2014 Imperial College Computing Student Workshop

ICCSW'14, September 25–26, 2014, London, United Kingdom

Edited by

Rumyana Neykova
Nicholas Ng

OASIcs – Vol. 43 – ICCSW'14 www.dagstuhl.de/oasics

Editors

Rumyana Neykova
Department of Computing
180 Queen's Gate, London, SW7 2AZ
United Kingdom
`rumyana.neykova10@imperial.ac.uk`

Nicholas Ng
Department of Computing
180 Queen's Gate, London SW7 2AZ
United Kingdom
`nickng@imperial.ac.uk`

ACM Classification 1998
B.1.4 Languages and Compilers, C.1.4 Parallel Architectures, D.1.1 Applicative (Functional) Programming, D.1.3 Parallel Programming, D.2.1 Requirements/Specifications D.2.4 Software/Program Verification, D.1.3 Concurrent Programming, F.1.3 Complexity Measures and Classes, F.3.1 Specifying and Verifying and Reasoning about Programs, F.4.1 Mathematical Logic, G.2.2 Graph Theory, I.2.10 Vision and Scene Understanding, I.2.7 Natural Language Processing, I.2 Artificial Intelligence, I.6 Simulation and Modelling, K.6.5 Security and Protection

ISBN 978-3-939897-76-7

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-939897-76-7.

Publication date
September, 2014

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 http://dnb.d-nb.de.

Digital Object Identifier: 10.4230/OASIcs.ICCSW.2014.i

ISBN /978-3-939897-76-7 **ISSN 2190-6807** **http://www.dagstuhl.de/oasics**

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.

ISSN 2190-6807

www.dagstuhl.de/oasics

Contents

Keynotes

Regular Papers

Imperial College Computing Student Workshop (ICCSW'14).
Editors: Rumyana Neykova and Nicholas Ng
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

We are pleased to present the proceedings of the fourth Imperial College Computing Student Workshop (ICCSW'14), which took place on 25th–26th September 2014 in London, and was hosted by Imperial College London.

ICCSW is an event organised with the "by students, for students" ethos in mind. The organisation work of the workshop was done by the committee formed of Ph.D. students from the ACM Student Chapter in the Department of Computing, Imperial College London.

The vision for ICCSW is to provide a venue for doctoral students to experience running and participating in an academic workshop, as well as providing a networking opportunity for researchers in similar stages of their career. All of the papers and reviews for ICCSW were written by students; we adopted a unique peer review system in which all authors are also programme committee members for the workshop.

We operate a highly successful ambassador programme for students in different institutions to promote ICCSW. This invaluable programme helps to increase the scope and prominence of the event both nationally and internationally.

These proceedings contain 12 contributions in various fields from across computer science. This year the workshop received 20 submissions over two tracks: technical papers track survey track. After a rigorous review and selection process, 12 papers were accepted to be presented at ICCSW'14, representing a 60% acceptance rate.

Both days of the workshop featured a keynote from a prominent researcher in computer science; We were truly grateful to have as keynote speakers (1) a winner of the POPL Most Influential Paper Award, an ACM Fellow and a Fellow of Royal Society of Edinburgh, Professor Philip Wadler from the University of Edinburgh and (2) one of the co-founders of Google DeepMind Technologies and a recipient of the Canadian Singularity Institute research prize, Dr. Shane Legg from Google.

The keynote talks were entitled:

- You and Your Research and the Elements of Style, by Prof. Philip Wadler;
- From academia to industry: The story of Google DeepMind, by Dr. Shane Legg.

On behalf of the organising committee, we wish to thank all authors and our ambassadors, who acted as reviewers in our unique peer review process. Furthermore, we also wish to thank our sponsors: Imperial College London, who provided us with more than just financial support; Google, our platinum-level sponsor, who have supported ICCSW since its inception in 2011; ARM for their bronze-level sponsorship. Without the support of our sponsors, ICCSW'14 would not have been possible.

Rumyana Neykova and Nicholas Ng
ICCSW'14 Editors

Imperial College Computing Student Workshop (ICCSW'14).
Editors: Rumyana Neykova and Nicholas Ng
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
ICCSW

Conference Organisation

Organising Committee

Petr Hošek	Imperial College London
Bertan Kavuncu	Imperial College London
Dan Liew	Imperial College London
Luo Mai	Imperial College London
Feryal Mehraban Pour Behbahani	Imperial College London
Silvia Vinyes Mora	Imperial College London
Rumyana Neykova	Imperial College London
Nicholas Ng	Imperial College London
Marily Nika	Imperial College London
Florian Rathgeber	Imperial College London
Claudia Schulz	Imperial College London
Călin-Rares Turliuç	Imperial College London
	Imperial College London ACM Student Chapter `http://acm.doc.ic.ac.uk/`

Imperial College Computing Student Workshop (ICCSW'14).
Editors: Rumyana Neykova and Nicholas Ng
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Ambassadors

Khulood Alyahya	University of Birmingham
Joao Amaral	University of Lisbon
Shaswar Baban	King’s Collge of London
Gabriel Barata	University of Lisbon
Tarek Besold	Osnabrück University
Tzu-Chun Chen	University of Turin
Dana Codreanu	University of Toulouse
Benoit Desouter	University of Ghent
Stefan Ellmauthaler	Leipzig University
Matthew Forshaw	Newcastle University
Johannes Fichte	TU Wien
Randy Goebel	University of Alberta
Evgenios Hadjisoteriou	University of Cyprus
Jesus Omana Iglesias	University College Dublin
Wilhelm Kleiminger	ETH Zurich
Chong-U Lim	Massachusetts Institute of Technology
Federico Mancinelli	University College London
Amin Mobasheri	Heidelberg University
Tiago Oliveira	University of Minho
Marco Paolieri	University of Florence
Marco Patrignani	University of Leuven
Alireza Pourranjbar	University of Edinburgh
Ken Satoh	National Institute of Informatics
Peter Schüller	Marmara University Istanbul
Zohreh Shams	University of Bath
Christof Spanring	University of Liverpool
Raoul-Gabriel Urma	University of Cambridge
Hu Xu	University of Dundee
Agnieszka Zbrzezny	University of Czestochowa
Huanzhou Zhu	University of Warwick
Maciej Zielenkiewicz	University of Warsaw

External Reviewers

Ahd Aljarf	Coventry University
Khulood Alyahya	University of Birmingham
Ethel Bardsley	Imperial College London
Evgenij Belikov	Heriot-Watt University Edinburgh
Tarek Besold	Osnabrück University
Andrew Cropper	Imperial College London
Jakub Daniel	Charles University in Prague
Pantazis Deligiannis	Imperial College London
Sophia Drossopoulou	Imperial College London
Stefan Ellmauthaler	Leipzig University
Johannes Fichte	TU Wein
Matthew Foster	Newcastle University
Romain Franceschini	University of Corsica
Juliana Franco	Imperial College London
Zhuoer Gu	University of Warwick
Peng Jiang	University of Warwick
Amin Mobasheri	University of Heidelberg
Tiago Oliveira	University of Minho
Marco Patrignani	University of Leuven
Alan Perotti	University of Turin
Jörg Pührer	Leipzig University
Quoc-Sang Phan	Queen Mary, University of London
Lei Shi	University of Warwick
Maciej Skorski	University of Warsaw
Christof Spanring	University of Liverpool
Niklas Wahlström	Linköping University
Agnieszka Zbrzezny	University of Czestochowa
Jian Zhang	University of Dundee
Maciej Zielenkiewicz	University of Warsaw

Supporters and Sponsors

Supporting Scientific Institutions

Imperial College London
`http://www.imperial.ac.uk/`

Platinum Sponsors

Google Inc.
`http://www.google.com/`

Bronze Sponsors

ARM Holdings, plc.
`http://www.arm.com/`

Imperial College Computing Student Workshop (ICCSW'14).
Editors: Rumyana Neykova and Nicholas Ng
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

From academia to industry: The story of Google DeepMind

Shane Legg

Google DeepMind

Abstract

Shane Legg left academia to cofound DeepMind Technologies in 2010, along with Demis Hassabis and Mustafa Suleyman. Their vision was to bring together cutting edge machine learning and systems neuroscience in order to create artificial agents with general intelligence. Following investments from a number of famous technology entrepreneurs, including Peter Thiel and Elon Musk, they assembled a team of world class researchers with backgrounds in systems neuroscience, deep learning, reinforcement learning and Bayesian statistics. In early 2014 DeepMind made international business headlines after it was acquired by Google. In this talk Shane covers some of the history behind DeepMind, his experience making the transition from academia to industry, how Google DeepMind performs research and finally some demos of the artificial agents that are under development.

1998 ACM Subject Classification I.2.11 Intelligent Agents

Keywords and phrases machine learning

Digital Object Identifier 10.4230/OASIcs.ICCSW.2014.1

Imperial College Computing Student Workshop (ICCSW'14).
Editors: Rumyana Neykova and Nicholas Ng; pp. 1–1
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

ICCSW

You and Your Research and the Elements of Style

Philip Wadler

University of Edinburgh

Abstract

This talk surveys advice from experts, including Richard Hamming, William Strunk, E. B. White, Donald Knuth, and others, on how to conduct your research and communicate your results.

1998 ACM Subject Classification K.3.2 Computer science education

Keywords and phrases research,communication

Digital Object Identifier 10.4230/OASIcs.ICCSW.2014.2

Imperial College Computing Student Workshop (ICCSW'14).
Editors: Rumyana Neykova and Nicholas Ng; pp. 2–2
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

History-Based Adaptive Work Distribution*

Evgenij Belikov

Heriot-Watt University, School of Mathematical and Computer Sciences
Riccarton, EH14 4AS, Edinburgh, Scotland
eb120@hw.ac.uk

Abstract

Exploiting parallelism of increasingly heterogeneous parallel architectures is challenging due to the complexity of parallelism management. To achieve high performance portability whilst preserving high productivity, high-level approaches to parallel programming delegate parallelism management, such as partitioning and work distribution, to the compiler and the run-time system. Random work stealing proved efficient for well-structured workloads, but neglects potentially useful context information that can be obtained through static analysis or monitoring at run time and used to improve load balancing, especially for irregular applications with highly varying thread granularity and thread creation patterns. We investigate the effectiveness of an adaptive work distribution scheme to improve load balancing for an extension of Haskell which provides a deterministic parallel programming model and supports both shared-memory and distributed-memory architectures. This scheme uses a less random work stealing that takes into account information on past stealing successes and failures. We quantify run time performance, communication overhead, and stealing success of four divide-and-conquer and data parallel applications for three different update intervals on a commodity 64-core Beowulf cluster of multi-cores.

1998 ACM Subject Classification C.1.4 Parallel Architectures, D.1.1 Applicative (Functional) Programming, D.1.3 Parallel Programming, D.3.4 Run-Time Environments, D.4.1 Scheduling

Keywords and phrases Adaptive Load Balancing, High-Level Parallel Programming, History, Work Stealing, Context-Awareness

Digital Object Identifier 10.4230/OASIcs.ICCSW.2014.3

1 Introduction

In the many-core era parallelism is one of the key sources of application performance [19]. Unfortunately, exploiting parallelism is challenging due to the added complexity of managing parallelism [2], in particular of partitioning and work distribution across the available processing elements (PEs). To preserve high programmer productivity, high-level approaches to structured parallel programming delegate parallelism management to the compiler and the run-time system (RTS). Moreover, manual adaptation to rapidly evolving and increasingly heterogeneous and hierarchical parallel architectures is deemed infeasible mandating adaptive solutions to achieve high performace portability [17, 7]. Furthermore, distribution is required for scalability beyond one physical machine as often required in important domains such as Large-Scale Data Analysis, Scientific Computing and Cloud Computing.

A state-of-the-art work distribution scheme is *random work stealing* where idle workers attempt to steal from victims chosen uniformly at random. This policy is scalable due to its decentralised and probabilistic nature and proved efficient for well-structured workloads [6].

* This work is supported by the Scottish Informatics and Computer Science Alliance (SICSA).

Imperial College Computing Student Workshop (ICCSW'14).
Editors: Rumyana Neykova and Nicholas Ng; pp. 3–10
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

However, due to randomness it neglects information that can be obtained through static analysis or monitoring at run time and potentially used to improve load balancing and locality, especially for applications that can be characterised as irregular due to highly varying thread granularity and thread creation patterns. In this paper, the term *thread* refers to a light-weight thread managed in user space, not to a fully-fledged OS-thread.

We investigate the effectiveness of an adaptive work distribution scheme that aims to improve load balancing in the context of a high-level non-strict functional language – an extension of Haskell [13] – which provides a deterministic parallel programming model and supports both shared-memory and distributed-memory architectures whilst dynamically managing work distribution. This scheme mostly uses a less randomised variant of work stealing that takes into account information on past stealing successes and failures to improve workload distribution. We quantify run time performance, communication overhead, and stealing success of four divide-and-conquer and data parallel functional applications on a modern 64-core Beowulf-class cluster consisting of 8-core nodes.

2 Background and Related Work

We describe Glasgow parallel Haskell that provides a unified high-level semi-explicit deterministic parallel programming model and its RTS that was extended to take additional contextual information into account when making policy decisions[1]. Additionally, most important related work is discussed along with the relevant concepts, policies, and mechanisms.

2.1 Parallel Functional Programming

Glasgow parallel Haskell (GpH) [20] provides the `par` combinator to express advisory parallelism, which takes two arguments and potentially executes the first argument in parallel whilst returning the second that is evaluated by the parent thread. Additionally, the `pseq` combinator is defined that fixes the evaluation order by evaluating the first argument and then the second. Lazy polymorphic higher-order functions are used to define *Evaluation Strategies* [22, 15] which further raise the level of abstraction by *separating coordination from computation*, similar to algorithmic skeletons [11].

Notably, the model is deterministic by design thus preventing the occurrence of race conditions and deadlocks that are notoriously difficult to detect and correct. For an overview of parallel programming models refer to recent surveys [4, 10]. GpH delegates most of the parallelism management to the RTS to provide architecture-independence at language level, whilst retaining optimisation flexibility at the system level. Below Listing 1 illustrates the use of the GpH combinators to parallelise the `QuickSort` algorithm. Note the conciseness and the close correspondence of the code to the common mathematical notation.

Listing 1 `QuickSort` Implementation in GpH.

```
par_qsort :: Ord a => [a] -> [a]
par_qsort [] = []
par_qsort (pivot:xs) = lower ‘par‘ higher ‘pseq‘ merge
  where lower  = par_qsort [y | y <- xs, y <  pivot]
        higher = par_qsort [y | y <- xs, y >= pivot]
        merge  = lower ++ (pivot:higher)
```

[1] the source code of this experimental RTS is available upon request

2.2 Adaptive Run-Time System Policy Control

GUM (Graph Reduction for a Unified Machine Model) [21] is an adaptive RTS for GpH that implements distributed graph reduction using a *virtual shared heap*. Figure 1 depicts the GUM control model which is based on observing system state as well as receiving some environmental information and controlling the policy decisions based on this information.

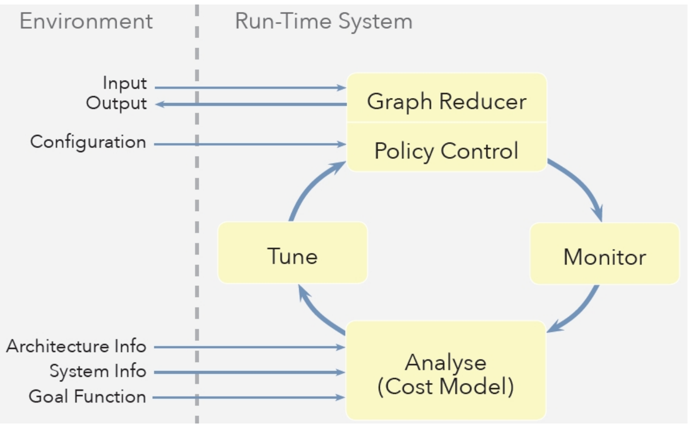


Figure 1 GUM Control Model Enables Adaptation.

The emphasis of the control model is on flexible adaptation at run time as opposed to commonly used static partitioning and work distribution schemes (e.g. in MPI or OpenCL). For every `par`, a *spark*, i.e. a pointer to an unevaluated closure in the shared graph representing potentially parallel work, is added to the spark pool. Sparks are cheap and can be turned into light-weight threads for parallel execution, which are more expensive as they store their state and a stack. A parent thread can *subsume* a child thread by evaluating the spark it has created sequentially to reduce thread creation overhead and increase granularity, a mechanism similar to lazy task creation [16].

Work Stealing Passive work distribution or work stealing is an established mechanism used in many language run-time systems to balance computational load across PEs [6]. It is scalable due to its decentralised nature and efficient for well-structured workloads since stealing efforts are amortised across idle PEs. Once no local threads are available the system sends a steal request to a PE chosen uniformly at random if the local spark pool is empty. If a PE receives a steal request (a so-called `FISH` in GUM terminology), it first looks for work in the spark pool and if there are sparks available donates the oldest and thus probably large spark in FIFO fashion [8]. Alternatively, if no work is available it forwards the request to another randomly chosen PE, unless the message exceeds its time-to-live limit. In that case the request is sent back to the original PE and is registered as a failed stealing attempt.

Recently, GUM has been ported to computational Grids and results on a heterogeneous cluster demonstrated benefits of using information such as computational power of the PEs, load, as well as latency between them [1]. In particular, simply placing main PE on the most powerful node of the most powerful cluster lead to increased performance. Similarly, a simulation study of divide-and-conquer applications on heterogeneous clusters comprising homogeneous PEs, showed that using load information is beneficial in a hybrid locally centralised and globally distributed scheme, where one PE is chosen to manage information as cluster head, whereas across clusters the heads communicate in a decentralised fashion [14]. Moreover, language extensions demonstrated the importance of improving locality [3].

3 Using Historical Information in Work Distribution Decisions

Several complementary policies were identified based on an application characterisation [5]. The key idea is to use monitored RTS-level information to *de-randomise* work stealing to increase the flexibility of adaptation to architectural and behavioural system-level changes. Here we investigate the effectiveness of using dynamic information about past stealing successes and failures. Additionally, we discuss the importance of selecting a suitable update interval. To our knowledge the use of this policy has not been previously explored in the context of a non-strict functional language with a semi-explicit parallel programming model.

3.1 History-Based Stealing

Work stealing is mainly concerned with the following decisions: a) which PE should one steal from as a thief (i.e. PE with no work); b) to which PE should one forward a `FISH` as a victim with no work; c) which of the available sparks should one donate as a victim with work.

Our policy extension is aimed at reducing communication overhead by increasing the fishing success ratio (i.e. the percentage of sent `SCHEDULE` messages containing work in relation to the total number of sent `FISH` messages requesting some work) by monitoring and storing information on recent stealing successes and failures. We investigate whether simply trying to steal from PEs where recent stealing attempts were successful yields any substantial benefits. If no most suitable PE could be selected due to either lack of successful stealing attempts or due to stale information, the algorithm falls back to random stealing. The policy is expected to work best in cases where a set of parallelism generators is fairly stable over time. The overhead is low as it involves counters and updating cost is amortised as it happens at garbage collection times and on arrival of `FISH` or `SCHEDULE` messages.

The key change to the mechanism is in victim selection: a table is maintained that records whether last stealing attempt from a given PE was successful. Logically, this can be viewed as a function $f(i :: PEid) \rightarrow (successInfo_i, timeStamp_i)$. Table 1 shows how the stored data is interpreted to select a PE with most consecutive successes, tie-breaking on the index.

Table 1 Overview of the Stored Historical Information.

information table field	value = 0	value > 0
history information	failed stealing attempt	number of consecutive successes
time stamp	information is stale	time of last update

3.2 Balancing Accuracy and Coverage

We also record a time stamp of the last update for each PE to judge whether the stored information is reliable and purge stale data at garbage collection times. An RTS flag is set to select an interval used to invalidate any table entries for PEs for which no update happened during the last interval.

The main challenge is the choice of a suitable interval such that highest *coverage* of the PEs is achieved whilst keeping *accurate* information. In fact, using stale information can be misleading and reduce fishing success ratio, which may lead to performance degradation.

4 Empirical Evaluation

We report application performance from a median run out of three on a cluster of multi-cores with 64 PEs and focus on relative speedups as we are interested in the behaviour of the parallel applications. We use CentOS 6.5, GHC 6.12.3, gcc 4.4.7, and PVM 3.4.6. Due to space limitations the focus is on cluster results as it is a more challenging architecture because of higher inter-node latency and hence higher associated communication costs.

The Beowulf cluster comprises a mix of 8-core Xeon 5504 nodes with two sockets with four 2GHz cores, 256 KB L2 cache, 4MB shared L3 cache and 12GB RAM, and 8-core Xeon 5450 nodes with two sockets with four 3GHz cores, 6MB shared L2 cache and 16GB RAM. The machines are connected via Gigabit Ethernet with an average latency of 150ns.

4.1 Parallel Applications

We use several applications from the *nofib* suite [18] and from a study of Evaluation Strategies [15]. The applications employ simple yet expressive *divide-and-conquer* (D&C) and *data-parallel* patterns which are considered representative of a wide range of applications [11, 9]. Using D&C, the final result is computed by merging the solutions of smaller problems obtained by recursively sub-dividing and solving the initial problem. A *threshold* can be used to increase granularity by limiting spark creation to a specified depth of the tree. Data-parallel applications exploit the parallelism by applying a function to the elements of a data structure. Granularity can be tuned by *chunking* several elements together. We measure run times, the number of messages sent, and fishing success ratio on up to 64 PEs.

