Report from Dagstuhl Seminar 21372

Behavioural Types: Bridging Theory and Practice

Edited by

Mariangiola Dezani[1], Roland Kuhn[2], Sam Lindley[3], and Alceste Scalas[4]

1 **University of Turin, IT,** dezani@di.unito.it
2 **Actyx AG – München, DE,** roland@actyx.io
3 **University of Edinburgh, GB,** sam.lindley@ed.ac.uk
4 **Technical University of Denmark – Lyngby, DK,** alcsc@dtu.dk

Abstract

Behavioural types specify the way in which software components interact with one another. Unlike data types (which describe the structure of data), behavioural types describe communication protocols, and their verification ensures that programs do not violate such protocols. The behavioural types research community has developed a flourishing literature on communication-centric programming, exploring many directions. One of the most studied behavioural type systems are session types, introduced by Honda et al. in the '90s, and awarded with prizes for their influence in the past 20 and 10 years (by the ESOP and POPL conferences, respectively). Other varieties of behavioural types include typestate systems, choreographies, and behavioural contracts; research on verification techniques covers the spectrum from fully static verification at compile-time to fully dynamic verification at run-time.

In the last decade, research on behavioural types has shifted emphasis towards practical applications, using both novel and existing programming languages (e.g., Java, Python, Go, C, Haskell, OCaml, Erlang, Scala, Rust). An earlier Dagstuhl Seminar, 17051 "Theory and Applications of Behavioural Types" (January 29–February 3, 2017), played an important role in coordinating this effort. Yet, despite the vibrant community and the stream of new results, the use of behavioural types for mainstream software development and verification remains limited.

This limitation is largely down to the rapid pace at which mainstream industrial practice for the design and development of concurrent and distributed systems evolves, often resulting in substantial divergence from academic research. In the absence of established tools to express communication protocols, widely used implementations concentrate solely on scalability and reliability. The flip side is that these systems are either overly loose, supporting any conceivable communication structure (via brokers), or overly restricted, supporting only simple request-response protocols (like HTTP or RPC).

In this seminar, experts from academia and industry explored together how best to bridge the gap between theory and mainstream practice. They tackled challenges that are fundamental in practical systems development, but are rarely or only partially addressed in the behavioural types literature – in particular, *failure handling*, *asynchronous communication*, and *dynamic reconfiguration*. Moreover they explored how the tools of behavioural types and programming languages theory (such as *linearity*, *gradual types*, and *dependent types*) can help to address these challenges.

Seminar September 12–17, 2021 – http://www.dagstuhl.de/21372
2012 ACM Subject Classification Theory of computation → Models of computation; Theory of computation → Process calculi; Theory of computation → Type structures
Keywords and phrases behavioural types, concurrency, programming languages, session types
Digital Object Identifier 10.4230/DagRep.11.8.52
Edited in cooperation with Jakobsen, Mathias

Behavioural Types: Bridging Theory and Practice, *Dagstuhl Reports*, Vol. 11, Issue 08, pp. 52–75
Editors: Mariangiola Dezani, Roland Kuhn, Sam Lindley, and Alceste Scalas

Dagstuhl Reports
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Executive Summary

Mariangiola Dezani (University of Turin, IT)
Roland Kuhn (Actyx AG – München, DE)
Sam Lindley (University of Edinburgh, GB)
Alceste Scalas (Technical University of Denmark – Lyngby, DK)

This seminar followed the earlier Dagstuhl Seminar 17051 "Theory and Applications of Behavioural Types". Whereas Seminar 17051 was quite broad, encompassing both theory and practice across a wide range of areas relating to behavioural types, this seminar was much more focused, concentrating on how best to enable the use of behavioural types for practical programming.

Initial preparations

We gathered initial lists of proposed talks and breakout topics prior to the start of the seminar via an online form. We added to these throughout the week. We scheduled talks and breakout groups daily depending on audience interest and participant availability. The first part of the week was primarily talks, with ample time for stimulating discussions; the second part included more time for breakout sessions.

Hybrid seminar logistics

Due to the SARS-CoV-2 pandemic, the seminar was organised in hybrid format, with both in-person and remote participants. As the virtual participants came from a wide range of time zones (from central US to Japan) we gave special consideration to the time slot 2pm–4pm CEST during which everyone could attend. Those in Europe and Japan were able to attend morning sessions and those in Europe and America to attend further afternoon sessions (and a special evening session on Monday).

In order to run a successful hybrid Dagstuhl seminar, we made essential use of the dedicated equipment available at Dagstuhl: a Zoom-based streaming setup, with multiple cameras and ceiling microphones in the seminar room. All talks were live-streamed to both virtual and in-person participants. Talks were recorded so that virtual participants from incompatible time zones could catch up, then deleted at the end of the week. Larger hybrid breakout sessions were held in the main seminar room, and smaller ones elsewhere using a more ad hoc setup.

Moreover, all participants (local and virtual) were invited to use Zulip (a chat application) to exchange messages and files, pose questions during presentations, and remain informed on the upcoming events, group activities, and schedule updates.

Activities and outcomes

Throughout the seminar, the participants gathered in focused breakout groups: the findings of the breakout groups are described in more detail elsewhere in the report. Here is a brief summary:

Typing non-channel-based models allowed researchers with a wide range of perspectives and backgrounds to exchange their views. A key observation was that modern concurrent systems that coordinate via streams of events are difficult to analyse and verify with using existing approaches, and new formalisms are needed.

Logic-based approaches reviewed the state of the art, and discussed new directions. One of the conclusions is that more research is needed to relate concurrent and distributed systems to a broader range of logics beyond classical and intuitionistic linear logic (which are the focus of most current publications).

Type-informed recovery strategies explored failure handling at different levels (from network to application), and summarised several open questions not addressed in existing work.

Session types with untrusted counter-parties focused on how to ensure that different processes interact under compatible protocols, establishing the beginning of new work on monitoring and adaptation.

Join patterns / synchronisation – the next generation collected a survey of various attempts to integrate join patterns in programming languages, and discussed why they have not yet become mainstream. The discussion highlighted the need for exploring the connections between join patterns and linear logic, and the use of the join calculus as a reference for new implementation attempts.

The participants of several breakout groups have agreed to continue their work and collaboration after the seminar.

In addition to these more structured breakout sessions there were further lively improvised meetings and discussions (especially after dinner) which are not summarised in the report.

Overall, we believe that the seminar activities were a success. Unfortunately the hybrid format did pose a barrier for remote participants, especially those in different time zones. But on the positive side, for many participants this was their first Dagstuhl seminar, and for the in-person participants it was their first in-person scientific gathering after many months of virtual events due to the SARS-CoV-2 pandemic: their feedback has been enthusiastic.

