extension expansion, contraction and revision, respectively, that computes an AF F' with semantics $S_\sigma(F')$ for which $\exists L' \in S_\sigma(F')$ such that $L'(a) = l$.

When performing an argument extension expansion (or contraction, or revision) for a certain argument a of an AF F, the operators of Definition 9 take into account a single labelling of the semantics σ and there is no control over the other labellings, for which a can have its label arbitrarily changed. For example, an argument extension expansion that increases the number of labels of a with respect to a chosen labelling L, may reduce that number in a different labelling. Therefore, we introduce a further definition that considers all the possible labellings $\mathcal{L}^F_\sigma$ of $S_\sigma(F)$. To compare the various labels an argument can have in different labellings, we refer to the order in Figure 4 and, calling $\mathcal{L}^F_{\sigma\downarrow_a}$ the multi-set of the labels a has in the various $L \in \mathcal{L}^F_\sigma$, we say that $\mathcal{L}^{F'}_{\sigma\downarrow_a} \supseteq \mathcal{L}^F_{\sigma\downarrow_a}$ if there exists an injective function $f : \mathcal{L}^F_\sigma \rightarrowtail \mathcal{L}^{F'}_\sigma$ such that $\forall l \in \mathcal{L}^F_{\sigma\downarrow_a}.l \leq f(l)$. Moreover, we use the notation $\mathcal{L}^F_{\sigma\downarrow_a}|_l$ to restrict to l labels in the multi-set $\mathcal{L}^F_{\sigma\downarrow_a}$, where $l = \{\emptyset, in, out, undec\}$.

▶ **Definition 10** (Argument semantics expansion, contraction, revision). *Let $F = \langle Arg, R\rangle$ be an AF on the universe U, $Arg \subseteq U$, $R \subseteq Arg \times Arg$, σ a semantics, and $a \in U$ an argument.*

- *An argument semantics expansion $\oplus^\sigma_a : AF \rightarrow AF$ computes a new AF $F' = \oplus^\sigma_a(F)$ with semantics $S_\sigma(F')$ such that $\mathcal{L}^{F'}_{\sigma\downarrow_a} \supseteq \mathcal{L}^F_{\sigma\downarrow_a}$.*
- *An argument semantics contraction $\oslash^\sigma_a : AF \rightarrow AF$ computes a new AF $F' = \oslash^\sigma_a(F)$ with semantics $S_\sigma(F')$ such that $\mathcal{L}^F_{\sigma\downarrow_a} \supseteq \mathcal{L}^{F'}_{\sigma\downarrow_a}$.*
- *An argument semantics revision $\circledast^\sigma_a : AF \rightarrow AF$ computes a new AF $F' = \circledast^\sigma_a(F)$ with semantics $S_\sigma(F')$ such that $\forall b \in Arg. \left|\mathcal{L}^F_{\sigma\downarrow_b}|_{undec}\right| \geq \left|\mathcal{L}^{F'}_{\sigma\downarrow_b}|_{undec}\right|$ (that is no inconsistencies are introduced), and:*
  - *in-to-out revision: $\left|\mathcal{L}^F_{\sigma\downarrow_a}|_{out}\right| < \left|\mathcal{L}^{F'}_{\sigma\downarrow_a}|_{out}\right| \wedge \left|\mathcal{L}^F_{\sigma\downarrow_a}|_{in}\right| > \left|\mathcal{L}^{F'}_{\sigma\downarrow_a}|_{in}\right|$;*
  - *out-to-in revision: $\left|\mathcal{L}^F_{\sigma\downarrow_a}|_{in}\right| < \left|\mathcal{L}^{F'}_{\sigma\downarrow_a}|_{in}\right| \wedge \left|\mathcal{L}^F_{\sigma\downarrow_a}|_{out}\right| > \left|\mathcal{L}^{F'}_{\sigma\downarrow_a}|_{out}\right|$;*

It is important to note that the formalism we present is not monotone: the *add* operation may lead to a contraction, reducing the number of arguments with the labels *in* and/or *out*. Similarly, the removal of an argument may lead to an expansion (this is the case of Figure 8).

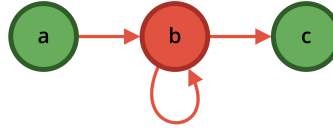


Figure 8 Example of argument extension expansion. Removing the *in* argument a makes both b and c *undec*.

AGM operators have already been studied from the point of view of their implementation in work as [5, 14], especially with regard to enforcement. However, in the previous literature, realisability of extensions and not of single arguments is considered. The implementation of an argument expansion/contraction/revision operator changes according to the semantics we take into account. In the following, we consider the grounded semantics and show how the operators of Definitions 9 can be implemented. For the grounded semantics, that only has one extension, Definitions 9 and 10 coincide. Notice also that there exist many ways to obtain expansion, contraction and revision. We chose one that leverage between minimality with respect to the changes required in the framework and linearity of implementation.

▶ **Proposition 11.** *Let $F = \langle Arg, R\rangle$ be an AF on the universe U, $Arg \subseteq U$, $R \subseteq Arg \times Arg$, $a \in U$ an argument, and L the unique grounded labelling. A possible argument extension expansion $\oplus^{gde,l}_{a,L}(F)$ could act as:*

- *if* $L(a) = \emptyset$ *and* $l = in$*, add* a *to* Arg
- *if* $L(a) = \emptyset$ *and* $l = out$*,*
  - *if* $\exists b \in Arg \mid L(b) = in$*, add* $\langle\{a\}, \{(b,a)\}\rangle$ *to* F
  - *otherwise, add* $\langle\{a,b\}, \{(b,a)\}\rangle$ *to* F
- *if* $L(a) = in$ *and* $l = undec$*,*
  - *if* $\exists b \in Arg \mid L(b) = undec$*, add* (b,a) *to* R
  - *otherwise, add* (a,a) *to* R
- *if* $L(a) = out$ *and* $l = undec$*,*
  - $\forall b \in Arg \mid L(b) = \{in\} \wedge (b,a) \in R$*, add* (a,b) *to* R

Proof. If a has an empty label, it means that $a \in U \setminus Arg$, since the grounded labelling assigns a label different from $\emptyset$ to all arguments in Arg. It is then sufficient to add a to the set of considered arguments Arg to make it in. If the freshly added argument is attacked by another in argument, it becomes out. Continuing, a is labelled $undec$ in the grounded labelling only if it is attacked by an $undec$ argument (included a itself), thus, to make an in argument a become $undec$ we can look for an argument b in Arg that is already labelled as $undec$. If we find such a b then it is sufficient to add the attack relation from b to a to the store. Otherwise, we make a attack itself. Finally, if we want an out argument a to become $undec$, we make it attack back all its in attackers. Doing so, we obtain three distinct complete labellings: one in which a is accepted and its attackers are not, another one in which the opposite situation occurs, and the third labelling in which neither a nor its attackers are fully accepted or rejected (that is they are $undec$). Hence, a will be $undec$ in the minimal complete labelling (that, by Definition 6, is also grounded). ◀

▶ **Proposition 12.** *Let* $F = \langle Arg, R\rangle$ *be an AF on the universe* U*,* $Arg \subseteq U$*,* $R \subseteq Arg \times Arg$*,* $a \in U$ *an argument, and* L *the unique grounded labelling. A possible argument extension contraction* $\oslash^{gde,l}_{a,L}(F)$ *could act as:*

- *if* $L(a) = undec$ *and* $l = in$*,* $\forall b \in Arg \mid L(b) = undec$*, remove* (b,a) *from* R
- *if* $L(a) = undec$ *and* $l = out$*,*
  - *if* $\exists b \in Arg \mid L(b) = in$*, add* (b,a) *to* R
  - *otherwise, add* $\langle\{b\}, \{(b,a)\}\rangle$ *to* F
- *if* $L(a) = in$ *and* $l = \emptyset$*, remove* a *(and all attacks involving* a*) from* F
- *if* $L(a) = out$ *and* $l = \emptyset$*, remove* a *(and all attacks involving* a*) from* F

Proof. Consider a grounded labelling. An $undec$ argument a can become in by removing all attacks coming from $undec$ arguments (included a itself). Indeed an argument is $undec$ only if it is attacked by another $undec$. Note that a cannot be attacked by in arguments, otherwise it would have been out. Therefore, after the changes a is only attacked by out arguments, and thus is in. Alternatively, a can become out when it is attacked by another in argument b (when the store does not contain in arguments, we add one from the universe). If a is either in or out, instead, we can contract its label to $undec$ through the removal of a itself form the store. ◀

▶ **Proposition 13.** *Let* $F = \langle Arg, R\rangle$ *be an AF on the universe* U*,* $Arg \subseteq U$*,* $R \subseteq Arg \times Arg$*,* $a \in U$ *an argument, and* L *the unique grounded labelling. A possible argument extension revision* $\circledast^{gde,l}_{a,L}(F)$ *could act as:*

- *if* $L(a) = in$*,*
  - *if* $\exists b \in Arg \mid L(b) = in$*, add* (b,a) *to* R *and then* $\forall c \in Arg \mid (a,c) \in R$*, add* (b,c) *to* R
  - *otherwise, add* $\langle\{b\}, \{(b,a)\}\rangle$ *to* F *and then* $\forall c \in Arg \mid (a,c) \in R$*, add* (b,c) *to* R
- *if* $L(a) = out$*,* $\forall b \in Arg \mid L(b) \in \{in, undec\}$*, remove* (b,a) *from* R *and then* $\forall c \in Arg \mid (a,c) \in R \wedge L(c) \in \{in, undec\}$*, remove* (a,c) *from* R

Proof. Given a grounded labelling we want to change the label of a from *in* to *out* (or vice versa), while preserving the labels of all other arguments. If a is *in*, we can look for another argument b labelled *in* and make b attack a, together with all other arguments attacked by a. If the store does not contain any *in* argument, we take one from the universe. If a is *out*, we remove all the attacks coming from *in* and *undec* arguments, so that the only attacks left come from *out* arguments and a becomes *in*. To preserve the labels of the other arguments, all attacks from a towards *in* and *undec* are removed, since they would have become *out* after the revision of a. *out* arguments attacked by a does not need further adjustments. ◀

Note that the argument extension revision we propose for grounded semantics in Proposition 13 is more restrictive than necessary, since ensure all the arguments different from a (that is the argument to be revised) to maintain the exact same labels, while Definition 9 only forbids to change the label to *undec*. For each operator, we also show how to implement it in our language.

▶ **Proposition 14.** *The argument extension expansion, contraction and revision in Propositions 12, 12 and 13, respectively, can be implemented in our language.*

Proof. We show an example of possible implementations in Tables 4, 5 and 6. We make use of some syntactic sugar to simplify the presentation of the results. Let be $|Arg| = n$:

- $E_1 \wedge E_2 \rightarrow A$ represents $E_1 \rightarrow E_2 \rightarrow A$;
- $E_1 \vee E_2 \rightarrow A$ represents $E_1 \rightarrow A + E_2 \rightarrow A$;
- $true$ represents a dummy $check(\{\},\{\})$;
- $\sum_{a \in Arg} (E(a))$ represents $E(a_1) + E(a_2) + \cdots + E(a_n)$, $\forall a_i \in Arg$;
- $\big\|_{G \atop a \in Arg} (E(a))$ represents $E(a_1) \|_G E(a_2) \|_G \ldots \|_G E(a_n)$, $\forall a_i \in Arg$;
- $test_c(a, S, \sigma) \rightarrow A$ represents $\sum_{l \in S} (test_c(a, l, \sigma))$.

We also use the letter u to identify fresh arguments taken from $U \setminus Arg$. ◀

We want to emphasize that guarded parallelism $\|_G$ and if then else constructs realised through $+_P$ are crucial for the implementation of the aforementioned operators. For instance, we use $\|_G$ in the argument extension contraction (Table 5) to remove all and only the attacks towards a coming from *undec* arguments. This behaviour cannot be achieved through classical parallelism (which only succeeds when all the branches terminates). The operator $+_P$, instead, is used in Table 4 to realise the expansion from $\emptyset$ to *out*: if an *in* argument b can be found in the framework, then we add an attack from b to a; otherwise we have to introduce, beforehand, an *in* argument. Without an if then else construct it is not possible to prioritise the choice of looking for an existing *in* argument and an agent could arbitrarily add a new argument even if it is not needed.

In devising operations of Definitions 9 and 10, that allow agents for changing the labels of arguments in a shared knowledge base with respect to a given semantics, we reinterpret AGM operators for expansion, contraction and revision. In particular, our operations are restricted to a single argument, rather than considering a set of beliefs as in other approaches like [14] and [5]. Nonetheless, we maintain similarities with the AGM theory, to the point that we can highlight some similarities with the original postulates of [1] that characterise rational operators performing expansion, contraction and revision of beliefs in a knowledge base. Consider for instance an argument a of an AF F and a semantics σ. An argument semantics expansion $\oplus_a^\sigma$ produces as output an AF F' for which no labelling $L' \in S_\sigma(F')$ is such that a has less labels in L' than in any labelling L of F (i.e., the number of labels assigned to a either remains the same or increases after the expansion).

Table 4 Argument extension expansion operator (Proposition 11) in CA syntax.

$$\oplus^{gde,in}_{\substack{a,L\\(L(a)=\emptyset)}}(F) : add(\{a\},\{\}) \rightarrow success$$

$$\oplus^{gde,out}_{\substack{a,L\\(L(a)=\emptyset)}}(F) : \sum_{b\in Arg} (test_c(b,in,gde) \rightarrow add(\{a\},\{(b,a)\})) \rightarrow success$$
$$+_P$$
$$add(\{a,u\},\{(u,a)\}) \rightarrow success$$

$$\oplus^{gde,undec}_{\substack{a,L\\(L(a)=in)}}(F) : \sum_{b\in Arg} (test_c(b,undec,gde) \rightarrow add(\{\},\{(b,a)\})) \rightarrow success$$
$$+_P$$
$$add(\{\},\{(a,a)\}) \rightarrow success$$

$$\oplus^{gde,undec}_{\substack{a,L\\(L(a)=out)}}(F) : \Big\|_{\substack{G\\b\in Arg}} (test_c(b,in,gde) \wedge check(\{\},\{(b,a)\})$$
$$\rightarrow add(\{\},\{(a,b)\})) \rightarrow success$$

Table 5 Argument extension contraction operator (Proposition 12) in CA syntax.

$$\oslash^{gde,in}_{\substack{a,L\\(L(a)=undec)}}(F) : \Big\|_{\substack{G\\b\in Arg}} (test_c(b,undec,gde) \rightarrow rmv(\{\},\{(b,a)\})) \rightarrow success$$

$$\oslash^{gde,out}_{\substack{a,L\\(L(a)=undec)}}(F) : \sum_{b\in Arg} (test_c(b,in,gde) \rightarrow add(\{\},\{b,a\})) \rightarrow success$$
$$+_P$$
$$add(\{u\},\{u,a\}) \rightarrow success$$

$$\oslash^{gde,\emptyset}_{\substack{a,L\\(L(a)=in)}}(F) : rmv(\{a\},\{\}) \rightarrow success$$

$$\oslash^{gde,\emptyset}_{\substack{a,L\\(L(a)=out)}}(F) : rmv(\{a\},\{\}) \rightarrow success$$

5 Related Work

A formalism for expressing dynamics in AFs is defined in [28] as a *Dynamic Argumentation Framework* (DAF). The aim of that paper is to provide a method for instantiating Dung-style AFs by considering a universal set of arguments U. A DAF consists of an AF $\langle U, R\rangle$ and a set of evidence, which has the role of restricting $\langle U, R\rangle$ to possible arguments and relations, so to obtain a static instance of the framework. DAFs are built starting from argumental structures, in which a tree of arguments supports a claim (corresponding to the root of the tree), and then adding attacks between argumental structures. The dynamic component of a DAF is thus the set of evidence. The introduced approach allows for generalising AFs,

Table 6 Argument extension revision operator (Proposition 13) in CA syntax.

$$\underset{(L(a)=in)}{\circledast^{gde,out}_{a,L}(F)}: \sum_{b\in Arg} (test_c(b, in, gde) \rightarrow add(\{\}, \{(b, a)\})$$
$$\rightarrow \|_{G\, c\in Arg} (check(\{c\}, \{a, c\}) \rightarrow add(\{\}, \{(b, c)\})) \|_G\, true \rightarrow success$$
$$+_P$$
$$add(\{b\}, \{(b, a)\})$$
$$\rightarrow \|_{G\, c\in Arg} (check(\{c\}, \{a, c\}) \rightarrow add(\{\}, \{(b, c)\})) \|_G\, true) \rightarrow success$$
$$\underset{(L(a)=out)}{\circledast^{gde,in}_{a,L}(F)}: \|_{G\, b\in Arg} (test_c(b, \{in, undec\}, gde) \rightarrow rmv(\{\}, \{(b, a)\}))$$
$$\rightarrow \|_{G\, c\in Arg} ($$
$$test_c(c, \{in, undec\}, gde) \wedge check(\{c\}, \{a, c\})$$
$$\rightarrow rmv(\{\}, \{(a, c)\}) \|_G\, true$$
$$) \rightarrow success$$

adding the possibility of modelling changes, but, contrary to our study, it does not consider how such modifications affect the semantics and does not allow to model the behaviour of concurrent agents.

The impact of modifications on an AF in terms of sets of extensions is studied in [13]. Different kinds of revision are introduced, in which a new argument interacts with an already existing one. The authors describe different kinds of revision differing in the number of extensions that appear in the outcome, with respect to a semantics: a *decisive* revision allows to obtain a unique non-empty extension, a *selective* revision reduces the number of extensions (to a minimum of two), while a *questioning* one increases that number; a destructive *revision* eliminates all extensions, an *expansive* revision maintain the number of extension and increases the number of accepted arguments; a *conservative* revision does not introduce changes on the semantics level (and is strictly connected to the notion of robustness [9]), and an *altering* revision insert and delete arguments in the extensions. All these revisions are obtained through the addition of a single argument, together with a single attack relation either towards or from the original AF, and can be implemented as procedures of our language. The review operator we define in the syntax of our language (as the other two operator for expansion and contraction), instead, does not consider whole extensions, but just an argument at a time, allowing communicating agents to modify their beliefs in a finer grain.

Focusing on syntactic expansion of an AF (the mere addition of arguments and attacks), [5] show under which conditions a set of arguments can be enforced (to become accepted) for a specific semantics. Moreover, since adding new arguments and attacks may lead to a decrease in term of extensions and accepted arguments, the authors also investigate whether an expansion behave in a monotonic fashion, thus preserving the status of all originally accepted arguments. The study is only conducted on the case of weak expansion (that adds further arguments which do not attack previous arguments). The notion of expansion we use in the presented work is very different from that in [5]. First of all, we take into

account semantics when defining the expansion, making it more similar to an enforcement itself: we can increment the labels of an argument so to match a desired acceptance status. Then, our expansion results to be more general, being able to change the status of a certain argument not only to accepted, but also rejected, undecided or undetermined. This is useful, for instance, when we want to diminish the beliefs of an opponent agent.

Enforcing is also studied in [14], where the authors consider an expansion of the AF that only allows the addition of new attack relations, while the set of arguments remains the same (differently from [5]). It is shown, indeed, that if no new argument is introduced, it is always possible to guarantee the success of enforcement for any classical semantics. Also in this case, we want to highlight the differences with our work. Starting from the modifications allowed into the framework, we are not limited to only change the set of relations, since we implement procedures that also add and remove arguments. Moreover, the operators we define are not just enforcement operators, since they allow to modify the acceptability status of a single argument of an AF.

In our model, AFs are equipped with a universe of arguments that agents use to insert new information in the knowledge base. The problem of combining AFs is addressed in [6], that study the computational complexity of verifying if a subset of argument is an extension for a certain semantics in incomplete argumentation frameworks obtained by merging different beliefs. The incompleteness is considered both for arguments and attack relations. Similarly to our approach, arguments (and attacks) can be brought forward by agents and used to build new acceptable extensions. On the other hand, the scope of [6] is focused on a complexity analysis and does not provide implementations for the merging.

6 Conclusion and Future Work

We introduced a concurrent language for argumentation, that can be used by (intelligent) agents to implement different forms of communications. The agents involved in the process share an abstract argumentation framework that serves as a knowledge base and where arguments represent the agreed beliefs. The framework can be changed via a set of primitives that allow the addition and the removal of arguments and attacks. All agents have at their disposal a universe of "unused" arguments to chose from when they need to introduce new information. In order to take into account the justification status of such beliefs (which can be accepted, rejected, undetermined and inconsistent) we considered a four-state labelling semantics. Besides operations at a syntactic level, thus, we also defined semantic operation that verify the acceptability of the arguments in the store. Finally, to allow agents for realising more complex forms of communication (like negotiation and persuasion), we presented three AGM-style operators, namely of expansion, contraction and revision, that change the status of a belief to a desired one; we also showed how to implement them in our language.

For the future, we plan to extend this work in many directions. First of all, given the known issues of abstract argumentation [27], we want to consider structured AFs and provide an implementation for our expansion, contraction and revision operators, for which a different store (structured and not abstract, indeed) need to be considered. The concurrent primitives are already general enough and do not require substantial changes. To obtain a spendable implementation, we will consider operations that can be done in polynomial time [20], for instance by using the grounded semantics, for which finding and checking extension is a easy task from the point of view of computational complexity. We also plan to provide a real implementation of our language that can be used for both research purposes and practical applications.

To further improve the capabilities of our agents and make it more appealing for real-life applications, we want to extend our language with the ability to handle processes involving time-critical aspects, in a similar way as CC is extended with temporal logic in [16, 15]. In this way, we could implement operations that also take into account time constraints. The shared store could also be shaped as a subsumptive hierarchy, able to handle various relations among the arguments.

On the operations level, we are currently only able to modify the acceptance status of the arguments, without further considerations on the obtained semantics. To gain control also over changes on the set of extensions, we want to introduce operators able to obtain a specified semantics (when possible) or to leave it unchanged (this can be done relying on the notion of robustness [9]). Another study we could conduct over the operators concerns their (non-)monotonicity. Since, in the current state of the work, operations like the removal of an argument can lead to an expansion into the considered AF, we would like to investigate the conditions under which, for instance, a contraction can be the only consequence of a removal. To this extent, also other operations on beliefs (like extraction, consolidation and merging) could be taken into account.

As a final consideration, whereas in real-life cases it is always clear which part involved in a debate is stating a particular argument, AFs do not hold any notion of "ownership" for arguments or attacks, that is, any bond with the one making the assertion is lost. To overcome this problem, we want to implement the possibility of attaching labels on (groups of) arguments and attacks of AFs, in order to preserve the information related to whom added a certain argument or attack, extending and taking into account the work in [24]. Consequently, we can also obtain a notion of locality (or scope) of the belief in the knowledge base: arguments owned by a given agents can be placed into a local store and used in the implementation of specific operators through hidden variables.

References

1. Carlos E. Alchourrón, Peter Gärdenfors, and David Makinson. On the logic of theory change: Partial meet contraction and revision functions. *The Journal of Symbolic Logic*, 50(02):510–530, June 1985. doi:10.2307/2274239.
2. Pietro Baroni, Martin Caminada, and Massimiliano Giacomin. An introduction to argumentation semantics. *Knowledge Eng. Review*, 26(4):365–410, 2011. doi:10.1017/S0269888911000166.
3. Pietro Baroni and Massimiliano Giacomin. On principle-based evaluation of extension-based argumentation semantics. *Artif. Intell.*, 171(10-15):675–700, 2007. doi:10.1016/j.artint.2007.04.004.
4. Ringo Baumann. What Does it Take to Enforce an Argument? Minimal Change in abstract Argumentation. *Frontiers in Artificial Intelligence and Applications*, pages 127–132, 2012. doi:10.3233/978-1-61499-098-7-127.
5. Ringo Baumann and Gerhard Brewka. Expanding argumentation frameworks: Enforcing and monotonicity results. In Pietro Baroni, Federico Cerutti, Massimiliano Giacomin, and Guillermo Ricardo Simari, editors, *Computational Models of Argument: Proceedings of COMMA 2010, Desenzano del Garda, Italy, September 8-10, 2010*, volume 216 of *Frontiers in Artificial Intelligence and Applications*, pages 75–86. IOS Press, 2010. doi:10.3233/978-1-60750-619-5-75.
6. Dorothea Baumeister, Daniel Neugebauer, Jörg Rothe, and Hilmar Schadrack. Verification in incomplete argumentation frameworks. *Artif. Intell.*, 264:1–26, 2018. doi:10.1016/j.artint.2018.08.001.

7 Stefano Bistarelli, Lars Kotthoff, Francesco Santini, and Carlo Taticchi. Containerisation and Dynamic Frameworks in ICCMA'19. In *Proceedings of the Second International Workshop on Systems and Algorithms for Formal Argumentation (SAFA 2018) Co-Located with the 7th International Conference on Computational Models of Argument (COMMA 2018), Warsaw, Poland, September 11, 2018*, volume 2171 of *CEUR Workshop Proceedings*, pages 4–9. CEUR-WS.org, 2018.

8 Stefano Bistarelli and Francesco Santini. Conarg: A constraint-based computational framework for argumentation systems. In *IEEE 23rd International Conference on Tools with Artificial Intelligence, ICTAI 2011, Boca Raton, FL, USA, November 7-9, 2011*, pages 605–612. IEEE Computer Society, 2011. `doi:10.1109/ICTAI.2011.96`.

9 Stefano Bistarelli, Francesco Santini, and Carlo Taticchi. On Looking for Invariant Operators in Argumentation Semantics. In *Proceedings of the Thirty-First International Florida Artificial Intelligence Research Society Conference, FLAIRS 2018, Melbourne, Florida, USA. May 21-23 2018.*, pages 537–540, 2018.

10 Guido Boella, Souhila Kaci, and Leendert W. N. van der Torre. Dynamics in Argumentation with Single Extensions: Attack Refinement and the Grounded Extension (Extended Version). In *Argumentation in Multi-Agent Systems, 6th International Workshop, ArgMAS 2009. Revised Selected and Invited Papers*, volume 6057 of *Lecture Notes in Computer Science*, pages 150–159. Springer, 2009. `doi:10.1007/978-3-642-12805-9_9`.

11 Martin Caminada. On the Issue of Reinstatement in Argumentation. In *Logics in Artificial Intelligence, 10th European Conference, JELIA 2006, Liverpool, UK, September 13-15, 2006, Proceedings*, volume 4160 of *Lecture Notes in Computer Science*, pages 111–123. Springer, 2006.