- The D&C `parfib` program computes the Nth Fibonacci number using arbitrary-length integers; we use $N = 50$ and a threshold of 23; used to assess thread subsumption capabilities of the RTS, this benchmark is representative of regular and flat D&C applications with a single source of parallelism; splitting and the combining phases require two arithmetic operations on integers of arbitrary length, the sequential work is exponential.
- The D&C `worpitzky` application from the symbolic computation domain checks the Worpitzky identity for two arbitrary-length integers; we take 19 to the exponent of 27 and use a threshold of 10; at the top level this requires one exponentiation, one equality comparison, and a sum of n intermediate results, which are computed in parallel and for the other part requires two arithmetic operations and binomial computation using three factorial and three arithmetic operations; parallel computations include a single source of parallelism and three arithmetic operation for both the combine and the split phase.
- The D&C `coins` program computes ways to pay out a specified amount from a set of coins; in our case the value is 5777; the program is similar to `parfib` as the split and the combine phases require one arithmetic operation each, whilst sequential solution requires finding suitable permutations of coins.
- The data-parallel `sumeuler` program computes the sum over euler totient numbers in a given integer interval (which plays the role of explicit chunking to control granularity) by applying a function to each element of the list generated from the given intervals in parallel, and is fairly irregular; we use interval from 0 to 100000 with a chunk size of 500; all the degree of parallelism with merely 200 sparks is relatively low and all the parallelism is generated in the beginning of the execution by the main PE.

4.2 Results and Evaluation

The early results shown below demonstrate the effects of using history-based stealing but are rather indicative than conclusive. In Figure 2, the left column of graphs shows the run times (in seconds, note the different scales) of the applications on 2, 4, 16, and 64 PEs and the right column presents the corresponding fishing success ratios (in percent of total number of `FISH` messages). We observe that run times decrease by an *order of magnitude* for all applications demonstrating scalability. However, for most applications the benefit from adding more PEs reduces with the number of PEs due to lack of work and increased overheads. We aim to reduce communication overhead and improve load balancing by using the enhanced policy.

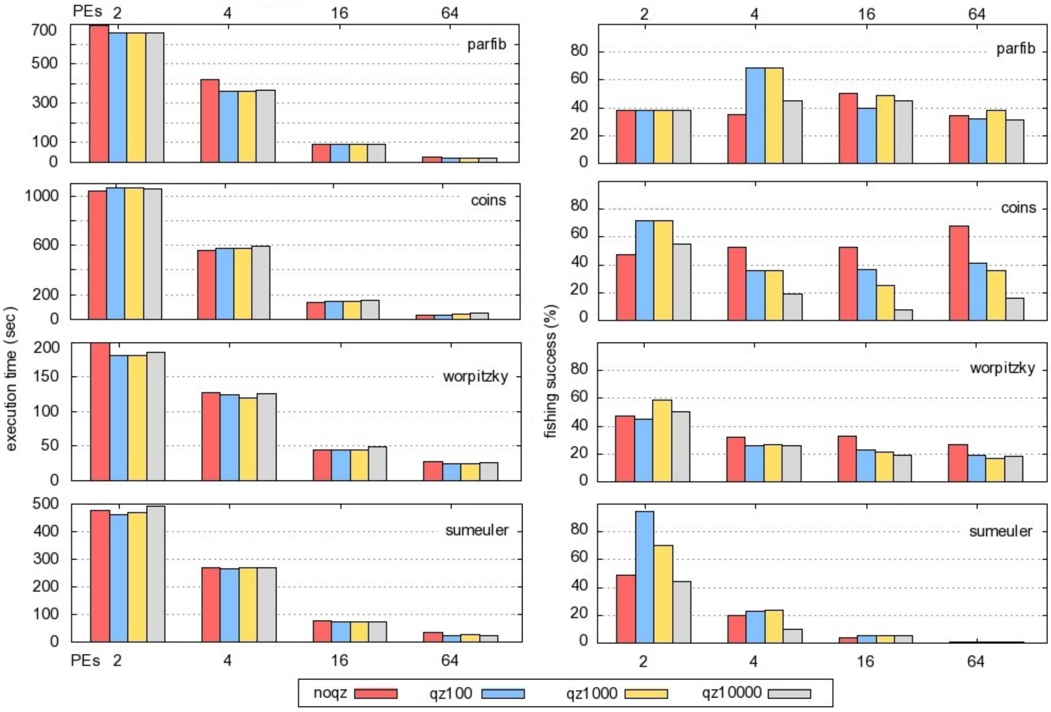


Figure 2 History-Based Stealing: Execution Times and Fishing Success Ratios.

Each bar for each number of PEs represents the baseline random stealing (leftmost of the four) or using history with a small (qz100 that stands for 100 ms), medium (qz1000) or large interval. A small interval leads to rapid invalidation of information so that we ensure high accuracy but coverage is often low, whereas opposite is the case if the interval is large. As noted above, the challenge is in finding optimal interval to balance accuracy and coverage.

On 64 PEs the run times for applications using history are consistently decreased by up to 34% (as for `sumeuler`). We can attribute the effectiveness of the policy for `sumeuler` to the fact that all parallelism is generated by the main PE, hence past behaviour appears predictive of future behaviour during the initial phase of the computation. In particular for low numbers of PEs the fishing success ratio is mostly higher if history is used. However, it is lower in most cases for D&C applications as new parallelism sources are created dynamically but are rather short-lived. There is not much difference in success ratio for `sumeuler` for higher PE numbers as there is not enough work available[2]. By contrast, more work is generated

[2] also indicated by the high percentage of `FISH`es in relation to the total number of messages (cf Table 2)

Table 2 Number of FISH Messages versus Total Sent Messages (on 64 PEs).

interval	noqz		qz100		qz1000		qz10000	
	FISHes	Total	FISHes	Total	FISHes	Total	FISHes	Total
sumeuler	34968	35714	18650	19451	20395	21200	15190	16004
parfib	5137	12134	3965	9027	4874	12196	4801	10793
coins	7710	28774	10541	27856	12694	30804	38715	63320
worpitzky	73278	153438	71437	125510	74370	125630	82155	139641

throughout a longer phase of the D&C computations. However, history appears misleading in this case as generators only create few sparks and further sparks are generated elsewhere.

As shown in Table 2, using information on past successes also significantly reduces the number of FISHes for the data-parallel sumeuler from 34968 (baseline) by 57% to 15190 (qz10000) for the rather regular D&C parfib from 5137 by 23% to 3965 (qz100), contributing to the reduction of communication overhead which helps reduce execution time. On the other hand, history does not reflect well the run time behaviour of worpitzky (modest 2.5% decrease) and coins (a disappointing 37% increase), where the threads are more numerous and more fine-grained than in the other applications.

5 Conclusion and Future Work

We have investigated the effectiveness of using information on past stealing successes to improve victim selection of random work stealing to increase stealing success ratio and reduce communication overhead. We quantify run time performance of four applications on a cluster of multi-cores and use profiling data to explain application behaviour. We find improved run time of up to 34% along with increased fishing success rate and reduced number of FISH messages for data-parallel and for regular D&C applications. However, this heuristic fails in cases where past application behaviour is not predictive of the future behaviour as it is the case for more irregular D&C applications with large number of very fine-grained threads.

Ongoing work is focused on investigating automated ways of parameter selection and tuning as well as on exploring complementary policies such as temporary switching to work-pushing and using information on the source of the sparks to avoid exporting sparks that could have been successfully subsumed by the parent and would otherwise cause additional communication overhead if the results are needed by the parent. In general, predicting the amount of work associated with a spark proved very challenging [12], hence we aim to use additional information to *co-locate* sparks from the same source of parallelism to improve locality and selection of the spark to donate according to implicit ancestry dependencies.

In the future, we plan to add larger applications to the set of our benchmarks, to enrich the used information by architectural characteristics and to use cost models as a more systematic way to adaptively control policies within a RTS to achieve high performance portability for a high-level non-strict functional parallel programming language.

Acknowledgements This work is part of a collaboration with Hans-Wolfgang Loidl and Greg Michaelson to whom the author is grateful for encouragement and helpful discussions. The author also thanks three anonymous reviewers for comments that improved this paper.

References

1 A. Al Zain, P. Trinder, G. Michaelson, and H.-W. Loidl. Evaluating a high-level parallel language (GpH) for computational Grids. *IEEE Transactions on Parallel and Distributed Systems*, 19(2):219–233, 2008.

2 K. Asanovic, R. Bodik, J. Demmel, et al. A view of the parallel computing landscape. *CACM*, 52:56–67, October 2009.

3 M. Aswad, P. Trinder, and H.-W. Loidl. Architecture aware parallel programming in Glasgow parallel Haskell (GpH). *Procedia Computer Science*, 9:1807–1816, 2012.

4 E. Belikov, P. Deligiannis, P. Totoo, et al. A survey of high-level parallel programming models. Technical Report HW-MACS-TR-0103, Heriot-Watt University, 2013.

5 E. Belikov, H.-W. Loidl, and G. Michaelson. Characterisation of Parallel Functional Applications. In *Draft Proceedings of the 2014 Symposium on Trends in Functional Programming*, Utrecht University, 2014.

6 R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

7 A. Brodtkorb, C. Dyken, T. Hagen, J. Hjelmervik, and O. Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18(1):1–33, May 2010.

8 F. Burton and M. Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the Conference on Functional Program Language and Computer Architecture*, pages 187–194. ACM, 1981.

9 J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

10 J. Diaz, C. Munoz-Caro, and A. Nino. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1369–1386, 2012.

11 H. Gonzalez-Velez and M. Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience*, 40(12), 2010.

12 T. Harris and S. Singh. Feedback directed implicit parallelism. *ACM SIGPLAN Notices*, 42(9):251–264, September 2007.

13 P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of Haskell: being lazy with class. In *Proc. of the 3rd ACM SIGPLAN History of Programming Languages Conference*, pages 1–55, June 2007.

14 V. Janjic and K. Hammond. How to be a successful thief. In *Euro-Par 2013 Parallel Processing*, pages 114–125. Springer, 2013.

15 S. Marlow, P. Maier, H.-W. Loidl, M. Aswad, and P. Trinder. Seq no more: better Strategies for parallel Haskell. In *Proc. of the 3rd ACM Symposium on Haskell*, pages 91–102, 2010.

16 E. Mohr, D. Kranz, and R. Halstead Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

17 J. Owens, D. Luebke, N. Govindaraju, et al. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

18 W. Partain. The nofib benchmark suite of Haskell programs. In *Functional Programming, Glasgow 1992*, pages 195–202. Springer, 1993.

19 H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

20 P. Trinder, E. Barry Jr., M. Davis, et al. GpH: An architecture-independent functional language. *IEEE Transactions on Software Engineering*, 1998.

21 P. Trinder, K. Hammond, J. Mattson Jr., A. Partridge, and S. Peyton Jones. GUM: a portable parallel implementation of Haskell. In *Proc. of PLDI'96 Conf.*, 1996.

22 P. Trinder, K. Hammond, H.-W. Loidl, and S. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.

Everything you know is wrong: The amazing time traveling CPU, and other horrors of concurrency*

Ethel Bardsley

Imperial College London, UK
emb2009@ic.ac.uk

Abstract

In this paper, we shall explore *weak memory models*, their insidious effects, and how it could happen to you! It shall explained how and why both compilers and CPUs rewrite your program to make it faster, the inevitable fallout of this, and what you can do to protect your code. We shall craft a lock, building from a naïve and broken implementation up to a safe and correct form, and study the underlying model that requires these modifications as we go.

1998 ACM Subject Classification D.1.3 Concurrent Programming

Keywords and phrases Concurrency, weak memory, compilers

Digital Object Identifier 10.4230/OASIcs.ICCSW.2014.11

1 A tale of things to come

Once upon a time, life was simple. Compilers were straightforward, CPUs comprehensible. And things were good, for a time. But then the users' desire for more speed overwhelmed them. Deals were made, souls were sold, and so came about the pipeline, and with it speculation, and instruction level parallelism, and out of order execution. But while the processor's hunger grew, memories and buses could not keep up enough to sate the beast. And so came caching and buffering, more and more levels, towering above the masses. At this time, compilers also resorted to trickery, performing fiendish transformations, twisting a program's natural form to make it a more palatable meal. However, these terrible secrets below the surface were hidden from the software above, which could go about its day, blissfully unaware of the madness below the streets.

But then, in our hubris, we wanted multiple threads. And we wanted multiple CPUs to run our new threads. We even started to shed the locks placed to keep us safe, proudly declaring ourselves "lock free and scalable". And so the insanity started to leak.

While many, even most, of the plains and roads above were the same as ever, sometimes a stray program might stumble into things it should not see. A crunch, a scream. Some are killed instantly, others stumble on before falling some time later. Yet others, worse, survive but are "changed", returning to their daily lives, occasionally finding themselves somewhere unexpected with no knowledge of how they got there, or why they're standing, dazed, over the corpse of a now hideously corrupted file system.

But there are ways to fight back! Barriers, used to help forge the same locks we abandoned, could save us. With careful understanding and placement, we could keep the unspeakable out, while still reaping many of the benefits they provided us. And so it comes to me to warn of the danger, and pass the rites of protection on to you!

* This paper was written in the course of work funded by Intel Corporation.

Imperial College Computing Student Workshop (ICCSW'14).
Editors: Rumyana Neykova and Nicholas Ng; pp. 11–18
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2 In the beginning

Below is a C implementation of Peterson's mutual exclusion algorithm for two threads, 0 and 1 [10]:

```
 1  unsigned flag [2] = {0,0};
 2  unsigned turn = 0;
 3
 4  void lock (unsigned thread) {
 5          unsigned other_thread = (thread+1)%2;
 6
 7          flag[thread] = 1;
 8          turn = other_thread;
 9          do {
10                  //spin
11          } while (flag[other_thread] && turn == other_thread);
12  }
13
14  void unlock (unsigned thread) {
15          flag[thread] = 0;
16  }
```

Each thread has a flag with which to state its intention to lock (line 1), and there is a variable to indicate whose turn it is in the event they both want the lock at once (line 2). Upon entering `lock()`, the thread first sets its associated flag and lets it be the other thread's turn (lines 7–8). Then they wait until either the other thread's flag is unset, or it becomes their turn (lines 9–11). To unlock, the calling thread unsets its flag (line 15).

This locking mechanism has been proven correct [3], and indeed it is under the assumption that the code as written is what will be executed. However, as Knuth would remind us, simply proving code correct can be insufficient, and we shall see what is necessary to maintain correctness in the face of an actual implementation.

3 The very hungry compiler

Optimizing compilers are very clever machines. They take often-inefficient human-readable code and transform it into an equivalent program that is much faster. Alas, much of this is performed under the assumption that the code is either single-threaded, or that nothing will be shared between threads. Consider the core of the above lock implementation:

```
        flag[thread] = 1;
        turn = other_thread;
        do {
                //spin
        } while (flag[other_thread] && turn == other_thread);
```

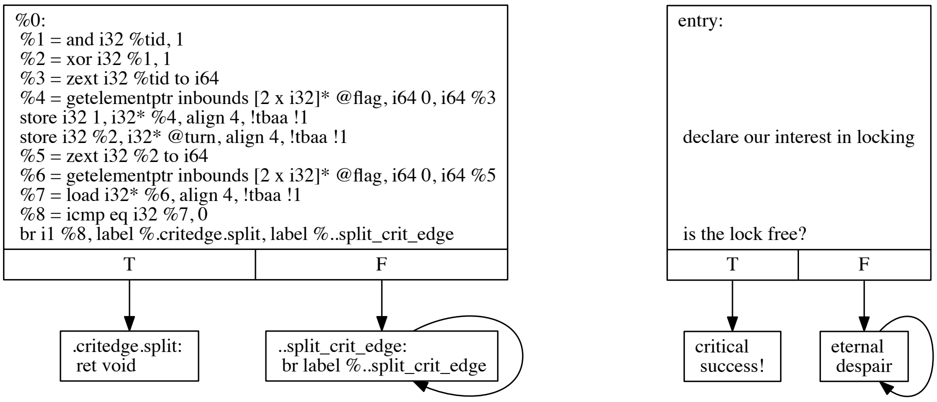
The compiler "knows" that `turn` is equal to `other_thread` because we set it that way before the loop, and so can remove that "redundant" part of the while condition:

```
        flag[thread] = 1;
        turn = other_thread;
        do {
                //spin
        } while (flag[other_thread]);
```

Similarly, `flag[other_thread]` is never modified by the loop, so is only checked once, before the loop:

```
        flag[thread] = 1;
        turn = other_thread;
        if (flag[other_thread]) {
          while (1) {
                //spin forever
          }
        }
```

And the compiler is totally "safe" to make these transformations — if this code were single threaded, these are optimizations you would even want your compiler to make. In fact, here is the control flow graph of the code generated by the Clang C compiler for the naïve implementation from Section 2 at optimization level O3:



```
%0:
 %1 = and i32 %tid, 1
 %2 = xor i32 %1, 1
 %3 = zext i32 %tid to i64
 %4 = getelementptr inbounds [2 x i32]* @flag, i64 0, i64 %3
 store i32 1, i32* %4, align 4, !tbaa !1
 store i32 %2, i32* @turn, align 4, !tbaa !1
 %5 = zext i32 %2 to i64
 %6 = getelementptr inbounds [2 x i32]* @flag, i64 0, i64 %5
 %7 = load i32* %6, align 4, !tbaa !1
 %8 = icmp eq i32 %7, 0
 br i1 %8, label %.critedge.split, label %..split_crit_edge
```

```
.critedge.split:
 ret void
```

```
..split_crit_edge:
 br label %..split_crit_edge
```

Figure 1 Control flow graph of LLVM bitcode for `lock()`, with English translation on the right

This obviously does not work. So what can we do to fix our lock? Some would suggest declaring the lock variables as `volatile`, as this provides two guarantees in C:

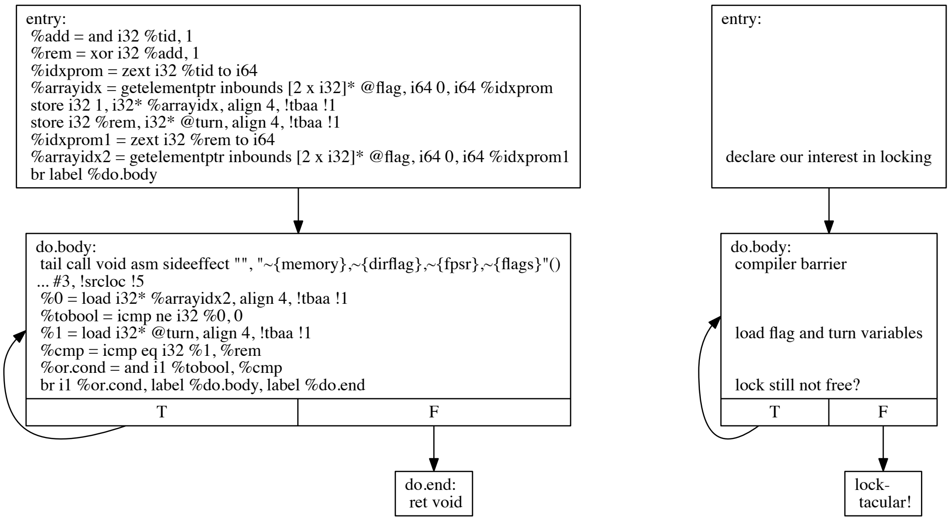
- The compiler may not omit a `volatile` access
- The compiler may not reorder `volatile` stores

However, we only require the first guarantee; reading `flag` or `turn` first is unimportant, and enforcing the second restricts acceptable optimization. Furthermore, `volatile` *does not* prevent the compiler reordering other accesses [2, 6], including whatever the lock may have been guarding, rendering the lock useless. Instead of `volatile`, we should insert a *compiler barrier*, which forces the compiler to not omit any reads or re-order accesses around it:

```
1         flag[thread] = 1;
2         turn = other_thread;
3         do {
4                 asm volatile ("":::"memory");
5         } while (flag[other_thread] && turn == other_thread);
```

The inserted line 4 adds a blank assembly instruction, tells the compiler not to remove it (`volatile`), with the `"memory"` parameter declaring that it can arbitrarily change, or "clobber" all of memory. This is sufficient for compiler to load `flag` and `turn` every iteration, while allowing it to optimize the condition itself. It also provides a stronger ordering guarantee than `volatile` variables, ensuring that any other code after the barrier stays there.

And to check the control flow graph:



```
entry:
 %add = and i32 %tid, 1
 %rem = xor i32 %add, 1
 %idxprom = zext i32 %tid to i64
 %arrayidx = getelementptr inbounds [2 x i32]* @flag, i64 0, i64 %idxprom
 store i32 1, i32* %arrayidx, align 4, !tbaa !1
 store i32 %rem, i32* @turn, align 4, !tbaa !1
 %idxprom1 = zext i32 %rem to i64
 %arrayidx2 = getelementptr inbounds [2 x i32]* @flag, i64 0, i64 %idxprom1
 br label %do.body
```

```
do.body:
 tail call void asm sideeffect "", "~{memory},~{dirflag},~{fpsr},~{flags}"()
... #3, !srcloc !5
 %0 = load i32* %arrayidx2, align 4, !tbaa !1
 %tobool = icmp ne i32 %0, 0
 %1 = load i32* @turn, align 4, !tbaa !1
 %cmp = icmp eq i32 %1, %rem
 %or.cond = and i1 %tobool, %cmp
 br i1 %or.cond, label %do.body, label %do.end
```

```
do.end:
 ret void
```

Figure 2 Control flow graph for `lock()` with compiler barrier inserted, with translation on right

Sweet!