At the end of the seminar the participants agreed to remain in contact to continue the discussions, and foster new collaborations. There was strong enthusiasm for organising a follow-up Dagstuhl seminar in the future, perhaps taking place in about two years time. To enable future collaborations the participants:

1. created a GitHub organisation where all seminar participants (and other researchers invited later) can exchange references and materials;
2. agreed to use the seminar's Zulip chat (mentioned above) as a starting point to set up a more permanent solution for continuing the interactions and exchanges (e.g., a mailing list);
3. nominated four people who will propose a new seminar, building upon the results of this one.

2 Table of Contents

3 Overview of Talks

3.1 Session Logical Relations for Noninterference

Stephanie Balzer (Carnegie Mellon University – Pittsburgh, US)

Joint work of Stephanie Balzer, Farzaneh, Derakhshan, Limin Jia

In this talk I introduce the audience to linear session types through the lens of noninterference. Session types, as the types of message-passing concurrency, naturally capture what information is learned by the exchange of messages, facilitating the development of a flow-senstive information flow control (IFC) type system guaranteeing noninterference. Noninterference ensures that an observer (adversary) cannot infer any secrets from made observations. I will explain the key ideas underlying the development of the IFC type system as well as the construction of the logical relation conceived to prove noninterference. The type system is based on intuitionistic linear logic and enriched with possible worlds to impose invariants on run-time configurations of processes, leading to a stratification in line with the security lattice. The logical relation generalizes existing developments for session-typed languages to open configuration to allow for a more subtle statement of program equivalence.

3.2 Session Types for Runtime Verification

Christian Bartolo Burló (Gran Sasso Science Institute, IT)

Joint work of Christian Bartolo Burló, Adrian Francalanza, Alceste Scalas, Catia Trubiani, Emilio Tuosto
Main reference Christian Bartolo Burlò, Adrian Francalanza, Alceste Scalas: "On the Monitorability of Session Types, in Theory and Practice", in Proc. of the 35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference), LIPIcs, Vol. 194, pp. 20:1–20:30, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.
URL https://doi.org/10.4230/LIPIcs.ECOOP.2021.20

Communication is central to present day computation. The expected communication protocols between parties can be formalised as session types, serving as specifications which systems can be verified against. We present our work on the runtime verification of communicating systems using session types, where we investigate their *monitorability* qualities in [1] and augment them with probabilities in [2].

From the work in [1], we show that it is impossible to achieve both *sound* (*i.e.,* only flag ill-typed processes) and *complete* (*i.e.,* flag all ill-typed processes) monitors for verifying the interaction between black-box components. Correspondingly, we prove that our autogenerated session monitors are sound and *weakly*-complete: *i.e.*, the monitors get stuck upon certain violations to the session type. On the practical side, we present a Scala toolkit, `STMonitor` [3], for the automatic generation of session monitors following our formal model. These executable monitors can be used as proxies to instrument communication across black-box processes written in any programming language. We also present the results of a series of benchmarks, showing that the synthesised monitors only introduce limited overheads.

Finally, we present a tool-based methodology from [2] that extends `STMonitor` by synthesising monitors from *probabilistic session types*. These types have each choice point augmented with a probability distribution describing how often each choice should be taken.

The synthesised monitors infer the probabilistic behaviour of a system at runtime, and based solely on the evidence observed up to the current point of execution, issue warnings when the observed behaviour deviates from the one specified by the type.

References

1 Christian Bartolo Burlò, Adrian Francalanza, and Alceste Scalas. *On the Monitorability of Session Types, in Theory and Practice*. ECOOP, LIPIcs, vol. 194, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, pp. 20:1–20:30.

2 Christian Bartolo Burlò, Adrian Francalanza, Alceste Scalas, Catia Trubiani, and Emilio Tuosto. *Towards Probabilistic Session-Type Monitoring*. COORDINATION, Lecture Notes in Computer Science, vol. 12717, Springer, 2021, pp. 106–120.

3 Christian Bartolo Burlò, Adrian Francalanza, and Alceste Scalas. *On the Monitorability of Session Types, in Theory and Practice (Artifact)*. Dagstuhl Artifacts Ser. 7 (2021), no. 2, 02:1–02:3.

3.3 A Model of Actors and Grey Failures

Laura Bocchi (University of Kent – Canterbury, GB) and Laura Voinea (University of Kent – Canterbury, GB)

Joint work of Laura Bocchi, Laura VoineaJulien Lange, Simon Thompson

We report on ongoing work on defining a model of failures for distributed systems, as a first step towards better detection and recovery. Looking at failures along an unpredictability axis, at the two extremes of the spectrum we find fail-stop (a component either works correctly or stops, and this latter case can be recognised and thus dealt with) and byzantine failure (a component behaves arbitrarily). In practice, systems can experience behaviours that lie somewhere between these two extremes, and this is often called grey failure: the system appears to be functional, but its overall performance is degraded in some way that may anticipate the full, fail-stop, failure of the system or some of its components.

In the last decade, several kinds of grey failures have been studied, such as transient failures (e.g., a component is down at periodic intervals), partial failures (only some subcomponents are affected), and slowdowns [2].

The symptoms of a grey failure tend to be subtle and ambiguous, involve any layer of the stack, and be signalled by different parts of the system having different perceptions of the health of some component, i.e., differential observation [1].

We present a model of grey failures for actor-based systems. Our model of failures consists of three inter-dependent models: (1) an (actor-based) systems model based on a process calculus, (2) a "curse" model of injected failures, and (3) a model awareness that components of the system have of each other, based on monitoring.

In (2) each curse is analogous to a trace or test of the system. Interesting developments include:

- establishing links to probabilistic functions (e.g., modelling patterns of failure distributions in real systems) as well as generalising to symbolic notions of curse to make model checking more tractable.
- appropriate definition of quality of a diagnosis, such as soundness, completeness, and timeliness, with respect to the injected failures.
- appropriate definition of recovery.