12 Martin Caminada. On the Issue of Reinstatement in Argumentation. In Michael Fisher, Wiebe van der Hoek, Boris Konev, and Alexei Lisitsa, editors, *Logics in Artificial Intelligence, 10th European Conference, JELIA 2006, Liverpool, UK, September 13-15, 2006, Proceedings*, volume 4160 of *Lecture Notes in Computer Science*, pages 111–123. Springer, 2006.

13 Claudette Cayrol, Florence Dupin de Saint-Cyr, and Marie-Christine Lagasquie-Schiex. Revision of an Argumentation System. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Eleventh International Conference, KR 2008, Sydney, Australia, September 16-19, 2008*, pages 124–134. AAAI Press, 2008.

14 Sylvie Coste-Marquis, Sébastien Konieczny, Jean-Guy Mailly, and Pierre Marquis. Extension enforcement in abstract argumentation as an optimization problem. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 2876–2882. AAAI Press, 2015. URL: `http://ijcai.org/Abstract/15/407`.

15 Frank S. de Boer, Maurizio Gabbrielli, and Maria Chiara Meo. Semantics and expressive power of a timed concurrent constraint language. In Gert Smolka, editor, *Principles and Practice of Constraint Programming – CP97, Third International Conference, Linz, Austria, October 29 – November 1, 1997, Proceedings*, volume 1330 of *Lecture Notes in Computer Science*, pages 47–61. Springer, 1997. `doi:10.1007/BFb0017429`.

16 Frank S. de Boer, Maurizio Gabbrielli, and Maria Chiara Meo. A timed concurrent constraint language. *Inf. Comput.*, 161(1):45–83, 2000. `doi:10.1006/inco.1999.2879`.

17 Sylvie Doutre, Andreas Herzig, and Laurent Perrussel. A Dynamic Logic Framework for Abstract Argumentation. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference, KR 2014, Vienna, Austria, July 20-24, 2014*, 2014.

18 Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence*, 77(2):321–357, September 1995. `doi:10.1016/0004-3702(94)00041-X`.

19 Florence Dupin de Saint-Cyr, Pierre Bisquert, Claudette Cayrol, and Marie-Christine Lagasquie-Schiex. Argumentation update in YALLA (Yet Another Logic Language for Argumentation). *International Journal of Approximate Reasoning*, 75:57–92, August 2016. `doi:10.1016/j.ijar.2016.04.003`.

20 Wolfgang Dvorák and Paul E. Dunne. Computational problems in formal argumentation and their complexity. *FLAP*, 4(8), 2017. URL: `http://www.collegepublications.co.uk/downloads/ifcolog00017.pdf`.

21 Jacob Glazer and Ariel Rubinstein. Debates and Decisions: On a Rationale of Argumentation Rules. *Games and Economic Behavior*, 36(2):158–173, 2001. `doi:10.1006/game.2000.0824`.

22 Hadassa Jakobovits and Dirk Vermeir. Robust semantics for argumentation frameworks. *J. Log. Comput.*, 9(2):215–261, 1999. `doi:10.1093/logcom/9.2.215`.

23 Magdalena Kacprzak, Katarzyna Budzynska, and Olena Yaskorska. A logic for strategies in persuasion dialogue games. In *Advances in Knowledge-Based and Intelligent Information and Engineering Systems – 16th Annual KES Conference, San Sebastian, Spain, 10-12 September 2012*, volume 243 of *Frontiers in Artificial Intelligence and Applications*, pages 98–107. IOS Press, 2012. `doi:10.3233/978-1-61499-105-2-98`.

24 Nicolas Maudet, Simon Parsons, and Iyad Rahwan. Argumentation in Multi-Agent Systems: Context and Recent Developments. In *Argumentation in Multi-Agent Systems, Third International Workshop, ArgMAS 2006, Hakodate, Japan, May 8, 2006, Revised Selected and Invited Papers*, pages 1–16, 2006. `doi:10.1007/978-3-540-75526-5_1`.

25 Henry Prakken. Models of Persuasion Dialogue. In *Argumentation in Artificial Intelligence*, pages 281–300. Springer, 2009. `doi:10.1007/978-0-387-98197-0_14`.

26 Henry Prakken. An abstract framework for argumentation with structured arguments. *Argument & Computation*, 1(2):93–124, 2010. `doi:10.1080/19462160903564592`.

27 Henry Prakken and Michiel De Winter. Abstraction in argumentation: Necessary but dangerous. In Sanjay Modgil, Katarzyna Budzynska, and John Lawrence, editors, *Computational Models of Argument – Proceedings of COMMA 2018, Warsaw, Poland, 12-14 September 2018*, volume 305 of *Frontiers in Artificial Intelligence and Applications*, pages 85–96. IOS Press, 2018. `doi:10.3233/978-1-61499-906-5-85`.

28 Nicolas D. Rotstein, Martın O. Moguillansky, Alejandro J. Garcia, and Guillermo R. Simari. An abstract argumentation framework for handling dynamics. In *Proceedings of the Argument, Dialogue and Decision Workshop in NMR 2008, Sydney, Australia*, pages 131–139, 2008.

29 Vijay A. Saraswat and Martin Rinard. Concurrent constraint programming. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages – POPL '90*, pages 232–245, San Francisco, California, United States, 1990. ACM Press. `doi:10.1145/96709.96733`.

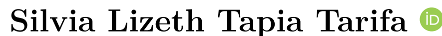
30 Francesca Toni. A tutorial on assumption-based argumentation. *Argument & Computation*, 5(1):89–117, 2014. `doi:10.1080/19462166.2013.869878`.

Inseguendo Fagiani Selvatici: Partial Order Reduction for Guarded Command Languages

Frank S. de Boer
CWI Amsterdam, The Netherlands
f.s.de.boer@cwi.nl

Einar Broch Johnsen
Department of Informatics, University of Oslo, Norway
einarj@ifi.uio.no

Rudolf Schlatte
Department of Informatics, University of Oslo, Norway
rudi@ifi.uio.no

Silvia Lizeth Tapia Tarifa 
Department of Informatics, University of Oslo, Norway
sltarifa@ifi.uio.no

Lars Tveito
Department of Informatics, University of Oslo, Norway
larstvei@ifi.uio.no

Abstract

This paper presents a method for testing whether objects in actor languages and active object languages exhibit locally deterministic behavior. We investigate such a method for a class of guarded command programs, abstracting from object-oriented features like method calls but focusing on cooperative scheduling of dynamically spawned processes executing in parallel. The proposed method can answer questions such as whether all permutations of an execution trace are equivalent, by generating candidate traces for testing which may lead to different final states. To prune the set of candidate traces, we employ partial order reduction. To further reduce the set, we introduce an analysis technique to decide whether a generated trace is schedulable. Schedulability cannot be decided for guarded commands using standard dependence and interference relations because guard enabledness is non-monotonic. To solve this problem, we use concolic execution to produce linearized symbolic traces of the executed program, which allows a weakest precondition computation to decide on the satisfiability of guards.

2012 ACM Subject Classification Software and its engineering → Automated static analysis; Software and its engineering → Software testing and debugging; Software and its engineering → Semantics; Theory of computation → Semantics and reasoning

Keywords and phrases Testing, Symbolic Traces, Guarded Commands, Partial Order Reduction

Digital Object Identifier 10.4230/OASIcs.Gabbrielli.2020.10

1 Introduction

Let us open this paper with the allegory of the pheasant-chasing wine-maker:

> *A vineyard is a place where wild pheasants are gobbling up the grapes and where wine-makers chase these pheasants off the land. During this Sisyphean undertaking, a theoretically inclined wine-maker may wonder: "will the order in which I chase the pheasants affect the yield at season's end?" Overwhelmed by the existential dimensions of this question, the wine-maker could but observe the unfolding of the feast.*

© Frank S. de Boer, Einar Broch Johnsen, Rudolf Schlatte, S. Lizeth Tapia Tarifa, and Lars Tveito;
licensed under Creative Commons License CC-BY

Leaving aside its existential dimensions, the astute reader may observe that the problem of the pheasant-chasing wine-maker bears a remarkable similarity to the problem of testing whether dynamically spawned processes operating on a shared state space exhibit deterministic behavior in the sense that the final state is independent of the local scheduling decisions. We hope that we, by studying a particular instance of the latter problem, can also shed some light on the former.

Different forms of dynamically spawned processes have been studied extensively by Gabbrielli [12–14]. The problem that we address in this paper is to test whether in an *imperative* context cooperatively scheduled tasks executing on a *shared state space* exhibit *deterministic* behavior. This problem lies at the heart of the development of a testing theory for a single active object, or a single actor in an actor system. For a general overview of different programming languages and paradigms, we refer to Gabbrielli's course notes [28].

Actor systems [2] are inherently prone to social distancing, based on a strong sense of isolation and asynchronous communication: by design, one actor cannot directly affect the local state of another actor, it only sends messages. Active objects [10] extend these attractive features of actors to an object-oriented setting with asynchronous method calls and futures [22, 26]. As a consequence, both actor and active object systems are *almost* confluent: if there are no communication races and local scheduling is deterministic, asynchronously communicating objects have been shown to have deterministic behavior [17].

Active object languages such as ABS [33] and Encore [11] allow methods to be cooperatively scheduled: a task executing on an object may choose to suspend itself and allow other tasks to be scheduled, such that the original task can only be rescheduled once an associated Boolean condition holds. This extension makes the behavior highly non-deterministic because suspended tasks that are enabled can be arbitrarily selected by the scheduler when an object is idle. The authors have previously shown that even for cooperatively scheduled active objects, it is sufficient to control the local behavior of each actor to ensure global confluence [9, 42] and developed an axiomatic semantics of trace reachability for active objects [23]. However, neither line of work addresses the problem of testing determinacy for active objects with cooperatively scheduled tasks.

In this paper, we study the problem of testing whether an object with cooperatively scheduled tasks locally exhibits deterministic behavior in the sense that the final state of the object is independent of the local scheduling of tasks. The cooperatively scheduled tasks of a single object can be abstracted in terms of a guarded command language over shared state. The paper develops a behavioral theory of guarded commands in a slight variation of Dijkstra's guarded command language GCL [24]. The problem of deterministic behavior can be formulated as determining whether all feasible schedulings of tasks (also called processes in the sequel) will produce the same final state, or as testing whether, given a trace of a guarded command program, are there permutations of the trace that are executable but not observationally equivalent to the original trace? We tackle this problem by means of concolic execution [15, 16, 31], such that the concrete run produces a linearized, symbolic trace of the executed program. We then combine techniques for weakest-precondition calculation [7, 25] and for partial order reduction [19, 29] to compute, for a given symbolic trace, all executable traces which only differ in the interleaving of the individual local computations of the tasks (threads) of the given trace and which may result in a different final state.

2 Partial Order Reduction

Partial order reduction (POR) is a technique to reduce the size of the search space when exploring the different executions of a parallel program by exploiting the commutativity of concurrently executed *independent* transitions [19, 29]. This commutativity relation between transitions is lifted to an equivalence relation $\sim$ on traces over these transitions. Given a trace θ reflecting an interleaved execution of a number of parallel processes, we denote by $[\theta]$ the set of traces equivalent to θ according to the equivalence relation $\sim$; the traces in this set agree on the sequential order of transitions for the individual processes, but the processes may be interleaved in different ways. Thus, all equivalent traces have the same length and contain the same labeled transition steps. Let $s \xrightarrow{\theta} s'$ denote that a state s' can be reached from a state s by a sequence of labeled transition steps, where the trace θ is the corresponding sequence of labels representing scheduling events for the different transition steps. We use the following notation for traces: ε denotes the empty trace and $\theta \cdot \tau$ the composition of a trace θ and an event τ. With a slight abuse of notation, we will write $\theta_1 \cdot \theta_2$ for the composition of traces θ_1 and θ_2 and $\tau \cdot \theta$ for the trace which starts with event τ and continues as trace θ.

The pruning of the search based on traces during model exploration can be justified when the traces are sufficiently expressive to make sure that equivalent traces lead to equal states; i.e., the following must be a theorem [29]:

▶ **Theorem 1.** *If* $s_0 \xrightarrow{\theta_1} s_1$*,* $s_0 \xrightarrow{\theta_2} s_2$ *and* $\theta_2 \in [\theta_1]$*, then* $s_1 = s_2$*.*

Observe that, given a trace θ, the elements of $[\theta]$ can be enumerated by successively permuting adjacent commuting (i.e., independent) transition steps. An additional problem is to identify syntactic criteria to approximate this semantic notion of equivalence. This can be done by identifying transitions that correspond to *interference-free* statements [6]; e.g., two transitions are independent if their corresponding statements do not affect each others' program variables.

▶ **Example 2** (Independent processes)**.** Consider a program

```
x↦1,y↦2, { x := x+1 || y := 3 }
```

with two parallel statements `x := x+1` and `y := 3` and a shared state that is initialized with program variables `x` with value `1` and `y = 2`. Assume that the two statements are executed by processess ι_1 and ι_2, respectively, and, for simplicity, that the execution of each assignment is atomic. The final result of this program is a state in which `x` has value `2` and `y` has value `3`. However, there are two traces θ_1 and θ_2 of this program, reflecting that either of the two processes ι_1 and ι_2 can be scheduled first without affecting the outcome of the program. Thus, θ_1 and θ_2 belong to the same equivalence class.

POR can be used to explore the different equivalence classes of executions, without exploring every execution path of each equivalence class. If we can decide whether two traces are in the same equivalence class, we can stop the analysis of a candidate execution path if we know that its trace is equivalent to the trace of an execution that has already been explored.

▶ **Example 3** (Interfering processes)**.** Consider a program

```
x↦1, { x := x+1 || x := 3 }
```

which describes two processes `x := x+1` and `x := 3` executing interleaved on a shared state where program variable `x` has value `1`. This program may have two outcomes: in the final state, `x` will have as value either `3` or `4`, depending on the scheduling of the processes. Let us assume that the two statements are executed by processes ι_1 and ι_2, respectively. Then these two executions can be represented by execution traces θ_1 and θ_2 in which the scheduling of transition steps (which we can represent as events) for ι_1 and ι_2 occur in different orders. Since the outcome of executing the program depends on the selected trace, θ_1 and θ_2 do not belong to the same equivalence class.

The interference of two scheduling events can be approximated by syntactic criteria. A common approach is to decorate the scheduling events with the sets of read and written variables in the executed atomic blocks, such that a standard notion of interference [6] can be applied (i.e., write-variables in one process interfere with both read- and write-variables in the other process). In this case, scheduling events τ will have the format $\iota\langle W, R\rangle$, where ι is a process identifier and W and R are the write- and read-sets of the underlying transition. Thus, we get the traces $\iota_1\langle\{x\},\{x\}\rangle \cdot \iota_2\langle\{x\},\emptyset\rangle$ and $\iota_2\langle\{x\},\emptyset\rangle \cdot \iota_1\langle\{x\},\{x\}\rangle$ for the executions of Example 3. With events on this format, we can syntactically approximate non-interference by comparing write- and read-sets:

$$\iota_i\langle W_i, R_i\rangle \sim \iota_j\langle W_j, R_j\rangle \iff W_i \cap (R_j \cup W_j) = \emptyset \wedge (W_i \cup R_i) \cap W_j = \emptyset.$$

By Theorem 1, two traces $\theta_1 \cdot \tau_1 \cdot \tau_2 \cdot \theta_2$ and $\theta_1 \cdot \tau_2 \cdot \tau_1 \cdot \theta_2$ are equivalent and lead to the same final state s_1 from a given initial state s_0 if the events τ_1 and τ_1 are non-interfering, as captured by $\tau_1 \sim \tau_2$. In this case, there is no need to execute both traces. In contrast, if $\tau_1 \not\sim \tau_2$, the two traces may be in different equivalence classes and can lead to different final states from s_0. However, we cannot in general know that the events in $\tau_1 \cdot \theta_2$ can be executed after $\theta_1 \cdot \tau_2$. For example, with dynamically spawned processes, the execution associated with τ_1 may create the process scheduled by τ_2. This dependency between events can be captured by a so-called *must happen before* relation, which is a transitive relation over the events of the traces: if two events τ_1 and τ_2 associated with different processes are in a causal ordering, they cannot be permuted even if $\tau_1 \sim \tau_2$. Thus, there is a clear resemblance between equivalence classes in our setting and Mazurkiewicz traces [35].

POR can be used to systematically generate traces which correspond to all possible behaviors of a program up to trace equivalence without executing all the traces of the program, for example for the purpose of systematic testing [4]. The basic idea is to ensure that all equivalence classes are visited by at least one execution. Given a trace θ that corresponds to some execution, and $\theta_1 \cdot \tau_1 \cdot \tau_2 \cdot \theta_2 \in [\theta]$ (where $[\theta]$ is an equivalence class) such that $\tau_1 \not\sim \tau_2$, we know that the any trace that extends $\theta_1 \cdot \tau_2$ is a candidate trace for a different equivalence class than $[\theta]$. An algorithm can successively run executions which extend a given trace prefix (e.g., $\theta_1 \cdot \tau_2$), such that all equivalence classes will eventually be visited. The best known such algorithm is perhaps DPOR [27], which is usually implemented by directly manipulating the data structures and the scheduler of the runtime system of a targeted language.

3 GCL: A Language with Guarded Commands

This section presents a guarded command language (hereafter GCL), which is a slight simplifying variation on Dijkstra's original language [24]. The syntax of GCL is given in Figure 1. A program *Prog* consists of a state σ and a guarded statement g. The state σ binds program variables x to values v. Guarded statements g are **`skip`**, sequential composition

$$
\begin{aligned}
\mathit{Prog} &::= \sigma,\ g \\
\sigma \in \mathit{State} &::= \epsilon \mid \sigma[x \mapsto v] \\
g \in \mathit{GrdStm} &::= \texttt{skip} \mid e \rhd s \mid s \lhd e \rhd s \mid g; g \\
s \in \mathit{Stm} &::= x := e \mid \texttt{spawn}(g) \\
e \in \mathit{Exp} &::= \texttt{True} \mid \texttt{False} \mid x \mid v \mid op(e, \ldots, e)
\end{aligned}
$$

Figure 1 The syntax of *GCL*.

$g; g$ and the two statements $e \rhd s$ and $s_1 \lhd e \rhd s_2$, where e is an expression and s, s_1, s_2 are simple statements. Simple statements s are assignments $x := e$ and $\texttt{spawn}(g)$. The guarded statement $e \rhd s$ allows s to be executed when the guard e holds. The guarded statement $s_1 \lhd e \rhd s_2$ will execute s_1 when e holds, otherwise s_2. The guarded statement `skip` is used to denote both internal actions and termination, i.e., we identify g with $g; \texttt{skip}$ and define a solitary $\iota(\texttt{skip})$ (explained below) to represent a terminated process. Expressions e include basic propositions `True` and `False`, program variables x, values v (such as Boolean values `true` and `false`, and numbers), and operations over expressions (such as addition of numbers and logical operators over Booleans). Expressions are assumed to be well-typed. Their syntax is standard and not further detailed in Figure 1. The GCL language is kept intentionally simple, but see Section 6 for some straightforward extensions such as loops, nested guarded statements and procedure calls.

Runtime syntax

The execution of GCL programs is organized around a set of processes in the form of guarded commands. Processes are executed in an interleaved way. Any enabled process may be selected to execute at any scheduling point, which makes GCL highly non-deterministic. The runtime syntax extends Fig. 1 as follows:

$$
\begin{aligned}
rs \in \mathit{RuntimeState} &::= \sigma, P \\
P \in \mathit{ProcessSet} &::= \emptyset \mid \{\iota(g)\} \mid P \cup P
\end{aligned}
$$

A runtime configuration rs is a tuple which consists of a state σ and a set P of processes. A state σ assigns values to the program's shared variables. A state update $\sigma[x \mapsto v]$ denotes the state resulting from assigning the value v to the variable x. By $\sigma(e)$ we denote the value resulting from the evaluation of the expression e in state σ. Note that evaluating expressions is free of side effects. Processes are written $\iota(g)$ and consist of a process identifier ι and a guarded statement g (which can be a compound statement $g; g$). All processes in a runtime configuration are required to have unique process identifiers. The initial process, which is not created by a `spawn` statement, is assigned the process identifier ι_0.

The operational semantics of GCL is defined as a transition system $rs \to rs'$ between runtime configurations rs and rs', shown in Fig. 2. Rule Assign updates the global state with the effect of an assignment under the assumption that the guard is true. Rule Spawn creates a new process. The process identifier ι' of the spawned process is non-deterministically chosen (uniqueness is guaranteed by the Interleaving rule and the above requirement that all its processes in a runtime configuration have unique process identifiers). Rule Skip can always reduce since its implicit guard is taken to be `True`. Each of the Choice rules schedules the enabled guarded statement (note that the premises of both rules result from the execution of the enabled statement). Rule Interleaving nondeterministically chooses a process to execute, which will trigger the execution of one of the other rules, depending on the guarded statement g.

$$\frac{\sigma(e) = \texttt{True} \quad \sigma' = \sigma[x \mapsto \sigma(e')]}{\sigma, \{\iota(e \rhd x := e'; g)\} \to \sigma', \{\iota(g)\}} \ (\textsc{Assign})$$

$$\frac{\sigma, \{\iota(e \rhd s_1; g)\} \to \sigma', P}{\sigma, \{\iota(s_1 \lhd e \rhd s_2; g)\} \to \sigma', P} \ (\textsc{Choice1})$$

$$\frac{\sigma(e) = \texttt{True}}{\sigma, \{\iota(e \rhd \texttt{spawn}(g'); g)\} \to \sigma, \{\iota(g), \iota'(g')\}} \ (\textsc{Spawn})$$

$$\frac{\sigma, \{\iota(\neg e \rhd s_2; g)\} \to \sigma', P}{\sigma, \{\iota(s_1 \lhd e \rhd s_2; g)\} \to \sigma', P} \ (\textsc{Choice2})$$

$$(\textsc{Skip}) \quad \sigma, \{\iota(\texttt{skip}; g)\} \to \sigma, \{\iota(g)\}$$

$$\frac{\sigma, \{\iota(g)\} \to \sigma', P'}{\sigma, P \cup \{\iota(g)\} \to \sigma', P \cup P'} \ (\textsc{Interleaving})$$

Figure 2 Operational semantics of GCL.

The *initial runtime configuration* of a program σ, g is given by $\sigma, \{\iota_0(g)\}$ with the initial process identifier ι_0. A successful execution of a program from an initial runtime configuration $\sigma, \{\iota_0(g)\}$ is a sequence of transitions which ends in a *terminal* configuration where all processes are of the form $\iota(\texttt{skip})$. An execution *deadlocks* if it reaches a non-terminal configuration in which no rule is applicable. This can happen when the guards of all the initial statements evaluate to `False`. We denote by $\to^*$ the transitive closure of the transition relation $\to$ and by σ, $g \to^* \sigma'$ the existence of a successful execution of the program σ, g with initial state σ and final state σ'.

4 A Concolic Semantics for GCL

A concolic semantics for GCL can be defined by lifting the non-deterministic operational semantics of Fig. 2 to a labeled transition system in which the labeled transitions of each rule of the operational semantics capture the symbolic execution step corresponding to the concrete transition. The labeled transition relation $\xrightarrow{l}$ is given in Fig. 3; apart from the labeling the rules are the same as their non-labeled versions in Fig. 2.

▶ **Definition 4** (Labels). *A label l takes one of the following forms:*

$$l ::= \tau \mid \iota(e \rhd x := e') \mid \iota(e \rhd \texttt{spawn}(\iota'))$$

Here, τ denotes the empty label and, in the other labels, ι corresponds to the identifier of the process that was executed.

The guarded statements in labels are non-branching (i.e., a guarded assignment or spawn). However, in case of the execution of a spawn instruction we record in the label the identifier of the new process (encoded by $\texttt{spawn}(\iota')$, where ι' denotes the new process identifier).

The *trace* θ generated by an execution in this transition system is the sequence of (non-empty) labels of the corresponding transition steps. The trace records a *symbolic linearization* of the executed program; i.e., the trace ignores the branching points in the control flow of the source program. Thus, there may be many traces which correspond to the different executions of a GCL program.