4 The Call of Cthoncurrency

Alas, despite bending the compiler to our will, the hardware itself, the very fabric of execution, will conspire to undo us. This is where the true madness lies.

The increasing gap between CPU and RAM performance has required a number of solutions to keep the processor fed. As multiprocessor systems become the norm, the overhead required to maintain coherency between the many component parts spirals. And so rules were relaxed. Guarantees were loosened. Causality was called into question. And this is usually OK — like the very hungry compiler, this only matters when threads are sharing data, and only for those shared locations. For everything else, this speeds execution, simplifies CPU design, lowers power consumption, and generally makes everyone happy.

However, for those cases of shared data between threads, confusing and seemingly "impossible" bugs can arise. We shall describe the culprits, how they lead us here, and what we can do about it.

Sequential consistency The intuitive notion of multi-threaded programming [8], where instructions execute in the order you wrote them and all processors see memory the same way at the same time. Under this notion, executing a multi-threaded program on multiple processors is the same as doing so on a single one, and concurrency is easy to reason about. A simpler time, when the world made sense.

Instruction reordering Many modern CPUs implement *out-of-order execution*, meaning that instructions may not actually happen in program order. This allows many important performance optimizations, e.g. if a processor encounters a sequence `X;Y`, has the data for `Y` but not `X`, and `Y` does not depend upon `X`, then it can perform `Y` while waiting. The CPU will have a *reorder window*, within which it can perform dependence analysis and rearrange instructions for optimal throughput. While some designs (e.g. x86) will ensure external effects such as writes, are put back in the original order, many do not (e.g. ARM).

Memory buffering In modern computers, memory is very far away and very slow; reading from cache can be an order of magnitude slower than executing an instruction, with a full trip to main memory being an order slower again. Rather than stalling waiting for a write to finish, writes go into a per-core buffer to be resolved later while the core continues executing. Similarly, rather than waiting for a read to traverse the cache hierarchy, the reorder window allows the processor to see what locations will be required next and start reading them in ahead of time to be ready for when the instruction is actually executed.

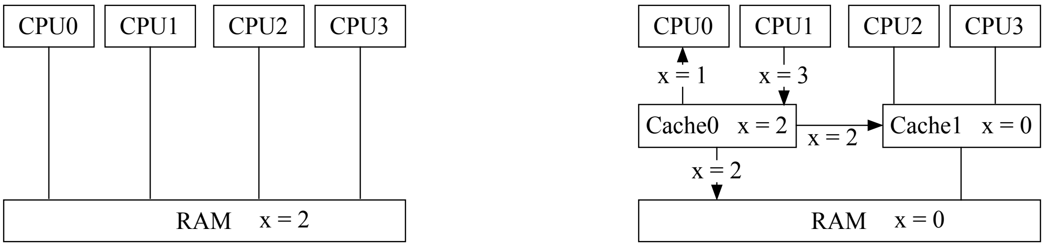


Figure 3 Intuitive view of memory in a multiprocessor system (left) vs harsh reality (right)

On the right,

- CPU_0 read an old value `x = 1`.
- CPU_1 just wrote `x = 3`, and will see that.
- Cache_0 contains the `x = 2` that CPU_1 wrote earlier in the program, and is propagating that to RAM and Cache_1.
- RAM and Cache_1 contain the original $x = 0$.

As a result of reordering and buffering, each CPU conceptually works in its own disconnected time bubble, reading from the distant past, writing to the distant future. Sometimes their timelines will cross, but, like the time traveller's wife, they will never know when, or how far the other has progressed.

Example: x86-TSO A (comparatively) simple example of weak memory is the model describing Intel and AMD's x86 and x86-64 processors, known as *x86-TSO* (Total Store Order) [9]. It differs from sequential consistency in only two ways: a processor might read old data, and writes take an unspecified amount of time to be globally visible. In all other ways, it is sequentially consistent:

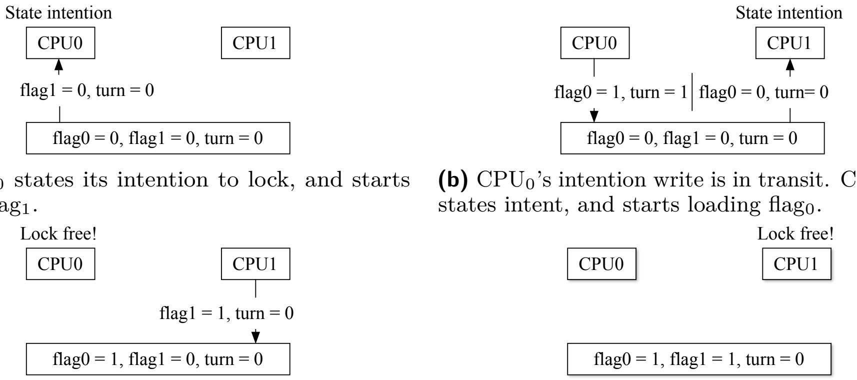
- Writes from a given processor, while arbitrarily delayed, are always in program order (e.g. Sparc's PSO (Partial Store Order) will not ensure this)
- Once a value is globally visible, it is immediately globally visible to all others (e.g. ARM does not provide this, allowing 'write atomicity relaxation')
- A read cannot have an older value than a prior read to the same location (e.g. DEC Alpha can do this, known as 'read-after-read reordering')
- A read cannot violate causality, reading data that no thread wrote (e.g. C++11 and Java allow 'out-of-thin-air reads')

But it still sufficiently deviates from sequential consistency to be problematic.

For example, while we prevented the compiler from discarding memory accesses, the hardware may yet reorder them. Due to the effects of read and write buffering, it may appear as though our lock was actually:

```
	do {
		asm volatile (""::: "memory");
	} while (flag[other_thread] && turn == other_thread);
	flag[thread] = 1;
	turn = other_thread;
```

The reads will have started ahead of time, and the writes will take a while to propagate down (see Figure 4 for a step-by-step of how this may happen). As a result, both threads can read stale data, incorrectly see the lock as free, and both proceed into the critical section, wreaking havoc as they do so.



(a) CPU$_0$ states its intention to lock, and starts loading flag$_1$.

(b) CPU$_0$'s intention write is in transit. CPU$_1$ also states intent, and starts loading flag$_0$.

(c) CPU$_0$ checks the copy of flag$_1$ it read in (a), sees it clear, progressing into the critical section. CPU$_1$'s write is in transit.

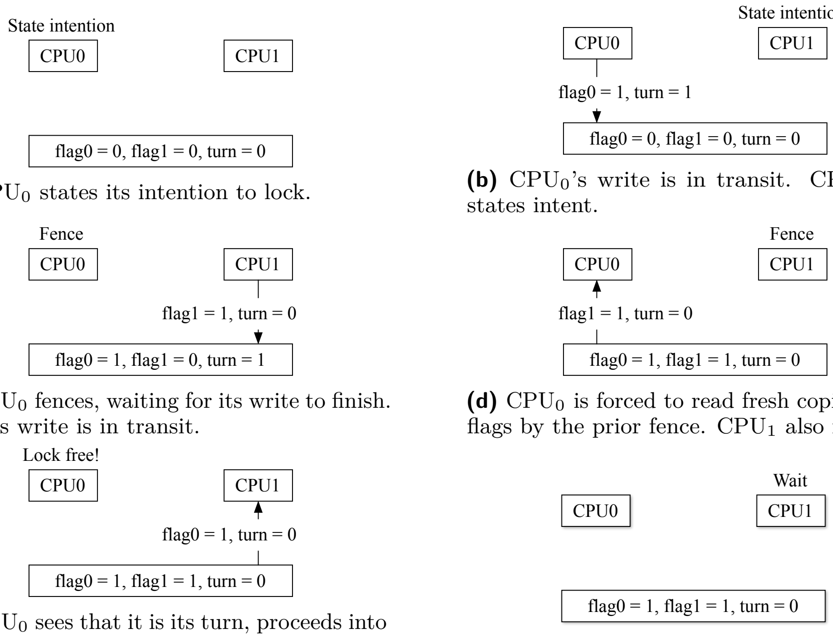
(d) CPU$_1$ checks the copy of flag$_0$ it read in (b), sees it clear. Both CPUs are now in the critical section.

Figure 4 Two CPUs entering the lock simultaneously under x86-TSO

So what can we do? Much like the compiler barrier, we can insert a *barrier instruction* to tell the processor to leave our program alone. An `MFENCE` instruction on x86 CPUs will stall until its write buffer has been fully flushed to memory, and forces it to perform reads after, rather than loading them up ahead of time. This allows us to regain just enough sequential consistency to ensure correctness. We can insert one between declaring intent to lock and checking for lock freedom (see Figure 5 for the step-by-step):

```
	flag[thread] = 1;
	turn = other_thread;
	asm volatile ("mfence")
	do {
		asm volatile (""::: "memory");
	} while (flag[other_thread] && turn == other_thread);
```

x86 also offers `SFENCE` and `LFENCE` to only enforce write or read ordering, respectively, and other architectures with weaker models often provide many more for fine-grained control. In "hot" or high performance code, it may be preferable to use the weakest possible (barriers force the CPU to slow down, and stronger means slower), but a full barrier like `MFENCE` will always be safe.


State intention
CPU0
CPU1
flag0 = 0, flag1 = 0, turn = 0
State intention
CPU0
CPU1
flag0 = 1, turn = 1
flag0 = 0, flag1 = 0, turn = 0
Fence
CPU0
CPU1
flag1 = 1, turn = 0
flag0 = 1, flag1 = 0, turn = 1
Fence
CPU0
CPU1
flag1 = 1, turn = 0
flag0 = 1, flag1 = 1, turn = 0
Lock free!
CPU0
CPU1
flag0 = 1, turn = 0
flag0 = 1, flag1 = 1, turn = 0
Wait
CPU0
CPU1
flag0 = 1, flag1 = 1, turn = 0

(a) CPU_0 states its intention to lock.

(b) CPU_0's write is in transit. CPU_1 also states intent.

(c) CPU_0 fences, waiting for its write to finish. CPU_1's write is in transit.

(d) CPU_0 is forced to read fresh copies of the flags by the prior fence. CPU_1 also fences.

(e) CPU_0 sees that it is its turn, proceeds into the critical section. CPU_1 reads fresh copies of the flags.

(f) CPU_1 sees it is not its turn, and waits.

Figure 5 Successful mutual exclusion!

5 The moral of the story

In this paper we have seen the effects of some compiler and hardware optimizations, and how these can cause unexpected behaviors in concurrent code. The simple take-home message is to avoid writing your own low-level concurrency primitives or "lock-free" concurrent data structures where possible. If you must, let paranoia be your guide: check compiler output for sensitive regions, and read in-depth descriptions of what your compilers and architectures will do [4]. Tools that assist in detecting where fences are required may also be of help [1, 5].

For some managed platforms, such as Java, the model is consistent between environments regardless of host architecture, simplifying the conceptual overheadfor managing concurrency without locks. Additionally, in the case of Java and C#, `volatile` does behave more like a barrier without the unintuitive potential for reordering of other accesses [7].

Alas, for C and C++ this is unfortunately both compiler and architecture specific. While the `asm` lines used in this paper are specific to GCC and Clang, equivalents exist for other compilers, such as `_ReadWriteBarrier()` compiler barrier and `MemoryBarrier()` hardware barrier in MSVC. To achieve more portable code, you should wrap these in compile-time macros, e.g. `compiler_barrier` and `full_barrier` [4]. There is also some support for atomic accesses and barriers introduced in the C11 and C++11 standards, however not all compilers support these yet, and some never will, and so they are also effectively non-portable.

On the other hand, if you write single-threaded code, or multi-threaded code without shared data structures, or use the locking mechanisms provided by your environment (e.g. pthreads), none of the above matters. You can go about your ways, unconcerned and carefree.

References

1 Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence: A static analysis approach to automatic fence insertion. *CoRR*, abs/1312.1411, 2013.

2 Jonathan Corbet. Volatile considered harmful. `https://www.kernel.org/doc/Documentation/volatile-considered-harmful.txt`. Retreived from Linux 3.15-rc7.

3 Micha Hofri. Proof of a mutual exclusion algorithm—a classic example. *ACM SIGOPS Operating Systems Review*, 24(1):18–22, 1990.

4 David Howells and Paul E. McKenney. Linux kernel memory barriers. `https://www.kernel.org/doc/Documentation/memory-barriers.txt`. Retreived from Linux 3.15-rc7.

5 Jeff Huang, Charles Zhang, and Julian Dolby. Clap: Recording local executions to reproduce concurrency failures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 141–152, New York, NY, USA, 2013. ACM.

6 ISO. Programming Language C (C11). Standard ISO/IEC 9899:2011, International Organization for Standardization, 2011.

7 JCP. JSR 133: JavaTM Memory Model and Thread Specification Revision. Standard, Java Community Process, 2004.

8 L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

9 Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 391–407. Springer Berlin Heidelberg, 2009.

10 Gary L. Peterson. Myths About the Mutual Exclusion Problem. *Information Processing Letters*, 12(3):115–116, 1981.

Identifying and inferring objects from textual descriptions of scenes from books

Andrew Cropper

Department of Computing, Imperial College London
a.cropper13@imperial.ac.uk

Abstract

Fiction authors rarely provide detailed descriptions of scenes, preferring the reader to fill in the details using their imagination. Therefore, to perform detailed text-to-scene conversion from books, we need to not only identify explicit objects but also infer implicit objects. In this paper, we describe an approach to inferring objects using Wikipedia and WordNet. In our experiments, we are able to infer implicit objects such as *monitor* and *computer* by identifying explicit objects such as *keyboard*.

1998 ACM Subject Classification I.2.10 Vision and Scene Understanding, I.2.6 Learning, I.2.7 Natural Language Processing

Keywords and phrases Text-to-Scene Conversion, Natural Language Processing, Artificial Intelligence

Digital Object Identifier 10.4230/OASIcs.ICCSW.2014.19

1 Introduction

Since the release of Toy Story in 1995, animation films have become increasingly popular [21]. As animators strive for photorealistic animation, budgets are spiralling [16]. Similarly, with next-generation consoles pushing the boundaries of computer graphics, expectations for visually aesthetic video games are increasing. Consequently, to generate more detailed content, games companies have to recruit more game artists, increasing the cost of development [17].

A text-to-scene conversion system (TTSCS) generates a 2D or 3D scene from a textual description provided by the user. For example, figure 1 shows the scene generated by Wordseye [7], a TTSCS described in section 2, for the text "The lawn mower is 5 feet tall. John pushes the lawn mower. The cat is 5 feet behind John. The cat is 10 feet tall". TTSCSs have the potential to reduce budgets and production times for animation films and video games by automatically generating scenes from user-provided scripts.

In this paper, we focus on textual descriptions of scenes from fiction books, where authors rarely provide detailed descriptions, preferring the reader to fill in the details using their imagination. Therefore, to perform detailed text-to-scene conversion from fiction books, we need to not only identify explicit objects but also infer implicit objects. For example, for the extract "I was going to email Van and Jolu to tell them about the hassles with the cops, but as I put my fingers to the keyboard, I stopped again.", a TTSCS needs to identify the keyboard and then infer that a computer is a likely accompaniment. In the following sections, we describe an approach using Wikipedia and WordNet.

Imperial College Computing Student Workshop (ICCSW'14).
Editors: Rumyana Neykova and Nicholas Ng; pp. 19–26
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

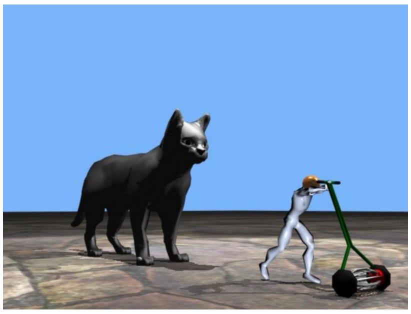


Figure 1 3D scene generated by WordsEye [7] for the text "The lawn mower is 5 feet tall. John pushes the lawn mower. The cat is 5 feet behind John. The cat is 10 feet tall.".

1.1 Contributions

To our knowledge, this is the first paper to investigate inferring objects in a TTSCS using Wikipedia as a source of world-knowledge. Our main results are as follows:

- we demonstrate a system able to identify explicit objects using WordNet
- we demonstrate a system able to infer implicit objects using Wikipedia

1.2 Paper outline

This paper is organised as follows. We detail related work in section 2. In section 3, we discuss the limitations of the existing work and propose a solution that uses Wikipedia and WordNet. This is followed in section 4 by a description of the Python implementation used in the experiments. In the experiments in section 5 we show that explicit objects can be identified and implicit objects can be inferred using our technique. Finally, we conclude the paper and propose future work in section 6.

2 Related work

Often cited as the first TTSCS, SHRDLU [22] allows users to enter natural language commands, which are reissued to a virtual robot arm which moves blocks around in a small 'blocks world'. SHRDLU has a basic vocabulary of objects and properties, and basic semantics for interpreting them and the environment to which they apply [20].

Following SHRDLU came two microworld systems (highly restricted sets of objects or ideas that operate in accordance with a limited set of rules). The Clowns of Microworld [18] processes user-commands concerning spatial descriptions of clowns and their actions to generate a simple diagram of a scene. Similarly, NALIG (Natural Language Image Generation) [1] takes sequences of descriptions regarding the spatial relations between objects to generate a scene.

WordsEye [7] generates 3D scenes from user descriptions. WordsEye works by converting a parse tree to a dependency representation, which is then converted into a semantic representation. Depiction rules convert the semantic representations into a set of low-level depictors (representing 3D objects, poses, spatial relations, colour attributes, etc). Conflicting and illegal depictors are removed, and a final 3D scene is generated.

Carsim [8, 2, 9] takes textual police traffic accident reports and generates 3D animations from them. It uses a number of natural language techniques such tokenising, part-of-speech tagging, noun group detection, and domain-specific multi-words to parse a report's text into a formal representation outlining what happened. Using temporal relations between the events, Carsim is able to generate and then animate the scene.

Other TTSCSs include: [6] which also focuses on spatial relationships and uses restricted keywords such as *in*, *on*, and *at* to manipulate spatial arrangements of existing objects within a scene; [10] who propose an automatic text illustration system which automatically extracts keywords from the text and matches these keywords to a database of pictures; [14] and [23] who both present systems to generate pictures from a natural language text; and [4] who focus on automatically translating natural language patient instructions into pictures.

Sproat [19] looks into inferring the environment in a TTSCS. For example, for the sentence "John was eating breakfast", Sproat tries to infer that the scene is taking place in a kitchen. The system is able to match events such as 'wash clothes' to 'laundry room' and 'wash hands' to 'bathroom'. However, the system does not attempt to populate matched rooms with objects you would expect to find within them.

To our knowledge, the only other work that focuses on inferring implicit relations is [5], who use an existing 3D scene database to create a spatial knowledge-base with priors on the hierarchy of objects in scenes. Specifically, they parse the input text into a scene template, which places constraints on what objects must be present and the relationships between them. They then use the priors from the spatial knowledge-base to expand the scene template by inferring additional implicit constraints. These implicit constraints are then used to select objects from an object dataset.

3 Limitations and proposed solutions

Existing TTSCS tend to focus on extracting specific information from text, then matching this information to a pre-determined and often basic vocabulary of objects and properties. Once matched, most systems emphasise the spatial relationships of objects to position them within a scene. To our knowledge, only one other paper [5] considers objects not explicitly mentioned within the text. In addition, the majority of existing systems, including [5], ignore the dynamic nature of natural language, and often the only dynamic content within these systems are the heavily constrained, and in some cases, simple narratives supplied by the user.

We propose that by first identifying explicit objects, we can then infer implicit objects. For example, for the sentence "She placed the pen on the desk", we suggest that we can infer a chair by identifying the desk. However, exploring the semantic properties of a desk suggests nothing concerning its relationship to a chair. A desk has draws, legs, and is a type of table, but there is nothing in its semantic properties referring to its relationship to a chair. Humans make this inference using world-knowledge. For a machine to perform this inference, we need a machine-readable source of world-knowledge. In the following section, we describe an approach to this problem using Wikipedia. This approach differs from [5] since we use a dynamic source of world-knowledge which is adaptable to changes in natural language, whereas [5] use a static database of objects collected from 133 small indoor scenes.

4 Implementation

We now describe the Python implementation of our system. We use the Natural Language Toolkit [3] for the natural language processing tasks. Note that in this paper, we ignore the problem of scene detection, i.e. we take a book already parsed into scenes as input. See [12] for work on scene detection.

4.1 Object identification

The input to the system is a collection of scenes from a book, where each scene is a string. For each scene, we identify nouns by tokenising and part-of-speech (POS) tagging [11] the text. Plurals are singularised and word frequencies aggregated, so that each scene is reduced to a set of nouns and associated frequencies.