References

1 Huang et al. *Gray Failure: The Achilles' Heel of Cloud-Scale Systems.* In Proc. HotOS. Association for Computing Machinery, Whistler, BC, Canada, 2017

2 Gunawi et al. *Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems.* ACM Trans. Storage, vol. 14, no. 3, 2018

3.4 Quantitative Types in Idris 2

Edwin Brady (University of St Andrews, GB)

Main reference Edwin C. Brady: "Idris 2: Quantitative Type Theory in Practice", in Proc. of the 35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference), LIPIcs, Vol. 194, pp. 9:1–9:26, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.
URL https://doi.org/10.4230/LIPIcs.ECOOP.2021.9

Dependent types allow us to express precisely what a function is intended to do. Recent work on Quantitative Type Theory (QTT) extends dependent type systems with linearity, also allowing precision in expressing when a function can run. This is promising, because it suggests the ability to design and reason about resource usage protocols, such as we might find in distributed and concurrent programming, where the state of a communication channel changes throughout program execution. As yet, however, there has not been a full-scale programming language with which to experiment with these ideas. Idris 2 is a new version of the dependently typed language Idris, with a new core language based on QTT, supporting linear and dependent types. This talk described Idris 2, and how QTT has influenced its design. I gave examples of the benefits of QTT in practice including: expressing which data is erased at run time, at the type level; and, resource tracking in the type system leading to type-safe concurrent programming with session types.

3.5 Global Types and Event Structure Semantics for Asynchronous Multiparty Sessions

Ilaria Castellani (INRIA – Sophia Antipolis, FR), Mariangiola Dezani (University of Turin, IT), and Paola Giannini

Main reference Ilaria Castellani, Mariangiola Dezani-Ciancaglini, Paola Giannini: "Global types and event structure semantics for asynchronous multiparty sessions", CoRR, Vol. abs/2102.00865, 2021.
URL https://arxiv.org/abs/2102.00865

We propose an interpretation of asynchronous multiparty sessions as Flow Event Structures. We also introduce a new notion of type for asynchronous multiparty sessions, ensuring the expected properties for sessions, including progress.

Our types, which reflect asynchrony more directly than standard global types and are more permissive, are themselves interpreted as Prime Event Structures.

The main result is that the Event Structure interpretation of a session is equivalent, when the session is typable, to the Event Structure interpretation of its type.

3.6 An Overview of Explicit Cancellation

Simon Fowler (University of Glasgow, GB)

Main reference Simon Fowler, Sam Lindley, J. Garrett Morris, Sára Decova: "Exceptional asynchronous session types: session types without tiers", Proc. ACM Program. Lang., Vol. 3(POPL), pp. 28:1–28:29, 2019.
URL https://doi.org/10.1145/3290341

Safely implementing behavioural types typically requires some form of linearity, typically in the form of a linear type system, in order to rule out errors such as using an endpoint more than once or failing to complete a set of actions.

Unfortunately, linear type systems are difficult to integrate with exceptions or failures, which are inevitable in real-world applications. This talk gives an overview of explicit cancellation as introduced by Mostrous & Vasconcelos in 2014, and how the idea has since been applied to support exception handling in functional languages with applications in web programming, graphical user interfaces, and actor systems.

3.7 The STARDUST project: Session Types for Reliable Distributed Systems

Simon Gay (University of Glasgow, GB)

Joint work of Simon Gay, Phil Trinder, Simon Fowler, Nobuko Yoshida, Laura Bocchi, Simon Thompson, Laura Voinea
URL https://epsrc-stardust.github.io

The STARDUST project (Session Types for Reliable Distributed Systems) is funded by the UK EPSRC (grants EP/T014512/1, EP/T014628/1, EP/T014709/1) from 1st October 2020 to 30th September 2024. It is a collaboration between the University of Glasgow, the University of Kent and Imperial College London. The key objective is to combine the communication-structuring mechanism of session types with the scalability and fault-tolerance of actor-based software architectures. The result will be a well-founded theory of reliable actor programming, supported by a collection of libraries and tools, and validated on a range of case studies. Key aims are to deliver tools that provide lightweight support for developers – e.g. warning of potential issues – and to allow developers to continue to use established idioms. By doing so we aim to deliver a step change in the engineering of reliable distributed software systems.

3.8 A Multiparty Session Typing Discipline for Fault-Tolerant Event-Driven Distributed Programming

Raymond Hu (Queen Mary University of London, GB)

Joint work of Malte Viering, Raymond Hu, Patrick Eugster, Lukasz Ziarek
Main reference Malte Viering, Raymond Hu, Patrick Eugster, Lukasz Ziarek: "A multiparty session typing discipline for fault-tolerant event-driven distributed programming", Proc. ACM Program. Lang., Vol. 5(OOPSLA), pp. 1–30, 2021.
URL https://doi.org/10.1145/3485501

This paper presents a formulation of multiparty session types (MPSTs) for practical fault-tolerant distributed programming. We tackle the challenges faced by session types in the context of distributed systems involving asynchronous and concurrent partial failures – such as supporting dynamic replacement of failed parties and retrying failed protocol segments in an ongoing multiparty session – in the presence of unreliable failure detection. Key to our approach is that we develop a novel model of event-driven concurrency for multiparty sessions. Inspired by real-world practices, it enables us to unify the session-typed handling of regular I/O events with failure handling and the combination of features needed to express practical fault-tolerant protocols. Moreover, the characteristics of our model allow us to prove a global progress property for well-typed processes engaged in multiple concurrent sessions, which does not hold in traditional MPST systems.

To demonstrate its practicality, we implement our framework as a toolchain and runtime for Scala, and use it to specify and implement a session-typed version of the cluster management system of the industrial-strength Apache Spark data analytics framework. Our session-typed cluster manager composes with other vanilla Spark components to give a functioning Spark runtime; e.g., it can execute existing third-party Spark applications without code modification. A performance evaluation using the TPC-H benchmark shows our prototype implementation incurs an average overhead below 10%.

3.9 Papaya: Global Typestate Analysis of Aliased Objects

Mathias Jakobsen (University of Glasgow, GB) and Ornela Dardha (University of Glasgow, GB)

Joint work of Mathias Jakobsen, Alice Ravier, Ornela Dardha
Main reference Mathias Jakobsen, Alice Ravier, Ornela Dardha: "Papaya: Global Typestate Analysis of Aliased Objects", in Proc. of the PPDP 2021: 23rd International Symposium on Principles and Practice of Declarative Programming, Tallinn, Estonia, September 6-8, 2021, pp. 19:1–19:13, ACM, 2021.
URL https://doi.org/10.1145/3479394.3479414

Typestates are state machines used in object-oriented programming to specify and verify correct order of method calls on an object. To avoid inconsistent object states, typestates systems often enforce linear typing, which eliminates – or at best limits – aliasing. However, aliasing is an important feature in programming, and the state-of-the-art on typestates is too restrictive if we want typestates to be adopted in real-world software systems.

In this talk, we present a type system for an object-oriented language with typestate annotations, which allows for unrestricted aliasing, and as opposed to previous approaches it does not require linearity constraints. The typestate analysis is global and tracks objects throughout the entire program graph, which ensures that well-typed programs conform to and complete the declared protocols.