▶ **Example 5** (Traces). Consider a GCL program with an initial process ι_0 that spawns two processes ι_1 and ι_2, where ι_1 doubles the value of a counter `x` if a guard `flag` is true, and ι_2 increments the value of `x` by one and negates the value of `flag` four times, then increments

$$\text{(ASSIGN)}\quad \frac{\sigma(e) = \texttt{True} \quad \sigma' = \sigma[x \mapsto \sigma(e')]}{\sigma, \{\iota(e \triangleright x := e'; g)\} \xrightarrow{\iota(e \triangleright x := e')} \sigma', \{\iota(g)\}}$$

$$\text{(CHOICE1)}\quad \frac{\sigma, \{\iota(e \triangleright s_1; g)\} \xrightarrow{l} \sigma', P}{\sigma, \{\iota(s_1 \triangleleft e \triangleright s_2; g)\} \xrightarrow{l} \sigma', P}$$

$$\text{(SPAWN)}\quad \frac{\sigma(e) = \texttt{True}}{\sigma, \{\iota(e \triangleright \texttt{spawn}(g'); g)\} \xrightarrow{\iota(e \triangleright \texttt{spawn}(\iota'))} \sigma, \{\iota(g), \iota'(g')\}}$$

$$\text{(CHOICE2)}\quad \frac{\sigma, \{\iota(\neg e \triangleright s_2; g)\} \xrightarrow{l} \sigma', P}{\sigma, \{\iota(s_1 \triangleleft e \triangleright s_2; g)\} \xrightarrow{l} \sigma', P}$$

$$\text{(SKIP)}\quad \sigma, \{\iota(\texttt{skip}; g)\} \xrightarrow{\tau} \sigma, \{\iota(g)\}$$

$$\text{(INTERLEAVING)}\quad \frac{\sigma, \{\iota(g)\} \xrightarrow{l} \sigma', P'}{\sigma, P \cup \{\iota(g)\} \xrightarrow{l} \sigma', P \cup P'}$$

Figure 3 Concolic operational semantics for GCL.

an unrelated variable y. Initially, we let the counters x and y have value 0 and flag has value true. In the surface syntax of GCL, the program looks like this:

```
x↦0, y↦0, flag↦true,
True ▷ spawn(x==0∨flag ▷ x:=2*x);                              // ι1
True ▷ spawn(x:=x+2◁flag ▷ x:=x-1; True ▷ flag:=¬flag;    // ι2
             x:=x+2◁flag ▷ x:=x-1; True ▷ flag:=¬flag;
             x:=x+2◁flag ▷ x:=x-1; True ▷ flag:=¬flag;
             x:=x+2◁flag ▷ x:=x-1; True ▷ flag:=¬flag;
             True ▷ y:=y+1)
```

A possible execution of this program, resulting in a state x↦4, y↦1, flag↦true, schedules ι_2 until it completes before scheduling ι_1. The trace θ_0, of the concolic transitions corresponding to this execution is

$$\begin{aligned}\theta_0 : \quad & \iota_0(\texttt{True} \triangleright \texttt{spawn}(\iota_1)) \cdot \iota_0(\texttt{True} \triangleright \texttt{spawn}(\iota_2)) \\ & \cdot \iota_2(\texttt{flag} \triangleright \texttt{x:=x+2}) \cdot \iota_2(\texttt{True} \triangleright \texttt{flag:=}\neg\texttt{flag}) \\ & \cdot \iota_2(\neg\texttt{flag} \triangleright \texttt{x:=x-1}) \cdot \iota_2(\texttt{True} \triangleright \texttt{flag:=}\neg\texttt{flag}) \\ & \cdot \iota_2(\texttt{flag} \triangleright \texttt{x:=x+2}) \cdot \iota_2(\texttt{True} \triangleright \texttt{flag:=}\neg\texttt{flag}) \\ & \cdot \iota_2(\neg\texttt{flag} \triangleright \texttt{x:=x-1}) \cdot \iota_2(\texttt{True} \triangleright \texttt{flag:=}\neg\texttt{flag}) \\ & \cdot \iota_2(\texttt{True} \triangleright \texttt{y:=y+1}) \cdot \iota_1(\texttt{x==0}\vee\texttt{flag} \triangleright \texttt{x:=2*x})\end{aligned}$$

Note that well-formed permutations in traces θ (e.g., permutations respecting program order in the different processes) will generate different linearizations with possible different final states due to the non-deterministic nature of GCL. Such possible well-formed permutations will be explored in the next section.

Let $e[e'/x]$ denote the substitution operation which replaces all occurrences of x in e by e' (it binds stronger than any other logical operation/connective). The path condition of a symbolic trace θ expresses all the guards of θ in terms of the initial state. We define path conditions symbolically as follows.

▶ **Definition 6** (Path condition). *Let θ be a trace over labels l. The path condition $path(\theta)$ is defined inductively:*

$$
\begin{aligned}
&path(\epsilon) = \texttt{True}\\
&path(\tau \cdot \theta) = path(\theta)\\
&path(\iota(e \rhd x := e') \cdot \theta) = e \wedge path(\theta)[e'/x]\\
&path(\iota(e \rhd \texttt{spawn}(\iota')) \cdot \theta) = e \wedge path(\theta)
\end{aligned}
$$

▶ **Example 7** (Path conditions). We compute the path conditions for the trace in Example 5.

$$
\begin{aligned}
path(\theta_0) =& path(\iota_0(\texttt{True} \rhd \texttt{spawn}(\iota_1)) \cdot \iota_0(\texttt{True} \rhd \texttt{spawn}(\iota_2))\\
&\cdot \iota_2(\texttt{flag} \rhd \texttt{x:=x+2}) \cdot \iota_2(\texttt{True} \rhd \texttt{flag:=}\neg\texttt{flag})\\
&\cdot \iota_2(\neg\texttt{flag} \rhd \texttt{x:=x-1}) \cdot \iota_2(\texttt{True} \rhd \texttt{flag:=}\neg\texttt{flag})\\
&\cdot \iota_2(\texttt{flag} \rhd \texttt{x:=x+2}) \cdot \iota_2(\texttt{True} \rhd \texttt{flag:=}\neg\texttt{flag})\\
&\cdot \iota_2(\neg\texttt{flag} \rhd \texttt{x:=x-1}) \cdot \iota_2(\texttt{True} \rhd \texttt{flag:=}\neg\texttt{flag})\\
&\cdot \iota_2(\texttt{True} \rhd \texttt{y:=y+1}) \cdot \iota_1(\texttt{x==0}\vee\texttt{flag} \rhd \texttt{x:=2*x}))\\
=& \texttt{flag} == \texttt{true}
\end{aligned}
$$

We can see how the computation produces the weakest precondition for the guards to hold; thus, any state in which the initial state of `flag` has value `true` allows the execution of θ_0.

For a symbolic trace θ, let $g(\theta)$ denote the corresponding guarded statement obtained simply by dropping the empty labels τ, removing the process identifiers ι (such that $e \rhd \texttt{spawn}(\iota)$ becomes $e \rhd \texttt{spawn(skip)}$), and connecting the resulting sequence of guard statements via sequential composition. We have the following basic property for path conditions.

▶ **Theorem 8** (Formal justification of path conditions). *For any symbolic trace θ and initial state σ, if $\sigma(path(\theta)) = \texttt{True}$ then there exists a state σ' such that $\sigma, g(\theta) \rightarrow^* \sigma'$.*

Proof. The proof proceeds by a straightforward induction on the length of θ (assuming that for the base case $g(\epsilon) = \texttt{skip}$). ◀

5 Partial Order Reduction for GCL

This section describes an algorithm which, from a given run, constructs all scheduling policies which respect the local flow of control of the individual processes. In order to reduce the search space we apply a partial order reduction based on a non-interference relation between the labels of the concolic operational semantics.

5.1 Symbolic Traces and Equivalence

We first define equivalence for the symbolic traces of the concolic semantics of GCL. Let vars(e) denote the program variables in an expression e. For a label l, we define the write- and read-sets, written as $W(l)$ and $R(l)$ respectively, as follows:

$$
\begin{aligned}
&W(\iota(e_1 \rhd x := e_2)) = \{x\} \qquad && R(\iota(e_1 \rhd x := e_2)) = \text{vars}(e_1) \cup \text{vars}(e_2)\\
&W(\iota(e \rhd \texttt{spawn}(\iota'))) = \emptyset && R(\iota(e \rhd \texttt{spawn}(\iota'))) = \text{vars}(e)
\end{aligned}
$$

Building on the general explanation of non-interference in Section 2, we can now define the non-interference relation $\sim$ between labels for GCL as follows:

▶ **Definition 9** (Non-interference). *For any two labels* $l_1 = \iota_1(e_1 \rhd s_1)$ *and* $l_2 = \iota_2(e_2 \rhd s_2)$ *we denote by* $l_1 \sim l_2$ *that*

$$
\begin{aligned}
&W(l_1) \cap (R(l_2) \cup W(l_2)) = \emptyset \wedge\ (W(l_1) \cup R(l_1)) \cap W(l_2) = \emptyset \\
&\wedge\ \iota_1 \neq \iota_2 \\
&\wedge\ s_1 \neq \texttt{spawn}(\iota_2) \wedge s_2 \neq \texttt{spawn}(\iota_1)
\end{aligned}
$$

This definition of non-interference captures both the non-interference of independent transitions and the must-happen-before relation for the semantics of GCL: two events of the same process must happen in program order, a process cannot be scheduled before it is created, and events with overlapping write- and read-sets cannot be reordered.

Recall from Section 2 that an equivalence relation between events can be extended to an equivalence relation on traces over those events. We extend the non-interference relation of Definition 9 to the smallest equivalence relation between symbolic traces such that $\theta' \cdot l_1 \cdot l_2 \cdot \theta'' \sim \theta' \cdot l_2 \cdot l_1 \cdot \theta''$, if $l_1 \sim l_2$.

▶ **Example 10** (Trace permutations and equivalence.). Consider the following traces, which are permutations of trace θ_0 from Example 5. In the traces, the changing position of the label $\iota_1(\texttt{x==0}\vee\texttt{flag} \rhd \texttt{x:=2*x})$ is highlighted.

$$
\begin{aligned}
\theta_1 :\quad & \iota_0(\texttt{True} \rhd \texttt{spawn}(\iota_1)) \cdot \iota_0(\texttt{True} \rhd \texttt{spawn}(\iota_2)) \\
& \cdot \iota_2(\texttt{flag} \rhd \texttt{x:=x+2}) \cdot \iota_2(\texttt{True} \rhd \texttt{flag:=}\neg\texttt{flag}) \\
& \cdot \iota_2(\neg\texttt{flag} \rhd \texttt{x:=x-1}) \cdot \iota_2(\texttt{True} \rhd \texttt{flag:=}\neg\texttt{flag}) \\
& \cdot \iota_2(\texttt{flag} \rhd \texttt{x:=x+2}) \cdot \iota_2(\texttt{True} \rhd \texttt{flag:=}\neg\texttt{flag}) \\
& \cdot \iota_2(\neg\texttt{flag} \rhd \texttt{x:=x-1}) \cdot \iota_2(\texttt{True} \rhd \texttt{flag:=}\neg\texttt{flag}) \\
& \cdot \boxed{\iota_1(\texttt{x==0}\vee\texttt{flag} \rhd \texttt{x:=2*x})} \cdot \iota_2(\texttt{True} \rhd \texttt{y:=y+1})
\end{aligned}
$$

$$
\begin{aligned}
\theta_2 :\quad & \iota_0(\texttt{True} \rhd \texttt{spawn}(\iota_1)) \cdot \iota_0(\texttt{True} \rhd \texttt{spawn}(\iota_2)) \\
& \cdot \iota_2(\texttt{flag} \rhd \texttt{x:=x+2}) \cdot \iota_2(\texttt{True} \rhd \texttt{flag:=}\neg\texttt{flag}) \\
& \cdot \iota_2(\neg\texttt{flag} \rhd \texttt{x:=x-1}) \cdot \iota_2(\texttt{True} \rhd \texttt{flag:=}\neg\texttt{flag}) \\
& \cdot \iota_2(\texttt{flag} \rhd \texttt{x:=x+2}) \cdot \iota_2(\texttt{True} \rhd \texttt{flag:=}\neg\texttt{flag}) \\
& \cdot \iota_2(\neg\texttt{flag} \rhd \texttt{x:=x-1}) \cdot \boxed{\iota_1(\texttt{x==0}\vee\texttt{flag} \rhd \texttt{x:=2*x})} \\
& \cdot \iota_2(\texttt{True} \rhd \texttt{flag:=}\neg\texttt{flag}) \cdot \iota_2(\texttt{True} \rhd \texttt{y:=y+1})
\end{aligned}
$$

$$
\begin{aligned}
\theta_3 :\quad & \iota_0(\texttt{True} \rhd \texttt{spawn}(\iota_1)) \cdot \iota_0(\texttt{True} \rhd \texttt{spawn}(\iota_2)) \\
& \cdot \iota_2(\texttt{flag} \rhd \texttt{x:=x+2}) \cdot \iota_2(\texttt{True} \rhd \texttt{flag:=}\neg\texttt{flag}) \\
& \cdot \iota_2(\neg\texttt{flag} \rhd \texttt{x:=x-1}) \cdot \iota_2(\texttt{True} \rhd \texttt{flag:=}\neg\texttt{flag}) \\
& \cdot \iota_2(\texttt{flag} \rhd \texttt{x:=x+2}) \cdot \boxed{\iota_1(\texttt{x==0}\vee\texttt{flag} \rhd \texttt{x:=2*x})} \\
& \cdot \iota_2(\texttt{True} \rhd \texttt{flag:=}\neg\texttt{flag}) \cdot \iota_2(\neg\texttt{flag} \rhd \texttt{x:=x-1}) \\
& \cdot \iota_2(\texttt{True} \rhd \texttt{flag:=}\neg\texttt{flag}) \cdot \iota_2(\texttt{True} \rhd \texttt{y:=y+1})
\end{aligned}
$$

Trace θ_1 is in the same equivalence class as θ_0, denoted by $\theta_1 \in [\theta_0]$, since $\iota_1(\texttt{x==0}\vee\texttt{flag} \rhd \texttt{x:=2*x}) \sim \iota_2(\texttt{True} \rhd \texttt{y:=y+1})$. However, $\theta_2 \notin [\theta_0]$ since $\iota_1(\texttt{x==0}\vee\texttt{flag} \rhd \texttt{x:=2*x}) \nsim \iota_2(\texttt{True} \rhd \texttt{flag:=}\neg\texttt{flag})$, and similarly $\theta_3 \notin [\theta_0]$, $\theta_3 \notin [\theta_2]$, etc.

▶ **Example 11** (Path conditions continued). We compute path conditions for the traces from Example 10. By computing $path(\theta_2)$ we get the constraint

$$\begin{aligned} path(\theta_2) = \quad & path(\iota_0(\texttt{True} \rhd \texttt{spawn}(\iota_1)) \cdot \iota_0(\texttt{True} \rhd \texttt{spawn}(\iota_2)) \\ & \cdot \iota_2(\texttt{flag} \rhd \texttt{x:=x+2}) \cdot \iota_2(\texttt{True} \rhd \texttt{flag:=}\neg\texttt{flag}) \\ & \cdot \iota_2(\neg\texttt{flag} \rhd \texttt{x:=x-1}) \cdot \iota_2(\texttt{True} \rhd \texttt{flag:=}\neg\texttt{flag}) \\ & \cdot \iota_2(\texttt{flag} \rhd \texttt{x:=x+2}) \cdot \iota_2(\texttt{True} \rhd \texttt{flag:=}\neg\texttt{flag}) \\ & \cdot \iota_2(\neg\texttt{flag} \rhd \texttt{x:=x-1}) \cdot \iota_1(\texttt{x==0}\vee\texttt{flag} \rhd \texttt{x:=2*x}) \\ & \cdot \iota_2(\texttt{True} \rhd \texttt{flag:=}\neg\texttt{flag}) \cdot \iota_2(\texttt{True} \rhd \texttt{y:=y+1})) \\ = \quad & \texttt{flag} == \texttt{true} \wedge \texttt{x} == \texttt{-2} \end{aligned}$$

Thus, θ_2 is not executable from the initial program state of Example 5, but it would be executable from states satisfying this constraint (i.e., for initial states in which the value of `flag` is true and the value of `x` is -2). By computing $path(\theta_3)$, we see that θ_3 is executable from the initial state of Example 5, since its path condition reduces to `flag` == `true`. Observe that although θ_0 and θ_3 have the same path condition, their final states differ. The final state for θ_3 when executed from the initial program state of Example 5 will be `x`$\mapsto$`5, y`$\mapsto$`1, flag`$\mapsto$`true`.

To easily decide whether two traces are in the same equivalence class, we define a *canonical representation* for the traces of GCL executions. In general, a lexicographic ordering on traces can be used to select the smallest in a set of traces, assuming a total order on the elements of the traces. Traces in the same equivalence class differ only in the ordering of adjacent, commuting labels [35]. Hence, a partial order on labels that commute will suffice to define the canonical representative for the traces in an equivalence class. This partial order can for example be expressed by lifting a strict total order on the process identifiers, since commuting events must belong to different processes.

▶ **Definition 12** (Canonical representatives for GCL traces). *Assume a strict total order $<$ on process identifiers $\iota_0, \iota_1, \iota_2, \ldots$ For any labels l_1 and l_2 with process identifiers ι_1 and ι_2, respectively, let $l_1 < l_2$ if and only if $\iota_1 < \iota_2$. The canonical representative of a trace θ, denoted $canon(\theta)$, is defined inductively as follows:*

$$\begin{aligned} & canon(\varepsilon) = \varepsilon \\ & canon(\varepsilon \cdot l) = \varepsilon \cdot l \\ & canon(\theta \cdot l_1 \cdot l_2) = canon(\theta \cdot l_1) \cdot l_2 && \textit{if } l_1 \not\sim l_2 \textit{ or } l_1 < l_2 \\ & canon(\theta \cdot l_1 \cdot l_2) = canon(canon(\theta \cdot l_2) \cdot l_1) && \textit{if } l_1 \sim l_2 \textit{ and } l_2 < l_1 \end{aligned}$$

If we consider the traces from Examples 5 and 10 and a strict total order $\iota_0 < \iota_1 < \iota_2$ on process identifiers, we can observe that $canon(\theta_0) = \theta_1$.

The set of processes in a runtime state of a GCL program can be derived from the trace leading to that state. We define a function which, for a given trace, returns the set of process identifiers for these processes.

▶ **Definition 13** (Active processes). *Let θ be a symbolic trace representing the execution of a GCL program. The set of active process identifiers $proc(\theta)$ is defined inductively as follows:*

$$\begin{aligned} & proc(\epsilon) = \{\iota_0\} \\ & proc(\theta' \cdot e \rhd \texttt{spawn}(\iota)) = proc(\theta') \cup \{\iota\} \\ & proc(\theta' \cdot l) = proc(\theta') \textit{ for } l \in \{\tau, \iota(e \rhd x := e')\} \end{aligned}$$

This definition is easily justified: If θ is a trace of $\sigma, \{\iota_0(g)\} \to^* \sigma', P$, then $P = proc(\theta)$. The proof is straightforward by induction over the length of θ.

We can now formalize what it means for a symbolic trace to be executable. Let $\leq$ denote the prefix relation on symbolic traces. and $\theta \downarrow_\iota$ the projection of a symbolic trace θ unto all labels of of process identifier ι (i.e., labels of the form $\iota(\ldots)$). For a symbolic trace θ, let $length(\theta)$ denote the length of θ and θ_n the initial prefix of θ of length n (i.e., $\theta_n \leq \theta$ and $length(\theta_n) = n$).

▶ **Definition 14** (Trace executability). *Let g be a GCL program, σ a state, l a label with process identifier ι, and assume that θ is a permutation of a symbolic trace of the execution $\sigma, \{\iota_0(g)\} \to^* \sigma', P$. The trace θ is* executable *in σ if $\sigma(path(\theta)) = \mathtt{True}$ and $\iota \in proc(\theta_{n-1})$ for all $n \leq length(\theta)$, where $\theta_n = \theta_{n-1} \cdot l$.*

Trace executability allows us to strengthen Theorem 8 by expressing that the source program can produce executable permutations of one of its traces. We say that a permutation θ' of a symbolic trace θ *preserves local order* if $\forall \iota \in proc(\theta) : \theta' \downarrow_\iota = \theta \downarrow_\iota$.

▶ **Theorem 15** (Soundness). *For any GCL program g and state σ, such that θ is a symbolic trace of $\sigma, \{\iota_0(g)\} \to^* \sigma', P$ and let θ' be a permutation of θ which preserves local order. If θ' is executable in σ then there exists a σ'' such that θ' is a symbolic trace of $\sigma, \{\iota_0(g)\} \to^* \sigma'', P$.*

Proof. The proof proceeds by induction on the length of θ (assuming that for the base case $g(\epsilon) = \mathtt{skip}$). ◀

5.2 The Fagiani Algorithm

We now present an algorithm[1] which, given an initial state σ and an initial symbolic trace θ generated by a successfully terminating concolic execution in the above labeled transition system, constructs a set I of all the canonical representatives of permutations of θ which are executable in σ; i.e., the algorithm generates one representative for each equivalence class of the traces which are permutations of θ. For simplicity, we assume that θ is in canonical form. Considering our running example, if we start the algorithm with state σ and trace $canon(\theta_0)$ from Example 5, the algorithm will compute traces such as θ_1, θ_2 and θ_3 from Example 10. The algorithm will detect that $canon(\theta_1) = canon(\theta_0)$ so $canon(\theta_1)$ can be discarded, that $canon(\theta_2)$ is not executable from the initial state σ of Example 5 and can therefore discarded, and that $canon(\theta_3) \neq canon(\theta_0)$ and therefore $canon(\theta_3)$ is added to I.

The algorithm is presented in the form of pseudo code in Algorithm 1. In the pseudo code we abstract from the data structures representing states and traces. Let θ' be a symbolic trace such that for any process ι its local computation in θ' is a prefix of its local computation in the initial symbolic trace θ. We then denote by $next(\iota, \theta')$ the next instruction of process ι as defined by θ. Formally, the next instruction of a process can be defined as follows:

▶ **Definition 16** (Process-local next). *Let θ and θ' be symbolic traces, $\iota \in proc(\theta)$, and assume that $\theta' \downarrow_\iota \leq \theta \downarrow_\iota$. The next ι-event after θ' is defined as follows:*

$$next(\iota, \theta') = l \text{ if } (\theta' \cdot l) \downarrow_\iota \leq \theta \downarrow_\iota$$
$$next(\iota, \theta') = nil \text{ if } \theta' \downarrow_\iota = \theta \downarrow_\iota$$

We use the process-local next to ensure that new traces preserve local order. In the code we also assume the inductive definitions of the path condition $path(\theta)$ (see Definition 6), the canonical representative $canon(\theta)$ (see Definition 12) and the active processes $proc(\theta)$

[1] A prototype implementation of the concolic semantics of GCL and the Fagiani algorithm is available at https://github.com/larstvei/GCL.

(see Definition 13) of a symbolic trace θ, and the inductive definition of the value $\sigma(e)$ of expression e in a state σ. As before, $\mathit{length}(\theta)$ denotes the length of the trace θ, i.e., the number of instructions it contains, and ϵ the empty trace.

The algorithm itself iterates over the length of the initial trace θ in canonical form. Each such iteration in turn iterates over all the traces currently stored in I. Since this inner iteration updates the set I, we "freeze" the initial value of I upon each iteration of the outer for-loop. The set I will then store the newly updated canonical traces.

Algorithm 1 The Fagiani Algorithm.

```
Input: θ0: a global symbolic trace in its canonical form
Input: σ: an initial state
Auxiliaries: path, canon, proc, next: see Definitions 6, 12, 13, 16
Result: I: A set of executable permutations of θ0
I := {ε};
for i := 1 to length(θ0) do
    I' := I;
    I := {};
    foreach θ' ∈ I' do
        foreach ι ∈ proc(θ') do
            if next(ι, θ') ≠ nil then
                θ'' := canon(θ' · next(ι, θ'));
                if θ'' ∉ I ∧ σ(path(θ'')) then I := I ∪ {θ''};
            end
        end
    end
end
```

Formally, the computed set I of permutations, which respect the local order of the input trace θ, satisfies

$$I = \{\mathit{canon}(\theta') \mid \sigma(\mathit{path}(\theta')) \wedge \forall \iota \in \mathit{proc}(\theta') : \theta' \downarrow_\iota = \theta \downarrow_\iota\}.$$

To prove this, it suffices to show that after the i'th iteration, $i = 1, \ldots, \mathit{length}(\theta)$, I satisfies

$$I = \{\mathit{canon}(\theta') \mid \mathit{length}(\theta') = i \wedge \sigma(\mathit{path}(\theta')) \wedge \forall \iota \in \mathit{proc}(\theta') : \theta' \downarrow_\iota \leq \theta \downarrow_\iota\}.$$

The computed set I forms the basis for a set of test cases. We use this set for testing whether these different traces have an observable effect on the state. It is easy to see that the set I consists of executable traces, such that Theorem 15 applies. For each trace the corresponding test case simply consists of the underlying scheduling policy, which is represented by a sequence of process identifiers. The execution of such a test case then consists of an execution from the given initial state σ following the specified scheduling policy, and checking the final state.

6 Language Extensions

This section presents some conservative extensions to the language: nested guarded statements, a looping construct, named procedures, and local scopes for variables.

Nesting

Guarded statements may be nested without adding any particular complexity to the calculus. Let us consider statements with the syntax $e \rhd g$ and $g_1 \lhd e \rhd g_2$. The guarded skip $e \rhd \texttt{skip}$, which corresponds to an assert-statement, needs an additional label of the form $\iota(e \rhd \texttt{skip})$. With this extension, it has the obvious semantics of Skip provided that the guard e holds in the current state, and with the above label. We get the following rules in the semantics:

$$\text{(NestedGuard)}\quad \frac{\sigma, \{\iota((e_1 \wedge e_2) \rhd g; g')\} \xrightarrow{l} \sigma', P}{\sigma, \{\iota(e_1 \rhd (e_2 \rhd g); g')\} \xrightarrow{l} \sigma', P}$$

$$\text{(NestedChoice1)}\quad \frac{\sigma, \{\iota((e_1 \wedge e_2) \rhd g_1; g)\} \xrightarrow{l} \sigma', P}{\sigma, \{\iota(e_1 \rhd (g_1 \lhd e_2 \rhd g_2); g)\} \xrightarrow{l} \sigma', P}$$

$$\text{(NestedChoice2)}\quad \frac{\sigma, \{\iota(\neg(e_1 \wedge e_2) \rhd g_2; g)\} \xrightarrow{l} \sigma', P}{\sigma, \{\iota(e_1 \rhd (g_1 \lhd e_2 \rhd g_2); g)\} \xrightarrow{l} \sigma', P}$$

Loops

We can add a loop construct $e \rhd^* s$ to repeat a guarded statement zero or more times, which can be captured by the following transition rules:

$$\text{(While1)}\quad \frac{\sigma, \{\iota(e \rhd s; e \rhd^* s; g)\} \xrightarrow{l} \sigma', P}{\sigma, \{\iota(e \rhd^* s; g)\} \xrightarrow{l} \sigma', P}$$

$$\text{(While2)}\quad \frac{\neg\sigma(e)}{\sigma, \{\iota(e \rhd^* s; g)\} \xrightarrow{\epsilon} \sigma', \{\iota(g)\}}$$

Note that such a loop construct would require a straightforward generalization of our concolic testing theory to non-terminating computations by imposing a bound on the length of the computations.

Procedures

It is also straightforward to spawn new processes by procedure calls: We can add procedure definitions $\texttt{proc}\ p\{\,g\,\}$ and add syntax $p()$ for procedure calls to the statements. If we assume given a mapping PT from procedure names p to procedure bodies g, it suffices to add the following rule to the semantics:

$$\text{(Proc)}\quad \frac{PT(p) = g' \qquad \sigma, \{\iota(e \rhd \texttt{spawn}(g'); g)\} \xrightarrow{l} \sigma', P}{\sigma, \{\iota(e \rhd p(); g)\} \xrightarrow{l} \sigma', P}$$

To model procedure calls locally in the context of a single process can be described by inlining the procedure body.