Only extracting nouns is insufficient to identify objects. For example, in the sentence "I went to the shop on Wednesday", the word *Wednesday* is a noun, but not a physical object. We use WordNet [15], a lexical semantic dictionary, to identify physical objects. WordNet places words into one or more logical categories, of which there are 45, including the following:

- noun.artifact: denoting man-made objects
- noun.communication: denoting communicative processes and contents
- noun.location: denoting spatial position
- noun.person: denoting people

We use the *noun.artifact* category to decide whether a word refers to a physical object.

A reader might question the need to POS tag the text if we use WordNet to identify objects. POS tagging is necessary because words of the *noun.artifact* category can appear in a scene but not as nouns. For example, in the sentence, "The politician wishes to table an amendment to the proposal", the word *table* is of the type *noun.artifact* but is used as a verb, not an object.

As mentioned, WordNet places words into one or more logical categories. If a word is in multiple categories, then WordNet orders the categories by estimated frequency of use[1]. For example, the word *mouth* has eight entries in WordNet (entries 3-7 are omitted for brevity):

1. noun.body: mouth, oral cavity, oral fissure, rima oris (the opening through which food is taken in and vocalizations emerge)
2. noun.body: mouth (the externally visible part of the oral cavity on the face and the system of organs surrounding the opening)
8. noun.artifact: mouth (the opening of a jar or bottle)

Here, the *noun.artifact* category is the least likely interpretation. But how do we know which interpretation corresponds to the sense of the word in the scene? To investigate this, we checked WordNet for the 20 most frequent nouns in a chapter of a book (described in section 5.1). We found that the correct interpretation of the word was in the first three results returned by WordNet for 19 of the 20 words. Therefore, we only use the first three results to decide whether a word refers to a object. Developing a more sophisticated approach, such as using word-sense disambiguation, remains a topic for future work.

As a final step, we aggregate synonyms (identified using WordNet) to end with a set of words and frequencies that potentially refer to objects in a scene.

4.2 Object inference

Having identified potential objects in a scene, the task is to infer implicit objects. To do this, we look at the corresponding Wikipedia page for each potential object in a scene. For example, if we identify the word *chair* as being a potential object, we look at the

[1] https://wordnet.princeton.edu/wordnet/man/wn.1WN.html

corresponding Wikipedia page for the word *chair*[2]. We extract the contents of the page using the Wikipedia export pages[3] and repeat the object identification steps described in section 4.1. Following this step, we have a set of words and frequencies that potentially relate to objects in a scene.

4.3 Object ranking

We assign each object a score to reflect how important it is in a scene. This ensures that less common items are represented in the scene. For example, we might assign the word *clarinet* a higher score than the word *chair* because a *clarinet* is a less common object. The tfidf (term-frequency inverse-document-frequency) weighting scheme [13] is used in information retrieval and text mining to rank documents based on their similarity to a query. This scheme assigns each word a value which increases proportionally to the number of times the word appears in a document, but is offset by the frequency of the word in the corpus, which helps to discriminate against words that generally appear more often than others. Specifically, in the tfidf scheme, the term frequency component *tf* is calculated by summing the instances of a term t in a document d. The document frequency component *df* is computed by dividing the total number of documents n by the number of documents that term t occurs in d, then taking the log of this value. These two values are multiplied to give the weighting for a term. We use this weighting scheme to rank the objects in a scene. Specifically, we say that a scene and its inferred objects form a document, and that the collection of all scenes forms the corpus. The term frequency component is the frequency of an object in a document (scene), and the document frequency component is the number of documents (scenes) in which each object appears. Using this scheme, we run the object identification and object inference steps for all scenes in our dataset and rank the objects using their tfidf score.

5 Experiments

The experiments investigate whether our system is able to identify and infer objects in a scene.

5.1 Materials

In this paper, we work with unlabelled data, i.e. we are given a book and the task is to identify and infer objects without any supervision. Therefore, the results must be evaluated using intuition. Experimentation using labelled data is left for future work.

For the choice of textual material, we experimented with several texts, all in the public domain. However, we found that the system often inferred anachronisms. Therefore, for the following experiments, we use Corey Doctorow's 'Little Brother'[4], chosen specifically due to its modern content, e.g. it includes objects such as *radio* and *Xbox*.

5.2 Methods

For the experiments, the input is chapter 7 from Corey Doctorow's 'Little Brother' manually parsed into scenes. The output is a list of potential objects for each scene ordered by their tfidf score.

[2] http://en.wikipedia.org/wiki/Chair
[3] http://en.wikipedia.org/wiki/Special:Export
[4] http://craphound.com/littlebrother/download/

Table 1 Words and inferred related words ordered in descending order by tfidf.

keyboard	**telephone**	**computer**	**screen**	**bed**
key	microphone	machine	computer	mattress
computer	receiver	microprocessor	panel	box
typewriter	coil	telephone	tube	frame
screen	handset	transistor	cathode	bedding
keypad	bell	keyboard	stand	mortise
laptop	telegraph	disc	keyboard	cot
joystick	candlestick	webcam	telephone	bedpost

5.3 Results

We now describe the results for one scene from chapter 7 of Corey Doctorow's 'Little Brother'. Results for other scenes are omitted for brevity.

5.3.1 Object identification

For the scene, we identified the following objects:

bed, computer, jail, camp, picture, telephone, room, ceiling, projector, wall, filter, screen, microscope, bag, radar, keyboard

Most objects are present in the scene, but there are exceptions. For the sentence "If anyone knew how to keep our butts out of jail, it would be him", we incorrectly identified that the word *jail* is an object in the scene. This example highlights a problem with using the tfidf weighting scheme because the word *jail* was not in any other scene, and was thus given a high weighting. In addition, we missed several objects. For example, the scene starts with the sentence "I hooked up my Xbox as soon as I got to my room". However, we did not detect the word *Xbox* as an object because this word does not exist in WordNet. Neither do many commonplace technological objects such as *iPhone*, *iPod*, *Wii*, *Mac*, etc. This is a limitation of the work, i.e. we are restricted to objects in WordNet.

5.3.2 Object inference

Having identified explicit objects, the next task is to infer implicit objects. Table 1 shows a sample of the results for this step where the column header is an explicit object identified in the scene and row items are inferred objects, displayed in descending order by their tfidf score. The results for *keyboard* are particularly good, with *computer*, *laptop*, and *joystick* all inferred. In comparison, the results for *telephone* are less impressive, with objects such as *coil*, *telegraph*, and *candlestick* inferred.

6 Conclusions and future work

In this paper, we have described a technique which uses Wikipedia and WordNet to identify explicit objects and infer implicit objects from textual descriptions of a scenes from a book. No known existing literature has attempted this. Our results show potential, and we are able to infer implicit objects such as *keyboard* and *screen* by identifying explicit objects such as *computer*.

The main issue complicating the work is the ambiguity of natural language, which is a common problem across all areas of natural language processing [11]. For example, when identifying explicit objects from text, we did not disambiguate if an object was in the scene or if it was just being discussed. Future work should integrate more sophisticated natural language processing techniques to resolve such ambiguity, such as using word-sense disambiguation techniques. In addition, we described an unsupervised learning technique. In future work, it would be interesting to investigate whether a semi-supervised machine learning approach would work.

References

1 Giovanni Adorni, Mauro Di Manzo, and Fausto Giunchiglia. Natural language driven image generation. In *Proceedings of the 10th international conference on Computational linguistics*, pages 495–500. Association for Computational Linguistics, 1984.

2 Ola Åkerberg, Hans Svensson, Bastian Schulz, and Pierre Nugues. Carsim: an automatic 3d text-to-scene conversion system applied to road accident reports. In *Proceedings of the tenth conference on European chapter of the Association for Computational Linguistics-Volume 2*, pages 191–194. Association for Computational Linguistics, 2003.

3 Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python*. " O'Reilly Media, Inc.", 2009.

4 Duy Bui, Carlos Nakamura, Bruce E Bray, and Qing Zeng-Treitler. Automated illustration of patients instructions. In *AMIA Annual Symposium Proceedings*, volume 2012, page 1158. American Medical Informatics Association, 2012.

5 Angel X Chang, Manolis Savva, and Christopher D Manning. Semantic parsing for text to 3d scene generation. *ACL 2014*, page 17, 2014.

6 Sharon Rose Clay and Jane Wilhelms. Put: Language-based interactive manipulation of objects. *Computer Graphics and Applications, IEEE*, 16(2):31–39, 1996.

7 Bob Coyne and Richard Sproat. Wordseye: an automatic text-to-scene conversion system. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 487–496. ACM, 2001.

8 Sylvain Dupuy, Arjan Egges, Vincent Legendre, and Pierre Nugues. Generating a 3d simulation of a car accident from a written description in natural language: The carsim system. In *Proceedings of the workshop on Temporal and spatial information processing-Volume 13*, page 1. Association for Computational Linguistics, 2001.

9 Richard Johansson, David Williams, Anders Berglund, and Pierre Nugues. Carsim: a system to visualize written road accident reports as animated 3d scenes. In *Proceedings of the 2nd Workshop on Text Meaning and Interpretation*, pages 57–64. Association for Computational Linguistics, 2004.

10 Dhiraj Joshi, James Z Wang, and Jia Li. The story picturing engine—a system for automatic text illustration. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP)*, 2(1):68–89, 2006.

11 Dan Jurafsky and James H Martin. *Speech & language processing*. Pearson Education India, 2000.

12 Marie Louise Lingaya. Automatic scene extraction from natural language text. Master's thesis, Nottingham Trent University School of Science and Technology, UK, 2008.

13 Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

14 Rada Mihalcea and Chee Wee Leong. Toward communicating simple sentences using pictorial representations. *Machine Translation*, 22(3):153–173, 2008.

15 George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

16 K Onstad. Pixar gambles on a robot in love. `http://www.nytimes.com/2008/06/22/movies/22onst.html`, 2008. Accessed: 25-06-2014.

17 J Reimer. Cross-platform game development and the next generation of consoles. `http://arstechnica.com/old/content/2005/11/crossplatform.ars/7`, 2005. Accessed: 25-06-2014.

18 Robert F Simmons. The clowns microworld. In *Proceedings of the 1975 workshop on Theoretical issues in natural language processing*, pages 17–19. Association for Computational Linguistics, 1975.

19 Richard Sproat. Inferring the environment in a text-to-scene conversion system. In *Proceedings of the 1st international conference on Knowledge capture*, pages 147–154. ACM, 2001.

20 Daniel Allen Tappan. *Knowledge-based spatial reasoning for automated scene generation from text descriptions*. PhD thesis, New Mexico State University, 2004.

21 Meng Wang. Research on the relationship between story and the popularity of animated movies. Master's thesis, Purdue University, United States, 2012.

22 Terry Winograd. Procedures as a representation for data in a computer program for understanding natural language. Technical report, DTIC Document, 1971.

23 Xiaojin Zhu, Andrew B Goldberg, Mohamed Eldawy, Charles R Dyer, and Bradley Strock. A text-to-picture synthesis system for augmenting communication. In *AAAI*, volume 7, pages 1590–1595, 2007.

Predicate Abstraction in Program Verification: Survey and Current Trends*

Jakub Daniel[1] and Pavel Parízek[2]

1 **Charles University in Prague**
daniel@d3s.mff.cuni.cz
2 **Charles University in Prague**
parizek@d3s.mff.cuni.cz

Abstract

A popular approach to verification of software system correctness is model checking. To achieve scalability needed for large systems, model checking has to be augmented with abstraction. In this paper, we provide an overview of selected techniques of program verification based on predicate abstraction. We focus on techniques that advanced the state-of-the-art in a significant way, including counterexample-guided abstraction refinement, lazy abstraction, and current trends in the form of extensions targeting, for example, data structures and multi-threading. We discuss limitations of these techniques and present our plans for addressing some of them.

1998 ACM Subject Classification D.2.4 Software/Program Verification, F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases program verification, model checking, predicate abstraction, refinement

Digital Object Identifier 10.4230/OASIcs.ICCSW.2014.27

1 Introduction

Software systems are nowadays ubiquitous and therefore it is important that they work correctly. An erroneous behavior may lead to loss of lives, money, and resources especially in safety-critical systems that are used, for example, in the domains of transportation and healthcare. It is necessary that as many errors as possible are detected and fixed before the deployment of a given system. We need techniques and tools able to reason about the behavior of programs in order to check the program safety and find errors. One approach to detection of errors is the use of program verification. The state-of-the-art verification techniques work for simple programs, but it is desirable that they scale also to large programs, have a good performance, and require little manual effort (automation).

In this paper, we provide an overview of selected recent contributions in the area of program verification that address these challenges. A very popular approach to program verification is model checking [15], which decides whether a given program satisfies a desired property by exhaustively exploring its state space. The state space of non-trivial programs is generally too large, and therefore an abstraction of some kind is needed to reduce its size. In particular, predicate abstraction [22] has been a subject of extensive research (e.g., [5, 11, 24, 28, 29]). We introduce the reader to successful verification techniques based on predicate abstraction and discuss possible directions of their further improvement.

We use the example program in Figure 1 throughout the paper to illustrate the basic principles of checking program safety with predicate abstraction. The program consists

* This work was partially supported by the Grant Agency of the Czech Republic project 14-11384S.

Imperial College Computing Student Workshop (ICCSW'14).
Editors: Rumyana Neykova and Nicholas Ng; pp. 27–35
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
1: procedure MAIN()
2:     $a \leftarrow$ an arbitrary array of integers
3:     $i \leftarrow$ FINDGREATER$(a, 10)$
4:     assert $i <$ LENGTH$(a) \implies a[i] > 10$