3.10 Session Types as Program Logics

Eduard Kamburjan (University of Oslo, NO)

Main reference Eduard Kamburjan: "Behavioral Program Logic", in Proc. of the Automated Reasoning with Analytic Tableaux and Related Methods – 28th International Conference, TABLEAUX 2019, London, UK, September 3-5, 2019, Proceedings, Lecture Notes in Computer Science, Vol. 11714, pp. 391–408, Springer, 2019.
URL https://doi.org/10.1007/978-3-030-29026-9_22

This talk was based on a conference presentation at TABLEAUX 2019. We present Behavioral Program Logic (BPL), a dynamic logic for trace properties that incorporates concepts from behavioral types and allows reasoning about nonfunctional properties within a sequent calculus. BPL uses behavioral modalities, to verify statements against behavioral specifications. Behavioral specifications generalize both postconditions and behavioral types. They can be used to specify other static analyses, e.g., data flow analyses. This enables deductive reasoning about the results of multiple analyses on the same program, potentially implemented in different formalisms. Our calculus for BPL verifies the behavioral specification gradually, as common for behavioral types. This vastly simplifies specification, calculus and composition of local results. We present a sequent calculus for object-oriented actors with futures that integrates a pointer analysis and bridges the gap between behavioral types and deductive verification.

3.11 Priorities as a Graded Monad

Wen Kokke (University of Edinburgh, GB) and Ornela Dardha (University of Glasgow, GB)

Main reference Wen Kokke, Ornela Dardha: "Prioritise the Best Variation", in Proc. of the Formal Techniques for Distributed Objects, Components, and Systems – 41st IFIP WG 6.1 International Conference, FORTE 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings, Lecture Notes in Computer Science, Vol. 12719, pp. 100–119, Springer, 2021.
URL https://doi.org/10.1007/978-3-030-78089-0_6

In this talk, I will present PGV, a variant of Wadler's GV which decouples channel creation from thread spawning, and restores deadlock freedom by adding priorities. Notably, I will show how PGV embeds deadlock free communication and concurrency primitives with priorities in a standard linear functional language using a graded monad.

References

1 Kokke W., Dardha O. (2021) Prioritise the Best Variation. In: Peters K., Willemse T.A.C. (eds) Formal Techniques for Distributed Objects, Components, and Systems. FORTE 2021. Lecture Notes in Computer Science, vol 12719. Springer, Cham. https://doi.org/10.1007/978-3-030-78089-0_6

2 Wen Kokke and Ornela Dardha. 2021. Deadlock-free session types in linear Haskell. Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell. Association for Computing Machinery, New York, NY, USA, 1–13. DOI:https://doi.org/10.1145/3471874.3472979

3.12 Asymmetric Replicated State Machines

Roland Kuhn (Actyx AG – München, DE), Hernán Melgratti (University of Buenos Aires, AR), and Emilio Tuosto (Gran Sasso Science Institute, IT)

Joint work of Roland Kuhn, Daniela Marottoli, Hernán Melgratti, and Emilio Tuosto

A factory hall is a place where many humans and machines work together to turn input materials into finished goods. The efficiency of this collaboration – structured into many loosely coupled cells – is of vital importance to the business, hence our system favours availability and local progress over (global) correctness. Conflicts are allowed and can be recognised and compensated.

We present a formal model for machines emitting events to their local logs, where communication occurs eventually by shipping log prefixes between machines. A global type governs the desired protocol and can – if well-formed – be projected to local Mealy machines who will then faithfully realise the protocol.

3.13 Choreographic Programming in Choral

Fabrizio Montesi (University of Southern Denmark – Odense, DK)

This talk is an introduction to Choral (`https://www.choral-lang.org`), the first language for programming choreographies (multiparty protocols) based on mainstream programming abstractions: in Choral, choreographies are objects [2].

Choral's interpretation of choreographies is made possible by new types that enhance standard object types with a notion of locality. Every object is located at some roles (Alice, Bob, etc.), which denotes that the object is implemented collaboratively by them. Thus, objects become choreographic.

Choral is a choreographic programming language [1]: given a choreography that defines interactions among some roles, an implementation for each role in the choreography is automatically generated by a compiler. These implementations are libraries in pure Java, which developers can modularly compose in their own programs to participate correctly in choreographies. Crucially, Choral gives back to the programmer control over the APIs exposed to the users of the generated libraries. For the first time in the application of choreographic languages, this feature enables the generation of libraries that support information hiding, in the sense that the generated libraries hide the communication behaviour that they enact. An important consequence is that updates to the communication behaviour of a choreography might not alter the APIs of the generated libraries, avoiding the need for updating the client code that uses the (code generated from the) choreography.

Leveraging the interpretation of choreographies as objects, Choral brings higher-order composition to choreographic programming. The key novelty is that Choral allows for higher-order composition without the need for global synchronisations or central coordination, which is required by other current models. Choral's foundations have been recently modelled as an extension of the λ-calculus, Chorλ [3]. Chorλ has new reduction and rewriting rules that formalise the principles that underpin decentralised higher-order composition of choreographies.

References

1 Fabrizio Montesi. *Choreographic Programming*. PhD Thesis, IT University of Copenhagen, 2013.
2 Saverio Giallorenzo and Fabrizio Montesi and Marco Peressotti. *Choreographies as Objects*. CoRR abs/2005.09520, 2020.
3 Luís Cruz-Filipe and Eva Graversen and Lovro Lugović and Fabrizio Montesi and Marco Peressotti. *Choreographies as Functions*. CoRR abs/2111.03701, 2022.

3.14 Effpi: verified message-passing programs in Scala 3

Alceste Scalas (Technical University of Denmark – Lyngby, DK)

Joint work of Alceste Scalas, Nobuko Yoshida, Elias Benussi
Main reference Alceste Scalas, Nobuko Yoshida, Elias Benussi: "Verifying message-passing programs with dependent behavioural types", in Proc. of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019, pp. 502–516, ACM, 2019.
URL https://doi.org/10.1145/3314221.3322484

I will talk about Effpi: an experimental toolkit for strongly-typed concurrent and distributed programming in Scala 3.

Effpi addresses a main challenge in the development of concurrent programs: errors like protocol violations, deadlocks, and livelocks are often spotted late, at run-time, when applications are tested or (worse) deployed. Effpi aims at finding such errors early, when code is written and compiled.

Effpi provides: (1) a set of Scala classes for describing communication protocols as types; (2) an embedded DSL for concurrent programming (reminiscent of Akka actors); (3) a Scala compiler plugin to verify whether protocols and programs enjoy desirable properties, such as deadlock-freedom; and (4) an efficient run-time system for executing Effpi programs.