Local scopes

Local scopes $\{\sigma', g\}$ can also be added as guarded statements. Here, we are not interested in seeing the local variables in the labels, because they are private and do not affect other processes. For this reason, we remove local variables from the labels by applying the local

substitution *before* we (in most cases) reuse rules without local scope to create the labels. We need additional rules for scoped assignment and spawning, for leaving an empty scope and we assume that an inner scope takes precedence over an outer scope to unfold nested scopes.

$$\textsc{(LeaveScope)}\quad \sigma, \{\iota(\{\sigma', \texttt{skip}\}; g)\} \xrightarrow{\epsilon} \sigma, \{\iota(g)\}$$

$$\textsc{(ScopedAssign1)}\quad \frac{x \in \mathrm{dom}(\sigma) \qquad \sigma, \{\iota(\sigma'(e) \rhd x := \sigma'(e'); g)\} \xrightarrow{l} \sigma'', \{\iota(g)\}}{\sigma, \{\iota(\{\sigma', e \rhd x := e'; g\}; g'')\} \xrightarrow{l} \sigma'', \{\iota(\{\sigma', g\}; g')\}}$$

$$\textsc{(ScopedSpawn)}\quad \frac{P' = P \cup \{\iota(\{\sigma', g'\}; g'')\} \qquad \sigma, \{\iota(\sigma'(e) \rhd \texttt{spawn}(g); \texttt{skip})\} \xrightarrow{l} \sigma, P}{\sigma, \{\iota(\{\sigma', e \rhd \texttt{spawn}(g); g'\}; g'')\} \xrightarrow{l} \sigma, P'}$$

$$\textsc{(ScopedAssign2)}\quad \frac{x \notin \mathrm{dom}(\sigma) \qquad e'' = (e \wedge (e' == e')) \qquad \sigma', \{\iota(\sigma(e) \rhd x := \sigma(e'); g)\} \xrightarrow{l} \sigma'', \{\iota(g)\}}{\sigma, \{\iota(\{\sigma', e \rhd x := e'; g\}; g')\} \xrightarrow{\iota(\sigma'(e'') \rhd \texttt{skip})} \sigma, \{\iota(\{\sigma'', g\}; g')\}}$$

$$\textsc{(UnfoldScope)}\quad \frac{\sigma, \{\iota(\{\sigma'', g\}; \{\sigma', g'\}; g'')\} \xrightarrow{l} \sigma''', P}{\sigma, \{\iota(\{\sigma', \{\sigma'', g\}; g'\}; g'')\} \xrightarrow{l} \sigma''', P\}}$$

7 Related Work

Parallel and distributed systems are difficult to analyze because of their inherent non-determinism. Both testing and formal verification have their limitations for these systems. Model checking, which can be situated between testing and formal verification, here suffers from state explosion [19]; in practice, model checking relies on analyzing models with a tractable state space. Software model checking techniques either adapt model checking into techniques for systematic testing of programs (e.g., [4, 18, 30, 44]) or abstract programs into models for which traditional model checking techniques apply (e.g., [8, 20, 32, 36, 38]). Our work fits into the former category.

Stateless model checking avoids state space explosion by exploring the executions of a program without explicitly storing all the program states [29], and has been implemented in tools including VeriSoft [30] and CHESS [37]. Stateless model checking can be realized by combining a runtime scheduler which controls the program execution with an algorithm which explores the different ways in which processes can be scheduled. The combinatorial explosion of different executions for parallel programs can be reduced by means of partial order reduction (POR) [19, 29, 39], which introduces an equivalence relation on executions based on Mazurkiewicz traces [35]. POR explores at least one execution in each equivalence class. Ideally, only one trace of each equivalence class needs to be explored; the precision (i.e., performance) of a particular algorithm depends on the number of execution paths visited in each equivalence class. Dynamic partial order reduction (DPOR) [1, 27, 34, 39, 40, 43] makes POR more precise by detecting and exploiting interference dynamically. DPOR assumes access to the scheduler and state of the runtime system, both to guide execution and to decide whether an action is enabled. Our work uses partial order reduction, but it does not need access to the scheduler of the runtime system.

Actor systems [2] are well-suited for systematic testing using POR because their inherent isolation of local state limits the number of races in a program. TransDPOR [40] extends DPOR to explore that the dependency relation of actor systems is transitive. SYCO [4, 5] is a testing tool for actor-based concurrency which combines the transitivity exploited by

TransDPOR with a dependency relation based on process interference [6], similar to the non-interference relation discussed in Section 2, and to consider synchronization primitives as found in active object languages [10], such as ABS [33]; i.e., they handle await-statements which synchronize on the resolution of futures. ContextDPOR [3] introduces a context-sensitive notion of non-interference between events. This is achieved by deciding on the equivalence between subsequences of the traces and the next action (resulting in so called sleep sequences). Technically, this is done by storing the state resulting from the one trace together with the sleep sequence. In contrast, our motivation for studying GCL stems from the problem of swapping events in the traces with intra-actor synchronization based on non-monotonic Boolean await-statements (in contrast to futures, which keep an enabled state after reaching it). Similar to ContextDPOR, we had to go beyond the read- and write-sets traditionally used to determine interference in order to decide at the level of traces whether the permutation of an observed trace is executable. In contrast to ContextDPOR, our work is based on a weakest-prefix calculation over symbolic traces to decide on their executability. The relationship between concrete and symbolic execution with partial-order reduction has previously been studied by the authors in [21]; that previous work focussed on soundness and correctness of the symbolic execution framework but not on weakest-precondition computation for executability as we address in this paper.

A major limitation of DPOR algorithms is that they are implemented inside the runtime system of the language. We are currently developing ExoDPOR, a stateless model checker for ABS which is implemented outside the runtime system, such that it can perform parallel stateless model checking by exogenously coordinating the runs of a number of instances of the runtime system [41]. This is enabled by extending the backend of ABS with a trace record and replay mechanism [42] and manipulating traces directly to trigger new runs. Whereas ExoDPOR can handle most of ABS (including deployment components and real-time behavior), it does not yet handle await statements with Boolean conditions (a non-monotonic guard statement). We expect the work in this paper to provide a basis to address this currently missing piece in our tool.

8 Conclusion

This paper presented a method for testing the deterministic behavior of dynamically spawned processes executing on a shared state. We have developed an algorithm which, starting from the symbolic trace of an initial run of a program, generates all traces which may result in different final states. Each trace represents an execution with a different local scheduling of the program's processes, but the traces may result in the same final state because the non-interference relation is an approximation. Therefore, the generated traces need to be tested to determine whether the program outcome is independent of the local scheduling decisions. We rely on recording traces of executions, partial order reduction to eliminate traces which are obviously equivalent to previously generated traces, and weakest precondition calculation to eliminate infeasible (non-executable) traces. The weakest precondition calculation allows the proposed method to handle the non-monotonic enabledness conditions of guarded commands without explicitly computing the different states of the program.

This proposed method can be extended in a straightforward way to generate "seed traces", executable trace prefixes that lead to different end states than the one in the original recorded run. To formulate seed traces, the concolic operational semantics of Section 4 can simply be extended by a new label type that records the conditional in the choice expression. These seed traces can be used to implement stateless model checking for parallel systems given a controllable scheduler. We plan to implement this method as an extension of ExoDPOR [41], our stateless model checker for the active object language ABS.

References

1. Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Source sets: A foundation for optimal dynamic partial order reduction. *J. ACM*, 64(4):25:1–25:49, 2017. doi:10.1145/3073408.
2. Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, 1986.
3. Elvira Albert, Puri Arenas, Maria Garcia de la Banda, Miguel Gómez-Zamalloa, and Peter J. Stuckey. Context-sensitive dynamic partial order reduction. In Rupak Majumdar and Viktor Kuncak, editors, *Proc. 29th International Conference on Computer Aided Verification (CAV 2017), Part I*, volume 10426 of *Lecture Notes in Computer Science*, pages 526–543. Springer, 2017. doi:10.1007/978-3-319-63387-9_26.
4. Elvira Albert, Puri Arenas, and Miguel Gómez-Zamalloa. Systematic testing of actor systems. *Softw. Test. Verification Reliab.*, 28(3), 2018. doi:10.1002/stvr.1661.
5. Elvira Albert, Miguel Gómez-Zamalloa, and Miguel Isabel. SYCO: a systematic testing tool for concurrent objects. In Ayal Zaks and Manuel V. Hermenegildo, editors, *Proc. 25th International Conference on Compiler Construction (CC 2016)*, pages 269–270. ACM, 2016. doi:10.1145/2892208.2892236.
6. Gregory R. Andrews. *Concurrent programming - principles and practice*. Benjamin/Cummings, 1991.
7. Krzysztof R. Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. Texts in Computer Science. Springer, 3 edition, 2009. doi:10.1007/978-1-84882-745-5.
8. Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A decade of software model checking with SLAM. *Commun. ACM*, 54(7):68–76, 2011. doi:10.1145/1965724.1965743.
9. Nikolaos Bezirgiannis, Frank S. de Boer, Einar Broch Johnsen, Ka I Pun, and Silvia Lizeth Tapia Tarifa. Implementing SOS with active objects: A case study of a multicore memory system. In *Proc. 22nd Intl. Conf. on Fundamental Approaches to Software Engineering (FASE 2019)*, volume 11424 of *Lecture Notes in Computer Science*, pages 332–350. Springer, 2019. doi:10.1007/978-3-030-16722-6_20.
10. Frank De Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. A survey of active object languages. *ACM Comput. Surv.*, 50(5):76:1–76:39, October 2017. doi:10.1145/3122848.
11. Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I. Pun, S. Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. Parallel objects for multicores: A glimpse at the parallel language Encore. In Marco Bernardo and Einar Broch Johnsen, editors, *Formal Methods for Multicore Programming*, volume 9104 of *Lecture Notes in Computer Science*, pages 1–56. Springer, 2015. doi:10.1007/978-3-319-18941-3_1.
12. Nadia Busi, Maurizio Gabbrielli, and Gianluigi Zavattaro. Replication vs. recursive definitions in channel based calculi. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP 2003)*, volume 2719 of *Lecture Notes in Computer Science*, pages 133–144. Springer, 2003. doi:10.1007/978-3-540-27836-8_28.
13. Nadia Busi, Maurizio Gabbrielli, and Gianluigi Zavattaro. Comparing recursion, replication, and iteration in process calculi. In *Proc. 31st International Colloquium on Automata, Languages and Programming (ICALP 2004)*, volume 3142 of *Lecture Notes in Computer Science*, pages 307–319. Springer, 2004. doi:10.1007/978-3-540-27836-8_28.
14. Nadia Busi, Maurizio Gabbrielli, and Gianluigi Zavattaro. On the expressive power of recursion, replication and iteration in process calculi. *Math. Struct. Comput. Sci.*, 19(6):1191–1222, 2009. doi:10.1017/S096012950999017X.
15. Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *Proc. 13th ACM Conf. on Computer and Communications Security (CCS'06)*, pages 322–335. ACM, 2006. doi:10.1145/1180405.1180445.

16 Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013. `doi:10.1145/2408776.2408795`.

17 Denis Caromel, Ludovic Henrio, and Bernard Serpette. Asynchronous and deterministic objects. In *Proc. 31st Symposium on Principles of Programming Languages (POPL 2004)*, pages 123–134. ACM Press, 2004. `doi:10.1145/964001.964012`.

18 Maria Christakis. *Narrowing the Gap between Verification and Systematic Testing*. PhD thesis, ETH Zurich, 2015.

19 Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 2001. `doi:10.3233/978-1-60750-711-6-260`.

20 James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In Carlo Ghezzi, Mehdi Jazayeri, and Alexander L. Wolf, editors, *Proc. 22nd International Conference on Software Engineering (ICSE 2000)*, pages 439–448. ACM, 2000. `doi:10.1145/337180.337234`.

21 Frank S. de Boer, Marcello Bonsangue, Einar Broch Johnsen, Ka I Pun, Silvia Lizeth Tapia Tarifa, and Lars Tveito. SymPaths: Symbolic execution meets partial order reduction. In Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, and Mattias Ulbrich, editors, *Deductive Verification - The Next 70 Years*, volume 12345 of *Lecture Notes in Computer Science*. Springer, 2020. To appear.

22 Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2007. `doi:10.1007/978-3-540-71316-6_22`.

23 Frank S. de Boer and Hans-Dieter A. Hiep. Axiomatic characterization of trace reachability for concurrent objects. In Wolfgang Ahrendt and Silvia Lizeth Tapia Tarifa, editors, *Proc. 15th Intl. Conf. on Integrated Formal Methods (IFM 2019)*, volume 11918 of *Lecture Notes in Computer Science*, pages 157–174. Springer, 2019. `doi:10.1007/978-3-030-34968-4_9`.

24 Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975. `doi:10.1145/360933.360975`.

25 Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

26 Kiko Fernandez-Reyes, Dave Clarke, Ludovic Henrio, Einar Broch Johnsen, and Tobias Wrigstad. Godot: All the benefits of implicit and explicit futures. In Alastair F. Donaldson, editor, *Proc. 33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, volume 134 of *LIPIcs*, pages 2:1–2:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. `doi:10.4230/LIPIcs.ECOOP.2019.2`.

27 Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In Jens Palsberg and Martín Abadi, editors, *Proc. 32nd Symp. on Principles of Programming Languages (POPL 2005)*, pages 110–121. ACM, 2005. `doi:10.1145/1040305.1040315`.

28 Maurizio Gabbrielli and Simone Martini. *Programming Languages: Principles and Paradigms*. Springer, 2010. `doi:10.1007/978-1-84882-914-5`.

29 Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996. `doi:10.1007/3-540-60761-7`.

30 Patrice Godefroid. Model checking for programming languages using Verisoft. In Peter Lee, Fritz Henglein, and Neil D. Jones, editors, *Proc. 24th Symp. on Principles of Programming Languages (POPL 1997)*, pages 174–186. ACM, 1997. `doi:10.1145/263699.263717`.

31 Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In Vivek Sarkar and Mary W. Hall, editors, *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, pages 213–223. ACM, 2005. `doi:10.1145/1065010.1065036`.

32 Gerard J. Holzmann and Margaret H. Smith. A practical method for verifying event-driven software. In Barry W. Boehm, David Garlan, and Jeff Kramer, editors, *Proc. International Conference on Software Engineering (ICSE 1999)*, pages 597–607. ACM, 1999. doi:10.1145/302405.302710.

33 Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2011. doi:10.1007/978-3-642-25271-6_8.

34 Shmuel Katz and Doron A. Peled. An efficient verification method for parallel and distributed programs. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 489–507. Springer, 1988. doi:10.1007/BFb0013032.

35 Antoni W. Mazurkiewicz. Trace theory. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Advances in Petri Nets 1986*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer, 1987. doi:10.1007/3-540-17906-2_30.

36 Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A pragmatic approach to model checking real code. In David E. Culler and Peter Druschel, editors, *Proc. 5th Symposium on Operating System Design and Implementation (OSDI 2002)*. USENIX Association, 2002. URL: http://www.usenix.org/events/osdi02/tech/musuvathi.html.

37 Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In Richard Draves and Robbert van Renesse, editors, *Proc. 8th Symposium on Operating Systems Design and Implementation (OSDI 2008)*, pages 267–280. USENIX Association, 2008. URL: http://www.usenix.org/events/osdi08/tech/full_papers/musuvathi/musuvathi.pdf.

38 Kedar S. Namjoshi and Robert P. Kurshan. Syntactic program transformations for automatic abstraction. In E. Allen Emerson and A. Prasad Sistla, editors, *Proc. 12th International Conference on Computer Aided Verification (CAV 2000)*, volume 1855 of *Lecture Notes in Computer Science*, pages 435–449. Springer, 2000. doi:10.1007/10722167_33.

39 Doron A. Peled. All from one, one for all: on model checking using representatives. In Costas Courcoubetis, editor, *Proc. 5th International Conference on Computer Aided Verification (CAV'93)*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer, 1993. doi:10.1007/3-540-56922-7_34.

40 Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. TransDPOR: A novel dynamic partial-order reduction technique for testing actor programs. In Holger Giese and Grigore Rosu, editors, *Proc. Formal Techniques for Distributed Systems (FORTE/FMOODS 2012)*, volume 7273 of *Lecture Notes in Computer Science*, pages 219–234. Springer, 2012. doi:10.1007/978-3-642-30793-5_14.

41 Lars Tveito, Einar Broch Johnsen, and Rudolf Schlatte. ExoDPOR: Exogenous stateless model checking. Submitted for publication, 2020.

42 Lars Tveito, Einar Broch Johnsen, and Rudolf Schlatte. Global reproducibility through local control for distributed active objects. In Heike Wehrheim and Jordi Cabot, editors, *Proc. 23rd International Conference on Fundamental Approaches to Software Engineering (FASE 2020)*, volume 12076 of *Lecture Notes in Computer Science*, pages 140–160. Springer, 2020. doi:10.1007/978-3-030-45234-6_7.

43 Antti Valmari. Stubborn sets for reduced state space generation. In Grzegorz Rozenberg, editor, *Proc. 10th International Conference on Applications and Theory of Petri Nets*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer, 1989. doi:10.1007/3-540-53863-1_36.

44 Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003. doi:10.1023/A:1022920129859.

Derivation of Constraints from Machine Learning Models and Applications to Security and Privacy

Moreno Falaschi

Department of Information Engineering and Mathematics, University of Siena, Italy
https://www3.diism.unisi.it/people/person.php?id=485
moreno.falaschi@unisi.it

Catuscia Palamidessi

Inria, Palaiseau, France
LIX, Ecole Polytechnique, Institut Polytechnique de Paris, Palaiseau, France
https://www.lix.polytechnique.fr/~catuscia/
catuscia@lix.polytechnique.fr

Marco Romanelli

Inria, Palaiseau, France
LIX, Ecole Polytechnique, Institut Polytechnique de Paris, Palaiseau, France
University of Siena, Italy
http://www.lix.polytechnique.fr/Labo/Marco.Romanelli/
marcoromane@gmail.com

Abstract

This paper shows how we can combine the power of machine learning with the flexibility of constraints. More specifically, we show how machine learning models can be represented by first-order logic theories, and how to derive these theories. The advantage of this representation is that it can be augmented with additional formulae, representing constraints of some kind on the data domain. For instance, new knowledge, or potential attackers, or fairness desiderata. We consider various kinds of learning algorithms (neural networks, k-nearest-neighbours, decision trees, support vector machines) and for each of them we show how to infer the FOL formulae. Then we focus on one particular application domain, namely the field of security and privacy. The idea is to represent the potentialities and goals of the attacker as a set of constraints, then use a constraint solver (more precisely, a solver modulo theories) to verify the satisfiability. If a solution exists, then it means that an attack is possible, otherwise, the system is safe. We show various examples from different areas of security and privacy; specifically, we consider a side-channel attack on a password checker, a malware attack on smart health systems, and a model-inversion attack on a neural network.

2012 ACM Subject Classification Computing methodologies → Learning paradigms; Computing methodologies → Symbolic and algebraic manipulation; Security and privacy → Formal security models; Security and privacy → Privacy-preserving protocols; Security and privacy → Information flow control

Keywords and phrases Constraints, machine learning, privacy, security

Digital Object Identifier 10.4230/OASIcs.Gabbrielli.2020.11

Funding This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme. Grant agreement № 835294.

Acknowledgements We thank the anonymous reviewers for their detailed suggestions and comments that helped us to improve the presentation of our paper.

1 Introduction

Machine learning (ML) is pervasive in nowadays society: systems based on this technology run in hospitals to help diagnose diseases, in cars to help avoid car accidents, in banks to evaluate loans and manage investments, at insurance agencies to evaluate coverage suitability and costs for clients.

© Moreno Falaschi, Catuscia Palamidessi, and Marco Romanelli;
licensed under Creative Commons License CC-BY

Recent Developments in the Design and Implementation of Programming Languages.
Editors: Frank S. de Boer and Jacopo Mauro; Article No. 11; pp. 11:1–11:20

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Machine learning refers to the automated pattern detection from data and it is is strictly linked to the idea of "generalized learning" as opposed to "memorization learning". The latter is based on storing information in the shape of data and running a comparison between new input data and the memorized one in order to make a decision. In contrast, ML aims at finding complex patterns that not only fit the available data but also generalize to new input, going beyond the mere comparison. Therefore, ML is particularly useful for tasks in which decision rules are particularly difficult to be hard coded and when adaptivity is necessary, like for instance speech and hand writing recognition.

In this paper, we propose to combine the power of machine learning with the flexibility of constraint systems, which is one of the most successful methodologies for solving hard discrete optimization problems. Nowadays, constraint programming is a mature technology, there are very good commercial and open-source solvers, and the range of applications is quite wide. In order to exploit a coordinated combination of the best solvers, portfolios of collaborative solvers have been designed [4].

Recently, much research on constraint solving has been devoted to the satisfiability modulo theories (SMT) problem. SMT solvers can deal with the satisfaction problem of logical formulae formalized in first-order logic (FOL), often with equality, in combination with some background theories. Examples of background theories can be the theory of real numbers, integers, some data structures, etc. SMT solvers are useful for many applications, such as verification of programs and software testing based on symbolic execution. Several recent efficient SMT solvers are currently available, see for instance [5, 16].

As stated above, the goal of this paper is to combine ML and constraint systems. More specifically, we show how machine learning models can be represented by first-order logic theories, and how to derive these theories. We argue that this representing ML by FOL can have numerous applications:

First of all, the representation can help to understand the ML model (*explainability*), which, especially in the case of deep neural networks, could be quite mysterious. Providing an explanations of the decision taken by the system is important from the point of view of the users, especially since these decisions are often critical for the concerned people (diagnosing the right disease, obtaining a loan, etc.).

Second, the representation of ML models in terms of constraint provides an *automatic* way of creating new knowledge, since the learning process in ML is totally automatized and simply consists in applying an algorithm to the available data (*training data*). Since this new knowledge is represented in terms of constraints, it can be processed, queried, checked for satisfiability or implications, etc..

Third, the representation can be augmented with additional formulae, representing constraints of some kind on the data domain. For instance, new knowledge, or potential attackers, or fairness desiderata.

In this paper, we focus on this latter application domain. In particular, we show how our proposed approach can be used to detect potential security and privacy breaches on the ML system, or to prove that they do not exist and the system is therefore secure. The idea is to represent the system as a FOL theory, and the specific user and the attacker as a set of constraints. A SMT solver can then be applied. The existence of a solution then reveals the possibility of an attack, and its nature. The non satisfiability means that the system is secure[1].

[1] It could happen that the SMT solver is not able to find a solution, but cannot prove the non-satisfiability either. For the more complicated theories, indeed, it could happen that the SMT is not decidable, which means that in some cases neither the satisfiability nor the unsatisfiability can be proved. In this case, however, the SMT solver should return a warning, and in this case the diagnosis of the security of the system is not conclusive, but at least we know it. Namely, the approach is correct, even though it may not be complete.

1.1 Related work

The relation between constraint rules and automatic learning is a widely investigated topic in the fields of artificial intelligence and machine learning. In particular, large bodies of work have explored the possibility of extending the ability of learning from data to learn constraints (see, among others, [19, 24, 34, 35])[2]. In other words, while in many applications a trained model is considered as a black-box that creates a descriptive representation of the problem by summarizing the knowledge coming from data, in this case, the focus is on the symbolic interpretation of the model by extracting rules from the model itself (cfr. [9, 30, 43]).

According to the popular framework of *learning from constraints*, it is possible to reinterpret the learning theory based on supervised learning. For instance, the empirical risk minimization approach (cfr. 2.1) can be seen as learning by constraining the error between prediction and supervision to be minimal. Through first order logic (FOL) it is possible to model families of logic constraints which, via the so called task functions, can be mapped onto the *real valued* constraints which are typical of machine learning. The idea was first introduced in [28] and then it has been adapted to kernel base machine learning models in [19].

Building on these basic notions, the works in [25] and [31] propose a solution to learn constraints from observable samples and solve search optimization problems for which the constraints either are not given or need to be estimated. The solution is based and builds on the idea of using data to select constraints from a finite domain, a problem that was already addressed in [6]. In [34] and [35], the authors tackle the problem from an inductive logic programming (ILP) standpoint, building on the framework first introduced in [28]. On the one hand, they propose, once again, to learn constraints within a finite domain which can be modeled by a certain language $\mathcal{L}_c$. On the other hand, by using ILP, they work in a specific machine learning in which first-order logic is used to represent the data as well as the learned hypotheses which, in turn, can be expressed via $\mathcal{L}_c$. Therefore, the solution to the problem can be reduced to finding a particular hypothesis $\in \mathcal{L}_c$ such that it holds for all the positive samples and for none of the negative samples. It is important to notice that the fact that the hypothesis is to be selected as one possible choice within a set of hypothesis recalls the typical machine learning requirement of probably approximately correct (PAC) learning [44].

The common denominator of the works referenced so far is that the only constraints that are involved in the learning are somehow known a priori, and the learning involves taking a decision on which constraints represent the hypotheses learnt from the samples.

In [13], a different framework, which represents a step in a new direction, is introduced. As in the previous work (cfr. [25, 31, 34, 35]), the authors aim at learning constraints from a learnable set. However, they propose to use information theoretic principles (maximizing the information transfer from the concept space (hypotheses) to the rule space (constraints)), and to model the constraints as neural networks. Doing so, they realize that this process leads to the unsupervised development of new constraints that they analyze from the standpoint of FOL.

[27] investigates how to optimise the ML process, and has some similarities with our approach as they present some coding of Artificial Neural Network and Decision Trees in Local Search, Constraint Programming, Mixed Integer Non-Linear Programming (only ANNs) and SAT Modulo Theory (only DTs).

[2] A meaningful distinction must be highlighted between this and the idea of using constraints to drive the learning of a model, for instance by including constraints in the optimized loss functions in a way that recalls the minimization (maximization) of an objective functions according to constraints [36].

1.2 Structure of the paper

In Section 2 we present some preliminary technical definitions on Machine Learning, Constraint systems, and SMT solvers. Then, in Section 3 we consider various machine learning algorithms and we show how the resulting model can be represented as a FOL theory. In Section 4 we show how to apply our methodology to some examples from the field of security. In Section 5 we consider one application to a model-inversion attack. Finally, Section 6 draws some conclusions and discusses some future work.

2 Preliminaries

2.1 Machine learning

We give here a brief introduction to the learning process and the derivation of the model. We will focus on the supervised learning scenario in the context of classification problems, which cover all examples considered in this paper. We describe the basic elements common to all learning algorithms, and to this purpose we introduce a generic learner model based on a well established statistic framework. The details specific to the various algorithms will be described in the next section.