5: function FINDGREATER$(a, c)$
6:     for $0 \leq j <$ LENGTH$(a)$ do
7:         if $a[j] > c$ then
8:             return $j$
9:     return LENGTH$(a)$
```

Figure 1 Example program.

of the procedure MAIN, which creates an array of an arbitrary length, calls the function FINDGREATER, and subsequently asserts a specific property of the returned value i. The function FINDGREATER takes array a and constant c as arguments, and by iterating over the array a it finds an index j such that the array element $a[j]$ is greater than c. If no such index exists it returns the length of the array a. There are two locations, associated with lines 4 and 7, where the program execution may fail. The assertion at line 4 is violated if the value of the variable i points to an element of the array a whose value is less then or equal to 10. An error might arise also when attempting to access elements outside the bounds of the array at line 7.

The rest of the paper is structured in the following way. In Section 2, we give an overview of the state-of-the-art verification techniques based on predicate abstraction, and describe a procedure that can be used to verify the example program in Figure 1. In Section 3, we discuss limitations of the state-of-the-art techniques and the corresponding challenges. Finally, we present our plans and goals in this research area, including our current work to date (Section 4). Note that although in this paper we focus only on techniques related to predicate abstraction, there are many other approaches to program verification such as bounded model checking [13], and techniques tailored to efficient detection of errors of specific types, such as wrong usage of data structures and pointers [7, 20].

2 Overview

All the techniques presented in this section apply model checking to abstract programs. Their general idea is to automatically construct the most coarse-grained abstraction of an input program that is sufficient to prove the program safe. Such an abstraction captures all feasible behaviors of the program and also some infeasible behaviors.

First, we describe the basic procedure for model checking with predicate abstraction in Section 2.1. This is followed by a description of techniques that build on the basic procedure. We focus on verification techniques that can be divided into the following four categories: counterexample-guided abstraction refinement (Section 2.2), lazy abstraction (Section 2.3), combinations of multiple approaches (Section 2.4), and techniques targeting data structures, concurrency, and modular design (Section 2.5). The order of the subsections follows roughly the chronological development of the respective techniques.

2.1 Model Checking with Predicate Abstraction

The use of predicate abstraction enables efficient reasoning over an abstract state space that is much smaller than the original concrete one, because each abstract state represents a possibly large set of concrete states. Each abstract state corresponds to a specific valuation of *abstraction predicates* that express relationships between program variables.

In order to verify that the example program (Figure 1) is safe, it is necessary to determine whether there exists an execution under which any of the error locations is reached. The first

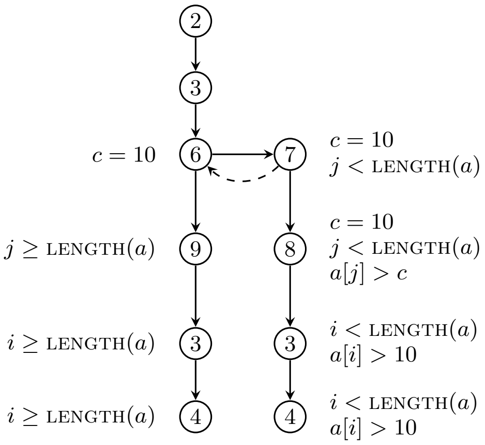


Figure 2 Abstract state space.

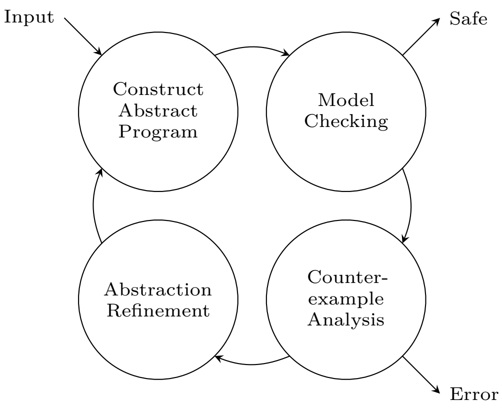


Figure 3 CEGAR loop.

step is to reduce the size of the state space to be explored by constructing an abstraction in the form of a boolean program [5, 17]. A set of abstraction predicates must be defined and given as input to the construction of the abstract boolean program. In the case of the example program, the set of abstraction predicates necessary to prove its safety includes the following: $i <$ LENGTH(a), $a[i] > 10$, $j \geq 0$, $j <$ LENGTH(a), $a[j] > c$, and $c = 10$. Although the predicates can be defined manually, the general aim is to design automated techniques for their inference and construction of the abstraction.

Now, we describe a run of the model checking procedure over the example program in Figure 1. The states of the abstract program correspond to different valuations (*true*, *false*, or *unknown*) of the abstraction predicates at different program locations. Figure 2 shows the whole reachable abstract state space. Each node represents an abstract state — it is labeled with the corresponding program location (source code line number) and predicates that evaluate to *true*.

The process of verification of this program starts at the entry point of MAIN (line 2) with all the predicates valuated to *unknown*. The program statements are executed symbolically until the invocation of FINDGREATER at line 3. At this point, the valuation of $c = 10$ is set to *true*, and the program execution continues at line 6, where branching takes place; one branch corresponds to the variable j pointing beyond the bounds of the array a, and the second branch corresponds to j pointing to some element of the array. The verification procedure needs to explore both state space branches. Suppose the procedure takes the first branch, so that the program execution continues to line 9 (left side of Figure 2). After returning to MAIN, the abstraction predicate $i <$ LENGTH(a) valuates to *false*, and therefore the assertion at line 4 holds. Next, the verification procedure explores the second branch, i.e. the body of the loop, and the program execution reaches line 7 (right side of Figure 2). The loop invariant $0 \leq j <$ LENGTH(a) ensures that only array elements at valid indices are accessed. Again, both branches of the `if`-statement have to be explored. We describe only the branch in which the body of the `if`-statement is executed because the other leads to an already visited state (through the dashed arc). In this branch, both the abstraction predicates $a[j] > c$ and $c = 10$ valuate to *true* at line 8. After returning to MAIN, the value of j is assigned to the variable i, hence $a[i] > 10$ is *true* and the assertion is satisfied. All the traces in the abstract program have been explored without reaching an error state, and thus the example program is safe.

One of the key requirements for automation of such verification procedure is that effects of individual statements on valuation of abstraction predicates are encoded precisely. A natural approach is to use logic formulæ. Obviously, the choice of a suitable logic plays an important role and affects the precision and complexity of the verification technique that relies on it. The building blocks of commonly used theories of the first-order logic include: equality logic, linear integer arithmetic, and uninterpreted functions. SMT solvers are often used to decide validity of formulæ that capture effects of program statements. However, invocations of an SMT solver are typically expensive, and therefore some of the techniques that we describe later try to minimize the number of invocations performed during the verification of a given program.

2.2 Counterexample-Guided Abstraction Refinement

The first technique for automated program verification based on predicate abstraction that we describe is *Counterexample-Guided Abstraction Refinement* (CEGAR) [16]. Figure 3 shows the main loop of the algorithm, which consists of these four steps: construction of an abstract program, model checking, analysis of a counterexample, and refinement of the abstraction. The initial abstraction does not consider any data values and the control-flow at branching points is entirely non-deterministic. Such an abstraction usually permits spurious executions, which do not correspond to real executions of the original program. This is true especially for infeasible counterexamples, which are eliminated by abstraction refinement. Note that the abstract program is constructed from scratch in each iteration.

In the rest of this section, we describe the individual steps of the main loop:

1. An abstract boolean program is constructed for the given input program. The boolean program contains boolean variables that represent values of the abstraction predicates, and captures the effects of statements from the original program on the values of predicates. Every assignment statement in the original program is modeled in the abstract program by an assignment of boolean values to variables representing individual predicates. Branching conditions are modeled by the boolean variable that represents the corresponding abstraction predicate. If the value of the predicate is *unknown* or no such abstraction predicate is defined, a non-deterministic choice is used.
2. A model checker is used to verify the abstract program. In case an error state is reachable, the model checker produces a counterexample in the form of a trace that represents the program execution from the initial state to the error state. If the abstract program is safe then the original program is *safe* as well.
3. The next step is to analyze the counterexample (error trace). The error trace is associated with a trace formula, which is constructed by conjoining subformulæ that express the semantics of individual statements in the trace. Satisfiability of this formula is checked to determine feasibility of the counterexample. If there exists a satisfying assignment to variables of the trace formula, then we get *a real counterexample*, which is then reported to the user.
4. Infeasible counterexamples are eliminated through refinement of the abstraction. New predicates are inferred based on the counterexample, and used in the next iteration to improve precision of the generated abstraction, so that the trace representing the infeasible counterexample is no longer permitted.

The verification procedure terminates when a real counterexample is encountered (step 3), or the program is proven safe (step 2). CEGAR is the basis of a large number of automated approaches to program verification, some of which we describe in the next three sections.

2.3 Lazy Abstraction

The construction of the full abstract boolean program from scratch in each iteration of the CEGAR loop is costly. More recent techniques [24, 28] use an incremental approach to abstraction refinement. Their key differences from the techniques described in the previous section is that (1) both construction and refinement of the abstraction are performed on-the-fly during model checking, and (2) the refinement does not affect already explored parts of the abstract state space, which saves a large computational effort. The basic idea is to iteratively unroll a control-flow graph of the given program into an *abstract reachability graph* (ARG). The nodes of ARG correspond to abstract states and edges reflect the program control-flow. At the beginning of the verification procedure, the ARG contains only the initial state. In each iteration, the procedure expands a leaf state and adds its children states into the graph. If an error state is added into the ARG, the corresponding error trace is analyzed for feasibility. The error may be either real, in which case it is reported and the procedure terminates, or spurious. In the case of a spurious error, the abstract states along the corresponding trace are refined with new predicates in order to eliminate the error. The new predicates for each location on the trace are obtained using interpolants [25, 28]. The locality of the newly added predicates makes it possible to keep the abstract state space reasonably small, as the verification procedure refines the precision only where necessary. The procedure uses a *covering* relation to ensure that, in each step, it explores only those abstract states that have not yet been processed before.

When unrolling the ARG, the abstraction may be constructed either in an eager or lazy fashion. The eager approach [25] computes the reachable abstract states using an *abstract post operator*, which involves multiple SMT queries. Blast [9] is an example of a verification tool that uses lazy predicate abstraction with an eager construction of the abstraction. The lazy approach [28], called Impact, overapproximates all abstract states with *true* and postpones the construction of a precise abstraction to the refinement step.

2.4 Combinations of Multiple Approaches

Lazy abstraction is an efficient technique as long as a relatively coarse abstraction is sufficient to find a real error or prove safety of the given program. Otherwise, the verification procedure has to perform a large number of refinement steps, which is computationally expensive. Loops in the program code, in particular, often require a complete unrolling of the ARG and numerous refinement steps, which reduce the benefits of abstraction. In this section, we describe some techniques that address this problem and improve scalability.

Yogi [23] is a tool that implements the Dash algorithm [6], which combines testing and verification in order to achieve better performance. The Dash algorithm works in an iterative fashion, and maintains two data structures: (1) an abstraction of the input program, which serves for proving the program's safety, and (2) a collection of already executed tests. At the start of each iteration, the algorithm inspects the current abstraction to see if there is any abstract error trace (counterexample). For the counterexample, Dash identifies the longest prefix that is covered by previously executed tests, and attempts to construct a new test that follows the prefix and reaches the next state in the counterexample after the prefix. If there is such a test, either its execution confirms the presence of a real error, or it guides abstraction refinement along the spurious counterexample. Otherwise, if such a test does not exist, the last abstract state of the prefix is refined by adding predicates that are derived from the transition between the prefix and the rest of the trace. An advantage of this approach is that it is computationally much less expensive for two reasons: (1) execution of tests may save many refinement steps, and (2) predicates are derived without the use of SMT.

Ufo [2] is a verification framework, which combines a configurable forward computation of the program abstraction with usage of interpolants to achieve sufficient precision. Depending on the particular configuration, the abstraction may be constructed in an eager or lazy manner (as in lazy abstraction). Ufo differs from the previously mentioned approaches in that it does not construct a separate trace formula for every trace that reaches an error location. Instead, it captures multiple traces in the ARG with a single formula, and thus significantly reduces the number of necessary refinement steps. A satisfying assignment of the formula's variables yields a real counterexample, whereas unsatisfiability of the formula enables the use of interpolants to refine the abstraction. In this way, the ARG is refined globally rather than along a single trace at a time, and each abstract state is refined as in lazy abstraction.

Another notable framework is CPAchecker [11], which supports custom *configurable program analyses* (CPA) [10]. The definition of each analysis consists of an abstract domain, transfer relation, merge operator, and a stop operator. The merge operator specifies if and how to merge abstract states when two control-flow paths meet. The stop operator detects whether a newly reached state is already covered and does not have to be explored again. Multiple custom analyses can be put together to form a combined analysis. The verification algorithm of CPAChecker performs a simple reachability analysis parametrized with a given CPA. It is implemented in the tool with the same name [11], which provides the CPA's with a compact interface to other necessary pieces of modern program verification frameworks, such as the input parser and SMT solvers.

2.5 Data Structures, Concurrency, and Modularity

In this section, we provide an overview of recent notable advancements within verification techniques towards support of realistic programs. This support is important for broader practical applicability, precision, and scalability of program verification. Specifically, we show techniques that can handle some of the following aspects of realistic programs: heap and data structures, object-oriented constructs, modular design (libraries), and concurrency.

Basically, every larger program uses data structures such as arrays, linked lists, and trees. This makes verification more challenging because more complex logics have to be used for reasoning about such programs. There exist verification techniques that address arrays [3, 26] and data structures of other kinds [14, 27].

The approach proposed in [3] focuses on precise reasoning over arrays of unknown length. Interesting properties of arrays and their elements (e.g., whether a given array is sorted) can be expressed only using quantifiers. The basic idea of [3] is to compute an overapproximation of the set of states backward-reachable from error states [21]. Then, the task of verifying safety of a given program reduces to a check for an empty intersection with the set of initial states. Spurious errors are eliminated through lazy abstraction. This approach was implemented in the SAFARI tool [4].

Modular verification is a popular approach to achieve scalability. Parts of the program can be analyzed separately in order to reduce the total cost of the verification. In particular, library code can be analyzed just once, even in the case of methods that are called at multiple locations in a given program. Authors of [27] propose to replace the calls of library methods that manipulate data structures with summary transitions that are derived from their specifications. The Whale algorithm [1] computes method summaries, which are necessary for the proof of correctness, using an iterative approach based on interpolation.

A difficult challenge related to programs with multiple threads is the need to model concurrent updates of shared data. The first step towards overcoming this challenge was proposed in [19]. The central idea is to perform the CEGAR loop on a parallel composition of multiple instances of the boolean program that abstracts the input program. Predicates referring just to thread-local variables are modeled with local boolean variables (a fresh copy per thread is used), and predicates involving only shared variables are modeled with shared boolean variables. In the case of predicates that refer to a mixture of local and shared variables, this verification procedure also uses local boolean variables to represent such predicates but each update of such a variable is broadcast to all the threads. To compute the correct value to be broadcast from an active thread t_a to a given thread t, the procedure uses predicates associated with t_a, predicates associated with t, and all shared predicates. In contrast, the approach proposed in [30] extends the IMPACT algorithm with support for concurrency, and uses partial order reduction [15] to achieve reasonable performance.

3 Limitations and Challenges

Our survey of the state-of-the-art verification techniques based on predicate abstraction indicates that great advancements have been made in the last decade, but the techniques still have certain limitations in terms of their support of features used in realistic programs, performance, scalability, and automation. In particular, they are applicable only to programs of a moderate size (up to tens of thousands of source code lines in C) and they can handle only programs with a limited usage of data structures.

The comparison [12] of the two fundamental approaches to lazy abstraction [24, 28] analyzes the effects of their individual features on performance. The authors implemented and compared several different configurations of the approaches using the same framework. Results show that the two approaches have similar performance in configurations that involve larger block encodings [8]. The general conclusion is that less refinement steps and cheaper covering tests lead to better performance.

4 Our Research Goals

Our main goal is to create a framework for verification of multi-threaded Java programs that involve loops, recursion, and usage of data structures. We especially intend to investigate automated generation of abstraction predicates by combination of static analysis with dynamic analysis, which is known to handle programs with loops very well.

We have already done some initial work on using predicates to capture the state and behavior of data collections. In [29], we defined a predicate language for modeling lists, sets, and maps, and implemented the language in the J2BP tool that automatically generates abstract programs [31]. We also started investigating the capabilities of predicate abstraction in the context of on-the-fly state space traversal combined with a dynamic analysis. We implemented basic support for predicate abstraction into the tool called Abstract Pathfinder [18], which is an extension to the Java Pathfinder verification framework [32].

References

1 A. Albarghouthi, A. Gurfinkel, and M. Chechik. Whale: An Interpolation-Based Algorithm for Inter-procedural Verification. In Proceedings of VMCAI 2012, LNCS, vol. 7148.

2 A. Albarghouthi, A. Gurfinkel, and M. Chechik. From Under-Approximations to Over-Approximations and Back. In Proceedings of TACAS 2012, LNCS, vol. 7214.

3 F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. Lazy Abstraction with Interpolants for Arrays. In Proceedings of LPAR 2012, LNCS, vol. 7180.

4 F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. SAFARI: SMT-Based Abstraction for Arrays with Interpolants. In Proceedings of CAV 2012, LNCS, vol. 7358.

5 T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic Predicate Abstraction of C Programs. In Proceedings of PLDI 2001, ACM.

6 N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from Tests. In Proceedings of ISSTA 2008, ACM.

7 J. Berdine, B. Cook, and S. Ishtiaq. SLAyer: Memory Safety for Systems-Level Code. In Proceedings of CAV 2011, LNCS, vol. 6806.

8 D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software Model Checking via Large-Block Encoding. In Proceedings of FMCAD 2009, IEEE.

9 D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The Software Model Checker BLAST. STTT, 9(5-6), 2007.

10 D. Beyer, T. A. Henzinger, and G. Theoduloz. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In Proceedings of CAV 2007, LNCS, vol. 4590.

11 D. Beyer and M. E. Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In Proceedings of CAV 2011, LNCS, vol. 6806.

12 D. Beyer and P. Wendler. Algorithms for software model checking: Predicate abstraction vs. Impact. In Proceedings of FMCAD 2012, IEEE.

13 A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, Bounded Model Checking. Advances in Computers, 58, Elsevier, 2003.

14 A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. On Inter-Procedural Analysis of Programs with Lists and Data. In Proceedings of PLDI 2011, ACM.

15 E. M. Clarke, O. Grumberg, and D. A. Peled. Model Checking. MIT Press, 1999.

16 E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In Proceedings of CAV 2000, LNCS, vol. 1855.

17 E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-Based Predicate Abstraction for ANSI-C. In Proceedings of TACAS 2005, LNCS, vol. 3440.

18 J. Daniel, P. Parizek, and C. Pasareanu. Predicate Abstraction in Java Pathfinder. Java Pathfinder Workshop 2013, ACM SIGSOFT Software Engineering Notes, 39(1).

19 A. Donaldson, A. Kaiser, D. Kroening, and T. Wahl. Symmetry-Aware Predicate Abstraction for Shared-Variable Concurrent Programs. In CAV 2011, LNCS, vol. 6806.

20 K. Dudka, P. Peringer, and T. Vojnar. Byte-Precise Verification of Low-Level List Manipulation. In Proceedings of SAS 2013, LNCS, vol. 7935.

21 S. Ghiraldi and S. Ranise. Backward Reachability of Array-based Systems by SMT solving: Termination and Invariant Synthesis. LMCS, 6(4), 2010.

22 S. Graff and H. Saidi. Construction of abstract state graphs with PVS. In Proceedings of CAV 1997, LNCS, vol. 1254.

23 B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: A New Algorithm for Property Checking. In Proceedings of FSE 2006, ACM.

24 T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In Proceedings of POPL 2002, ACM.

25 T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from Proofs. In Proceedings of POPL 2004, ACM.

26 R. Jhala and K. L. McMillan. Array Abstractions from Proofs. CAV 2007, LNCS, vol. 4590.

27 D. Kapur, R. Majumdar, and C.G. Zarba. Interpolation for Data Structures. In Proceedings of FSE 2006, ACM.

28 K. L. McMillan. Lazy Abstraction with Interpolants. CAV 2006, LNCS, vol. 4144.

29 P. Parizek and O. Lhotak. Predicate Abstraction of Java Programs with Collections. In Proceedings of OOPSLA 2012, ACM.

30 B. Watcher, D. Kroening, and J. Ouaknine. Verifying Multi-threaded Software with Impact. In Proceedings of FMCAD 2013, IEEE.

31 J2BP: Predicate Abstraction for Java, http://plg.uwaterloo.ca/~pparizek/j2bp

32 Java Pathfinder, http://babelfish.arc.nasa.gov/trac/jpf

Automatic Verification of Data Race Freedom in Device Drivers* (Extended Abstract)

Pantazis Deligiannis and Alastair F. Donaldson

Imperial College London {p.deligiannis, alastair.donaldson}@imperial.ac.uk

Abstract

Device drivers are notoriously hard to develop and even harder to debug. They are typically prone to many serious issues such as data races. In this paper, we present *static pair-wise lock set analysis*, a novel sound verification technique for proving data race freedom in device drivers. Our approach not only avoids reasoning about thread interleavings, but also allows the reuse of existing successful sequential verification techniques.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Device Drivers, Verification, Concurrency, Data Races

Digital Object Identifier 10.4230/OASIcs.ICCSW.2014.36

1 Introduction

Device drivers are complex pieces of system-level software responsible for the interaction between an operating system and any hardware devices that are attached to a computer [10]. Drivers are notoriously hard to develop and even harder to debug. Even after a device driver has shipped, it is typically prone to many serious errors [6, 31]. Regarding *concurrency bugs*, a recent study [28] found that they account for 19% of the total bugs in Linux drivers, showcasing their significance. The majority of these concurrency bugs were found to be *data races* or *deadlocks* in various configuration functions and hot-plugging handlers.

The main focus of this paper is on data races, which can lead to nondeterministically occurring bugs that can be challenging to reproduce, isolate and debug. Well-known Linux kernel analysers, such as sparse [8], coccinelle [22] and lockdep [9], have successfully found deadlocks in kernel code, but are typically unable to detect data races. Techniques such as [1, 7, 25, 12, 14, 16, 27, 17] have been used to analyse Linux and Windows device drivers, but primarily focus on sequential program properties. Furthermore, most previous techniques that attempt to reason about thread interleavings face significant scalability issues, because of the exponentially large state-space of realistic concurrent programs [19].

This work-in-progress paper presents *static pair-wise lock set analysis*, a novel technique for automatically verifying data race freedom in device drivers. The key idea behind our approach is that a driver can be proven free from data races by (i) deriving a sound *sequential* model that *over-approximates* the originally concurrent driver, (ii) instrumenting it for lock set analysis and race checking, and (iii) asserting that all accesses to the same shared resource are protected by at least one common lock. The immediate benefit is that our approach not only avoids reasoning about thread interleavings, and thus has the potential to scale well, but also allows the reuse of existing successful sequential verification techniques.

* This work is part of the research project "Automatic Synthesis of High-Assurance Device Drivers" and is generously funded by a gift from Intel Corporation.

Imperial College Computing Student Workshop (ICCSW'14).
Editors: Rumyana Neykova and Nicholas Ng; pp. 36–39
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2 Static Pair-Wise Lock Set Analysis

Our technique involves reasoning about the *lock sets* of a driver. Lock set analysis has its roots in Eraser [29], a dynamic data race detector that tracks the set of locks that consistently protect a memory location during program execution. If that set ever becomes empty, the tool reports a potential data race. This is because an inconsistent lock set suggests that a memory location can be accessed simultaneously by two or more threads.

Eraser avoids reasoning about arbitrary thread interleavings, and thus can scale well in realistic concurrent programs, but suffers from imprecision (i.e. can report false bugs), because a violation of the locking discipline does not always correspond to a real data race [29, 23, 20, 11, 13]. Furthermore, as a dynamic analyser, Eraser's bug finding ability is limited to the execution paths that the tool explores. To counter the second limitation, we apply the idea of Eraser's lock set analysis in a static verification context.

Static pair-wise lock set analysis begins by performing *two-thread reduction*, an abstraction that removes all but two arbitrary threads, each running an entry point of the originally concurrent driver. This reduction was inspired by a reduction to two threads employed in the GPUVerify tool for verification of GPU kernels [5, 2]. The technique then proceeds with *pair-wise sequentialisation*, which combines the two arbitrary threads in a single sequential pair. The pair is finally instrumented for lock set analysis with assertions that check if each memory location is consistently protected by at least one common between the two threads lock. This process repeats until all pairs of entry points have been sequentialised. To achieve soundness, each time an entry point performs a read access to a shared resource, we return a nondeterministic value. This over-approximates any effects from all the unmodeled threads on the driver shared state.

We have prototyped this technique in Whoop, a practical tool for automatic concurrency verification of Linux drivers written in C [15]. Whoop initially compiles the driver source code, together with an environmental model, to LLVM-IR using Clang/LLVM [18]. The program is then compiled to the Boogie [3] verification language using SMACK [26], an LLMV to Boogie translator which can efficiently model heap manipulating programs. Next, the Boogie program is sequentialised and instrumented, using static pair-wise lock set analysis. The abstract program is finally send to the Boogie verifier, which generates verification conditions [4] and discharges them to a theorem prover. Successful verification implies that the original driver is free of data races, while an error denotes a *potential* data race.

The main limitation of our approach is that it can potentially report many false positives, as we over-approximate the shared state. To tackle this problem, we plan to investigate (i) invariant generation for taming our coarse abstraction and (ii) counterexample feasibility checking to evaluate if a reported bug is real or spurious.

3 Related Work

Notable previous works on static analysis for race detection include the static analysers Warlock [30] and LockLint [21], which, however, heavily rely on user annotations. Whoop does not require any source code modifications, and thus can be applied with zero effort.

Most related to our work are the static lock set analysers RELAY [32] and Locksmith [24]. Both tools, though, have significant limitations. RELAY uses unsound post-analysis filters to limit the false positives, but these can filter out true races. Although Locksmith successfully detected data races in 7 medium-sized Linux device drivers, it reported a significant number of false positives. The authors also reported that Locksmith was unable to run on several large programs, showcasing its limited scalability.

References

1 Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. *ACM SIGOPS Operating Systems Review*, 40(4):73–85, 2006.

2 Ethel Bardsley, Adam Betts, Nathan Chong, Peter Collingbourne, Pantazis Deligiannis, Alastair F Donaldson, Jeroen Ketema, Daniel Liew, and Shaz Qadeer. Engineering a static verification tool for GPU kernels. In *Proceedings of the 26th International Conference on Computer Aided Verification*, 2014.

3 Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal methods for Components and Objects*, pages 364–387. Springer, 2006.

4 Mike Barnett and K Rustan M Leino. Weakest-precondition of unstructured programs. *ACM SIGSOFT Software Engineering Notes*, 31(1):82–87, 2005.

5 Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. GPUVerify: a verifier for GPU kernels. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 113–132. ACM, 2012.

6 Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 73–88. ACM, 2001.

7 Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design*, 25(2-3):105–127, 2004.

8 Jonathan Corbet. Finding kernel problems automatically. `https://lwn.net/Articles/87538/`, 2004.

9 Jonathan Corbet. The kernel lock validator. `https://lwn.net/Articles/185666/`, 2006.

10 Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux device drivers (Third Edition)*. O'Reilly, 2005.

11 Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: A race and transaction-aware Java runtime. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 245–255. ACM, 2007.

12 Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation*. USENIX, 2000.

13 Cormac Flanagan and Stephen N Freund. FastTrack: Efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 121–133. ACM, 2009.

14 Thomas A Henzinger, George C Necula, Ranjit Jhala, Gregoire Sutre, Rupak Majumdar, and Westley Weimer. Temporal-safety proofs for systems code. In *Computer Aided Verification*, pages 526–538. Springer, 2002.

15 Brian W Kernighan, Dennis M Ritchie, and Per Ejeklint. *The C programming language*, volume 2. prentice-Hall Englewood Cliffs, 1988.

16 Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea. Testing closed-source binary device drivers with DDT. In *Proceedings of the 2010 USENIX Annual Technical Conference*. USENIX, 2010.

17 Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. A solver for reachability modulo theories. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 427–443. Springer, 2012.

18 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, pages 75–86. IEEE, 2004.

19 Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 267–280. USENIX, 2008.

20 Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 167–178. ACM, 2003.

21 Oracle Corporation. Analyzing program performance with Sun WorkShop, Chapter 5: Lock analysis tool. `http://docs.oracle.com/cd/E19059-01/wrkshp50/805-4947/6j4m8jrnd/index.html`, 2010.

22 Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in Linux device drivers. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, Eurosys '08, pages 247–260. ACM, 2008.

23 Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 179–190. ACM, 2003.

24 Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. LOCKSMITH: Context-sensitive correlation analysis for race detection. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 320–331. ACM, 2006.

25 Shaz Qadeer and Dinghao Wu. KISS: Keep it simple and sequential. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 14–24. ACM, 2004.

26 Zvonimir Rakamaric and Alan J Hu. Automatic inference of frame axioms using static analysis. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 89–98. IEEE, 2008.

27 Matthew J Renzelmann, Asim Kadav, and Michael M Swift. SymDrive: Testing drivers without devices. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*. USENIX, 2012.

28 Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: Taming device drivers. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 275–288. ACM, 2009.

29 Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

30 Nicholas Sterling. WARLOCK - A static data race analysis tool. In *Proceedings of the 1993 Winter USENIX Conference*, pages 97–106. USENIX, 1993.

31 Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 207–222. ACM, 2003.

32 Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. RELAY: Static race detection on millions of lines of code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 205–214. ACM, 2007.

A survey of modelling and simulation software frameworks using Discrete Event System Specification

Romain Franceschini[1], Paul-Antoine Bisgambiglia[1], Luc Touraille[2], Paul Bisgambiglia[1], and David Hill[2]

1 University of Corsica Computing Laboratory
UMR SPE 6134 CNRS, UMS Stella Mare 3460, Campus Grimaldi, 20250 Corti
{r.franceschini,bisgambiglia,bisgambi}@univ-corse.fr

2 CNRS, UMR 6158, ISIMA LIMOS, Blaise Pascal University
BP 10448, F-63000 Clermont-Ferrand, France
{touraille@isima.fr, david.hill}@univ-bpclermont.fr

Abstract