The combination of (1) and (2) allows the Scala 3 compiler to check whether an Effpi program implements a desired protocol/type; and this, together with (3), means that many concurrent programming errors are found and reported at compile-time. Further, (4) allows for running highly concurrent Effpi programs with millions of interacting processes/actors, by scheduling them on a limited number of CPU cores.

In this talk, I will give an overview of Effpi, illustrate its design and main features, and explain how it leverages the capabilities of Scala 3. I will also discuss ongoing and future work.

References

1 Scalas, A., Yoshida, N., & Benussi, E. (2019). *Verifying Message-Passing Programs with Dependent Behavioural Types*. In Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (pp. 502–516). Association for Computing Machinery. `https://doi.org/10.1145/3314221.3322484`
2 Scalas, A., Yoshida, N., & Benussi, E. (2019). Effpi: Verified Message-Passing Programs in Dotty. In Proceedings of the Tenth ACM SIGPLAN Symposium on Scala (pp. 27–31). Association for Computing Machinery. `https://doi.org/10.1145/3337932.3338812`

3.15 Algebraic Session Types

Peter Thiemann (Universität Freiburg, DE) and Vasco T. Vasconcelos (University of Lisbon, PT)

Joint work of Peter Thiemann, Bernardo Almeida, Andreia Mordido, Janek Spaderna, and Vasco T. Vasconcelos

Current session type systems rely on recursive types to model recursive protocols. This approach requires nontrivial algorithms for checking type equivalence. For traditional recursive session types, type equivalence can be reduced to equivalence of finite automata. For context-free session types, it amounts to equivalence of deterministic context-free languages.

The system of algebraic session types overcomes these problems while keeping the expressivity of context-free session types. By modeling recursive protocols as an extension of recursive algebraic datatypes, we can replace a structural approach to recursive types by a nominal one. This shift in perspective avoids the expensive algorithms for type equivalence and replaces them with a linear-time test in the size of the type.

We demonstrate with examples that algebraic session types enable new ways of parameterizing protocols, which were difficult to achieve with previous systems.

3.16 Polymorphic Context-free Session Types

Peter Thiemann (Universität Freiburg, DE) and Vasco T. Vasconcelos (University of Lisbon, PT)

Joint work of Bernardo Almeida, Andreia Mordido, Peter Thiemann, Vasco T. Vasconcelos
Main reference Bernardo Almeida, Andreia Mordido, Peter Thiemann, Vasco T. Vasconcelos: "Polymorphic Context-free Session Types", CoRR, Vol. abs/2106.06658, 2021.
URL https://arxiv.org/abs/2106.06658

Context-free session types provide a typing discipline for recursive structured communication protocols on bidirectional channels. They overcome the restriction of regular session type systems to tail recursive protocols. This extension enables us to implement serialisation and deserialisation of tree structures in a fully type-safe manner. We present the theory underlying the language FreeST 2, which features context-free session types in an extension of System F with linear types and a kind system to distinguish message types and channel types. The system presents some metatheoretical challenges, which we address, contractivity in the presence of polymorphism, a non-trivial equational theory on types, and decidability of type equivalence. We also establish standard results on type preservation, progress, and a characterisation of erroneous processes.

3.17 A Joyful Empirical Study on Session Types

Nobuko Yoshida (Imperial College London, GB)

Rust is a modern systems language focused on performance and reliability. Complementing Rust's promise to provide "fearless concurrency", asynchronous message passing is widely used thanks to its efficient and intuitive communication model—although it is also vulnerable to many concurrency errors such as deadlocks. Particularly, developers frequently exploit asynchronous message reordering, where sends and receives are reordered to maximise computation-communication overlap. Unfortunately, these kinds of optimisations open up a Pandora's box of further subtle concurrency bugs.

To guarantee deadlock-freedom by construction, we present rumpsteak: a new Rust framework based on session types. Previous session type implementations in Rust are either (1) built upon synchronous and blocking communication, which incurs a substantial performance cost; and/or (2) limited to two-party interactions, which risks introducing deadlocks. Crucially, none of these implementations can support the safe message reordering we seek.

rumpsteak instead uses multiparty session types and targets asynchronous applications using async/await code. Its unique feature is the ability to practically offer asynchronous message reordering while preserving deadlock-freedom. For this, rumpsteak incorporates two recent advanced session type theories: (1) k-multiparty compatibility (k-MC), which globally verifies safety properties for a set of participants and (2) asynchronous multiparty session subtyping, which locally verifies optimisations in the context of a single participant. Specifically, we propose a novel algorithm for asynchronous subtyping that is both sound and decidable.

3.18 Monitoring Protocol Conformance with Multiparty Session Types and OpenTelemetry

Fangyi Zhou (Imperial College London, GB) and Nobuko Yoshida (Imperial College London, GB)

Joint work of Fangyi Zhou, Francisco Ferreira, and Nobuko Yoshida

In this talk, we demonstrate our ongoing work of monitoring protocol conformance using multiparty session types and OpenTelemetry. OpenTelemetry is a new observability framework for distributed cloud software, and we utilise its distributed tracing functionality to validate traces against a prescribed global type.

3.19 Statically Verified Refinements for Multiparty Protocols

Fangyi Zhou (Imperial College London, GB), Francisco Ferreira, Raymond Hu (Queen Mary University of London, GB), Rumyana Neykova (Brunel University – Uxbridge, GB), and Nobuko Yoshida (Imperial College London, GB)

Joint work of Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida
Main reference Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, Nobuko Yoshida: "Statically verified refinements for multiparty protocols", Proc. ACM Program. Lang., Vol. 4(OOPSLA), pp. 148:1–148:30, 2020.
URL https://doi.org/10.1145/3428216

With distributed computing becoming ubiquitous in the modern era, safe distributed programming is an open challenge. To address this, multiparty session types (MPST) provide a typing discipline for message-passing concurrency, guaranteeing communication safety properties such as deadlock freedom.

While originally MPST focus on the communication aspects, and employ a simple typing system for communication payloads, communication protocols in the real world usually contain constraints on the payload. We introduce refined multiparty session types (RMPST), an extension of MPST, that express data dependent protocols via refinement types on the data types.

We provide an implementation of RMPST, in a toolchain called Session*, using Scribble, a multiparty protocol description toolchain, and targeting F*, a verification-oriented functional programming language. Users can describe a protocol in Scribble and implement the endpoints in F* using refinement-typed APIs generated from the protocol. The F* compiler can then statically verify the refinements. Moreover, we use a novel approach of callback-styled API generation, providing static linearity guarantees with the inversion of control. We evaluate our approach with real world examples and show that it has little overhead compared to a naive implementation, while guaranteeing safety properties from the underlying theory.