A learning problem is defined by:

- a domain $\mathcal{X}$ of objects, represented as a vector of *features* (aka *attributes*) that we would like to classify;
- a set of *labels* (aka *classes*) $\mathcal{Y}$;
- a set of training data, i.e., a sequence $\mathcal{S} = ((\vec{x}_1; y_1) \dots (\vec{x}_m; y_m))$ of pairs in $\mathcal{X} \times \mathcal{Y}$;
- a correct labelling function $f : \mathcal{X} \to \mathcal{Y}$, such that, for all i, $y_i = f(\vec{x}_i)$;
- a distribution $\mathcal{D}$ of type $\mathcal{X} \times \mathcal{Y}$, according to which the samples are generated;
- the *prediction rule* or *hypothesis* $h : \mathcal{X} \to \mathcal{Y}$, that can be used to predict the label of new domain points;
- a measure of success that quantifies the predictor's error.

Ideally, the goal of the learning process is to select an h that minimizes the *risk*, defined as:

$$L_{\mathcal{D},f}(h) \stackrel{\text{def}}{=} \mathbb{P}_{\vec{x}\sim\mathcal{D}}\left[h(\vec{x}) \neq f(\vec{x})\right], \tag{1}$$

which represents the *expected probability* ($\mathbb{P}$) of a mismatch between h and f, measured with respect to the distribution $\mathcal{D}$.

In practice however we cannot compute analytically the h that minimizes (1), because we do not have a mathematical description of $\mathcal{D}$. What we can do, instead, is to use the training set $\mathcal{S}$, that, being generated from $\mathcal{D}$, represents an approximation of it. Then h is selected so to minimize the *empirical risk* over m samples, which is defined as:

$$L_{\mathcal{S}}(h) \stackrel{\text{def}}{=} \frac{|\{i \in [m] : h(\vec{x}_i) \neq y_i\}|}{m}. \tag{2}$$

This principle is called *empirical risk minimization* (ERM). The way this minimization is achieved depends on the specific algorithm, and the function h that is derived is called *model*.

For an extended discussion of the topic as well as a more complete overview of the learning problem we refer to [41, 44]. For further information about ML and the most popular algorithms and applications we refer to [21], while for a more theoretical and statistical background on the learning problem we refer to [15, 22].

2.2 Constraints and SMT solvers

In this paper we consider constraint systems as first order logic formulae. We note that there are alternative approaches in the literature, for instance using Scott's information systems [40, 39].

Following [23], we define a constraint system as a 4-tuple $(\Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T})$, where Σ is a signature, $\mathcal{D}$ is a Σ-structure, $\mathcal{L}$ is a set of Σ-formulae, and $\mathcal{T}$ is a first order Σ theory.

The idea is that Σ defines the syntax of the functions and predicates with their arities, $\mathcal{D}$ is the (Σ-)structure on which the computation is performed, and which allows to give a semantic interpretation to the functions and predicates defined in Σ, $\mathcal{L}$ are the constraints which can be syntactically expressed, and $\mathcal{T}$ is an axiomatization of some properties of $\mathcal{D}$.

The pair $(\mathcal{D}, \mathcal{L})$ is called a *constraint domain*.

We make the following assumptions:

- the terms and constraints in $\mathcal{L}$ are defined in a first-order language.
- the binary predicate symbol $=$ is always in Σ and is interpreted as the identity in $\mathcal{D}$.
- there are two constants $true$ and $false$ in $\mathcal{L}$ which are respectively identically true and identically false in $\mathcal{D}$.
- the set of constraints $\mathcal{L}$ is closed under variable renaming, conjunction, and existential quantification.

Examples of constraint domains [23] include, for instance, sets of linear equations and/or inequations over real numbers, the domain of word equations on strings, the finite domains over integers, where linear equations/inequations are built over variables which can assume values on intervals of integers, boolean constraints, constraints over finite trees (namely logic programming syntactic equations on data terms) etc.

We assume that our domains support a test of *consistency* or satisfiability. So we assume that it is possible in any moment to perform the following check:

$$(\mathcal{D}, \mathcal{L}) \vdash c \tag{3}$$

meaning that there exists a solution for the conjunction of constraints c, or that the variables in c can be instantiated in such a way to be solvable.

We also assume that we can perform an *entailment* (or *implication*) test of one constraint c' by another one c:

$$(\mathcal{D}, \mathcal{L}) \vdash c \Rightarrow c'. \tag{4}$$

Sometimes we use the equivalent notation $c \vdash c'$.

A conjunction of constraints can be simplified by using several formal techniques, maintaining the same set of solutions for the constraints to be solved.

Not for all constraint domains are available solvers for all possible cases, as the problem might be undecidable in the general case. In this case the solution of some constraints can be delayed until (and if) they become easier and (possibly) solvable.

Several refined techniques for solving constraints have been defined. We mention just a few of them:

- constraint propagation, exploiting local consistency conditions of subsets of constraints. This normally allows to reduce the search space of solutions.
- constraint simplification, trying to replace constraints by equivalent ones easier to solve.
- backtracking search, which allows to solve incrementally a subset of the constraints, backtracking on some assignments as soon as the set of constraints clearly gets not solvable.

- local search, which tries to modify a value of a variable at each step, choosing assignments close to the previous one in the search space.

For a more extensive overview please consult [38].

2.2.1 SMT solvers

SMT solvers are decision procedures for satisfiability of fragments of first-order logic with equality, where variables range over SMT data types, such as Booleans, integers, and reals. Satisfiability Modulo Theories (SMT) problem for a theory T [29] expressed as a set of closed first order formulae, can be intuitively defined as follows: given a formula F (it can be a propositional formula, or a ground formula in first order logic, or a formula in first order logic), determine whether F is T-satisfiable, i.e., whether there exists a model of T that is also a model of F. A formula F is T-satisfiable or T-*consistent* if $F \wedge T$ is satisfiable in the first-order sense. If the theory T is empty the problem reduces to satisfiability of a set of propositional/first order logic formulae. SAT solvers are available for this subproblem. A lot of research has been devoted to develop efficient SMT solvers [29]. There exist eager or lazy approaches. In eager approaches the input formula is transformed in a satisfiability equivalent one, usually in conjunctive normal form and then SAT solvers are applied. In a lazy approach the atoms of T are considered as propositions by the SAT solver. If the SAT solver returns a propositional model M of F, then this assignment (seen as a conjunction of literals) is checked by a T-solver. If M is found T-consistent then it is a T-model of F. Otherwise the process restarts. Incremental techniques, theory propagation and simplification can be also applied. Some examples of efficient SMT solvers are CVC4 [5], Yices 2 [16] and Z3 [14]. It is also possible to use efficient open source constraint programming solvers containing constraint solvers integrated with SAT modules, such as Chuffed [11] and OR-tools [32].

3 FOL theories from machine learning models

In this section we consider various machine learning algorithms and we show how the resulting model can be represented as a FOL theory.

3.1 Decision trees

Decision Trees (DTs) are models of supervised learning rather simple to understand and to interpret, also thanks to their graphical representation as mathematical trees [33]. They are predictive models, i.e., they allow to derive the value of a target variable from the value of the features of a given sample, and they are of two main kinds: *classification trees*, if the target variable is categorical, or *regression tree* if the predicted outcome is a real number. Here we consider the first case.

In general, DTs are constructed via an algorithmic approach that tries to identify the best ways to split the data set according to various criteria. To find the optimal tree, however, is a NP problem, so usually a greedy approach is used, which, at each step, identifies a feature and a test on that feature. In general, we choose the feature that gives the best *information gain* at that point of the process, which should help to keep the weighted tree balanced, at least locally. The feature is then associated to an intermediate node of the tree[3] and the

[3] A generalization of this method consists in associating to a node a set of features, and in this case the test represents a relation among the features. Here however we consider only the simpler case of a single feature per node.

possible outcomes of the test, that determine a partition on the data, are associated to the subtrees children of that node, and label the corresponding edges. The process goes on until we reach a unique classification for the target variable of the remaining data, which is then associated to a leaf. Figure 1a shows an example of decision tree where each node x_i is submitted to a binary test, i.e., whether it satisfies or not a certain property P_i. More in general however, the test could have more than two outcomes; the important thing is that they are mutually exclusive and cover the whole range of possibilities.

The representation of the tree in terms of constraints is defined as follows. The constraint system $(\Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T})$ is such that:

- Each sort of Σ represents a feature, plus there is a sort for the class. The values of the features and of the class are represented in Σ by symbols of the corresponding sort. Furthermore Σ contains the propositional symbols corresponding to the properties decorating the edges of the tree.
- For each feature x, Σ contains a distinct variable X of the sort corresponding to x. Furthermore, Σ contains an additional variable L of sort class.
- $\mathcal{D}$ is a structure containing the domains of the features values, the domain of the class values, and the properties that decorate the edges. Because of the way the tree is constructed, each property is satisfied by at least one feature value.

In order to define the theory $\mathcal{T}$ we introduce the following notation: Given a path γ from the root to a leaf, we denote by γ_i the proposition associated to the i-th edge in the path, and let x_i be the node just before that edge. Let ℓ be the label decorating the leaf of that path. Then, for each path γ, we assume that $\mathcal{T}$ contains the following formula.

$$(\bigwedge_i \gamma_i(X_i)) \Rightarrow L = \ell \tag{5}$$

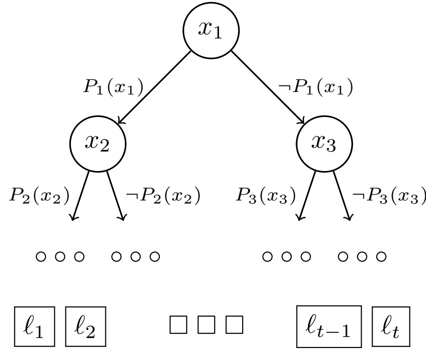
For example, the set of formulae constructed from the binary tree of Figure 1a is shown in Figure 1b. If the properties decorating the edges are equalities or negations of equality, then we don't need to add anything else in $\mathcal{T}$. Otherwise, we need to define the meaning of the propositions. For finite domains this can be done by adding to $\mathcal{T}$ all the statements of the form $P(v)$, where P is a proposition and v a value symbol, whenever the corresponding property holds for that value. Otherwise, we should enrich the theory with formulas defining P. For instance if P is an ordering $\leq$ on numbers, we should incorporate in the constraint system the axioms defining this relation.

The set of constraints $\mathcal{L}$ depends on the potential attack and on the victim. In general the victim is the object that we want to classify, hence it is specified by a tuple $(v_1, v_2, \ldots, v_n)$ representing the value of each feature. The constraints relative to the victim are therefore equalities of the form $X_i = v_i$. As for the attacker, it is characterized by its prior knowledge and its capabilities, usually consisting of the properties on the features that the attacker knows or can infer from observing the system, Hence, typically the attacker can be represented by FOL contraints constructed on the $\gamma_i(X_i)$'s atomic formulas. In Sections 4 and 5 we will see examples of attacks and how to derive the corresponding constraints.

3.2 Support vector machines

Support-vector machines are able to support both classification and regression problems. Their foundation relies on the theory of Vapnik and Chervonekis [8].

In this learning method, each object is represented as a point in the n-dimensional space, where n is the number of features. The idea behind the (linear) support vector machine is to construct a hyperplane or a set of hyperplanes of dimension $n-1$, which partition the



(a) A binary decision tree.

$$P_1(X_1) \wedge P_2(X_2) \wedge \ldots \wedge P_l(X_l) \implies L = \ell_1$$

$$P_1(X_1) \wedge P_2(X_2) \wedge \ldots \wedge \neg P_l(X_l) \implies L = \ell_2$$

$$\vdots$$

$$\neg P_1(X_1) \wedge \neg P_3(X_3) \wedge \ldots \wedge P_r(X_r) \implies L = \ell_{t-1}$$

$$\neg P_1(X_1) \wedge \neg P_3(X_3) \wedge \ldots \wedge \neg P_r(X_r) \implies L = \ell_t$$

(b) The formulae derived from the decision tree.

Figure 1

space in such a way that, ideally, each subspace contains only elements of the same class. In practice however a perfect partition is usually not possible, so the presence of elements of different classes is tolerated, we just try to minimize their number[4].

A hyperplane is described by a formula like

$$\vec{w} \cdot \vec{x} - a = 0 \tag{6}$$

where $\vec{w}$ is a vector of numerical coefficients, and a is a numerical constant. Both $\vec{w}$ and a are determined by applying the minimization explained above.

For instance, in a binary classification problem, we would determine $\vec{w}$ and a so that as many elements as possible of class ℓ_1 are below the hyperplane defined by (6) ($\vec{w} \cdot \vec{x} - a < 0$), and as many elements as possible of class ℓ_2 are above it ($\vec{w} \cdot \vec{x} - a > 0$). An example is shown in Figure 2: no linear hyperplane can completely separate the classes ℓ_1 and ℓ_2, but the hyperplane h_2 is optimal.

To derive a constraint system from such model we proceed in a way similar to the one described in Subsection 3.1. In this case, however, Σ must contain also the symbols $+$, $\times$ (numerical addition and multiplication), and $<$ (strict ordering), and $\mathcal{T}$ must contain also the axioms defining these operations and the ordering relation. The specific axioms representing the classification are then:

$$\vec{w} \cdot \vec{X} < a \Rightarrow L = \ell_1 \qquad \vec{w} \cdot \vec{X} > a \Rightarrow L = \ell_2 \tag{7}$$

In case of more than 2 labels, each of these clauses will contain a conjunction of premises of the form $\vec{w} \cdot \vec{X} < a$ and $\vec{w} \cdot \vec{X} > a$, representing discriminations operated by various hyperplanes. In some cases, which might be interesting for certain applications, failing to recognize a sample as belonging to a certain class, for instance ℓ_1, might be less severe than failing to recognize a sample as belonging to ℓ_2. A typical scenario is that of preliminary tests meant to quickly tell sick patients (ℓ_2) and healthy ones (ℓ_1) apart in order to assign the first ones to further tests. In this case, classifying an healthy person as sick is usually

[4] One method to improve the result is to consider more features, which corresponds to increasing the number of dimensions. Another possibility is to consider also non-linear separation surfaces. A generalizazion of support vector machines in this sense has been proposed as well.

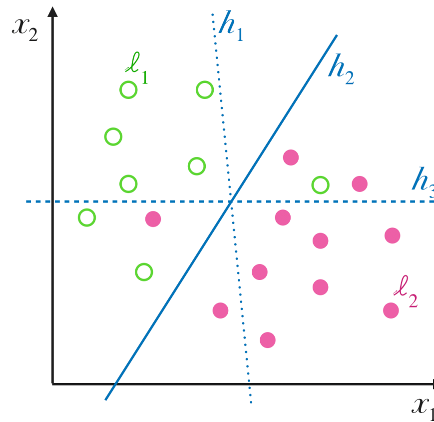


Figure 2 Separating the classes ℓ_1 and ℓ_2 by hyperplanes. Among the three hyperplanes h_1, h_2 and h_3, the one that produces the best discrimination is h_2. The resulting partition is not perfect but it is optimal, because no linear hyperplane can completely separate the classes ℓ_1 and ℓ_2.

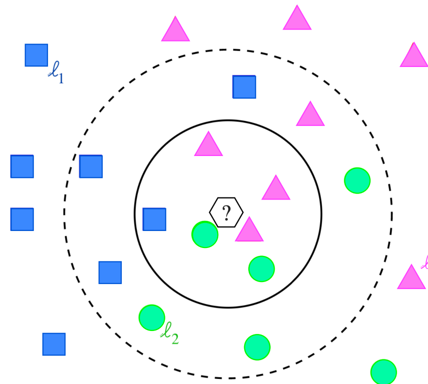


Figure 3 Classifying a new sample '?' using k-NN. If $k = 1$ the new object will be classified as ℓ_2. If $k = 6$ it will be classified as ℓ_3. If $k = 13$ it will be classified as ℓ_2 again.

considered less severe than misclassifying a sick patient who needs to be thoroughly checked through further tests. Therefore, among two (or more) hyperplanes providing the same overall misclassification rate (i.e. number of wrongly classified samples over the total amount of samples), the one(s) with the lowest misclassification rate on the samples of class ℓ_2 is to be preferred.

3.3 Nearest neighbors

The k-nearest neighbors algorithm (k-NN), where k is a numerical parameter, is a learning method proposed by Thomas Cover that supports both classification and regression [2]. In both cases we assume that the space of the features is equipped with a notion of distance d. The basic idea is the following: every time we need to classify a new sample, we find the k samples in the training set whose features are closest to those of the new one (nearest neighbors). Once the k nearest neighbors are selected, a majority vote over their class labels is performed to decide which class should be assigned to the new sample. See Figure 3 for an example.

To derive a constraint system from such model we proceed in a way similar to the one described in Subsection 3.1. In this case, however, Σ must contain also the symbol d (distance), and $\leq$ (ordering), and $\mathcal{T}$ must contain also the axioms defining the notion of distance and the ordering relation. Let us consider for simplicity the case $k = 1$. The specific axioms representing the classification are as follows: for any tuple $(\vec{v}, \ell)$ in the training set:

$$\forall \vec{w}.\ d(\vec{X}, \vec{v}) \leq d(\vec{X}, \vec{w}) \Rightarrow L = \ell. \tag{8}$$

Note that the training set is finite, hence the quantification on $\vec{w}$ is done over a finite domain, which means that we can eliminate it by reducing to a conjunction of formulae. Note also that we need to take a bit of care to avoid conflicts in the classification where there are two or more samples that are at the same minimal distance from the new object. In this case, we must establish some priority among samples, and add to $\mathcal{T}$ only the clause that sets the labeling along with the sample of highest priority.

The above idea can be extended to the case of a generic k. For any set of k tuples $\{(\vec{v}_1, \ell_1), \ldots, (\vec{v}_k, \ell_k)\}$ in the training set, let ℓ be the most frequent among the labels $\ell_1, \ldots, \ell_k$ (again, in case of equality we must establish some kind of priority). The axioms are:

$$\forall \vec{w} \neq \vec{v}_1, \ldots \vec{v}_k.\ (d(\vec{X}, \vec{v}_1) \leq d(\vec{X}, \vec{w}) \wedge \ldots \wedge d(\vec{X}, \vec{v}_k) \leq d(\vec{X}, \vec{w})) \Rightarrow L = \ell$$

where the quantification $\forall \vec{w} \neq \vec{v}_1, \ldots \vec{v}_k$ is to be replaced by a conjunction of all the formulae in which $\vec{w}$ is different from $\vec{v}_1, \ldots \vec{v}_k$.

3.4 Neural networks

Artificial neural networks (ANNs) are computing systems inspired by the biological brains [7, 21, 22]. An ANN consists of a collection of connected units called nodes, simulating the behavior of neurons. The connections are called edges. Each node, like the synapses in a biological brain, can receive, process, and transmit a signal to the nodes connected to it. The "signal" is a real number, and the output of each node is computed by some function of its inputs. Nodes and edges typically have a weight that increases or decreases the strength of the signal at a connection, and is computed during the learning phase. Typically, neurons are aggregated into layers, and signals travel from the first layer (the input layer), to the last layer (the output layer).

The relation between neural networks and constraints has been the object of investigation of several works already, especially in the context of the *learning from constraints* paradigm, in which the idea is to use additional knowledge, represented as constraints, to guide and refine the learning process. Two of these works, [13] and [12], provide also an interesting point of view on the problem of *learning explainability*, which focuses on the interpretation of the decision making process of the neural black-box models.

The key idea for this approach is to realize a mapping from the real valued functions represented by neural networks models to the space of constraints. Let us consider an input space $\mathcal{X}$ of dimension n, i.e., the space of the samples fed to a net. A learning environment for a multi-task (aka multi-label) classification problem can be defined as a vector $f = [f_1, \ldots, f_t]$, where $f : \mathcal{X} \subset \mathbb{R}^n \to \mathcal{Y} \subset \mathbb{R}^t$ and t is the number of classes. The environment of the constraints can be defined as $\phi = [\phi_1, \ldots, \phi_c]$, where $\phi : \mathcal{Y} \subset \mathbb{R}^t \to \mathcal{Z}^c$, and c is the number of constraints. Typically each constraint ϕ_j is embedded into a non-negative penalty function $\widehat{\phi}_j$ for $j = 1, \ldots, c$, so that the optimal learning environment, f^* can be defined as

$$f^* = \underset{f}{\operatorname{argmin}} \sum_{j=1}^{c} \sum_{x_k \in X_{\phi_j}} \widehat{\phi}_j(f(x_k)) + \gamma_f, \tag{9}$$

where γ is a regularization term associated to f and X_{ϕ_j} is the sample space associated to the j-th constraint. For example, f^* could be defined in terms of the classical error minimization, which corresponds to imposing $\widehat{\phi}_j(f(x)) = (f(x)-y(x))^2$, where $y(x)$ represents the supervision. However, the role of ϕ_j in the considered learning framework is more general, since ϕ_j could represent different types of knowledge expressed by means of FOL formulae. In [19] the authors discuss how to use directly constraints as FOL, so to transfer logic knowledge to neural networks models.

The authors of [13] refine the above paradigm and their system can actually automatically learn new constraints ψ, initially modeled in terms of numerical functions of type $\mathbb{R}^n \to [0, 1]$. Then, they show how to convert them into logical formulae, so to obtain a symbolic representation of this new knowledge. The main idea consists in approximating the input of each neuron with the vertices of the Boolean hyper-cube, while the neuron output is approximated with a Boolean value.

The work [12] also refines the above paradigm, and focuses on how to interpret the outcome of the network in term of symbolic constraints, but the approach is quite different from the one in [13]. Specifically, their proposal consists in introducing another neural network that operates in the output space of the classifier, and whose purpose is to build the formulae that represent the explanation of the classifier. The two networks are trained jointly in the learning process, thus implicitly introducing a latent dependency between the development of the explanation mechanism and the development of the classifier.

Both these approaches can be used for our purposes. We refer to the corresponding papers [12] and [13] for the details.

4 Applications to Security

In this section we consider some examples of applications to security. The idea is to model the attacker's capabilities, its prior knowledge (aka *side knowledge*), and its goals in terms of constraints. Then, consider the union of these constraints and the theory coming from the machine learning model and use the SMT module to check the existence of a solution. In the negative case we can conclude that the system is safe. Otherwise, a threat exists, and the solution produced by the SMT provides the description of the potential attack and the level of vulnerability.

We will consider two examples from two different areas of computer security: information flow and malware.

4.1 Information flow

Secure information flow is concerned with the inference of secrets from information made publicly available by the system, or anyway, information that the attacker is able to obtain by observing its behavior. Typically, these are physical observations made during a run, such as the execution time, the level of energy consumption, etc., and the corresponding kind of security breach is called *side channel attack*. The leakage of information in these situations is due to the correlation between the secret and the observable, and, by definition, it cannot be prevented using the typical security defenses, such as encryption or access control: it usually requires a re-thinking of the system architecture. Given the situation, a verification

of the system in terms of constraints can be useful not only to show that the system is safe, but also, in case of the existence of a breach, to indicate its cause and help redesigning the system so to eliminate it.

The complete lack of any information leakage is called *non-interference*, and has been an active area of research since the seminal paper of Goguen and Meseguer [20]. However, in more recent times, it is recognized that for practical systems the non-interference property is usually impossible to achieve, because some correlation between the output and the secret is inherent to the specification of the system and therefore cannot be totally eliminated. For example, a password checker always reveals a little bit of information about the secret password: in case of success we know that the password is the string we have entered, and in case of failure we know that the password must be different from the string we have entered. Therefore, a more significant analysis is not to detect the existence of a leak, but, rather, its magnitude. These considerations have given rise to the "modern" approach to secrecy called *quantitative information flow*, which focuses on measuring the amount of leakage and the threat that it implies. The metrics used for the measurements are usually based on probabilistic aspects (typically, the probability of a successful attack) or on the complexity of the attack (typically, the number of times the attack needs to be repeated to ensure success) [3].

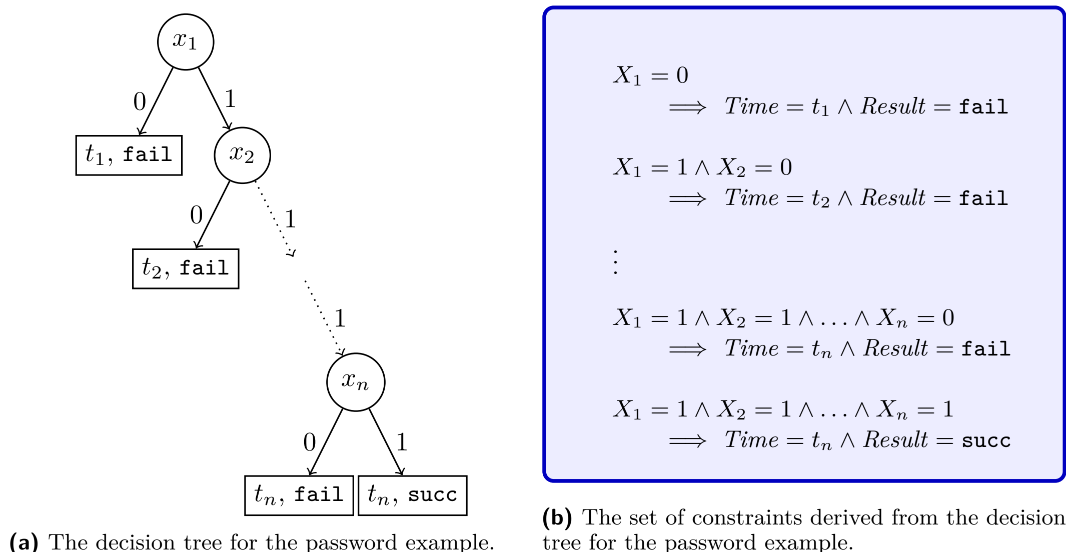
The constraint-based approach we propose can be used for verifying non-interference by proving that the capabilities of the attacker do not allow to derive any information about the value of the secret. This can be done by showing that there is no correlation between the properties on the features accessible to the attacker and the value of the secret (an implication from properties to values has no solution). More interestingly, our approach can also be used to *measure* the leakage of information, by computing the number of possible values for the secret from the point of view of the attacker. This number can then be interpreted as the probability of a successful attack (in one-try scenario) or as the complexity of the attack (in a repeated-try scenario). We show here an example of the latter in the form of a timing attack to a password checker[5]. A timing attack is a particular kind of side-channel attack based on the information that can be derived from the execution time.