Discrete Event System Specification is an extension of the Moore machine formalism which is used for modelling and analyzing general systems. This hierarchical and modular formalism is time event based and is able to represent any continuous, discrete or combined discrete and continuous systems. Since its introduction by B.P. Zeigler at the beginning of the eighties, most general modelling formalisms able to represent dynamic systems have been subsumed by DEVS. Meanwhile, the modelling and simulation (M&S) community has introduced various software frameworks supporting DEVS-based simulation analysis capability. DEVS has been used in many application domains and this paper will present a technical survey of the major DEVS implementations and software frameworks. We introduce a set of criteria in order to highlight the main features of each software tool, then we propose a table and discussion enabling a fast comparison of the presented frameworks.

1998 ACM Subject Classification I.6 Simulation and modeling

Keywords and phrases DEVS, Framework, Survey, Modelling, Simulation

Digital Object Identifier 10.4230/OASIcs.ICCSW.2014.40

1 Introduction

Simulation has become a popular tool to study a broad range of systems. The growing number and quality of simulation software requires expertise for their evaluation. Selecting an appropriate framework is an important issue to simulation practitioners.

Our study will be restricted to software based on discrete event systems (DES). A Discrete Event Sytem is a discrete-state, event driven dynamical system in which the state space is described by a discrete set, and states evolve in terms of asynchronous occurrence of discrete events over time. In DES theory, states, events and transition functions are defined by a five-tuple [6]: $M = \langle S, s_0, \lambda, \delta, \Sigma \rangle$ where: S is the set of states; s_0 is the initial state vector; Σ is the set of events; $\delta : S \times \Sigma \to S$ the state transition function; $\lambda : S \times \Sigma \to \Sigma_d$ the output function, where Σ_d and Σ_{ud} the set of detectable and undetectable events, respectively $\Sigma = \Sigma_d \cup \Sigma_{ud}$. Actually, DES represent many technological and engineering systems such as communication networks, computer networks, manufacturing systems, transportation systems, natural systems, and more. So far, many modelling approaches of DESs have been

Imperial College Computing Student Workshop (ICCSW'14).
Editors: Rumyana Neykova and Nicholas Ng; pp. 40–49
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

proposed and developed, notably including finite automata, fuzzy automata, Petri nets and the DEVS formalism.

The DEVS formalism [4, 36] is considered as universal for discrete event dynamic systems and is capable of representing a wide class of other dynamic systems. DEVS can simulate discrete time systems such as cellular automata and can also approximate, as closely as desired, differential equation systems, continuous systems, etc. DEVS is a modular formalism which permits the modelling of causal and deterministic systems. A DEVS atomic (behavioral) model is described by the following formula: $M = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta\rangle$, where X the list of the model inputs; Y the list of the model outputs; S the system states; ta the time advance function; $\delta_{ext} : S \times X \to S$ is the external transition function; $\delta_{int} : S \to S$ is the internal transition function and $\lambda : S \to S$ is the output function. A complex system can be designed as a coupling of several simpler systems using a DEVS coupled (structural) model, described by the following formula: $MC = \langle X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC, L\rangle$, with X being the input ports and values set; Y the output ports and values set; D the set of components; EIC the total set of input couplings ($X \to d$), which links the coupled model to its components; EOC the total set of output couplings ($d \to Y$), which links components to the coupled model; IC the total set of internal couplings, which links components with each other; L a list of priorities among components. To define the simulation semantics of DEVS models, Zeigler introduced abstract simulators. The advantage with such an approach lies in the separation between the models and the simulators. DEVS provides the algorithms used to simulate a model hierarchy, introduced in [36].

Selecting a software framework that meets the requirements needs a full examination of many factors and it is a difficult task. The aim of this paper is to propose a survey of research done in the field of M&S. The review is limited and the suggestion criteria are defined to assist decision makers in evaluating and selecting software. In this paper we will provide a broad comparison, and propose a benchmark on a subset of three tools. Section 2 describes our comparative criteria (programming language, version, GUI, DEVS extensions...). Section 3 lists the studied software [1, 5, 8, 9, 11, 14, 16, 20, 22, 26, 28, 30, 33, 34, 38]. Before concluding, Section 4 provides a comparison grid according to our selected criteria and a discussion presenting the comparison of DEVS software frameworks.

2 Discussion on criteria

Before any evaluations, it is essential to establish criteria of comparisons. Evaluating and selecting software frameworks that meet users requirements is a difficult process. The aim is to provide a basis to improve the process of selection of the software frameworks [13, 19]. Several methods have emerged to formulate a software evaluation process. Many papers have proposed their list of criteria with the lack of a standard common list. A standard list of criteria, an explanation and an example for each criterion could overcome some pitfalls. Due to progress in the field, new computer technology and software updates, it may not be possible to provide a standard list of criteria. This paper focus on 8 sets of criteria, chosen to give the best possible description framework. The latter were defined according to our application domain, i.e. the academic domain: **1**) Software version. If it is no longer updated or maintained it can become deprecated, as JDEVS [8]; **2**) The programming language is a very important criterion, especially if we want to make performance comparisons. This type of comparison has already been proposed [29]. Language popularity and libraries available are also important aspects to consider; **3**) The quality of documentation; **4**) Simulator algorithms and the proposed extensions: for 30 years the formalism has evolved, a parallel version of the simulator has been proposed [7], as numerous extensions for real-time systems, continuous, cellular [1, 31]; **5**) Model sets. DEVS formalism has been highlighted for its modular side. A framework should provide a set of models ready to be used; **6**) Network management. This is the opportunity to run models remotely using a web service [2, 17] or to provide models through middleware (HLA, SOA) [18, 32, 35]; **7**) Design tools. Like DEVS, these frameworks have evolved. A lot of frameworks are helping modelers with a GUI. Today, with the contributions of model driven engineering (MDE), very powerful tools (meta-modelling) are used to generate the model code, to mark them interoperable and to standardise simulators [10, 16, 25]; and **8**) Analysis and interpretation tools through visualization, statistics and reports. The next section will describe the major DEVS frameworks.

3 Framework description

In this section we study a non-exhaustive list of several DEVS-based simulators by giving their description.

CD++ [30] is a DEVS M&S toolkit that provides a library of C++ classes to specify models in several DEVS formalisms and simulate them. The supported formalisms include classical DEVS (CDEVS) and parallel DEVS (PDEVS), but the focus is on Cell-DEVS [31], an extension integrating cellular automata and DEVS. CD++ can handle in a single simulation several types of models, e.g. parallel DEVS and Cell-DEVS models. Simulations can be performed either locally or remotely, by sending model specifications to a simulation server. Depending on their types, CD++ models are specified either as C++ classes or through text files following a custom format. In addition to the simulation kernel, CD++ provides a plugin allowing edition of models both textually and graphically, as well as visualization of simulation results.

DEVSJava [23] is a Java library for modelling and simulating PDEVS, Dynamic-Structure DEVS and Real-Time-DEVS models. It is co-directed by B.P. Zeigler, H. Sarjoughian and R. Lysecky. DEVSJava provides a set of custom container classes for storing entities manipulated by models. These containers are used to develop DEVS models according to the class hierarchy defined by the library. Models are specified as Java classes and can then be simulated with the simulation processors implemented in the library. Local simulation, distributed simulation and real-time simulation are supported. Later versions of DEVSJava include a dynamic-structure modelling feature. The DEVSJava library is now included in a larger software suite for M&S called DEVS-Suite, which provides some graphical facilities for editing models, controlling simulation and visualising results [15]. Subsequently B.P. Zeigler created a company and incorporated some of DEVSJava and DEVS-Suite tools in MS4ME [33].

James II [12, 38] is a generic M&S platform, written in Java, which is developed under the coordination of A.M. Uhrmacher. Its aim is to provide an extensible platform that can integrate any modelling formalism, simulation algorithms, and tools. To do so, it uses a flexible plugin system that makes the addition of new features in the platform quite seamless. The architecture of James II focuses on minimizing coupling between modules. The core of the platform provides a set of services and classes for use by other packages, such as random number generation, data structures, mathematical functions, serialization, etc.). It also handles the graphical user interface and the plugin system. Other features are implemented as plugins. The most important types of plugins are modelling formalisms, simulation algorithms, editors and visualizers, but other plugin types can be defined. The last version at the time of this writing (v0.9.6) comes bundled with several formalisms, among others DEVS, PDEVS, PdynDEVS and cellular automata, along with various simulation algorithms (sequential, multi-threaded, etc.), editors and visualizers for each.

Virtual Laboratory Environment (VLE) [21] is an M&S platform based on PDEVS, written in C++ and mainly developed by Gauthier Quesnel. VLE is now integrated in the RECORD platform supported by the Applied Mathematics and Computer Science department of INRA (named MIA). Like James II, its architecture is quite modular to facilitate the addition of new features. The core of VLE consists in a set of class libraries (VFL), that implement several formalisms (Petri nets, 2D/3D cellular automata, Quantized State Systems (QSS), etc.). VLE provides a PDEVS simulator, different ways to observe graphically the models in real time, through the Eyes of VLE (EOV), and a graphical user interface (GVLE) for editing models and their couplings/hierarchy.

PyDEVS & PyPDEVS PyDEVS is a DEVS Modelling and Simulation Package implemented in Python [3] and mainly developed by The Modelling, Simulation and Design lab (MSDL) headed by Prof. Hans Vangheluwe. The package provides an easy way to model and simulate hierarchical DEVS. It is based on classic DEVS formalism. Newer version of the package called PyPDEVS [24] is available with PDEVS implementation and a distributed simulations feature. The package is also used as a basis for two other frameworks: AToM^3 [16], a multi-paradigm modelling tool with meta-modelling and model-transforming features that relies on model driven engineering (MDE) concepts and DEVSimPy [5], on open source project supported by University of Corsica that provides a GUI to facilitate both the coupling and reusability of models.

PowerDEVS is a software tool for classical DEVS M&S oriented toward the simulation of hybrid systems [1]. Developed by E. Kofman team of Rosario University, it allows defining atomic DEVS models in C++ which can be graphically coupled and translated in C++ code. It gives the possibility to perform simulations in real time, allowing the design and automatic implementation of digital controllers. It can be interconnected with Scilab.

SimStudio [27] is an architecture built upon the DEVS formalism that aims at integrating tools for M&S, analysis and collaboration through Model-Driven Engineering (MDE) features such as code generation.

DEVS-Ruby [9] is a library that allows formal specifications of CDEVS and PDEVS models. It provides an internal Domain-Specific Language (DSL) which can be extended to meet domain specific vocabulary of modelers.

4 Comparison and discussion

This section provides a comparison of the software listed in section 3 according to our selected criteria (see section 2) as long with a performance analysis of some frameworks.

4.1 Comparison

The aim of this comparison is to guide, according to its needs, any potential newcomer wishing to use a DEVS framework. As shown in Table 1, most of listed software are still maintained, except for SimStudio and AToM^3. Note that MS4Me is proprietary and that CD++ is not available to download except if you have a user access to the corresponding wiki. There is not a wide variety of languages used (C++, Java, Python, Ruby), but they are fundamentally various in terms of characteristics. We rate documentation from 1 to 5 based on code documentation, examples, tutorials, wikis of each simulator and provide a link to a website if available. Table 1 lists all approached frameworks, but subsequent

Table 1 Framework comparison grid based on criteria 1, 2 and 3.

DEVS Frameworks		Criteria			
Name	Ref.	1	2	3	
		Version	Lang.	Documentation	
aDEVS	[20]	2014	C++	2	`http://web.ornl.gov/~1qn/adevs/`
CD++	[30]	2013	C++	5	`http://cell-devs.sce.carleton.ca/`
PowerDEVS	[1]	2014	C++	4	`http://sourceforge.net/projects/powerdevs/files/`
SimStudio	[27]	2010	C++	1	
AToM^3	[16]	2006	Python	4	`http://atom3.cs.mcgill.ca/`
MS4Me	[37]		Java	5	`http://www.ms4systems.com/pages/main.php`
JAMES II	[38]	2014	Java	5	`http://wwwmosi.informatik.uni-rostock.de/mosi/projects/cosa/james-ii/`
VLE	[22]	2014	C++	5	`http://www.vle-project.org/wiki/Main_Page`
PyDEVS	[3]	2014	Python	4	`http://msdl.cs.mcgill.ca/projects/projects/DEVS/`
DEVSimPy	[5]	2014	Python	3	`http://devsimpy.univ-corse.fr/`
PyPDEVS	[24]	2014	Python	4	`http://msdl.cs.mcgill.ca/people/yentl/50_master`
DEVS-Ruby	[9]	2014	Ruby	3	`http://devs-ruby.github.io/devs_ruby/`

Table 2 Framework comparison grid based on 4th criterium.

DEVS Frameworks		Criteria		
Name	Ref.	4		
		Simulator Algorithms	Extensions Concepts	Systems modelling
aDEVS	[20]	PDEVS	Cell Space, dynDEVS	social, ecological, computer networks and computer architecture, military
CD++	[30]	CDEVS, PDEVS, distributed	Cell-DEVS, RT-DEVS, E-CD++, DEVStone	social, ecological, computer networks, environmental
PowerDEVS	[1]	CDEVS	QSS, QSS2, RT-DEVS	hybrid, physical, Scilab interconnexion
JAMES II	[38]	PDEVS, distributed	FDDEVS, SoS, components, agent	hybrid, space
VLE	[22]	PDEVS	DSDE	agent, agro-ecosystems
PyPDEVS	[24]	CDEVS, PDEVS, distributed	RT-DEVS, DSDEVS	
DEVS-Ruby	[9]	CDEVS, PDEVS	DEVStone, flat, decentralized	

tables 2 and 3 continue without: **1**) PyDEVS because newer and future version is PyPDEVS; **2**) AToM^3 which is a multi-paradigm tool where DEVS is supported through PyDEVS; **3**) DEVSimPy which provides a GUI on top of PyDEVS; **4**) SimStudio; and **5**) MS4Me. Many extensions of the formalism have been proposed over time. Table 2 lists supported formalisms for each framework and what particular domains of systems they are best suited for. Table 3 highlights when a repository of usable models is available, if distributed simulations are supported, how modelers can analyze simulations, if a GUI is available and which MDE concepts are supported.

4.2 Performance analysis

For our performance analysis, we compare DEVS-Ruby with aDEVS and PyPDEVS. We retained aDEVS for our comparison because it is written in a compiled, statically typed language (C++) whereas DEVS-Ruby is written with an interpreted, dynamically typed

Table 3 Framework comparison grid based on criteria 5, 6, 7 and 8.

DEVS Frameworks		Criteria				
Name	Ref.	5	6	7		8
		Model Sets	Network	GUI	MDE	Analysis
aDEVS	[20]	yes	yes	no	no	plot
CD++	[30]	yes	yes	yes	C++ modeler	DEVS view, 2d, 3d, visualization
PowerDEVS	[1]	yes	no	yes	no	quick scope
JAMES II	[38]	yes		yes		plot
VLE	[22]	yes		yes		visualization
PyPDEVS	[24]	no	yes	no		
DEVS-Ruby	[9]	yes	yes	no	DSL	plot, GIS

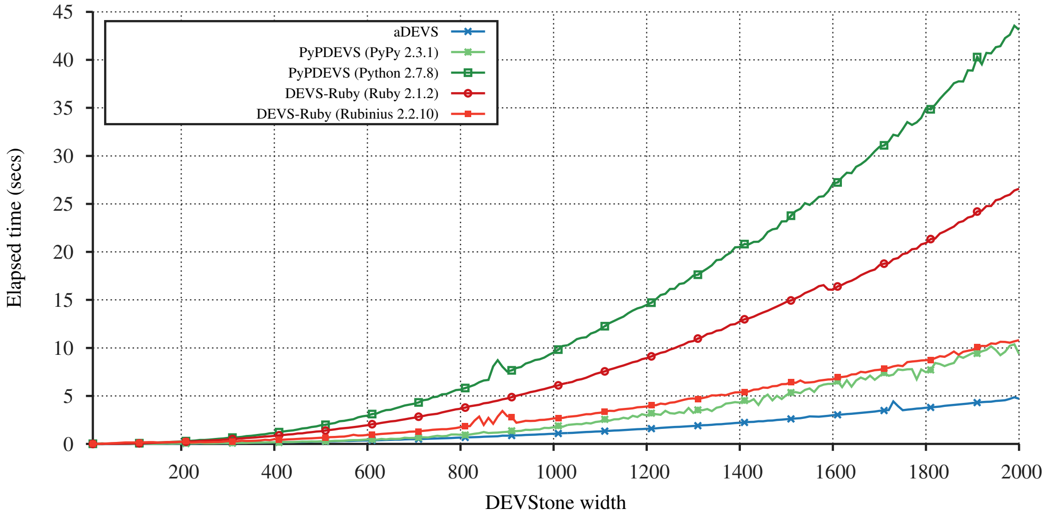


Figure 1 CPU time in seconds of a DEVStone simulation with a varying width and a fixed depth of 3, HI coupling type, δ_{ext} and δ_{int} times set to 0 secs.

language. We used aDEVS performances as an indicator, since [24] found it to be the fastest DEVS framework available. Concerning PyPDEVS, it is written in Python, which is very similar to Ruby. Moreover, DEVS-Ruby is closest to PyPDEVS according to [9].

The test environment is based on an Intel(R) Core(TM) i5-3360M CPU @ 2.80GHz (3MB L2 cache), 16 GB (2 x DDR3 - 1600 MHz) of RAM, a Toshiba MK5061GS hard drive, running on Ubuntu 14.04 (64bit). The software used for the benchmarking are: **1**) DEVS-Ruby 0.6 using the official Ruby VM (version 2.1.2) and an alternative Ruby implementation, called Rubinius (version 2.2.10); **2**) PyPDEVS 2.2.3 using the official Python VM (version 2.7.8) and an alternative Python implementation, called PyPy (version 2.3.1); and **3**) aDEVS 2.8.1 compiled using GCC 4.8.2 with -O3 optimizations flags. To compare performances of selected frameworks (criterium 2), we propose to present the results obtained with a DEVStone [29] benchmark. DEVStone is used to study the performance of DEVS-based simulators. We use it to generate automatically a suite of models with varied structure and behaviour automatically. Using DEVStone, we benefit from a common metric to compare the results obtained using different software. Since it is designed to evaluate the efficiency of DEVS simulation engines, we believe it is the go-to for such a performance analysis.

We measure elapsed wall clock time in seconds for a simulation involving a DEVStone suite of models. Events traverse all models of the generated structure with a fixed *depth* (number of nested coupled models in the hierarchy) of 3, *HI* coupling type (a type of models interconnections defined in [29]) and δ transitions times set to 0 seconds. The varying parameter is the *width* (number of components in each coupled model). All simulators are running the PDEVS formalism. Figure 1 shows the results obtained using the machine specified above. As you can see, the fastest benchmarked simulator is unsurprisingly aDEVS due to its implementation in C++. Concerning DEVS-Ruby and PyPDEVS, DEVS-Ruby is more performant when we are using the official Python and Ruby implementation. But if we switch language implementation to PyPy and Rubinius, both simulators are almost on an equal footing, with PyPDEVS being slightly superior. PyPy and Rubinius are two sophisticated alternative implementations written respectively in Python and in Ruby. They offer great performance thanks to a just-in-time (JIT) native machine code compiler. However, Python code from PyPy is then compiled in C to produce a native interpreter whereas Rubinius VM is written in C++ and nearly everything else is pure Ruby code (lexer, parser, AST to bytecode compiler, standard library). These differences lead Rubinius to poor performances. Another solution consists in using the Topaz Ruby implementation which is built with the same toolchain as PyPy, but it is unfortunately not stable enough yet.

4.3 Discussion

The study of these simulators highlights several points. The PDEVS formalism is more popular in the DEVS community for two reasons: it makes easier to parallelize/distribute simulations and it is more consistent and flexible when it comes to handle simultaneous events. There is no general purpose simulator, each simulator being developed by different institutions working on different aspects and extensions of the formalism, all tending to specialize like CD++ for cellular automata, VLE for agro-ecosystems, James II for agents. Programming languages are important because their syntax and specifications say how much we are ready to make concessions on flexibility and expressiveness, especially to implement models. Studied software either use compiled language (C++), interpreted languages (Python and Ruby) or compiled/interpreted language (Java). In terms of type systems, we have statically typed languages (C++, Java) and dynamically typed languages (Python, Ruby). But this paper emphasizes that the gap in terms of performances between a high performance language like C++ and slower languages like Ruby or Python tend to be reduced thanks to advances in language theory.

5 Conclusion

This paper reports a systematic review of M&S softwares published in journals and conference proceedings. The aim is to propose a survey in the field of simulation software frameworks. Based on an evaluation and selection method, we provide a basis to choose a discrete event simulation framework. Our selection method relies on criteria set related to academic fields. After evaluating several frameworks, we selected three (DEVS-Ruby, aDEVS, PyPDEVS) and we presented a performance benchmark.

Acknowledgements The present work was supported in part by the French Ministry of Research, the Corsican Region and the CNRS. DEVStone implementation for aDEVS and PyPDEVS is based from models originally written by Yentl Van Tendeloo and Prof. Hans Vangheluwe.

References

1 F. Bergero and E. Kofman. PowerDEVS: a tool for hybrid system modeling and real-time. *SIMULATION*, 87(1-2):113–132, January 2011.

2 P.-A. Bisgambiglia, R. Franceschini, F.-J. Chatelon, J.-L. Rossi, and P. A. Bisgambiglia. Discrete event formalism to calculate acceptable safety distance. In *Simulation Conference (WSC), 2013 Winter*, pages 217–228, December 2013.

3 J. S. Bolduc and H. Vangheluwe. A modeling and simulation package for classic hierarchical DEVS. Technical report, 2002.

4 L. Booker, S. Forrest, M. Mitchell, and R. Riolo. *Discrete Event Abstraction: An Emerging Paradigm For Modeling Complex Adaptive Systems by Bernard P. Zeigler Chapter 6 in Perspectives on Adaptation in Natural and Artificial Systems.* Oxford University Press, Oxford, England ; New York, February 2005.

5 L. Capocchi, J.-F. Santucci, B. Poggi, and C. Nicolai. DEVSimPy: A collaborative python software for modeling and simulation of DEVS systems. In *2011 20th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 170–175, June 2011.

6 C. G. Cassandrass and S. Lafortune. *Introduction to Discrete Event Systems.* 2nd ed. 2008. Kluwer Academic Publishers, springer edition, 1999.

7 A. C. Chow, B. P. Zeigler, and D. H. Kim. Abstract simulator for the parallel DEVS formalism. In *, Proceedings of the Fifth Annual Conference on AI, Simulation, and Planning in High Autonomy Systems, 1994. Distributed Interactive Simulation Environments*, pages 157–163, December 1994.

8 J.-B. Filippi, F. Bernardi, and M. Delhom. The JDEVS environmental modeling and simulation environment. *IEMSS, Integrated Assessment and Decision Support, Lugano Suisse*, page 283–288, 2002.

9 R. Franceschini, P.-A. Bisgambiglia, P. Bisgambiglia, and D. R. C. Hill. DEVS-Ruby: a Domain Specific Language for DEVS Modeling and Simulation (WIP). In *DEVS 14: Proceedings of the Symposium on Theory of M&S*, pages 393–398. SCS International, April 2014.

10 S. Gareddu, E. Vittori, J.-F. Santucci, and P.-A. Bisgambiglia. A Meta-Model for DEVS - Designed following Model Driven Engineering Specifications. In *SIMULTECH 2012*, pages 152–157, 2012.

11 N. Giambasi, B. Escude, and S. Ghosh. GDEVS: A generalized discrete event specification for accurate modeling of dynamic systems. In IEEE, editor, *Proceedings of the Fifth International Symposium on Autonomous Decentralized Systems ISADS*, page 464, 2001.

12 J. Himmelspach and A. M. Uhrmacher. Plug'n simulate. In *In Proceedings of the 40th Annual Simulation Symposium (2007)*, pages 137–143, 2007.

13 A. S. Jadhav and R. M. Sonar. Evaluating and selecting software packages: A review. *Information and Software Technology*, 51(3):555–563, March 2009.

14 V. Janoušek and E. Kironskỳ. Exploratory modeling with SmallDEVS. In *Proceedings of the 20th annual European Simulation and Modelling Conference*, page 122–126, 2006.

15 S. Kim, H. S. Sarjoughian, and V. Elamvazhuthi. DEVS-suite: a simulator supporting visual experimentation design and behavior monitoring. In *Proceedings of the 2009 Spring Simulation Multiconference*, page 161. Society for Computer Simulation International, 2009.

16 J. de Lara and H. Vangheluwe. AToM3: A tool for multi-formalism and meta-modelling. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering*, number 2306 in Lecture Notes in Computer Science, pages 174–188. Springer Berlin Heidelberg, January 2002.

17 S. Mittal, J. L. Risco, and B. P. Zeigler. DEVS-based simulation web services for net-centric t&e. In *Proceedings of the 2007 Summer Computer Simulation Conference*, SCSC '07, page 357–366, San Diego, CA, USA, 2007. Society for Computer Simulation International.

18 S. Mittal, J. L. Risco-Martín, and B. P. Zeigler. DEVS/SOA: A cross-platform framework for net-centric modeling and simulation in DEVS unified process. *SIMULATION*, 85(7):419–450, July 2009.

19 J. Nikoukaran, V. Hlupic, and R. J. Paul. Criteria for simulation software evaluation. In *Proceedings of the 30th Conference on Winter Simulation*, WSC '98, page 399–406, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

20 J. Nutaro. ADEVS (a discrete EVent system simulator). *Arizona Center for Integrative Modeling & Simulation (ACIMS), University of Arizona, Tucson. Available at http://www. ece. arizona. edu/nutaro/index.php*, 1999.

21 G. Quesnel, R. Duboz, and E. Ramat. The virtual laboratory environment – an operational framework for multi-modelling, simulation and analysis of complex dynamical systems. *Simulation Modelling Practice and Theory*, 17(4):641–653, April 2009.

22 E. Ramat and P. Preux. Virtual laboratory environment (VLE): a software environment oriented agent and object for modeling and simulation of complex systems. *Simulation Modelling Practice and Theory*, 11(1):45–55, March 2003.

23 H. S. Sarjoughian and B. P. Zeigler. DEVSJAVA: Basis for a DEVS-based collaborative m&s environment. *Simulation Series*, 30:29–36, 1998.

24 Y. Van Tendeloo and H. Vangheluwe. The Modular Architecture of the Python(P)DEVS Simulation Kernel Work In Progress paper. In *DEVS 14: Proceedings of the Symposium on Theory of M&S*, pages 387–392. SCS International, April 2014.

25 L. Touraille, M. K. Traoré, and D. R. C. Hill. A mark-up language for the storage, retrieval, sharing and interoperability of DEVS models. In *Proceedings of the 2009 Spring Simulation Multiconference*, SpringSim '09, page 163:1–163:6, San Diego, CA, USA, 2009. Society for Computer Simulation International.

26 L. Touraille, M. K. Traoré, and D. R. C. Hill. SimStudio : une infrastructure pour la modélisation, la simulation et l'analyse de systèmes dynamiques complexes. Technical Report RR-10-13, INRIA, May 2010.

27 M. K. Traoré. SimStudio: A next generation modeling and simulation framework. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, Simutools '08, page 67:1–67:6, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

28 H. Vangheluwe. The discrete EVent system specification DEVS formalism. Technical report, Modeling and Simulation (COMP522A), 2001. Published: Lecture Notes http://moncs.cs.mcgill.ca/.

29 G. Wainer, E. Glinsky, and M. Gutierrez-Alcaraz. Studying performance of DEVS modeling and simulation environments using the DEVStone benchmark. *SIMULATION*, 87(7):555–580, July 2011.

30 G. A. Wainer. CD++: a toolkit to define discrete-event models. *Software, Practice and Experience. Wiley*, 32(3):1261–1306, November 2002.

31 G. A. Wainer and N. Giambiasi. Application of the cell-DEVS paradigm for cell spaces modelling and simulation. *SIMULATION*, 76(1):22–39, January 2001.

32 G. Zacharewicz, M. El-Amine Hamri, C. Frydman, and N. Giambiasi. A generalized discrete event system (g-DEVS) flattened simulation structure: Application to high-level architecture (HLA) compliant simulation of workflow. *SIMULATION*, 86(3):181–197, March 2010.

33 B. P. Zeigler. *Guide to Modeling and Simulation of Systems of Systems - User's Reference*. Springer Briefs in Computer Science. Springer, 2013.

34 B. P. Zeigler, G. Ball, H. Cho, J. S. Lee, and H. S. Sarjoughian. The DEVS/HLA distributed simulation environment and its support for predictive filtering. Technical report, 1998.

35 B. P. Zeigler, S. B. Hall, and H. S. Sarjoughian. Exploiting HLA and DEVS to promote interoperability and reuse in lockheed's corporate environment. *SIMULATION*, 73(5):288–295, November 1999.

36 B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation, Second Edition*. 2000.

37 B. P Zeigler and H. S. Sarjoughian. DEVS integrated development environments. In *Guide to Modeling and Simulation of Systems of Systems*, page 11–26. Springer, 2013.

38 S. Zinn, J. Himmelspach, A. M. Uhrmacher, and J. Gampe. Building Mic-Core, a specialized M&S software to simulate multi-state demographic micro models, based on JAMES II, a general M&S framework. *J. Artificial Societies and Social Simulation*, 16(3), 2013.

Calculating communication costs with Sessions Types and Sizes*

Juliana Franco, Sophia Drossopoulou, and Nobuko Yoshida

Imperial College of London, Department of Computing
London, United Kingdom
{j.vicente-franco, s.drossopoulou, n.yoshida}@imperial.ac.uk

Abstract

We present a small object-oriented language with communication primitives. The language allows the assignment of binary session types to communication channels in order to govern the interaction between different objects and to statically calculate communication costs. Class declarations are annotated with size information in order to determine the cost of sending and receiving objects. This paper describes our first steps in the creation of a session-based, object-oriented language for communication optimization purposes.

1998 ACM Subject Classification Languages and compilers

Keywords and phrases Session types, communication, object-oriented, multicore

Digital Object Identifier 10.4230/OASIcs.ICCSW.2014.50

1 Introduction

In the near future, parallel machines with thousands of cores are expected and programming languages that easily scale for such machines are needed. Furthermore, it is possible to have concurrent computations with a large number of messages exchanged and develop programs with such communications can be a hard task and liable to several errors. Session types [9, 16] are a well-established mechanism to describe message-passing computations that allows to ensure communication safety and the absence of race conditions and deadlocks. When assigned to communication channels they can be used to express the kind of messages exchanged among the different partners of a communication and their order as well as to statically verify if the communication proceeds as specified by session types. There are two variants of session types: binary (or two-party) session types to describe the communication between two parties and multiparty session types to govern the communication among multiple participants.

In this work, we are interested in the design of a new object-oriented programming language that uses session types to govern the communication between different objects and to statically calculate the respective communication costs. We start only with two-party sessions and then when we achieve our goals we will expand our approach to multiparty session types. We present a Java-like syntax with communication primitives where objects may communicate using channels with session types assigned.

The contributions of this paper can be summarized as our first steps in the design of a session-based, object-oriented language that allows to express the size of data structures and to statically calculate communication costs, using session types, for optimization purposes. We believe that in the future we will be able to use this information to apply optimizations

* This work was supported by Upscale Project.

Imperial College Computing Student Workshop (ICCSW'14).
Editors: Rumyana Neykova and Nicholas Ng; pp. 50–57
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

$$
\begin{array}{rlcl}
\textbf{Class declarations:} & D & ::= & \mathsf{class}\ C[\langle N^{+}\rangle]\{F^{*};M^{*}\} \\
\textbf{Method declarations:} & M & ::= & (\mathsf{@set}\ sa\ (\mathsf{this}.f), v;)^{*} \quad \tau\ m((x:\rho)^{*})\{e\} \\
\textbf{Field declarations:} & F & ::= & [\mathsf{@}sa\ v\ :]T\ f; \\
\textbf{Size annotations:} & sa & ::= & \mathsf{has} \ \mid\ \mathsf{has\ up\_to} \\
\textbf{Values:} & v & ::= & \text{constant} \ \mid\ N \ \mid\ v+v \ \mid\ v*v \\
\textbf{Types:} & T & ::= & \mathsf{boolean} \ \mid\ \mathsf{integer} \ \mid\ \mathsf{char} \ \mid\ \mathsf{string}[v] \ \mid\ C[\langle v^{+}\rangle] \\
\textbf{Return types:} & \tau & ::= & T \ \mid\ \mathsf{void} \\
\textbf{Parameter types:} & \rho & ::= & T \ \mid\ ST
\end{array}
$$

Figure 1 Syntax of classes and types.

such as, to change the location of the objects—given that the location of objects has impact in the communication cost (if two objects are distant the communication cost is worst than if they are close) we can change the objects topology in order to obtain a better performance.

Structure of the document: The document is organised as follows. Section 2 describes our syntax of classes and types. Section 3 shows the syntax of session types followed by Section 4 with the syntax of session operations. The calculation of communication costs is described in Section 5. We present Section 6 and we finish this document with conclusion and future work in Section 7.

2 Syntax of classes and types

Our base language follows the syntax of Java; each program is a set of class declarations and each class declaration contains sets of fields and method declarations. However, we annotate our class declarations with information about the size of the objects (the number of elements of a data structure). Moreover we also annotate field and method declarations and the type parameters of our method declarations may be session types. The syntax of classes is shown in Figure 1, with the identifiers conventions: C for class identifiers, f for field identifiers, m for method identifiers, and N for size parameter identifiers.

Size parameters are annotated in class declarations and used in field declarations to describe (exactly or in maximum) how many elements a data structure has. For some class declaration C_3, consider, for instance, the class declarations:

$$\mathsf{class}\ C_1\langle N\rangle\ \{\ \ \mathsf{@has}\ N:\ C_3\ c_3;\ \ \}\qquad \mathsf{class}\ C_2\langle N\rangle\ \{\ \ \mathsf{@has\ up\_to}\ N:\ C_3\ c_3;\ \ \}$$

The type $C_1\langle 5\rangle$ represents an object that has exactly 5 elements of type C_3 while the type $C_2\langle 5\rangle$ represents an object that maximally has 5 elements of class C_3.

The method declaration is very similar to the Java method declaration where the returning type can be a type T or the void type and it has a set of parameters (each one with a type assigned). Note the possibility to have channel end-points as parameters (a variable assigned to a session type). The set annotation is used to change the size of an object. For instance if we add or remove elements of a data structure we need to indicate the new number of elements. A value v (a natural number or the result of an arithmetic expression) may denote either the number of elements of the object or the repetitions of a given communication

$$
\begin{array}{l}
\textbf{Session type declaration: } ST \ ::= \ \mathsf{session}\ S[\langle N^* \rangle] = SB \\
\textbf{Session type body: } SB \ ::= \ !T.SB \ \mid \ ?T.SB \ \mid \ \mathsf{end} \ \mid \\
\quad +\{l_i : SB_i\}_{i \in I} \ \mid \ \&\{l_i : SB_i\}_{i \in I} \ \mid \ \mathsf{rec}\ a.SB \ \mid \ a[v] \ \mid \ SB_1; SB_2 \\
\textbf{Session type invocation: } SI \ ::= \ S\langle v^* \rangle \ \mid \ \mathsf{dualof}\ S\langle v^* \rangle
\end{array}
$$

Figure 2 Syntax of session types.

pattern (recursive session types). We omit the syntax of expressions because it should be similar to that of Java expressions. However, note that our language supports also session operations. Session operations and session types are explained later. Our syntax allows five different types: the primitives boolean, integer, char and the string$[v]$ type (representing an array of chars of length v), and the object types, represented by a class identifier and the respective annotation for size.

In order to complement the explanation of our language we can use an example where we have a professor communicating with an administrative system. The professor should authenticate her credentials and after that she can provide an identifier of a course and ask to the system a list of students, or she can give a particular student identifier and obtain the respective student. Below, we show four class declarations useful to this example.