4 Working groups

4.1 Breakout Group: Typing Non-Channel-Based Models

Gul Agha (University of Illinois – Urbana-Champaign, US), Mariangiola Dezani (University of Turin, IT), Simon Fowler (University of Glasgow, GB), Philipp Haller (KTH Royal Institute of Technology – Kista, SE), Raymond Hu (Queen Mary University of London, GB), Eduard Kamburjan (University of Oslo, NO), Roland Kuhn (Actyx AG – München, DE), Hernán Melgratti (University of Buenos Aires, AR), Alceste Scalas (Technical University of Denmark – Lyngby, DK), and Peter Thiemann (Universität Freiburg, DE)

We started with the talk by Eduard Kamburjan (see materials), which describes a typing discipline and implementation for active objects based on asynchronous remote invocations. Thereafter, we briefly introduced the execution model later presented by Roland Kuhn and Emilio Tuosto in the plenary session on Thursday, namely evaluating state machines over an eventually consistent global log that is merged from locally produced append-only logs.

In such a system we have no channels to which session types can be attached. Instead, we need to characterise the event traces produced by the execution of a global protocol, where additional difficulty arises from the fact that the local behaviour can produce a greater variety of event traces due to local state machines having only a partial view of the global log – event dissemination is the mechanism by which non-determinism is introduced in this model.

The resulting "event soup" prompted the question of whether join calculus is a suitable model to tame such a system. The issue with this is that the setting is meant to guarantee 100% availability while join calculus treats messages as linear values that can only be consumed once; implementing this implies a consensus protocol between different participants that would ruin availability during network partitions. Events in the Actyx system have quantity omega, so join calculus is too strict.

The Actyx implementation currently has no mechanism for preventing conflicts (a consensus-based facility could be added in an opt-in fashion to coordinate important decisions). Instead, it allows conflicts to be detected since they are clearly visible in the event traces. This is okay for all cases where compensating actions can adequately fix the situation.

At this point we concluded that the discussed system is clearly distinct from the known body of previous work, so it is hard to transfer existing models or mechanisms to this setting.

We then switched to Actors as the other non-channel system that is widely used. Immediately after Eduard's talk we had already briefly discussed that removing the response (and thus the usage of Futures) from that Active Objects model would result in a very similar setting to the Actor model. We then looked at the join calculus again but moved on to the question of whether to type the behaviour of the processes or the contents of the mailbox.

There are two approaches: either use local types to govern the actions performed by the actor (which is done by Neykova and Yoshida's work on Multiparty Session Actors, and by Harvey et al.'s work on EnsembleS, for example); or to type the contents of the mailbox. The naive way of implementing the latter, namely typing a sequence of types to be received, is too restrictive to use in practice. Instead, work by de'Liguoro and Padovani considers a mailbox type system, where mailboxes are given a type described by a commutative regular expression (unordered sequencing, replication, and choice), which can rule out errors such as unhandled messages and deadlocks.

Gul Agha pointed out that in the absence of a typed channel it is more difficult to figure out failure cases because it is less obvious what happens when a given message does not arrive. This might require some dynamic (i.e. symbolic) analysis in addition to the static typing judgement, i.e. finding an undesirable configuration and working one's way backwards to find how this configuration could arise. Another issue is that the absence of a message cannot be clearly attributed to a single role, so the static analysis will need to model for example whether there can be at least one participant of a given role in a state that allows the needed interaction. Symbolic reasoning can then be used to figure out whether such state is actually realised.

Roland Kuhn pointed out that completeness of such a system is not required and will probably not even lead to the most useful system, because an application-dependent unhandled failure rate is acceptable in systems where humans can be called upon to fix things – which includes most systems today.

Since also such systems cannot statically prevent all conflicts, it is important to keep track of the local knowledge at the time a decision was made (like a causality trace) to be able to figure out the correct compensating actions. While this has a cost and is therefore

usually not done in real-world systems, the cost can be kept at a minimum for only tracking a single bit for each message that is expected within a given session (i.e. a version vector where each counter is 0 or 1).

One important message from Gul Agha was that we have nice means to separate

- how (which is the private implementation)
- what (which is the method, function signature, message, ... that selects the operation)
- when (which is governed by a session type)
- who (which can be pub-sub, static, assignment from a pool, etc.)

By keeping these well separated we get higher modularity and reuse, e.g. just needing to change a session type and reusing all methods and implementations.

4.2 Breakout Group: Logic-based approaches

Marco Carbone (IT University of Copenhagen, DK), Stephanie Balzer (Carnegie Mellon University – Pittsburgh, US), Ornela Dardha (University of Glasgow, GB), Wen Kokke (University of Edinburgh, GB), Sam Lindley (University of Edinburgh, GB), Fabrizio Montesi (University of Southern Denmark – Odense, DK), J. Garrett Morris (University of Iowa – Iowa City, US), Jorge A. Pérez (University of Groningen, NL), Bernardo Toninho (NOVA School of Science and Technology – Lisbon, PT), and Philip Wadler (University of Edinburgh, GB)

In the last decade, behavioural types, in particular Session Types, have been connected to Linear Logic. Back in 2010, Caires and Pfenning [1] proposed a proposition-as-types connection between intuitionistic linear logic and a session-typed variant of the pi calculus. Later, Wadler [2] used the same idea to draw a connection to classical linear logic. The two results have laid the basis for a stream of contributions in the area aiming at tightening the connection between behavioural type systems and linear logic. Thanks to this approach, it has been possible to represent some problems in behavioural types logically, solve them, and then map them back.

The goal of this breakout-group was not only to discuss the current state of the art, but also the directions for future research on the topic. Such directions can be summarised as follows:

- The community is interested in further exploiting the proposition-as-types approach for better understanding session/behavioural types.
- Different results are present in the literature: it is time that we also try to relate them formally so that we can transfer strengths from one approach to another.
- We should not restrict to Classical/Intuitionistic Linear Logic, but explore the relationship with other logics. This discussion has spawned a further discussion on what the minimum requirements are for a system to be a logic. An explicit and clear answer to "what is a logic?" can be found in Henry De Young's thesis [3], specifically on pag. 37, section 3.2.

In order to address the points above, people present at the meeting proposed to:

- Set up an online reading group in order to discuss relationships between single approaches
- Set up a sharing platform where results can actually be shared, and new collaborations can be started.