We consider a password of n bits, and we assume that the checker takes in input a string of n bits $b_1 b_2 \dots b_n$ from the user (who could be an attacker trying to crack the password), and checks it, bit by bit, against the stored password $p_1 p_2, \dots p_n$. We assume that the system stops as soon as it finds a mismatch. Namely, at each run the system performs k steps, with $1 \leq k \leq n$, where k is the position of the first bit that does not match the corresponding bit in the password. If all bits match, then $k = n$ and the system output *success*, otherwise *fail*. Namely:

$$k \stackrel{\text{def}}{=} \begin{cases} \operatorname*{argmin}_i \, (b_i \neq p_i) & \text{if } \exists i.\, b_i \neq p_i \\ n & \text{otherwise} \end{cases} \tag{10}$$

It is well known that such behavior produces a security breach. In fact, if an attacker is able to observe the exact execution time, it can infer correctly a password prefix: for $1 \leq k \leq n$, a k-steps run with output *fail* means that the first k bits of the password are $b_1 b_2 \dots b_{k-1}(b_k \oplus 1)$, where $\oplus$ is the sum modulo 2. (A n-steps run with output *success* means, of course, that the password is the same as the string.) As a consequence, the complexity of the attacks reduces from exponential to linear (in n), in the sense that after at most n re-iterations of the attack, the adversary can infer the entire password.

[5] This is a toy example, but it illustrates well a typical class of realistic and practical timing attacks, namely those against encryption keys. We have chosen a simpler example so to avoid introducing cryptographic notions, which are orthogonal to the issues considered here.



(a) The decision tree for the password example.

(b) The set of constraints derived from the decision tree for the password example.

Figure 4

One technique to mitigate the security breach is the so-called *bucketing* [26], which consists in letting the system run for a longer time, so to avoid revealing the exact step where the first mismatch occurs. More precisely, the bucketing technique partitions a system's possible execution times into intervals, called buckets, of variable length. Given the number of steps k as defined above, the system waits until the upper bound of the bucket containing k, then it ends the computation and returns the result. For instance, if all buckets have size h then the system stops after m time units, where m is the smallest number which is greater than or equal to k, and is a multiple of h. Often, however, the buckets are of different sizes, to further confuse the adversaries and/or to optimize the trade-off between execution time and security.

Suppose that we have a password checker produced by an untrusted third party, and we want to check how secure it is. We can interact with the system, but we don't know its code. Namely, we are in a *black box* scenario. In [10, 37] it was shown how to estimate the leakage of information in a black box situation using machine learning. In short, by interacting with the system the analyst collects a set of examples, representing pairs secret-observables, and uses them to train a classifier that, given an observable, tries to infer the corresponding secret. The expected error of the classifier gives a quantitative estimation of the threat.

We present here an approach alternative to [10, 37], based on the idea of representing the machine learning model, and the attacker, by constraints. The simplest model for our purposes, in the case of this example, is the decision tree. Assume that the features x_1, x_2, $\ldots$, x_n, express the result of the comparison of the string bits against those of the password, i.e., $x_i = 0$ if $b_i \neq p_i$, and $x_i = 1$ otherwise. Then, the model is represented by the tree in Figure 4a, where (not necessarily distinct) values t_1, t_2, $\ldots$, t_n indicate the time units (labels). The corresponding set of constraints is given in Figure 4b.

We now consider how to represent an attacker. We assume that the adversary can give in input a string and observe the number of time units before the system stops, and repeat the attack until it infers the whole password. In a given iteration step during the attack, the relevant background knowledge consists of the password bits p_1, p_2, $\ldots$, p_m that it has

$$(X_1 = 1 \wedge X_2 = 1 \wedge \ldots \wedge X_m = 1) \wedge$$
$$(\mathit{Time} = t_1 \vee \mathit{Time} = t_2 \vee \ldots \vee \mathit{Time} = t_n) \wedge (\mathit{Result} = \texttt{fail} \vee \mathit{Result} = \texttt{succ})$$

Figure 5 The attack in the password example.

already discovered. Obviously, the best strategy for the adversary consists in entering a string of the form $p_1 p_2 \ldots p_m b_{m+1} \ldots b_n$. Hence, this iteration step of the attack can be represented by the disjunctive formula in Figure 5.

The union of the constraints in Figure 4b and Figure 5 has many solutions, that can be produced by the SMT module, and that can be partitioned according to the value assigned to the variable *Time*. Let S_t be the set of solutions in which the variable *Time* is assigned the value t. Clearly, the largest such set is for $\mathit{Time} = t_{m+1}$. In this case, in fact, the attacker has no information about the remaining password bits, except that at least one among $b_{m+1}, b_{m+2}, \ldots b_{m+h}$ must be wrong, where h is the size of the bucket that contains the $m+1$ bit. Equivalently, one of the $X_{m+1}, X_{m+2}, \ldots X_{m+h}$ must be 0. The number of such configurations is $2^h - 1$, and can be determined automatically by counting the number of solutions for these variables[6]. Assuming that the substring of length h have the same probability $1/2^h$ to be part of the correct password, and that the attacker chooses uniformly the next one to try, the expected number of attempts that the attacker needs to perform to get to the next bucket is:

$$\sum_{i=1}^{2^h-1} \frac{1}{2^h}\, i \quad = \quad \frac{1}{2^h} \frac{(2^h-1)\, 2^h}{2} \quad = \quad \frac{(2^h-1)}{2}.$$

By repeating this process, we can determine the average-time complexity of the attack (the expected total number of attempts before discovering the entire password), which is given by $(2^{h_1} + 2^{h_2} + \ldots + 2^{h_s} - s)/2$, where $h_1, h_2, \ldots, h_s$ are the size of the buckets. We remark that s and $h_1, h_2, \ldots, h_s$ are not known, and that this result is computed automatically from the constraints derived from the machine learning model. Furthermore, for more complicated timing attacks, such as attacks to encryption key, the relation between the stopping time, the size of the buckets and the bits of the key is not necessarily known, or may be very complicated to compute. The method illustrated above gives a quick and easy way to determine the complexity of the attack also in these cases.

4.2 Malware

Machine learning models are often used in critical applications which involve decisions of significant personal, societal, or economical impact. Examples include network intrusion detection, spam and phishing detection, healthcare systems, production planning, etc. Malware attacks to such decisional models typically consist in altering the values of some of the features so to induce a misclassification, and therefore a wrong decision. In this section we consider an example in the healthcare domain, and we show how our methodology can help to detect a potential attack, or prove its impossibility.

[6] This can be done for example in CLP(FD) [23] by using the predicate **fd_size**, which returns the number of elements in the current domain of a variable.

$$Pulse > 160 - 0.5\,Age \quad \Rightarrow \quad Hearth_alarm = \texttt{on}$$

$$\left(\begin{array}{c} ((Sex = \texttt{female} \wedge Weight > .85(Height - 100)) \\ \vee \\ (Sex = \texttt{male} \wedge Weight > .90(Height - 100))) \\ \wedge \\ Blood_pressure > 130 \end{array}\right) \quad \Rightarrow \quad Diabetes_alarm = \texttt{on}$$

Figure 6 Representation of a SHS in form of FOL constraints.

In the last decade there has been a major evolution towards automatization in healthcare, thanks to advances of research in the Internet of Things that has allowed to connect body sensors and other implantable medicals devices to networks of computing and big data resources. Smart healthcare systems (SHS's) continuously collect data from the medical sensors connected to the human body and process them for making decisions accordingly. This trend will increase in the future due also to the development of personalised medicine. Typically, machine learning is used to classify possible health problems from the symptoms and prescribe the necessary treatment. Unfortunately, these devices are exposed to potential attacks, especially due to the possible presence of malware that can compromise the readings of the sensors, like in the case of MEDJACK [42].

We assume that the SHS uses a classifier that takes in input, as features, the physical characteristics of the patient (such as weight, height, age, etc.), and the readings from the body sensors, to decide whether there is some potential health threat, and consequently raise the corresponding alarm. In this example we assume that the classifier is implemented as a Support Vector Machine. Figure 6 shows some typical constraints that could be derived from the classifier, to signal critical situations that could bring to a diabetic attack or to a hearth attack. The first formula represents the monitoring of the pulse in relation to age, during physical exercise. In the second formula the relations between height and weight are based on the Broca equations for women and men, respectively, and indicate the condition of being overweight.

In this scenario, a particular patient can be represented by the values of his or her attributes. For instance:

$$\mathrm{P}_1 \qquad Sex = \texttt{female} \wedge Weight = 80 \wedge Height = 165 \wedge Age = 40 \tag{11}$$

$$\mathrm{P}_2 \qquad Sex = \texttt{male} \wedge Weight = 75 \wedge Height = 180 \wedge Age = 40 \tag{12}$$

An attacker can be represented by its goals and its capability in tampering with the readings of the sensors. For instance, an attacker that aims at thwarting the diabetes alarm and is able to alter the reading of the sensor $Blood_pressure$ up to ± 20, can be represented by the constraint:

$$R - 20 \leq Blood_pressure \leq R + 20 \wedge Diabetes_alarm = \texttt{off} \tag{13}$$

where R is the true reading of the sensor.

Consider the set of constraints C consisting of set of formulae in Figure 6 together with the one representing the patient ((11) or (12)), and the one representing the attacker (13). By applying the SMT to C we can verify the existence of a potential attack. For instance, consider the patient P_1. For any $r \leq 150$ C is satisfiable by the following assignment (where we use $\doteq$ to represent the association between a value and a variable):

$$\begin{aligned} &\mathit{Sex} \doteq \texttt{female}\ ,\ \mathit{Weight} \doteq 80\ ,\ \mathit{Height} \doteq 165\ ,\ \mathit{Age} \doteq 40\ , \\ &R \doteq r\ ,\ \mathit{Blood_pressure} \doteq r - 20\ ,\ \mathit{Diabetes_alarm} \doteq \texttt{off} \end{aligned} \tag{14}$$

which for $130 < r \leq 150$ represents a security breach because in such situation the alarm should be switched on.

Apart from reinforcing the security of the sensor, a possible countermeasure against this attack would be to specialize the SHS for patient P_1 so that the attacker could not succeed to prevent the activation of the alarm whenever $r > 130$. In order to ensure the non-satisfiability of (13) we derive that the threshold for *Blood_pressure* must be at most 110. Namely, we need to replace the constraint on *Blood_pressure* in Figure 6 with $\mathit{Blood_pressure} > 110$. As for patient P_2, the attack on the diabetes alarm is not possible, thanks to the fact that, due to his low weight, he is not a patient-at-risk. In both cases, the lack of an attack possibility can be automatically verified by proving, via the STM, that the set of constraints has no solution.

5 Applications to Privacy

5.1 Model inversion

Model-inversion (MI) attacks aim at deriving sensitive features of a target individual by taking advantage of their correlation with the output revealed by the machine learning model. The first work that pointed out the existence of such privacy threat was [18]. In that paper, the authors considered a linear regression model for personalized medicine (recommendation of the dosage of a drug called *warfarin*), and they showed that an attacker that has access to some of the non-sensitive attributes of the victim (age, race, height, and weight) and to the outcome of the model (the dosage), may infer the victim's private genomic attributes, especially if he or she has participated in the dataset used for training. (The study focused on two genes, VKORC1 and CYP2C9, that are associated with the mechanism with which the body metabolizes the drug.) Successive works (for instance, [17, 1]) have extended MI attacks to other settings, e.g., recovering an image of a person from a face recognition model given just their name, and other target models, e.g., logistic regression and neural networks.

Usually MI attacks are formalized in probabilistic terms, namely the attacker is supposed to know the distributions of the data, and its strategy consists in determining which possible value for the sensitive feature achieves the maximum likelihood, given its knowledge of the non-sensitive values and the result of the model. However, for the sake of simplicity, here we consider a non-probabilistic model of attacker.

Suppose that the model represents the classification $h : \mathcal{X} \rightarrow \mathcal{Y}$, and that each $x \in \mathcal{X}$ is a tuple of n features (attributes) $\langle x_1, x_2, \ldots, x_n \rangle$. Suppose that x_n is the sensitive attribute, and that the attacker is interested in discovering whether the value of x_n belongs to a target domain A. Suppose that the attacker knows the value $a_1, a_2, \ldots, a_{n-1}$ of the first $n-1$ attributes, and the classification outcome b. Then, the attack is effective if all the values a_n, such that $h(a_1, a_2, \ldots, a_{n-1}, a_n) = b$, belong to A. Otherwise, the attacker does not have enough evidence.

One way to model such attack by constraints is by representing its opposite, namely the possible existence of an alternative value for x_n which still satisfies the relation. This gives rise to the formula:

$$X_1 = a_1 \wedge X_2 = a_2 \wedge \ldots \wedge X_{n-1} = a_{n-1} \wedge Y = b \wedge \bigwedge_{a \in A} X_n \neq a \tag{15}$$

The outcome of the SMT solver in this case is to be interpreted as follows: no solutions implies that there is an attack, and viceversa one or more solution implies that the attacker has not enough evidence.

6 Conclusion

In this work we have discussed how to combine constraint solving and ML in a novel way. We have shown that several algorithms for ML, such as decision trees, support vector machines, k-nearest neighbours, can be transformed in a corresponding set of formulas in FOL to be solved by means of constraint solvers and SMT solvers. There are several advantages of this approach. One is to exploit the existing (and future) efficient solvers. Another one is to be able to combine the formulae in FOL with additional formulae expressing some critical facts on which we want to test the system. We have presented some examples of attempts of security and privacy breaches, such as an attempt to discover a password using information flow, an example of malware attack for smart health systems, and one model-inversion attack to a neural network for personalised medicine. We have then represented the attacker by an appropriate set of constraints, which we have combined together with the constraints of the original system. Finally, a check of satisfiability by a (SMT) solver of the full set of constraints allows us to derive whether the attack is possible, and how to prevent it.

As future work we intend to make some complexity analysis for the formulas that we derive with our methodology and perform experiments to evaluate the effectiveness of our approach. For this we need to encode the constraints generated with an appropriate constraint solver as mentioned in Section 2.2.

References

1 Ulrich Aïvodji, Sébastien Gambs, and Timon Ther. GAMIN: an adversarial approach to black-box model inversion. *CoRR*, abs/1909.11835, 2019. `arXiv:1909.11835`.

2 N. S. Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992. URL: `http://www.jstor.org/stable/2685209`.

3 Mário S. Alvim, Konstantinos Chatzikokolakis, Annabelle McIver, Carroll Morgan, Catuscia Palamidessi, and Geoffrey Smith. *The Science of Quantitative Information Flow*. Springer International Publishing, 2020. `doi:10.1007/978-3-319-96131-6`.

4 Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. Portfolio approaches for constraint optimization problems. *Ann. Math. Artif. Intell.*, 76(1-2):229–246, 2016. `doi:10.1007/s10472-015-9459-5`.

5 Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification – 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011. `doi:10.1007/978-3-642-22110-1_14`.

6 Nicolas Beldiceanu and Helmut Simonis. A model seeker: Extracting global constraint models from positive examples. In Michela Milano, editor, *Principles and Practice of Constraint Programming – 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, volume 7514 of *Lecture Notes in Computer Science*, pages 141–157. Springer, 2012. `doi:10.1007/978-3-642-33558-7_13`.

7. Christopher M. Bishop. *Pattern recognition and machine learning, 5th Edition*. Information science and statistics. Springer, 2007. URL: https://www.worldcat.org/oclc/71008143.
8. Bernhard E. Boser, Isabelle Guyon, and Vladimir Vapnik. A training algorithm for optimal margin classifiers. In David Haussler, editor, *Proceedings of the Fifth Annual ACM Conference on Computational Learning Theory, COLT 1992, Pittsburgh, PA, USA, July 27-29, 1992*, pages 144–152. ACM, 1992. doi:10.1145/130385.130401.
9. Will Bridewell, Pat Langley, Ljupco Todorovski, and Saso Dzeroski. Inductive process modeling. *Mach. Learn.*, 71(1):1–32, 2008. doi:10.1007/s10994-007-5042-6.
10. Giovanni Cherubin, Konstantinos Chatzikokolakis, and Catuscia Palamidessi. F-BLEAU: fast black-box leakage estimation. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 835–852. IEEE, 2019. doi:10.1109/SP.2019.00073.
11. Geoffrey Chu, Peter J. Stuckey, Andreas Schutt, Thorsten Ehlers, Graeme Gange, and Kathryn Francis. Chuffed, a lazy clause generation solver. URL: https://github.com/chuffed/chuffed.
12. Gabriele Ciravegna, Francesco Giannini, Marco Gori, Marco Maggini, and Stefano Melacci. Human-driven FOL explanations of deep learning. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 2234–2240. ijcai.org, 2020. doi:10.24963/ijcai.2020/309.
13. Gabriele Ciravegna, Francesco Giannini, Stefano Melacci, Marco Maggini, and Marco Gori. A constraint-based approach to learning and explanation. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020*, pages 3658–3665. AAAI Press, 2020. URL: https://aaai.org/ojs/index.php/AAAI/article/view/5774.
14. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Berlin, 2008. Springer-Verlag. doi:10.1007/978-3-540-78800-3_24.
15. Luc Devroye, László Györfi, and Gábor Lugosi. *A Probabilistic Theory of Pattern Recognition*, volume 31 of *Stochastic Modelling and Applied Probability*. Springer, 1996. doi:10.1007/978-1-4612-0711-5.
16. Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification – 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, 2014. doi:10.1007/978-3-319-08867-9_49.
17. Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 1322–1333. ACM, 2015. doi:10.1145/2810103.2813677.
18. Matthew Fredrikson, Eric Lantz, Somesh Jha, Simon M. Lin, David Page, and Thomas Ristenpart. Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, pages 17–32. USENIX Association, 2014. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/fredrikson_matthew.
19. Giorgio Gnecco, Marco Gori, Stefano Melacci, and Marcello Sanguineti. Foundations of support constraint machines. *Neural Comput.*, 27(2):388–480, 2015. doi:10.1162/NECO_a_00686.
20. J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 1982 Symposium on Security and Privacy (SSP '82)*, pages 11–20, Los Alamitos, Ca., USA, April 1990. IEEE Computer Society Press. doi:10.1109/SP.1982.10014.
21. Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016. URL: http://www.deeplearningbook.org/.

22 Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd Edition*. Springer Series in Statistics. Springer, 2009. doi:10.1007/978-0-387-84858-7.

23 Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994. doi:10.1016/0743-1066(94)90033-7.

24 Samuel Kolb, Stefano Teso, Andrea Passerini, and Luc De Raedt. Learning SMT(LRA) constraints using SMT solvers. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 2333–2340. ijcai.org, 2018. doi:10.24963/ijcai.2018/323.

25 Samuel M. Kolb. Learning constraints and optimization criteria. In Parisa Kordjamshidi, editor, *Declarative Learning Based Programming, Papers from the 2016 AAAI Workshop, Phoenix, Arizona, USA, February 13, 2016*, volume WS-16-07 of *AAAI Workshops*. AAAI Press, 2016. URL: http://www.aaai.org/ocs/index.php/WS/AAAIW16/paper/view/12651.

26 Boris Köpf and Markus Dürmuth. A provably secure and efficient countermeasure against timing attacks. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009*, pages 324–335. IEEE Computer Society, 2009. doi:10.1109/CSF.2009.21.

27 Michele Lombardi, Michela Milano, and Andrea Bartolini. Empirical decision model learning. *Artif. Intell*, 244:343–367, 2017. doi:10.1016/j.artint.2016.01.005.

28 Stephen Muggleton. Inductive logic programming. *New Gener. Comput.*, 8(4):295–318, 1991. doi:10.1007/BF03037089.

29 Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(T). *J. ACM*, 53(6):937–977, 2006. doi:10.1145/1217856.1217859.

30 Chunki Park, Will Bridewell, and Pat Langley. Integrated systems for inducing spatio-temporal process models. In Maria Fox and David Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press, 2010. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1742.

31 Tomasz P. Pawlak and Krzysztof Krawiec. Automatic synthesis of constraints from examples using mixed integer linear programming. *Eur. J. Oper. Res.*, 261(3):1141–1157, 2017. doi:10.1016/j.ejor.2017.02.034.

32 Laurent Perron and Vincent Furnon. Or-tools. URL: https://developers.google.com/optimization/.

33 J. Ross Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, 1986. doi:10.1023/A:1022643204877.

34 Luc De Raedt, Anton Dries, Tias Guns, and Christian Bessiere. Learning constraint satisfaction problems: An ILP perspective. In Christian Bessiere, Luc De Raedt, Lars Kotthoff, Siegfried Nijssen, Barry O'Sullivan, and Dino Pedreschi, editors, *Data Mining and Constraint Programming – Foundations of a Cross-Disciplinary Approach*, volume 10101 of *Lecture Notes in Computer Science*, pages 96–112. Springer, 2016. doi:10.1007/978-3-319-50137-6_5.

35 Luc De Raedt, Andrea Passerini, and Stefano Teso. Learning constraints from examples. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 7965–7970. AAAI Press, 2018. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17229.

36 Marco Romanelli, Konstantinos Chatzikokolakis, and Catuscia Palamidessi. Optimal obfuscation mechanisms via machine learning. In *33rd IEEE Computer Security Foundations Symposium, CSF 2020, Boston, MA, USA, June 22-26, 2020*, pages 153–168. IEEE, 2020. doi:10.1109/CSF49147.2020.00019.

37 Marco Romanelli, Konstantinos Chatzikokolakis, Catuscia Palamidessi, and Pablo Piantanida. Estimating g-leakage via machine learning. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20)*. ACM, 2020. To appear. Preprint in CoRR abs/2005.04399. arXiv:2005.04399.

38 Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006. URL: http://www.sciencedirect.com/science/bookseries/15746526/2.

39 Vijay Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993. doi:10.7551/mitpress/2086.001.0001.

40 Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. The semantic foundations of concurrent constraint programming. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 333–352. ACM Press, 1991. doi:10.1145/99583.99627.

41 Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning – From Theory to Algorithms*. Cambridge University Press, 2014. URL: http://www.cambridge.org/de/academic/subjects/computer-science/pattern-recognition-and-machine-learning/understanding-machine-learning-theory-algorithms.

42 Darlene Storm. MEDJACK: Hackers hijacking medical devices to create backdoors in hospital networks. Computerworld, June 2015. URL: https://www.computerworld.com/article/2932371/medjack-hackers-hijacking-medical-devices-to-create-backdoors-in-hospital-networks.html.

43 Geoffrey G. Towell and Jude W. Shavlik. Extracting refined rules from knowledge-based neural networks. *Mach. Learn.*, 13:71–101, 1993. doi:10.1007/BF00993103.

44 Leslie G. Valiant. A theory of the learnable. In Richard A. DeMillo, editor, *Proceedings of the 16th Annual ACM Symposium on Theory of Computing, April 30 – May 2, 1984, Washington, DC, USA*, pages 436–445. ACM, 1984. doi:10.1145/800057.808710.

Adaptive Real Time IoT Stream Processing in Microservices Architecture

Luca Bixio
Flairbit s.r.l, Genova, Italy
luca.bixio@flairbit.io

Giorgio Delzanno
DIBRIS, University of Genova, Italy
giorgio.delzanno@unige.it

Stefano Rebora
Flairbit s.r.l, Genova, Italy
stefano.rebora@flairbit.io

Matteo Rulli
Flairbit s.r.l, Genova, Italy
matteo.rulli@flairbit.io

Abstract

The Internet of Things (IoT) has created new and challenging opportunities for Data Analytics. IoT represents an infinitive source of massive and heterogeneous data, whose real-time processing is an increasingly important issue. Real-time Data Stream Processing is a natural answer for the majority of the goals of IoT platforms, but it has to deal with the highly variable and dynamic IoT environment. IoT applications usually consist of multiple technological layers connecting 'things' to a remote cloud core. These layers are generally grouped in two macro-levels: the edge-level (consisting of the devices at the boundary of the network near the devices that produce the data) and the core-level (consisting of the remote cloud components of the application). Real-time Data Stream Processing has to cope with a wide variety of technologies, devices and requirements that vary depending on the two IoT application levels. The aim of this work is to propose an adaptive microservices architecture for an IoT platform able to integrate real-time stream processing functionalities in a dynamic and flexible way, with the goal of covering the different real-time processing requirements that exist among the different levels of an IoT application. The proposal has been formulated for extending Senseioty, a proprietary IoT platform developed by FlairBit S.r.l., but it can easily be integrated in any other IoT platform. A preliminary prototype has been implemented as proof of concept of the feasibility and benefits of the proposed architecture.

2012 ACM Subject Classification Computer systems organization → Cloud computing

Keywords and phrases Cloud Computing, Service Oriented Computing, Internet of Things, Real-time Stream Processing, Query Languages

Digital Object Identifier 10.4230/OASIcs.Gabbrielli.2020.12

1 Introduction

Nowadays, with the rise of IoT, we have at our disposal a wide variety of smart devices able to constantly produce large volumes of data at an unprecedented speed. Sensors, smartphones and any other sort of IoT devices are able to measure an incredible range of parameters, such as temperature, position, motion, health indicators an so forth. More and more frequently, the value of these data highly depends on the moment when they are processed and the value diminishes very fast with time: processing them shortly after they are produced becomes a crucial aspect. Indeed, the aim of Real-time Stream Processing is to query continuous data streams in order to extract insights and detect particular conditions as quickly as possible, allowing a timely reaction. Possible examples are the alert generation of a medical

© Luca Bixio, Giorgio Delzanno, Stefano Rebora, and Matteo Rulli;
licensed under Creative Commons License CC-BY
Recent Developments in the Design and Implementation of Programming Languages.
Editors: Frank S. de Boer and Jacopo Mauro; Article No. 12; pp. 12:1–12:20

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

device or the real-time monitoring of a production line. In Stream Processing, data are no more considered as static and persistent data stored in a database, but as continuous and potentially unbounded sequences of data elements (i.e. data streams) from which static queries (a.k.a. rules) continuously extract information. The systems that execute this processing phase in a very short time span (milliseconds or seconds) are defined real-time stream processing engines. The IoT world offers an infinite set of use cases where real-time stream processing functionalities can be applied, but IoT applications provide at the same time a heterogeneous environment with respect to requirements, devices and technologies. For these reasons, integrating real-time processing engines in IoT platforms becomes a challenging operation that requires special attention.