```
class Login {
  string[50] username;
  string[20] password;
}
```

```
class Student {
  string[150] name;
  integer age;
}
```

```
class StudentList<N> {
  @has up_to N;
  Node head;
}
```

```
class Node {
  Node next;
  Student student;
}
```

The class Login is a class with two fields only, username and password. Each instance of the class Student has a name (which in maximum has 150 characters) and an age. The class StudentList represents a list with a maximum of N students. In fact, we have that the list has N nodes, however each node has 1 student and therefore we may say that there are N students in the list. The class Node has a reference to the next node in the list and a reference to its student. If we want to add operations to change the list (for instance, add a new student) we can write the following method declaration in the class StudentList.

```
@set has(this.head), N + 1; void addStudent(Student n) { ... }
```

This means that after the invocation of this method, the list will have one more Student.

3 Syntax of session types

We intend to integrate this language with the theory of session types to describe the communication during one session [9, 16]. We consider bidirectional channels, where each channel is composed of two end-points; when two processes/objects communicate, each one possesses one of the channel ends and each channel end has a session type assigned. For I some index set and with the identifiers conventions $N \in \mathsf{Id}$, for size and repetition parameters, and $S \in \mathsf{Id}$, for session identifiers, Figure 2 shows the syntax of session types.

A session type declaration, ST assigns an identifier and size information to a session type, the session type body. It binds the size variables, denoted by N, that may be used in the session type body. We may use the session type $!T.SB$ to describe a channel end that should send a value of type T and then proceed as SB, or the type $?T.SB$ to receive a value of type T and continue behaving as described in SB. To describe an end-point that selects an option from a fixed range of options, we use the type $+\{l_i : SB_i\}_{i \in I}$. The session type to describe an end-point that offers this menu of options is $\&\{l_i : SB_i\}_{i \in I}$. The type end should be assigned to end-points where no further interaction may occur. The syntax presented is very similar to other languages with support for binary session types, such as [6, 16]. However we introduce two constructs that we believe to be different from the other approaches: the limitation of recursion, through the type variable annotation (usually the type/recursion variable is not annotated with information about repetition), and the sequence of session types. Using recursive types of the form $\mathsf{rec}\ a.SB$ and the annotated type variable $a[v]$, it is also possible to write a session type that maximally allows the repetition of a given behaviour v times. For instance, the type $\mathsf{rec}\ a.!\mathsf{integer}.a[3]$ when assigned to an end-point, means that it can send 3 integers. We can consider that $\mathsf{rec}\ a.!\mathsf{integer}.a[3]$ is equivalent to $!\mathsf{integer}.!\mathsf{integer}.!\mathsf{integer}.\mathsf{end}$. The type $\mathsf{rec}\ a + \{l_1 :!T_1.a[3], l_2 :!T_2.a[2]\}$ governs a channel end that selects between l_1 and l_2 a total of 5 times, sends 3 values of type T_1 and 2 values of type T_2. An end-point assigned to $SB_1; SB_2$ will first behave as defined by SB_1 and then as SB_2. We may say that the session type $SB_1; SB_2$ is an abbreviation for SB_1 where we replace end by SB_2, which is $[SB_2/\mathsf{end}]SB_1$ [1]. In order to use a declared session type our syntax provides a constructor for the session type invocation composed of the name assigned to the session type, S, and the value or the result of the arithmetic expression, v, to be used in the session type SB. It also provides for a dual type invocation, that give the dual session type—it is important to ensure that when one of the channel ends sends a value of type T, the other must be ready to receive a value of type T and when one of the channel ends selects an option from a menu, the other must offer a menu that contains this option. This means that the end-points of a channel must have *dual* behaviours [2].

Our first session type declaration is the one to govern the administrative system communication side:

```
session Admin<u1, u2> =
  rec a. ?Login.
  +{authentication_denied: !string[u1].
    &{try_again: a[2], close: end},
    authentication_accepted:
      rec b. &{all_students: ?integer: !StudentList<u2>.end,
                    single_student: ?integer. !Student.b[u2]}
  }
```

First the system receives an object Login with the credentials of the professor, then it should select between authentication_denied and authentication_accepted. If it selects the first option, then it must send the string with the failure reason to the professor and offer a choice between try_again and close. The professor may only try to login 3 times; after that, the communication channel is closed. When the system selects the option authentication_accepted, it offers two options to the professor: all_students, where the system expects to receive an integer with course identifier to respond with a StudentList, or single_student where the system

[1] We still need to verify if the session type is well-formed. For instance the type `rec a. end; a` is allowed by our syntax however it is not contractive. Recursive types are required to be contractive, that is, it cannot be of the form or has any sub-type of the form `rec a1, ... rec an. a1`

[2] The duality function can be found in https://wp.doc.ic.ac.uk/jvicent1/files

Session operations: $O \ ::= \ x \ y \ = \ \mathsf{new} \ (SI_1@L_1, SI_2@L_2) \ \mid \ z.\mathsf{send}(e) \ \mid$

$z.\mathsf{receive}() \ \mid \ z.\mathsf{select}(l) \ \mid \ \mathsf{switch}(z)\{\mathsf{case} \ l_i : e_i\}_{i \in I} \ \mid \ \mathsf{startSession}\{e.m(e^*), e.m(e^*)\}$

where x, y, z are channel names, l denotes labels and L location identifiers.

Figure 3 Syntax of session operations.

expects an **integer** with the student identifier and replies with a Student. When the option single_student is selected, the menu will be available again, after the sending of the student, in a maximum of u2 times.

4 Session operations

In this section we present the syntax of session operations shown in Figure 3. We start with the channel creation of the form $x \ y \ = \ \mathsf{new} \ (SI_1@L_1, SI_2@L_2)$ to create a new channel defined by two end-points x and y. The end-point x is assigned to the resulting session type of the type invocation SI_1, while y is assigned to SI_2, which should be dual of SI_1. The channel end x will be used by an object in some location L_1 while y will be used from location L_2. An end-point can be used to send the result of an expression with $z.\mathsf{send}(e)$, to receive a value, $z.\mathsf{receive}()$, to select an option labelled as l, from a fixed range of options, with $z.\mathsf{select}(l)$ and to offer a menu of options, using $\mathsf{switch}(z)\{\mathsf{case} \ l_i : e_i\}_{i \in I}$, for I some index set. Our syntax support also a primitive to start the session between two participants of the communication by invoking two methods—we assume that the communication happens between those methods. Note that the channel ends must be passed as parameters of the method, so that the communication may proceed. For instance in order to start the communication in our example we can use the following code [3].