References

1 Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: CONCUR. pp. 222–236 (2010)
2 Wadler, P.: Propositions as sessions. In: ICFP. pp. 273–286 (2012)
3 Henry De Young. *Session Typed Ordered Logical Specifications*. PhD Thesis, 2020. `https://www.cs.cmu.edu/~hdeyoung/assets/papers/thesis20.pdf`

4.3 Breakout Group: Type-Informed Recovery Strategies

Fabrizio Montesi (University of Southern Denmark – Odense, DK), Laura Bocchi (University of Kent – Canterbury, GB), Marco Carbone (IT University of Copenhagen, DK), Ornela Dardha (University of Glasgow, GB), Mariangiola Dezani (University of Turin, IT), Philipp Haller (KTH Royal Institute of Technology – Kista, SE), Mathias Jakobsen (University of Glasgow, GB), Sam Lindley (University of Edinburgh, GB), J. Garrett Morris (University of Iowa – Iowa City, US), Philip Munksgaard (University of Copenhagen, DK), Laura Voinea (University of Kent – Canterbury, GB), Philip Wadler (University of Edinburgh, GB), and Fangyi Zhou (Imperial College London, GB)

Managing failures is important in distributed systems, but related research on behavioural types is still in the early stages. We are particularly interested in how types could help in the programming of strategies for recovering from failures.

The problem of failure recovery is multifaceted, because there are different categories of failures that can be encountered at runtime. These include crashes, message losses, and wrong ordering of messages or actions. Furthermore, depending on the level that software operates at (recall, for example, the layers of the OSI model), assumptions and focus might change.

For example, if we wish to write a low-level protocol like Ethernet or a distributed agreement protocol, then it might be desirable to use a fine-grained model that (a) exposes failures in detail, like single failures in the communication of each network packet, "alive" timeouts, etc., and (b) allows for programming recovery strategies to handle such failures, e.g., retransmission.

Differently, if we are reasoning about code designed for the application level, it is typical to make stronger reliability assumptions. For example, we might assume that network transmissions are reliable, delegating to the lower levels to deal with relevant failures there. A failure raised from the lower levels would then be managed using more abstract constructs on the application level, leading to more coarse recovery strategies (e.g., restart of the entire protocol or reconnection).

The multifaceted and multilevel nature of failure recovery is reflected by past and current research, including: types for managing fallible interactions and message loss in choreographic languages [3]; work on ensuring that data to be communicated can be meaningfully marshalled [4]; and exceptions for session types, where a reliable network is assumed [2, 1].

There are several open questions related to the principles of recovery strategies, including:

- How can we model the principles (for failure recovery) used in real-world software on different levels? How should we then interface lower-level models (with weak reliability assumptions) with higher-level models (with stronger reliability assumptions)?

- Can we modularly encode recovery strategies in high-level languages into lower-level languages that make weaker reliability assumptions?
- Can we extend session types to reason usefully about failures of different kinds?
- Can we make a fundamental calculus of failures? Is there a useful link to ongoing research on effect handlers (e.g., [5])?

References

1 Paul Harvey and Simon Fowler and Ornela Dardha and Simon J. Gay. *Multiparty Session Types for Safe Runtime Adaptation in an Actor Language*. In 35th European Conference on Object-Oriented Programming, ECOOP 2021, LIPIcs Vol. 194, 10:1–10:30, 2021.

2 Marco Carbone and Kohei Honda and Nobuko Yoshida. *Structured Interactional Exceptions in Session Types*. In CONCUR 2008 – Concurrency Theory, 19th International Conference, Lecture Notes in Computer Science, Vol. 5201, pp. 402–417, Springer, 2008.

3 Fabrizio Montesi and Marco Peressotti. *Choreographies meet Communication Failures*. CoRR abs/1712.05465, 2018.

4 Heather Miller and Philipp Haller and Martin Odersky. *Spores: A Type-Based Foundation for Closures in the Age of Concurrency and Distribution*. In 28th European Conference on Object-Oriented Programming, ECOOP 2014, Lecture Notes in Computer Science, Vol. 8586, pp. 308–333, Springer, 2014.

5 Sam Lindley. *Handler calculus*. Draft, 2021. `https://homepages.inf.ed.ac.uk/slindley/papers/handler-calculus-draft-may2021.pdf`

4.4 Breakout Group: Session types with untrusted counter-parties

Philip Munksgaard (University of Copenhagen, DK), Christian Bartolo Burló (Gran Sasso Science Institute, IT), Marco Carbone (IT University of Copenhagen, DK), Mariangiola Dezani (University of Turin, IT), Simon Fowler (University of Glasgow, GB), Mathias Jakobsen (University of Glasgow, GB), Roland Kuhn (Actyx AG – München, DE), Fabrizio Montesi (University of Southern Denmark – Odense, DK), Alceste Scalas (Technical University of Denmark – Lyngby, DK), Peter Thiemann (Universität Freiburg, DE), Emilio Tuosto (Gran Sasso Science Institute, IT), and Fangyi Zhou (Imperial College London, GB)

There is no question that session types and behavioral types are useful in situations where you control all the nodes in the network. In practice, however, that is not always the case, eg. in peer-to-peer networks or on the internet in general. Still, it would be helpful to be able to use session types to model the communication protocols in such cases.

Attempting to do so presents a range of problems, including:

- How do we know what protocol the counterparty is using?
- Can we trust that the counterparty will adhere to that protocol?
- How should we handle failures, both in the protocol and in the transport layer?

Starting from the end, exhaustive work has been performed in the area of *monitors*. Given a session type, a monitor sits between the user and an untrusted counterparty, mediating and making sure that the user only sees messages that adhere to the specified protocol. Depending on the particular implementation, it can handle timeouts, protocol errors, dropped connections and so on. With such a monitor, we can safely implement our side of the session type, without having to think about these problems.

However, what about a well-meaning counterparty whose protocol differs slightly from ours? A simple example would be a login server that accepts two strings, a username and a password (in that order), while your client expects to send a password and a username. Without additional information from the server, you would not be able to correctly interface with the server, even with a monitor.

For such a case to work, we would need to have some sort of meta-protocol for talking about the session type. In essence, each party should be able to tell the other what session type it expects to be using, and we should have a way of handling discrepancies in the protocol. This line of reasoning lead to some interesting debates at the seminar, and a stated intention from several attendees to continue work after the seminar.