An IoT platform provides tools, technologies and capabilities for simplifying the development, provisioning and management of IoT applications. Real-time stream processing engines are an increasingly popular and relevant technology, which the majority of the platforms are integrating in order to provide all the functionalities required by modern IoT applications. Indeed, Real-Time Stream Processing plays a crucial role in different and common IoT application scenarios, for instance: Anomaly and fraud detection; Remote monitoring; Predictive Maintenance; Real-time analytics (Sentiment analysis, Sports analytics, etc.). When integrating real-time stream processing engines in IoT platforms, the main difficulties arise from the high heterogeneity and dynamicity of the requirements and technologies of common IoT applications. At high level, a general IoT application consists of the following layers: The sensors/actuators layer, which includes the IoT devices; The edge layer, which includes all the devices near the sensors/actuators-level. These edge devices usually play the role of gateways, enabling the collection and the transmission of data; The core/cloud layer, which includes all the core functionalities and services of the application; The application/presentation layer, which includes all the client applications that have access to the core functionalities and services. Integrating real-time stream processing capabilities in IoT platforms imposes to face the following three main aspects:

- Twofold level of applicability. It is required often to apply Real-Time Stream Processing at two different levels: at edge level and at core/cloud level. Both approaches offer different benefits but the great difference between the devices and resources at edge level and core level imposes also quite different requirements that affect the choice of the stream processing engines.
- Technological pluralism. Due to the previous point, a natural consequence is to introduce different stream processing engines in the IoT platform because one stream processing technology rarely covers the edge level and the cloud level requirements. Having different stream processing engines means having different processing models and languages that must be handled for implementing stream processing rules.
- Rules' dynamicity. Usually, real-time IoT stream processing rules are based on a dynamic lifecycle. In the majority of IoT use cases, the functionalities implemented by real-time stream processing rules can be temporary functionalities (that are executed on demand and then removed after a while) or long-running functionalities never modified (e.g. a remote monitoring process). Moreover, it is often required to deploy rules directly on edge devices for reducing the response latency time or applying some pre-filtering operations, but when the workload increases, a scalable approach may be more preferable. For all these reasons, rules should have the possibility to be dynamically reallocated on different stream processing engines.

Considering these aspects, the goal of this work is to propose an adaptive solution for integrating real-time stream processing functionalities into an IoT platform, Senseioty by Flairbit [21], able to satisfy the different requirements imposed by the edge level and the

cloud/level. Moreover, the proposal offers a dynamic mechanism for facilitating the dynamic management and relocation of stream processing rules, hiding at the same time the complexity introduced by the presence of different and heterogeneous stream processing engines. The innovative aspect of our solution, with respect to common IoT platforms, consists in limiting the expressive power for defining stream processing rules to a predefined set of templates, in favor of a much more flexible and dynamic deployment model. The proposal architecture has been designed following a microservices architectural pattern. Microservices are a natural and widely adopted solution for implementing software platforms. The majority of IoT platforms are based on a microservices approach, even when it is not explicitly mentioned. This happens because the microservices architectural style is a chameleonic style, which can be implemented in different ways. Indeed, several technologies exist for implementing microservices, but for our purposes we have selected a particular technology able to guarantee a significant level of flexibility and dynamicity.

Plan of the paper

In Section 2 we give an overview of the main features of the microservices architectural style and of a particular Java technology named OSGi, the main technology applied in Senseioty, the proprietary IoT platform developed by FlairBit. In Section 3 and 4 we present our proposal and a prototype implemented as a possible extension of Senseioty based on Siddhi and Apache Flink. In Section 5 we address some conclusions and future work.

Our Contribution to Maurizio Gabbrielli's Festschrift

The reason for submitting the present work to Maurizio Gabbrielli's Festschrift was to provide a contribution related to recent work done by Maurizio Gabbrielli on Service Oriented Computing and IoT, see e.g. [26]. Our work is quite related to interoperability issues for IoT systems. Indeed our aim is to improve service interoperability via the definition of a core query language that abstracts from common features of existing stream processing tools and that facilitates their integration within the same IoT platform. The resulting query language is equipped with a control mechanism fully supported by a general purpose framework such as OSGi and by meta-data specified in JSON.

2 The Microservices Architectural Style and Java OSGi

The microservices architectural style, see e.g. [25], is born to address the problems of the traditional monolithic approach. When you start to design and build a new application, the easiest and most natural approach is to imagine the application as unit composed by several components. The application is logically partitioned in modules and each one represents a functionality, however it is packaged and deployed as a unit. This monolithic approach is very simple and comes naturally. Indeed, all the IDE's are necessarily designed to build a single application and the deployment of a single unit is easy and fast. Also scaling the application is trivial because it requires only running multiple instances of the single unit. This approach initially works and apparently quite well: the question is what happens when the application starts to grow. When the number of functionalities increases and the application becomes bigger and bigger, the monolithic approach shows its natural limit with respect to human capacities. In a short while, the dimensions of the application are such that a single developer is unable to fully understand it and this leads to serious problems. For example, implementing a new functionality becomes harder and time consuming and fixing bug even worse. The

whole code is inevitably too complex; therefore adopting new frameworks and technologies is discouraged. In addition, the deployment and the start-up time are obviously negative affected by the huge size of the application. The main consequence is the slowdown of the entire development phase and any attempts of continuous integration and other agile practises fails. Moreover, all the components run in the same process or environment causing serious problems of reliability: a failure or a bug in a single component can compromise the entire application. In few words, the overall complexity of a huge monolith overwhelms the developers. The microservices architectural style was created specifically to address this kind of problems and to tackle the complexity. The book [24] describes a three-dimensional scale model known as scale-cube: Horizontal Scaling (running multiple instances behind a load-balancer); Functional Scaling (decomposing a monolithic application into a set of services, each one implementing a specific set of functionalities); Scaling of Data Partitioning (data are partitioned among the several instances and each copy of the application). These concepts are strongly connected to the idea of Single Responsibility Principle (SRP) [31]. The functionalities are exposed through an interface, often a REST API, and can be consumed by other services increasing the composability. The communication between microservices can be indifferently implemented by synchronous or asynchronous communication protocol and each microservice can be implemented with a different and ad-hoc technology. Moreover, each microservice has its own database rather than sharing a single database schema with other services. This makes a microservice an actual independently deployable and loosely coupled component. In this setting communication is provided via an API Gateway [28]. The API Gateway is similar to the Facade pattern from object-oriented design: it is a software component able to hide and encapsulate the internal system details and architecture, providing a tailored API to the client. It is responsible for handling the client's requests and consequently invoking different microservices using different communication protocol, finally aggregating the results. Microservices architectures often provide a service discovery mechanism typically implemented via a shared registry which is basically a database that contains the network locations of the associated service instances. Two important requirements for a service registry are to be highly available and up to date, thus it often consists in a cluster of servers that use a replication protocol to maintain consistency. One of the main principles at the heart of the microservices architecture is the decentralization of data management: each microservice encapsulates its own database and data are accessible only by its API. This approach makes microservices loosely coupled, independently deployable and able to evolve independently from each other. In addition, each microservice can adopt different database technologies depending on its specific requirements, for example for some use cases a NoSQL database may be more appropriate than a traditional SQL database or vice versa. Therefore, the resulting architecture often uses a mixture of SQL and NoSQL databases, leading to the so-called polyglot persistence architecture. In this setting, data consistency is often achieved via an event-driven architecture [29]. A message broker is introduced into the system and each microservice publishes an event whenever a business entity is modified. Other microservices subscribe to these events, update their entities and may publish other events in their turn. The event-driven architecture is also a solution for the problem of queries that have to retrieve and aggregate data from multiple microservices. Indeed, some microservices can subscribe to event channels and maintain materialized views that pre-join data owned by multiple microservices. Each time a microservice publishes a new event, the view is updated. The last key aspect of the microservices architecture is how a microservices application is actually deployed. Three main different deployment patterns exist [30]: Multiple Service Instances per Host Pattern; Service Instance per Host Pattern sub-divided in (Service Instance per

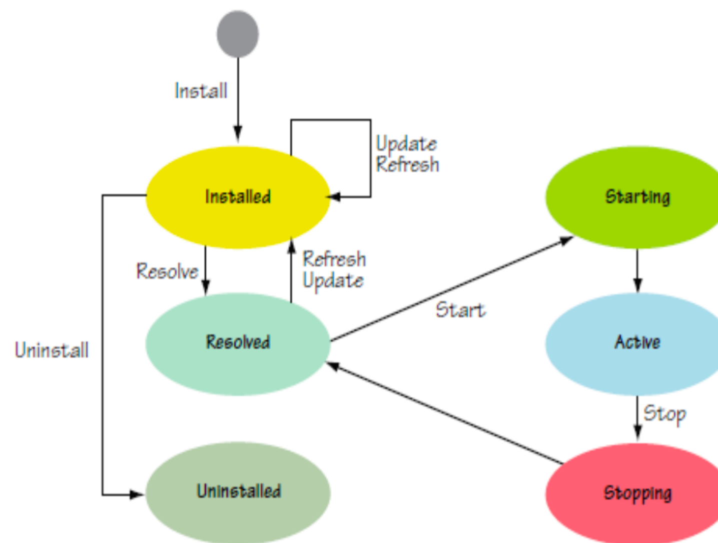


Figure 1 Bundle life cycle.

Virtual Machine Pattern/Container Pattern); Serverless Deployment Pattern (e.g. AWS Lambda; Google Cloud Functions; Azure Functions). Java OSGi OSGi [14] consists of a set of specifications established by the OSGi Alliance. The OSGi architecture [15] appears as a layered model. The bundles are the modules implemented by the developers. A bundle is basically a standard JAR file enriched by some metadata contained in a manifest [27]. The manifest and its metadata make possible to extend the standard Java access modifiers (public, private, protected, and package private). A bundle can explicitly declare on which external packages it depends and which contained packages are externally visible, meaning that the public classes inside a bundle JAR file are not necessarily externally accessible. The module, life cycle and services layer constitute the core of the OSGi framework: The module layer defines the concept of bundle and how a bundle can import and export code; The life cycle layer provides the API for the execution-time module management; The service layer provides a publish-find-bind model for plain old Java objects implementing services able to connect dynamically the bundles. Finally, the security layer is an optional layer, which provides the infrastructure to deploy and manage applications that must run in fine-grained controlled environments, and the execution environment defines the methods and classes that are available in a specific platform. A Bundle object logically represents a bundle into OSGi framework and it defines the API to manage the bundle's lifecycle. The BundleContext represents the execution context associated to the bundle. It basically offers some methods for the deployment and lifecycle management of a bundle and other methods for enabling the bundle interaction via services. It is interesting to notice that the BundleContext interface has methods to register BundleListener and FrameworkListener objects for receiving event notifications. These methods allow to monitor and to react to execution-time changes into the framework and to take advantage of the flexible dynamism of OSGi bundles. Finally, the BundleActivator offers a hook into the lifecycle layer and the ability to customize the code that must be executed when a bundle is started or stopped. The class implementing the BundleActivator inside a bundle is specified adding the Bundle-Activator header to the bundle manifest. As shown in Figure 1, firstly, a bundle must be installed into OSGi framework. Installing a bundle into the framework is a persistent operation that consists in

providing a location of the bundle JAR file to be installed (typically a URL) and then saving a copy of the JAR file in a private area of the framework called bundle cache. Then, the transition from installed to resolved state is the transition that represents the automated dependency resolution. This transition can happen implicitly when the bundle is started or when another bundle tries to load a class from it, but it can also be explicitly triggered using specific methods of lifecycle APIs. A bundle can be started after being installed into the framework. The bundle is started through the Bundle interface and the operations executed during this phase (e.g. operations of initialization) are defined by an implementation of the BundleActivator. The transition from the starting to the active state is always implicit. A bundle is in the starting state while its BundleActivator's start() method executes. If the execution of the start() method terminates successfully, the bundle's state transitions to active, otherwise it transitions back to resolved. Similarly, an active bundle can be stopped and an installed bundle can be uninstalled. When uninstalling an active bundle, the framework automatically stops the bundle first. The bundle's state goes to resolved and then to installed state before uninstalling the bundle. The OSGi environment is dynamic and flexible and it allows to update a bundle with a newer version even at execution-time. This kind of operation is quite simple for self-contained bundles but things get complicated when other bundles depend on the bundle being updated. The same problem exists when uninstalling a bundle, both the updating and uninstalling operations can cause a cascading disruption of all the other bundles depending on it. This happens because, in case of updating, dependent bundles have potentially loaded classes from the old version of the bundle, causing a mixture of loaded old classes and new ones. The same inconsistent situation occurs when a dependent bundle cannot load classes from a bundle that has been uninstalled. The solution for this scenario is to execute the updating and uninstalling operation as a two-step operation: the first step prepares the operation; the second one performs a refreshing. The refreshing allows to recalculate the dependencies of all the involved bundles, providing a control of the moment when the changeover to the new bundle version or removal of a bundle is triggered for updates and uninstalls. Therefore, each time an update is executed, in the first step the new version of the bundle is introduced and two versions of the bundle coexist at the same time. Similarly, for uninstalling operations, the bundle is removed from the installed list of bundles, but it is not removed from memory. In both cases, the dependent bundles continue to load classes from the older or removed bundle. Finally, a refreshing step is triggered and all the dependencies are computed and resolved again. In conclusion, the lifecycle layer provides powerful functionalities for handling, monitoring and reacting to the dynamic lifecycle of bundles. The next section presents the last but not the least layer of the OSGi framework: the service layer.

Java OSGi and Microservices OSGi allows the combination of microservices and nanoservices. Leveraging the OSGi service layer, it is possible to implement microservices internally composed by tiny nanoservices. The final resulting architecture will be composed by a set of microservices, each one running on its own OSGi runtime and communicating remotely with the other microservices. Internally, a single microservice may be implemented as a combination of multiple nanoservices that communicate locally as a simple method invocations. Secondly, OSGi offers an in-built dynamic nature. Developing microservices using OSGi means having a rich and robust set of functionalities specifically implemented for handling services with a dynamic lifecycle. Even more, it makes the microservices able to be aware of their dynamic lifecycle and react consequently to the dynamic changes. The OSGi runtime and its service layer were built upon this fluidity; therefore, the resulting microservices are intrinsically dynamicity-aware microservices. Last but not the list, OSGi makes the

microservices architecture a more flexible architecture with respect to service decomposition. One drawback of microservices is the difficulty of performing changes or refactoring operations that span multiple microservices. When designing a microservices architecture, understanding exactly how all the functionalities should be decomposed into multiple small microservices is an extremely difficult task, which requires defining explicit boundaries between services and establishing once for all the communication protocols that will be adopted. If in future, the chosen service decomposition strategy turns to be no more the best choice or only a modification involving the movement of one or multiple functionalities among different microservices is required, performing this change may become extremely difficult because of the presence of already defined microservices boundaries and communication protocols. On the contrary, the OSGi Remote Services offers a flexible approach for defining the microservices boundaries. Indeed, a set of functionalities implemented by an OSGi service can be easily moved from a local runtime to a remote one without any impact on other services. Therefore, an already defined microservices decomposition strategy can be modified by reallocating the functionalities offered by services at any time. For example, one of two OSGi services previously designed for being on the same runtime (i.e. within the same microservice boundary) can be moved on another remote OSGi runtime without any difficult changes. The interaction between a distribution provider and a distribution consumer in OSGi takes place always as the two entities were on the same and local runtime: the distribution manager provided by the Remote Services specification transparently handles the remote communication. Moreover, this remote communication is completely independent from the communication protocols; therefore, any previous choice is not binding at all. In conclusion, OSGi enriches the microservices architecture with new and powerful dynamic properties and a flexible model able to support elastic and protocol-independent service boundaries. Moreover, it provides a level of service granularity highly variable allowing the combination of microservices and nanoservices. The only but very relevant drawback of OSGi with respect to microservices architectural pattern is the complete cancellation of technological freedom that characterizes microservices. OSGi is a technology exclusively designed for Java and implementing a microservices architecture based on OSGi necessary requires to adopt Java for developing the microservices. This does not mean that a microservice implemented using OSGi cannot be integrated with other services implemented with different technology; an OSGi remote service, for example, can be exposed externally also for not-OSGi service consumers, loosing however all the OSGi service layer benefits. It actually means that if Java and OSGi are not widely adopted for implementing the majority of the microservices of the architecture, the OSGi additional features lose their effectiveness. OSGi represents also a very powerful and dynamic service-oriented platform due to the several features offered by its service layer [17].

Finally, the last relevant and powerful feature of the OSGi service layer is the flexibility offered by the Remote Services Specification [20]. The OSGi framework provides a local service registry for bundles to communicate through service objects, where a service is an object that one bundle registers and another bundle gets. However, the Remote Services Specification extends this behaviour in a very powerful and flexible manner, allowing the OSGi services to be exported remotely and independently from the communication protocols. The client-side distribution provider is able to discover remote endpoints and create proxies to these services, which it injects into the local OSGi service registry. The implementation of the discovery phase depends on the chosen distribution provider implementation (e.g. The Apache CXF Distributed OSGi [3] implementation provides discovery based on Apache Hadoop Zookeeper [9]). Another additional and powerful feature of OSGi Remote Services is

the ability to be independent from the underlying communication protocol adopted for the service exportation. A distribution provider may choose any number of ways to make the service available remotely. It can use various protocols (SOAP, REST, RMI, etc.), adopting a range of different security or authentication mechanisms and many different transport technologies (HTTP, JMS, P2P, etc.). The Remote Services specification offers a layer of indirection between the service provider and the distribution provider, leveraging the concepts of intents and configurations. They basically allow the service provider to specify just enough information to ensure that the service behaves as expected, then the task of the distribution provider is to optimize the communications for the environment in which they are deployed.

3 A Microservices Architecture for Adaptive Real-time IoT Stream Processing

Senseioty [21] is an IoT platform designed to accelerate the development of end-to-end solutions and verticals, revolving around the concept of insights-engineering, the seamless integration between data ingestion and distribution, data analytics and on-line data analysis. Senseioty is developed in Java as a set of highly cohesive OSGi microservices. Each Senseioty microservice can either be used together with Amazon AWS or Microsoft Azure managed services or deployed on private cloud or on-premises to accelerate and deliver full-fledged end-to-end IoT solutions for the customer. Senseioty features also an SDK to implement rapid verticalizations on top of its rich set of JSON RESTful APIs and analytics services. Senseioty automates the integration of IoT operational data with analytics workflows and provides a common programming model and semantics to ensure data quality, simplify data distribution and storage and enforce data access policies and data privacy. Senseioty is natively integrated with both Microsoft Azure IoT and Amazon AWS IoT and it can also operate on private and hybrid cloud to provide the maximum flexibility in terms of cloud deployment models Senseioty offers a wide variety of interesting and flexible functionalities that should give an idea of the flexibility and interoperability offered by a microservices architecture in the IoT context: Single-sign-on services for user and devices along with user management; Access policies microservice to protect resources and devices against unauthorized access and to guarantee data privacy; Flexible and unified programming interface to manage and provision connected devices.; Persistence of time series in Apache Cassandra clusters; Powerful and flexible way to communicate different microservices together and to implement remote services discovery based on the OSGi Remote Service specification; Senseioty microservices can be deployed at the three different layers of the hybrid-cloud stack (cloud layer, on-premises layer and edge layer); Deep-learning workflows based on neural networks and to push them on connected devices, in order to run analytics workflow on the edge. Senseioty integrates Apache Spark, a powerful Distributed Data Stream Processors engine to analyse data stream in real-time and provide on-line data analytics on the cloud, leveraging both neural network and statistical learning techniques to analyse data. Finally, Senseioty provides a rich set of IoT connectors to integrate standard and custom IoT protocols and devices.

FlairBit extensively adopts in its platform Apache Karaf [6], a powerful and enterprise ready applications runtime built on top two famous OSGi implementation (Apache Felix and Eclipse Equinox) that offers some additional and useful functionalities, such as the concept of feature.

One problem exposed by FlairBit, which is usually a problem common to the majority of IoT platform, is to have two different levels of data stream processing: The edge level; The core/cloud level. The two levels offer different benefits but impose quite different requirements.

The term “edge” in IoT platforms generally means the location at the boundary of the network near the devices that produce the data. Edge devices are usually quite simple devices that play the role of gateways, enabling the collection and the transmission of data. However, modern edge devices can also offer enough computational resources to enable more complex functionalities, such as pre-processing, monitoring or pre-filtering. Moving the stream processing elaboration directly on the edge of the IoT platform takes the name of Edge analytics and the consequent benefits are quite notable: Lowest possible latency, having a stream processing unit deployed directly on an edge devices makes possible to respond quickly to events produced locally, avoiding to send data to the remote cloud/core of the platform over the network; Improved reliability, moving the stream processing rules on the edge allows the edge devices to operate even when they lose the connection with the core platform; Reduced operational costs, pre-processing and pre-filtering data directly on the edge makes possible to save bandwidth, cloud storage and computational resources consequently lowering operational costs.

On the other hand, edge analytics imposes some stringent requirements in term of computational power. The modern edge devices are becoming more and more powerful, but the computational resources offered by this kind of devices are limited. Therefore, the technologies installed on the edge must be lightweight and it is likely that they are quite different technologies from those applied on the core platform. Indeed, the stream processing units on the edge usually deals with simple filtering rules and streams of data restricted to the local sensors or devices, without the need of scaling the stream processing job across multiple machines.

On the core platform, the context is completely different. In this scenario, the stream processing engines must be able to deal with different workloads and the computational resources abound. They must be able to scale the computation across a cluster of machines in order to handle large volume of data and more intensive tasks, for example joining and aggregating different events from different streams of data. Therefore, the need of scaling capabilities overcomes the limit of the computational resources. The main goals of the proposed extension of the Senseioty architecture are as follows: 1) Providing adaptivity, meaning that the stream processing units can be indifferently allocated on the edge or on the core and moved around. This makes possible to cover the two different levels of data stream processing, the edge level and the core/cloud level, and exploiting all their different benefits. 2) Providing flexibility, allowing a punctual and on-demand deployment of the stream processing units. The user or the client application/service defines when and where allocating, starting, stopping and deallocating the stream processing rules. 3) Providing a set of portable and composable rules that can be defined in a standard way and then automatically deployed on different stream processing engines without depending on their own languages and models. The rules can be combined together, in order to apply a sort of stream processing pipeline. The rules are not only dynamically manageable, but composable and engine-independent. The reference structure of the resulting architecture is shown in Fig. 2. There are two main components in our architecture:

- The proxy μ-service: the entry point of the architecture, offering a RESTful API for installing, uninstalling, starting, stopping and moving stream processing rules on demand.
- The adapter μ-service. It is responsible for physically executing the functionalities offered by the proxy μ-service, interacting with the different stream processing engines available on the edge and on the core of the architecture.

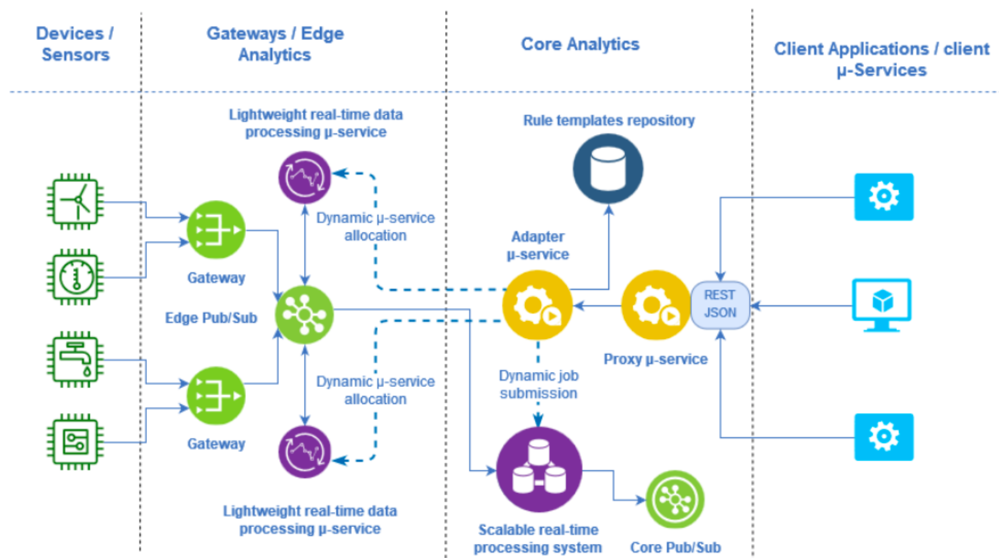


Figure 2 Reference Architecture.

The proxy μ-service represents the entry point of the architecture. It offers a RESTful JSON interface, a standard choice in microservices architecture (and it is usually adopted in Senseioty) in order to offer a solution as much compatible and reusable as possible. The REST API offers the following functionalities:

URL Method	Request Body	Response Body
/api/install POST	JSON installation object	JSON jobinfo object
/api/uninstall POST	JSON jobinfo object	JSON jobinfo object
/api/start POST	JSON jobinfo object	JSON jobinfo object
/api/stop POST	JSON jobinfo object	JSON jobinfo object
/api/move POST	JSON relocation object	JSON jobinfo object

The method *install* installs the rule on the required resource and engine defined by the json installation object. The method *uninstall* uninstalls the rule identified by the jobinfo request object. The method *start* runs the rule identified by the jobinfo request object. The method *stop* stops the execution of the rule identified by the jobinfo request object. The method *move* moves the rule identified by the jobinfo request object to the target runtime defined by the relocation object. Interaction with the proxy μ-service is carried out through the JSON objects of the following form:

```
// JSON INSTALLATION OBJECT
{
   "headers":{
      "runtime":<ENGINE>,
      "targetResource":<URL>,
      "jobType":<JOB_TYPE>
   },
   "jobConfig":{
      "connectors":{
         "inputEndpoint":<STRING>,
         "outputEndpoint":<STRING>
      },
```

```
      "jobProps":{
         "condition":< ">" | ">=" | "=" | "<" | "<=" >,
         "threshold":< INT | FLOAT | DOUBLE | STRING >,
         "fieldName":<STRING>,
         "fieldJsonPath":<JSON_PATH>
      }
   }
}
// JSON JOBINFO OBJECT
{
   "runtime":<ENGINE>,
   "jobId":<STRING>,
   "jobType":<JOB_TYPE>,
   "jobStatus":<INSTALLED|RUNNING|STOPPED|UNINSTALLED>,
   "configFileName":<STRING>
}
// JSON RELOCATION OBJECT
{
   "target_runtime":<ENGINE>,
   "targetResource":<URL>,
   "jobInfo":<JSON_JOBINFO_OBJECT>
}
```

The JSON installation object is the object that the client must provide to the proxy in order to describe the stream processing rule to be allocated. The headers field indicates the runtime engine that will execute the rule (the <ENGINE> value depends on the engines supported by the implementation), the target resource which is the machine on which allocating the rule (the value can be an URL or a simple ID, depending on the architecture implementation) and the job type, which indicates the kind of rule that the jobProps field contains. The implementation of the architecture supports a set of predefined rule templates identified by a unique name that must be inserted in the jobType field (e.g. single-filter, sum-aggregation, avg-aggregation, single-join etc.). Ideally, we would like to have a solution able to support any kind of rule expressible with a standard query stream language (e.g. the Stanford CQL [23]), but in practice this is not achievable because each stream processing engine has its own model and language with its own level of expressiveness. Therefore, it is extremely complicated to implement a compiler able to validate an arbitrary query and to compile and translate it to the model or language of the underlying stream processing engine. Considering this scenario, we provide an architecture able to support a set of predefined rule templates. A possible subset that should be compatible with the majority of stream processing engines includes (using a SQL like syntax):

- Filtering query (e.g. SELECT * FROM inputEvents WHERE field > threshold)
- Aggregation query over a window (e.g. SELECT SUM(field) FROM inputEvents[5 s])
- Joining query between two streams over windows (e.g. SELECT field1 field2 FROM stream1[1m] JOIN stream2[1m] ON stream1.field3 = stream2.field)

This is of course only a possible subset, which must be verified and extended considering the engines selected for the implementation.