4.5 Breakout Group: Join Patterns / Synchronization – The Next Generation

Claudio Russo (Dfinity – Cambridge, GB), Gul Agha (University of Illinois – Urbana-Champaign, US), Philipp Haller (KTH Royal Institute of Technology – Kista, SE), Eduard Kamburjan (University of Oslo, NO), Emilio Tuosto (Gran Sasso Science Institute, IT), Laura Voinea (University of Kent – Canterbury, GB), and Philip Wadler (University of Edinburgh, GB)

Intro

The attendees were a mixture of people with considerable experience of join patterns and the join calculus and those curious to hear learn more about them.

We kicked off with an introduction to join patterns as:

- a mechanism for pattern matching over the empty/non-empty state of co-located, (asynchronous) message queues.
- a special case of more general Logic Programming constructs with arbitrary predicates on channel contents from Concurrent Prolog.

Some stressed that the main advantage of joins is expressing synchronization & coordination, not necessarily pattern matching over the content of queues enabled by Concurrent Prolog.

We then re-iterated some of the history of join patterns – which have come in several guises from language extensions to libraries, for niche and mainstream languages. A (by no-means exhaustive) list includes the Join Calculus, JoCaml, Funnel (the precursor to Scala), Polyphonic C#, JErlang, JoinJava, Comega, the C# Joins library, Concurrent Basic, Scalable Joins, Scala, Akka, and Sharpie.

The different presentations of join patterns typical adopt one of :

- recursive function declarations with alternatives of conjoined function headers (JoCaml) and selective "return"s to different functions within the same body.
- object methods with alternatives of conjoined method headers (Polyphonic C#) and single returns.
- separate channels declarations with patterns over them (Joins library, Concurrent Basic)
- designs disguised as (library) extensions of pattern matching (Scala)
- various degrees of support for synchronous channels in patterns (on, sometimes several).

Limits to Adoption

Despite the plethora of implementations and designs, join patterns have never made into the mainstream the way, say, lambda abstraction and garbage collection have. Eisenbach has expressed similar difficulties [5]. We came up with following potential reasons:

- The early presentation of joins were all asynchronous, sometimes continuation based, and thus less approachable.
- Alternatives such as actors and futures are easier to explain, though likely less expressive (e.g. n-ary synchronization is difficult to achieve with usual primitives).
- Features such as lambdas and GC took several decades to gain wide-spread adoption. Perhaps join patterns just need more time.

Typing

We then briefly turned to discussing the typing aspects of join patterns. Phil Wadler asked "Is there a Curry-Howard isomorphism" for the join calculus as there is for linear types and process-calculi, citing Frank Pfenning's work on linear logic for typing processes. Since the inspiration for join patterns comes from chemistry (a restriction of the chemical abstract machine), not logic, uncovering a Curry-Howard isomorphism might prove challenging. Nevertheless, the atomic consumption of linear resources inherent to joins suggests a strong connection to linear typing: [1] applies separation logic to verify some join based programs, while the more recent work [2, 3, 4] on linear typing of joins might shed further light.

Expressivity

Discussion briefly turned to the question of expressivity: how do join patterns compare to say, pi-calculus or actors? Cedric Fournet's thesis gave a translation between pi and joins but we could not recall if it was modular or whole-program. Regardless of expressivity, there is strong reason to believe that join-calculus is easier to implement than, say, pi-calculus, especially in a distributed setting. The reason for this is that the scheduling decisions needed to be made in joins can be resolved locally, at a receiver, while typical implementations of the pi-calculus rely on reaching agreement between distributed parties and are thus more obviously suited to shared-memory, non-distributed implementations.

References

1 Kasper Svendsen, Modular specification and verification for higher-order languages with state, PhD thesis, `https://cs.au.dk/~birke/phd-students/SvendsenS-thesis.pdf`.
2 Silvia Crafa, Luca Padovani, The Chemical Approach to Typestate-Oriented Programming, ACM Transactions on Programming Languages and Systems, vol. 39(3), pages 13:1-13:45, 2017.
3 Luca Padovani, A Type Checking Algorithm for Concurrent Object Protocols, Journal of Logical and Algebraic Methods in Programming, vol. 100, pages 16-35, 2018.
4 Luca Padovani, Deadlock-Free Typestate-Oriented Programming, Programming Journal, vol. 2(3), pages article 15, 2018.
5 Susan Eisenbach, personnel communication with Russo, Microsoft Research, 2018.

Participants

- Marco Carbone
IT University of Copenhagen, DK
- Simon Fowler
University of Glasgow, GB
- Philipp Haller
KTH Royal Institute of Technology – Kista, SE
- Mathias Jakobsen
University of Glasgow, GB
- Eduard Kamburjan
University of Oslo, NO
- Roland Kuhn
Actyx AG – München, DE
- Sam Lindley
University of Edinburgh, GB
- Fabrizio Montesi
University of Southern Denmark – Odense, DK
- Philip Munksgaard
University of Copenhagen, DK
- Alceste Scalas
Technical University of Denmark – Lyngby, DK
- Peter Thiemann
Universität Freiburg, DE
- Emilio Tuosto
Gran Sasso Science Institute, IT

Remote Participants

- Gul Agha
University of Illinois – Urbana-Champaign, US
- Stephanie Balzer
Carnegie Mellon University – Pittsburgh, US
- Christian Bartolo Burló
Gran Sasso Science Institute, IT
- Laura Bocchi
University of Kent – Canterbury, GB
- Edwin Brady
University of St Andrews, GB
- Ilaria Castellani
INRIA – Sophia Antipolis, FR
- Tzu-Chun Chen
Evonik Industries – Hanau, DE
- Ornela Dardha
University of Glasgow, GB
- Mariangiola Dezani
University of Turin, IT
- Simon Gay
University of Glasgow, GB
- Raymond Hu
Queen Mary University of London, GB
- Atsushi Igarashi
Kyoto University, JP
- Jules Jacobs
Radboud University Nijmegen, NL
- Wen Kokke
University of Edinburgh, GB
- Dimitrios Kouzapas
University of Cyprus – Nicosia, CY
- Hernán Melgratti
University of Buenos Aires, AR
- J. Garrett Morris
University of Iowa – Iowa City, US

- Rumyana Neykova
Brunel University – Uxbridge, GB
- Marco Peressotti
University of Southern Denmark – Odense, DK
- Jorge A. Pérez
University of Groningen, NL
- Claudio Russo
Dfinity – Cambridge, GB
- Guido Salvaneschi
Universität St. Gallen, CH
- Bernardo Toninho
NOVA School of Science and Technology – Lisbon, PT
- Vasco T. Vasconcelos
University of Lisbon, PT
- Laura Voinea
University of Kent – Canterbury, GB
- Philip Wadler
University of Edinburgh, GB
- Nobuko Yoshida
Imperial College London, GB
- Fangyi Zhou
Imperial College London, GB