The connectors field specifies the information needed for reading and writing the events consumed by the rule from/to a pub-sub broker. Again, the format of these fields depends on the pub-sub broker adopted in the implementation, but it in general the required parameters

are a simple URL or a queue or topic name. It is important to notice that the presence of two pub-sub brokers (one on the edge and one on the core) makes possible to combine and compose the rules in order to obtain stream processing pipelines. The jobProps field contains the parameters needed for allocating the stream processing rules. The format of this field depends on the rule template specified in the jobType field. The JSON jobInfo object is the object that contains all the necessary information that must be provided in order to perform all the other operations (starting, stopping, uninstalling or moving the rule) and it is created by the adapter μ-service and returned to the client by the proxy μ-service. It contains some information specified by the JSON installation object, with the addition of a jobId (a unique identifier for the installed rule instance), a jobStatus (it indicates the current execution status of the rule) and a configFileName (the name of the configuration file that represents the materialization of the jobConfigs field specified in the JSON installation object). The role of the configuration file will be clarified shortly when describing the adapter μ-service. The JSON relocation object is the object required for moving a rule from the current runtime to a target runtime. It contains the jobinfo object describing the selected rule and information regarding the target runtime (the engine and the resource URL or ID identifying the target machine).

The adapter μ-service is responsible for actually executing the functionalities offered by the proxy μ-service. It offers the following procedures: A procedure for installing a new rule; A procedure for starting/stopping/uninstalling an existing rule; A procedure for moving an existing rule from its current runtime to another one. During the installation procedure, the adapter μ-service translates the information received from the proxy μ-service into executable rules via a sort of parametrization as shown below: The adapter μ-service has access to a repository from where it can download the rule template corresponding to the jobType and runtime fields expressed in the JSON installation. The rule template is any sort of predefined executable file (for our purposes will be a JAR archive) that can be modified injecting a configuration file containing the rule parameters specified by the JSON installation object. Therefore, in case of rule installation, the adapter μ-service downloads the relative rule template, creates and injects the configuration file and then install the rule on the target runtime. If the target runtime is a distributed stream processing engine, the executable template is actually an executable job that is submitted to the cluster manger. If the target runtime is a lightweight and non-distributed stream processing engine for the edge, the rule template is actually an independent μ-service that is installed on the target machine on the edge. Finally, the adapter μ-service creates the JSON jobinfo object with the necessary information that will be returned to the client. In case of starting, stopping and uninstalling operations, the adapter μ-service acts always depending on the runtime engine associated to the rule, as shown below.

In case of distributed stream processing engine, it communicates with the cluster manager for executing the required operation. On the other hand, in case of lightweight stream processing μ-service on the edge, the implementation must provide a mechanism to interact dynamically with target runtime. It is intuitive to understand that a technology like OSGi and its bundle lifecycle naturally fits this scenario. OSGi is the main technology adopted in the prototype that will be described in the next section, but this architecture description section is intentionally lacking of technical and implementation details in order to be as much general as possible. The idea is to offer a guideline proposal that must be refined with respect to technologies selected for the implantation, which may be completely different from those selected for our prototype. Indeed, one benefits of a microservices architecture is the technological freedom.

Finally, for moving an existing rule across different runtimes the adapter μ-service acts as follows. First, it checks if the rule to be moved is running and eventually it stops its execution. Secondly, it uninstalls the current rule, it downloads the new template for the new target runtime and it injects the previous configuration file. Finally, it installs the new rule on the new target runtime, starting the execution of the new rule if it was previously running. In our architecture we introduce two pub/sub brokers. Having two event dispatcher systems in the architecture, one for edge level and another one for the cloud/core level, makes possible to implement composable stream processing rules. Indeed, the connectors field in the JSON installation object allows to specify the queue or topic names from where reading events and where writing the output events. This means that any rule can be concatenated with other rules in order to implement a stream processing pipeline. For example, two filtering rules can be combined on the same edge-device leveraging the edge pub/sub broker in order to create a two-step filter. Moreover, a pre-filtering rule on the edge can be applied on top of an aggregation rule executed at cloud/core level in order to reduce the amount of data sent over the network.

The last aspect to consider is the client application layer. As already explained, the proxy μ-service offers a simple RESTful JSON API accessible from any kind of client. For this reason, the functionalities offered by the API can be employed by other μ-services in the context of a larger platform (e.g. Senseioty) and combined with other additional functionalities (e.g. the authentication and authorization μ-services offered by Senseioty). Moreover, it is possible to provide a web interface that lets a user to interact directly with the proxy μ-service, defining and managing the rules. The API offers all the functionalities needed for implementing an adaptive monitoring rule relocation procedure. The only prerequisites are: Having access to a stream of events logging statistics about the performance and workload of the edge-devices; Having the possibility to store and update the information mapping the rules (i.e. the JSON jobInfo objects) to the IDs of the edge devices that are executing the rules.

4 Prototype Implementation

In order to implement a prototype of the proposed architecture we considered a lightweight stream processing engine for edge analytics called Siddhi [22]. Siddhi is an effecting streaming processing engine that provides an SQL-like stream language with a rich expressive power. It allows any sort of stateful and stateless operation, timing and counting windows, aggregation and join functions. It also supports different event formats (JSON, XML, etc.) for specifying event patterns for complex event processing (CEP). It provides a rich set of external event source integration, such as Kafka, MQTT, RabbitMQ and other brokers and provides a lightweight runtime compatible, e.g., with Android devices. The Siddhi libraries were transformed and wrapped into well-defined OSGi bundles. The Senseioty SDK provides some Java project templates explicitly configured for applying OSGi specific tools (e.g. Bnd tools [10]) able to create a JAR with OSGi meta data (i.e. a bundle) based on instructions and the information in the class files. A feature for the Karaf runtime, collecting all the bundles needed by Siddhi as dependencies, was created. The Senseioty SDK offers some functionalities able to discover all the dependencies and transitive dependencies required by a bundle and then to materialize them in the form of a Karaf feature. Therefore, the provisioning phase of a Siddhi application on a Karaf runtime (by provisioning application, it means install all modules, configuration, and transitive applications) requires now only a simple and automatic feature installation. Although this step required a lot of technical passages, a

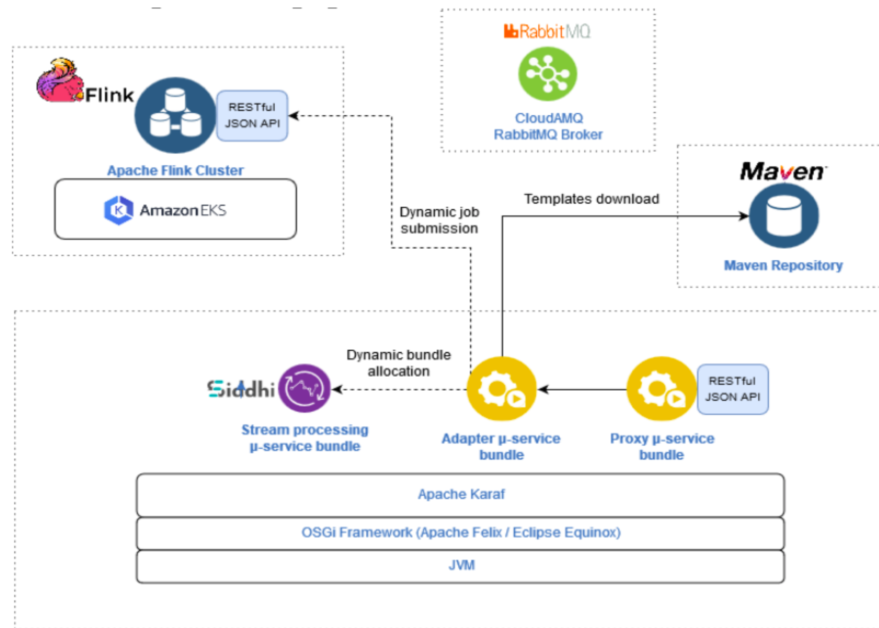


Figure 3 Prototype Structure.

detailed description is beyond the scope of the paper. Once obtained a fully OSGi-compliant stream processing engine for edge analytics purposes, the second step consisted in exploring and selecting another engine able to scale across a cluster of machines for core/cloud analytics purposes. During this phase, an analysis and some implementation of spike test programs were performed for the following stream processing technologies: Ignite [5]; Samza [7]; Flink [4]; Storm [8]; Streams [11]. Apache Flink turned out to be the most flexible available solution. It has a rich and complete API that follows a declarative model very similar to the Spark Streaming one and it has also a powerful additional library for complex event processing for specifying patterns of events. Moreover, it has a rich set of out-of-the-box external source connectors, a flexible resource allocation model based on slots independent from the number of CPU cores, and it is very easy to deploy a Flink cluster on Kubernetes. Based on all the above considerations, the structure of the implemented prototype is shown in Fig. 3. The proxy and the adapter μ-services are implemented as OSGi-bundles deployed on a Karaf runtime. The proxy μ-service offers a RESTful JSON API with the following functionalities. First, it provides an installation functionality for installing a filtering rule for events in JSON format. The rule can be indifferently instantiated as an independent Siddhi μ-service (implemented in the form of an OSGi bundle) or deployed as a distributed job on a Flink cluster. It also provides a starting, stopping and uninstalling functionalities for removing or handling the rule execution, and a moving functionality for relocating a rule from a Siddhi runtime to a Flink runtime or vice versa. The RESTful API was implemented using Apache CXF [2], an open-source and fully featured Web services framework. In this preliminary implementation, the runtime supported are Siddhi and Apache Flink and only one rule type is available: a threshold filter for events in JSON format. The client can specify a filtering rule defining the following jobProps in the JSON installation object:

```
// JSON INSTALLATION OBJECT

{ ...
      "jobProps":{
         "condition":< ">" | ">=" | "=" | "<" | "<=" >,
         "threshold":< INT | FLOAT | DOUBLE | STRING >,
         "fieldName":<STRING>,
         "fieldJsonPath":<JSON_PATH>
      }
   }
}
```

The prototype supports one rule type: a threshold filter for events in JSON format. The parameters specified by this rule type will be injected into two different rule templates that are implemented using the model and libraries provided by Siddhi and Flink. In practice, the rule parameters can be instantiated in two different rule templates:

```
// PROTOTYPE INSTALLATION ADAPTER SERVICE PROCEDURE
private bundleContext;

Install_rule (JsonInstallationObj req) {
   // Get and configure the right template
   mavenUrl = get_template_maven_url (req.headers.runtime, req.headers.jobType)
   ruleTemplate = download_from_maven_repo(mavenUrl)
   configurationFile = create_config_file(req.headers, req.jobConfig)
   configFileName = save(configurationFile)
   deployableRule = inject_config_file(ruleTemplate, configurationFile)

   // Install the rule as an independent  Siddhi service
   if (req.headers.runtime == SIDDHI)
      job_id = install_OSGi_bundle ( bundleContext, deployableRule)

   // Submit the rule to the remote Flink Cluster
   if (req.headers.runtime == FLINK)
      job_id = submit_to_cluster_manager (deployableRule)

   jobInfo = new JobInfo(runtime, jobId, jobType,
    status.INSTALLED, configurationFileName )
   return jobInfo
 }
```

In the form of an OSGi bundle (i.e. a μ-service) encapsulating a Siddhi runtime executing the filtering rule; In the form of a Flink job, which can be submitted to a Flink cluster. In this preliminary version of the prototype, the Siddhi bundles are installed and executed on the same OSGi runtime of the proxy and adapter μ-service. The remote installation on an edge-device can be easily integrated in future. Instead, the Flink runtime is installed on a remote Kubernetes cluster on the Amazon EKS service. The two rule templates previously cited are implemented in the form of a JAR file. Both templates are stored as Maven artifact into a Maven repository. Maven [11] is a tool used for building and managing any Java-based project and a Maven repository is basically a local or remote directory where Maven artifacts are stored. A Maven artifact is something that is either produced or used by a project (e.g. JARs, source, binary distributions, WARs etc.). In this case, both templates are implemented as JAR files. In order to download a Maven archetype from a Maven

repository, an OSGi bundle (i.e. the adapter μ-service) needs only to specify a Maven URL identifying the artifact. Then the URL resolution and the JAR download is handled by Pax URL [12], a set of URL handlers targeting the OSGi URL Handler Service. This mechanism is applied by the adapter μ-service for downloading the rule template for installing the rule on the required stream processing engine. The template to be download (and its relative Maven URL) depends on the jobType and runtime fields specified in the JSON installation object. Therefore, the adapter μ-service must have some predefined information that bind a Maven URL to a specific jobType and runtime. In this preliminary implementation, the above mentioned information are stored in memory into a simple hashTable, but for real purposes a simple database is required. The adapter μ-service provides the implementation of the procedures for installing, starting, stopping, uninstalling and moving the rules and it is responsible for injecting the rule parameters into the two different templates previously cited. When the adapter μ-service has to install a new rule, considering the jobType (in this case there is only one jobType: a filter) and the runtime (Siddhi or Flink) specified by the JSON installation object, it downloads the corresponding JAR file template from the Maven repository. Once obtained, the adapter μ-service translates the jobProps in a configuration file that is injected into the JAR template file. At this point, depending on the runtime chosen, the template rule is installed in two different ways. In case of a Flink job, the JAR template is sent to the Flink cluster manager using a REST API offered directly by Flink. On the other hand, in case of an OSGi bundle implementing the Siddhi filtering application, the bundle is installed on the Karaf runtime using the OSGi methods offered by the lifecycle layer. In this preliminary prototype, for the sake of simplicity, the OSGi bundle is installed on the same runtime of the proxy and adapter μ-service, but actually it should be installed on a remote runtime (i.e. a gateway device) on the edge of the IoT platform. Once the required rule is correctly installed on the target runtime, the adapter μ-service creates a JSON jobInfo object collecting all the relevant information about the just installed rule. In particular, it keeps trace of a jobId (corresponding to a bundle id for a Siddhi rule and to a jobId for Flink rule) and a configFileName (corresponding to a unique name of the generated configuration file, useful for reusing the file when moving the rule for one runtime to another). For all the other operations (starting, stopping, uninstalling and moving), the adapter μ-service uses the information provided by the jobInfo object and the methods offered by the OSGi lifecycle layer or the Flink REST API. The Siddhi OSGi bundles are installed on the same runtime of the proxy and adapter μ-service, but actually they should be installed on a remote runtime (i.e. a gateway device) on the edge of the IoT platform. This behaviour has been successfully implemented in Senseioty by FlairBit, which has extended the OSGi functionalities for communicating with remote runtime and it can be easily integrated in this prototype implementation in future. In practice, a remote OSGi runtime is connected to the core platform through two communication channels. A bidirectional channel used for communicating configurations options and statements. In this scenario, the adapter μ-service would use this channel to notify the target runtime about downloading the required bundle rule: it requires only a symbolic ID or URL to identify the target runtime. Possible communication protocols adopted for this channel are MQTT or TCP. A one-directional channel used by the remote OSGi runtime for download a remote resource, in this case the bundle rule notified by the adapter μ-service. A possible example of communication protocol adopted for this channel is FTP. This communication mechanism can be used by the adapter μ-service for executing all the required interactions with a remote OSGi runtime (installing, starting, stopping and uninstalling a Siddhi bundle). Another relevant feature implemented by this prototype is the rule composability, meaning that multiple filtering

rules can be concatenated in order to obtain a multiple-step filtering pipeline. Indeed, the currently supported filtering rules are easily composable because they read and write events from a RabbitMQ broker. RabbitMQ [19] is an open source message broker supporting multiple messaging protocols and it was chosen for this prototype implementation because both Siddhi and Flink provide out-of-the box connectors for consuming and writing event from a RabbitMQ broker. More specifically, RabbitMQ is adopted in this prototype for handling streams of events in JSON format using the AMQP protocol [1]. The role of an AMQP messaging broker is to receive events from a publisher (event producer) and to route them to a consumer (an application that processes the event). The AMQP messaging broker model relies on two main components:

- *Exchanges*, which are components of the broker responsible for distributing message copies to queues using rules called bindings. There are different exchange types, depending on the binding rules that they apply. This prototype uses only exchanges of type direct, which delivers messages to queues based on a message routing key included when publishing an event.
- *Queues*, which are the component that collect the messages coming from exchanges. A consumer reads the events from a queue in order to process the messages.

Therefore, when specifying the jobConfigs field in the JSON installation object, a client must provide in the connectors field the information needed for reading and writing events from/to an AMPQ queue. More specifically, is necessary to specify the parameters in the connectors field: For specifying the input source for the event, the following information are needed:

- *inputEndPoint*: the URL for connecting to the RabbitMQ broker (it might be different from outputEndPoint).
- *inputExchange*: the name of the exchange from which the input queue will read the messages. If the exchange does not already exist, it is created automatically.
- *inputQueue*: the name of the queue that will be bind to the inputExchange. If the queue does not already exist, it is created automatically.
- *inputRoutingKey*: the routing key that is used for binding the inputExchange to the InputQueue.

On the other hand, for specifying the output source of events, these information are required:

- *outputEndPoint*: the URL for connecting to the RabbitMQ broker.
- *outputExchange*: the name of the exchange where to publishing the events. If the exchange does not already exist, it is created automatically.
- *outputRoutingKey*: the routing key that is included to the event when publishing it.

Leveraging these features, the prototype allows to create stream processing pipelines of arbitrary complex. For example, multiple Siddhi filters can be concatenated with other filters executed on Flink. In practice, there is the need of two message brokers: one for the edge level and one for the cloud/core level. This aspect makes possible to concatenate multiple edge rules without the need of sending events to a remote broker in the core of the platform, avoiding to introduce unnecessary latency. RabbitMQ may be a reasonable choice for the cloud/core level scenario, but for the edge level, the choice must be carefully evaluated for not overloading the edge/gateway devices. For FlairBit and Senseioty purposes, considering that the edge/gateway devices are provided with an OSGi runtime, it may be a reasonable choice to take advantage of the OSGi Event Admin Service [16]: an inter-bundle communication mechanism based on an event publish and subscribe model. This sort of OSGi message broker can be easily paired with the Remote Service functionalities in order to connect multiple OSGi runtimes. This solution makes possible to obtain a message broker at edge level, without the need of adding an external and additional technology. The drawback is that we have to develop a customized connector implementation for each stream processing engine, in order to consume events from the OSGi broker.

5 Related Work and Conclusions

In this paper we have proposed an adaptive solution for satisfying the dynamic and heterogeneous requirements that IoT platforms are inevitably facing. During the research and development path that led to our proposal, we investigated all the features of the microservices architectural pattern, with the aim of deeply understanding the level of flexibility and dynamicity that this approach is able to offer. OSGi turned out to be the perfect booster for those dynamic and flexible features that we were looking for. Then, on the basis of the industrial experience of FlairBit, we formulated a proposal architecture accompanied by a preliminary prototype implementation. Our solution meets the need to introduce different real-time stream processing technologies in IoT platforms, in order to offer streaming analytic functionalities on the different architectural levels of IoT applications. The innovative aspect resides in a limitation of the expressiveness power for defining stream processing rules, in favour of a much more flexible and dynamic deployment model. Streaming rules are restricted to a predefined and manageable set of templates, which allows to handle rules as resources dynamically allocable, composable and engine independent. These resources can be indifferently deployed at edge-level or core-level and moved around at any time.

Comparing our proposal with similar real-time streaming functionalities offered by the IoT platforms of Amazon, Azure and Google, the dynamic features of our solution can be potentially promising and innovative. Amazon offers AWS IoT Greengrass [16] as a solution for moving analytical functionalities directly on edge devices. It is basically a software that once installed on an edge device enables the device to run AWS Lambda functions locally. AWS Lambda enables to run code without provisioning or managing servers. They offer a great level of expressivity with respect to our proposal because they support function implemented with all the most common programming languages. However, AWS IoT Greengrass does not provide any functionality for dynamically moving the Lambda computation back and forth between the edge-level and cloud-level and it is bound to the Lambda execution model. It does not offer any integration with external stream processing engines, which on the other hand can be integrated in our solution as pluggable components as long as template implementations of the supported rule types are provided. Microsoft Azure offers similar functionalities with Azure Stream Analytics on IoT Edge [13]. It empowers developers to deploy near-real-time analytical intelligence, developed using Azure Stream Analytics, to IoT devices. The principle is the same of AWS IoT Greengrass: installing the Azure IoT Edge software we enable the edge devices to locally execute Azure Stream Analytics rules. Azure Stream Analytics is a real-time analytics and complex event-processing engine where streaming rules and jobs are defined using a simple SQL-based query language. Again, the power of expressiveness is much wider with respect to our proposal, but the resulting solution is inevitably bound to the only Azure Stream Analytics engine and no mechanisms for the dynamic relocation of rules between edge and cloud are provided. Finally, Google Cloud IoT [18] integrates the Apache Beam SDK [21], which provides a rich set of windowing and session analysis primitives. It offers a unified development model for defining and executing data processing pipelines across different stream processing engines, including Apache Flink, Apache Samza, Apache Spark and other engines. However, Apache Beam supports only scalable engines suitable for the core-cloud level and it is not designed for supporting edge analytics.

Concerning future research directions, one possibility is to improve the architecture by investigating possible solutions for simplifying the rules' definition. Our API requires to define a JSON object containing the rules' parameters, but for example a web interface or an

SDK similar to Apache Beam may offer a higher level approach. In this case, it is required to identify the right trade-off between the level of expressiveness offered by a possible unified model or language and the limits imposed by the presence of predefined rule types and templates. Another interesting point consists in integrating a monitoring μ-service in the application. This monitoring functionality, which is presented as possible application scenario of our proposal, can be formalized in more details in order to become an integral part of our solution. Providing an out-of-the-box monitoring behaviour can be a powerful additional feature useful in many IoT use cases.

References

1. AMQP protocol. `https://www.amqp.org/`.
2. Apache CXF. `http://cxf.apache.org/`.
3. Apache CXF Distributed OSGi. `https://cxf.apache.org/distributed-osgi.html`.
4. Apache flink. `https://flink.apache.org/`.
5. Apache ignite. `https://ignite.apache.org/`.
6. Apache karaf. `https://karaf.apache.org/`.
7. Apache samza. `http://samza.apache.org/`.
8. Apache storm. `https://storm.apache.org/`.
9. Apache zookeeper. `https://zookeeper.apache.org/`.
10. Bnd tools. `https://bnd.bndtools.org/`.
11. Kafka streams. `https://kafka.apache.org/documentation/streams/`.
12. Kubernetes. `https://kubernetes.io/`.
13. Maven. `https://maven.apache.org/`.
14. Osgi alliance. `https://www.osgi.org`.
15. Osgi architecture. `https://www.osgi.org/developer/architecture/`.
16. Osgi event admin service. `https://osgi.org/specification/osgi.cmpn/7.0.0/service.event.html`.
17. Osgi service layer. `https://osgi.org/specification/osgi.core/7.0.0/framework.service.html`.
18. Pax url. `https://ops4j1.jira.com/wiki/spaces/paxurl/overview`.
19. RabbitMQ. `https://www.rabbitmq.com/`.
20. Remote services specification. `https://osgi.org/specification/osgi.cmpn/7.0.0/service.remoteservices.html`.
21. Senseioty. `http://senseioty.com/`.
22. Siddhi streaming and complex event processing system. `https://siddhi.io/`.
23. B. Shivnath A. Arvind and W. Jennifer. The cql continuous query language: Semantic foundations and query execution. `http://ilpubs.stanford.edu:8090/758/1/2003-67.pdf`, 2003.
24. M. Abbott and M. Fisher. *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*. Addison-Wesley Professional, 2015.
25. M. Fowler and J. Lewis. Microservices - a definition of this new architectural term. `https://martinfowler.com/articles/microservices.html`, 2014.
26. M. Gabbrielli, S. Giallorenzo, I. Lanese, and S. P. Zingaro. A language-based approach for interoperability of iot platforms. In Tung Bui, editor, *51st Hawaii International Conference on System Sciences, HICSS 2018, Hilton Waikoloa Village, Hawaii, USA, January 3-6, 2018*, pages 1–10. ScholarSpace / AIS Electronic Library (AISeL), 2018.
27. R. Hall, K. Pauls, S. McCulloch, and D. Savage. *OSGi in Action, Creating Modular Applications in Java.* Manning, 2011.
28. C. Richardson. Building microservices: Using an api gateway. `https://www.nginx.com/blog/introduction-to-microservices/`, 2015.

29 C. Richardson. Event-driven data management for microservices. https://www.nginx.com/blog/event-driven-data-management-microservices/, 2015.

30 C. Richardson. Choosing a microservices deployment strategy. https://www.nginx.com/blog/deploying-microservices, 2016.

31 M. Robert. *Agile Software Development: Principles Patterns And Practices*. Pearson, 2003.