```
// declaration of maxStudents and failureReason variables
Administrator a = new Administrator(maxStudents);
Professor p = new Professor(maxStudents);
x y = new (Admin<failureReason.length, maxStudents> @ L1,
       dualof Admin<failureReason.length, maxStudents> @ L2);
startSession{a.communicate(x), p.communicate(y)}
```

5 Calculation of communication cost

In this section we present the methodology to calculate the communication cost of a given channel using its session type. We assume that we have: a function sending_cost that expects two abstract locations L_1 and L_2 and returns the cost of sending 1 word from the concrete location of L_1 to the concrete location of L_2; a function receiving_cost that also expects two abstract locations L_1 and L_2 and returns the cost of receiving 1 word from the concrete location of L_2 to the concrete location of L_1. We also consider that a label of a choice type fits in 1 byte (for instance, the selection of an option is equivalent to the output of a label and we can use the function sending_cost). Furthermore, we assume that we have a function sizeOf that expects a type T and returns the number of bytes of a value or object of this type[4].

[3] What we envision as the full code of our example can be found in `https://wp.doc.ic.ac.uk/jvicent1/files`.

[4] We are currently working in this function and we do not include it in the document due space limitation.

In this section we show how to calculate the communication cost of a channel with the Admin session type assigned. The first step is to normalise the session type, that is, for a session type SB_1 we need to obtain a different session type SB_2 that has the same communication cost as SB_1 and respects the following properties:

- In a session type of the form $\mathsf{rec}\ a.SB$ the type variable a appears free in SB only once;
- In a choice type (selection or branching) if there is an option SB_i which has a free type variable, then the menu of options is composed only by SB_i.

This step is important when we have a branching type with recursion. A naive approach would be to calculate the cost of all options and choose the worst case (most expensive communication). However if we analyse, for instance, our session type Admin we can conclude that it would be an optimistic cost given that the authentication can be denied twice before to be accepted. After the normalisation, we are able to calculate the communication cost. The intuition behind the communication cost function is very simple. For the input/output types we only need to calculate the size of the parameter and sum to the cost of the continuation type. And given that the type is normalised, the cost of a branch type is the maximum cost of all branches. Both definitions of the functions normalisation and communication cost can be found in `https://wp.doc.ic.ac.uk/jvicent1/files`.

The normalisation

Informally, we may say that in the worst case, an end-point governed by the Admin session type receives three objects of type Login, meaning that the professor failed the first two authentication attempts. For each failed authentication the server sends the failure reason and offers the option to try to authenticate again. Then it selects the label authentication_accepted. After that, the end-point should be used to send b objects of type Student after receive b values of type **integer** (when the option single_student is selected by the professor). Finally the channel end should be used to receive an **integer** and send the complete list of students, called StudentList. If we normalise the Admin session type, we obtain:

$$\begin{aligned}
&\mathsf{rec}\ a.?\mathsf{Login}.+\{\mathsf{authentication\_denied}:!\mathsf{string}[u_1].\&\{\mathsf{try\_again}:a[2]\}\};\\
&?\mathsf{Login}.+\{\mathsf{authentication\_denied}:\&\{\mathsf{close}:\mathsf{end}\},\\
&\qquad\qquad\mathsf{authentication\_accepted}:\mathsf{rec}\ b.\&\{\mathsf{single\_student}:?\mathsf{integer}.!\mathsf{Student}.b[u_2]\};\\
&\qquad\qquad\qquad\&\{\mathsf{all\_students}:?\mathsf{integer}.!\mathsf{StudentList}.\mathsf{end}\}\}
\end{aligned}$$

The result is a session type that governs an end-point that performs the operations described above. As we can see the above session type includes two subtypes that terminate the communication in the channel end—the types $\&\{\mathsf{all\_students}:?\mathsf{integer}.!\mathsf{StudentList}.\mathsf{end}\}$ and $!\mathsf{string}[u_1].\&\{\mathsf{close}:\mathsf{end}\}$— and we only want to consider the worst case, that is, the first subtype. Therefore this should be taken into account in the communication cost function.

The communication cost calculation

Now that we have our normalised type we can calculate the communication cost of the first subtype (the calculation for the rest of the types is similar).

$$\begin{aligned}
&\mathsf{costc}(\mathsf{rec}\ a.?\mathsf{Login}.+\{\mathsf{authentication\_denied}:!\mathsf{string}[u_1].\&\{\mathsf{try\_again}:a[2]\}\};L_1,L_2)=\\
&2*g(L_1,L_2)*(\mathsf{sizeOf}(\mathsf{Login})+1)+2*f(L_1,L_2)*(1+\mathsf{sizeOf}(\mathsf{string}[u_1]))
\end{aligned}$$

6 Related work

There are already several papers about the incorporation of session types in object-oriented languages. Dezani et al. proposed the first integration of a class-based object-oriented programming language with session types [5]. This work was followed by the programming language Moose [4], a multi-threaded object-oriented core language augmented with session types, and by an extension of Moose [3], with bounded session types for object oriented languages. Hu et al. introduced an extension of Java, called SJ [10], a session-based distributed programming language. Following a different approach, Gay et al. [7] presented a distributed object-oriented programming language where it is possible to specify the possible sequences of method calls attaching session types to class definitions. Based on this work, Bica [1] is an extension to Java5 that allows the verification of Java code against session types, and Mool [2] is a minimal object-oriented language with support for concurrency. There is also an integration of multiparty session types with a Java extension, inspired on SJ, presented by Sivaramakrishnan et al. [14, 15].

In the scope of high-performance computing, Ng et al. created Session C, a programming framework for message-passing parallel programming that combines multiparty session types with the C programming language and its respective runtime libraries [12]. Based on Session C, Ng et al. presented a programming framework for safe and reconfigurable parallel designs and Pable a parametrised protocol description language [13, 11]. Honda et al. also proposed a methodology for MPI programs based on session types [8]. As in Session C, the idea is to check MPI programs against local protocols (obtained from multiparty session types), in polynomial time, to ensure type safety, communication safety and deadlock freedom.

At the best of our knowledge, all of these languages, and other which we do not mention, do not use session types to calculate communication costs. Some of them consider some optimizations using session types, such as [14, 15], however none of them do it as we intend to do in the future: use these costs to optimise the communication.

7 Conclusion and further work

Conclusion. We present our first steps towards the creation of a new object oriented programming language with primitives for communication over bidirectional channels. Session types will be assigned to Channels allowing to statically verify if the exchanged messages respect the expected order and type and in addition to calculate the communication cost of an end-point when this is located at some location. We believe that this communication cost information can be used to improve the communication in a concurrent computation, even that it only gives us the communication cost for the worst case.

Further Work. This is work is still in the beginning and there are yet several questions that we have to answer and several points to improve, such as, about the correctness of our functions and how we can check that the number of repetitions of session types and the number of elements of a data structure are correct. Moreover we want to integrate the sizeOf function and find a better solution for the annotated type variable in the duality function. Furthermore, we started our work with binary session types however our main goal is to have a object-oriented programming language with multiparty session types. Given that multiparty session types can be projected into local session types, we decided to start our work with binary session types and after we achieve our goals, we can extend our ideas to multiparty session types. In addition, we intend to study what optimizations we can do using this cost information.

References

1 Alexandre Caldeira and Vasco T. Vasconcelos. Bica. `http://gloss.di.fc.ul.pt/bica`.

2 Joana Campos and Vasco T. Vasconcelos. Channels as objects in concurrent object-oriented programming. In *PLACES 2010*, volume 69 of *EPTCS*, pages 12–28, 2011.

3 Mariangiola Dezani-Ciancaglini, Elena Giachino, Sophia Drossopoulou, and Nobuko Yoshida. Bounded session types for object oriented languages. In *Formal Methods for Components and Objects*, volume 4709 of *LNCS*, pages 207–245. Springer Berlin Heidelberg, 2007.

4 Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session types for object-oriented languages. In *European Conference on Object-Oriented Programming*, 2006.

5 Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, Alexander Ahern, and Sophia Drossopoulou. A distributed object-oriented language with session types. In *Trustworthy Global Computing*, pages 299–318, 2005.

6 Juliana Franco and VascoThudichum Vasconcelos. A concurrent programming language with refined session types. In *Software Engineering and Formal Methods*, LNCS, pages 15–28. Springer International Publishing, 2014.

7 Simon Gay, Vasco T. Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Modular session types for distributed object-oriented programming. In *Principles of Programming Languages*, pages 299–312. ACM Press, 2010.

8 Kohei Honda, EduardoR.B. Marques, Francisco Martins, Nicholas Ng, Vasco T. Vasconcelos, and Nobuko Yoshida. Verification of mpi programs using session types. In *Recent Advances in the Message Passing Interface*, LNCS, pages 291–293. Springer Berlin Heidelberg, 2012.

9 Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *European Symposym on Programming*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

10 Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in Java. In *European Conference on Object-Oriented Programming*, volume 5142 of *LNCS*, pages 516–541. Springer, 2008.

11 Nicholas Ng and Nobuko Yoshida. Pabble: Parameterised scribble for parallel programming. In *PDP 2014*, pages 707–714. IEEE Computer Society, 2014.

12 Nicholas Ng, Nobuko Yoshida, and Kohei Honda. Multiparty session C: safe parallel programming with message optimisation. In *Conference on Objects, Models, Components, Patterns*, volume 7304 of *LNCS*, pages 202–218. Springer, 2012.

13 Nicholas Ng, Nobuko Yoshida, Xinyu Niu, and Kuen Hung Tsoi. Session types: towards safe and fast reconfigurable programming. *SIGARCH Computer Architecture News*, 40(5):22–27, 2012.

14 K. C. Sivaramakrishnan, Mohammad Qudeisat, Lukasz Ziarek, Karthik Nagaraj, and Patrick Eugster. Efficient sessions. *Sci. Comput. Program.*, pages 147–167, 2013.

15 K.C. Sivaramakrishnan, Karthik Nagaraj, Lukasz Ziarek, and Patrick Eugster. Efficient session type guided distributed interaction. In *Coordination Models and Languages*, LNCS, pages 152–167. Springer Berlin Heidelberg, 2010.

16 Vasco T. Vasconcelos. Fundamentals of session types. *Information and Computation*, 217:52–70, 2012.

Symbolic Execution as DPLL Modulo Theories

Quoc-Sang Phan

Queen Mary University of London
q.phan@qmul.ac.uk

Abstract

We show how Symbolic Execution can be understood as a variant of the DPLL($\mathcal{T}$) algorithm, which is the dominant technique for the Satisfiability Modulo Theories (SMT) problem. In other words, Symbolic Executors are SMT solvers. This view enables us to use an SMT solver, with the ability of generating all models with respect to a set of Boolean atoms, to explore all symbolic paths of a program. This results in a more lightweight approach for Symbolic Execution.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Symbolic Execution, Satisfiability Modulo Theories

Digital Object Identifier 10.4230/OASIcs.ICCSW.2014.58

1 Introduction

Symbolic Execution (SE) [11] is now popular. It is increasingly used not only in academic settings but also in industry, such as in Microsoft, NASA, IBM and Fujitsu [4]. In the success of SE, the efficiency of SMT solvers [9] is a key factor. In fact, while SE was introduced more than three decades ago, it had not been made practical until research in SMT made significant advances [5].

State-of-the-art SMT solvers, e.g. [8, 2], implement the DPLL($\mathcal{T}$) algorithm [15] which is an integration of two components as follows. The first component is a propositional satisfiability (SAT) solver, based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [7], to search on the *Boolean skeleton* of the formula. The second component is a decision procedure, called the $\mathcal{T}$-solver, to check the consistency w.r.t. the theory $\mathcal{T}$ of conjunctions of literals, each literal is an atomic formula or the negation of an atomic formula. The path conditions generated by a Symbolic Executor, e.g. Symbolic PathFinder (SPF) [14], are also conjunctions of literals. Therefore, when an SMT solver checks such a path condition, only the $\mathcal{T}$-solver works on it, and the SAT component is not used.

On the other hand, a *classical* Symbolic Executor [11] can be divided into two components. The first component, dubbed as *Boolean Executor* hereafter, executes the instructions, and updates the path condition. The second component is a $\mathcal{T}$-solver (since the SAT solver is not used) to validate the consistency of the path condition. This paper shows that a Boolean Executor does the same work as the DPLL algorithm. Thus, SE is a variant of DPLL($\mathcal{T}$). This view is important since it connects two communities and can give an insight for future research.

Based on this new insight, we propose a lightweight approach for SE which uses the DPLL component of an SMT solver to explore symbolic paths instead of a Boolean Executor. Our approach relies on an SMT solver with the ability of generating all models w.r.t. a set of Boolean atoms.

Imperial College Computing Student Workshop (ICCSW'14).
Editors: Rumyana Neykova and Nicholas Ng; pp. 58–65
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2 Background

Before showing the correspondence between a Symbolic Executor and an SMT solver, we recall some background on Symbolic Execution and the SMT problem.

2.1 Symbolic Execution

Symbolic Execution [11] (SE) is a programming analysis technique which executes programs on unspecified inputs, by using symbolic values instead of concrete data. For each executed program path, SE builds a path condition *pc* which represents the condition on the inputs for the execution to follow that path.

```
function EXECUTOR(Program $P$){
    PathCondition pc = TRUE;
    InstructionPointer i = NULL;
    update($P, i$);
    if ($i$ == RETURN)  return;
    while (TRUE) {
        $l$ = chooseLiteral($i$);
        $pc = pc \wedge l$;
        update($P, i$);
        if ($i$ == RETURN)
            if (allStatesAreExplored())
                return;
            else backtrack($pc, i$);
} }
```

Figure 1 A simplified Boolean Executor.

For an **if** statement with condition c, there are three possible cases: (**i**) $pc \vdash c$: SE chooses the **then** path; (**ii**) $pc \vdash \neg c$: SE chooses the **else** path; (**iii**) $(pc \nvdash c) \wedge (pc \nvdash \neg c)$: SE executes both paths: in the **then** path, it updates the path condition $pc_1 = pc \wedge c$, in the **else** path it updates the path condition $pc_2 = pc \wedge \neg c$. The satisfiability of the path condition is checked by SMT solvers, for example the symbolic executor KLEE [3] uses the SMT solver STP [10], while SPF provides a parameter to select one of the SMT solvers: CVC3, Yices, Z3. In this way, only feasible program paths are explored. Test generation is performed by solving the path conditions.

2.2 SMT and DPLL($\mathcal{T}$)

Satisfiability Modulo Theories (SMT) is the problem of checking the satisfiability of logical formulas over one or more first-order theories $\mathcal{T}$.

Our setting is standard first-order logic. Boolean variables are called Boolean atoms or simply atoms, and atomic formulas are called theory atoms or $\mathcal{T}$-atoms. A *truth assignment* μ for a formula φ is a truth value assignment to the $\mathcal{T}$-atoms of φ. We define a bijective function $\mathcal{BA}$ (*Boolean abstraction*) which maps Boolean atoms into themselves and $\mathcal{T}$-atoms into fresh Boolean atoms. The *Boolean refinement* function $\mathcal{BR}$ is then defined as the inverse of $\mathcal{BA}$, which means $\mathcal{BR} = \mathcal{BA}^{-1}$.

At a high level, an SMT solver is the integration of two components: a SAT solver and $\mathcal{T}$-solvers. SMT solving can be viewed as the iteration of the two following steps. First, the

SAT solver searches on the Boolean abstraction of the formula, $\varphi^P = \mathcal{BA}(\varphi)$, and returns a (partial) truth assignment μ^P. The $\mathcal{T}$-solvers then check the Boolean refinement of the candidate, $\mathcal{BR}(\mu^P)$, whether it is consistent with the theories $\mathcal{T}$.

```
function DPLL(BooleanFormula $\varphi$){
    $\mu$ = TRUE; status = propagate($\varphi, \mu$);
    if (status == SAT)  return SAT;
    else if (status == UNSAT)  return UNSAT;
    while (TRUE) {
        $l$ = chooseLiteral($\varphi$);
        $\mu = \mu \wedge l$;
        status = propagate($\varphi, \mu$);
        if (status == SAT)  return SAT;
        else if (status == UNSAT)
            if (allStatesAreExplored())
                return UNSAT;
            else backtrack($\varphi, \mu$);
} }
```

Figure 2 DPLL algorithm.

The dominant approach for SAT solvers is the DPLL family of algorithms [7]. The simplest form of DPLL is depicted in Figure 2, its input is a propositional formula φ in Conjunctive Normal Form (CNF), which means φ takes the form:

$$\varphi = \bigwedge (l_1 \vee l_2 \cdots \vee l_k)$$

A (finite) disjunction of literals $(l_1 \vee l_2 \cdots \vee l_k)$ is called a clause, and a literal l_i is an atom or its negation. A clause that contains only one literal is called a *unit clause*. At a high level, DPLL is a stack-based depth-first search procedure which iteratively performs the two following steps: first choose a literal l_i from the remaining clause, and add it to the current truth assignment; then apply Boolean Constraint Propagation (BCP), backtracking if there is a conflict. These two steps are repeated until a model is found or all states are explored without finding a model.

The procedure BCP for a literal l_i removes all the clauses containing l_i, and removes $\neg l_i$ from the remaining clauses. If the removal results an empty clause, the search encounters a conflict.

3 Illustration of DPLL($\mathcal{T}$)

A complete formal description of first-order theories and the DPLL($\mathcal{T}$) algorithm can be found in, e.g., [15]. Here we briefly introduce necessary preliminaries via a running example as follows.

$$\begin{aligned}\varphi := {} & (\neg(x_0 > 5) \vee T_1) \wedge ((x_0 > 5) \vee T_2) \wedge (\neg(x_0 > 5) \vee (x_1 = x_0 + 1)) \wedge \\ & (\neg(x_1 < 3) \vee T_3) \wedge (\neg(x_1 < 3) \vee (x_2 = x_1 - 1)) \wedge \\ & ((x_1 < 3) \vee T_4) \wedge ((x_1 < 3) \vee (y_1 = x_1 + 1))\end{aligned} \tag{1}$$

φ is a Linear Arithmetic formula. Boolean variables, $T_1 \ldots T_4$, are called Boolean atoms, and atomic formulas, e.g. $(x_0 > 5)$, are called theory atoms or $\mathcal{T}$-atoms. A first-order formula can be abstracted into a Boolean skeleton by replacing all the $\mathcal{T}$-atoms with new Boolean atoms, which is often called *Boolean abstraction*. For the example above, we define new Boolean variables G_1, G_2, A_1, A_2, A_3 for the Boolean abstraction of $\mathcal{T}$-atoms, and the abstraction can be expressed as:

$$\begin{aligned}\mathcal{BA} := G_1 = (x_0 > 5) \wedge G_2 = (x_1 < 3) \wedge \\ A_1 = (x_1 = x_0 + 1) \wedge A_2 = (x_2 = x_1 - 1) \wedge A_3 = (y_1 = x_1 + 1)\end{aligned} \tag{2}$$

As the result, we obtain a formula φ^P (^{P} stands for propositional) as the Boolean skeleton of φ. Obviously, φ is logically equivalent to $\varphi^P \wedge \mathcal{BA}$.

$$\begin{aligned}\varphi^P := (\neg G_1 \vee T_1) \wedge (G_1 \vee T_2) \wedge (\neg G_1 \vee A_1) \wedge \\ (\neg G_2 \vee T_3) \wedge (\neg G_2 \vee A_2) \wedge \\ (G_2 \vee T_4) \wedge (G_2 \vee A_3)\end{aligned} \tag{3}$$

The DPLL($\mathcal{T}$) algorithm is the integration of the DPLL algorithm with a $\mathcal{T}$-solver. The DPLL algorithm searches on φ^P, returning a conjunction of Boolean literal μ^P. Replacing all the new Boolean atoms, G_i and A_i, in μ^P with their corresponding $\mathcal{T}$-atoms, we obtain the conjunction μ in $\mathcal{T}$. The $\mathcal{T}$-solver then checks whether μ is consistent with the theory $\mathcal{T}$. Below is the illustration of DPLL($\mathcal{T}$) on φ (for the limit of space, only decision literals are shown in μ^P):

0. $\mu^P = \texttt{True}$	φ^P
1. $\mu^P = G_1$	$\varphi^P = (\neg G_2 \vee T_3) \wedge (\neg G_2 \vee A_2) \wedge (G_2 \vee T_4) \wedge (G_2 \vee A_3)$
2. $\mu^P = G_1 \wedge G_2$	$\varphi^P = \texttt{True}$; $\mathcal{T}$-solver(μ) $= \texttt{Inconsistent}$
3. $\mu^P = G_1$	$\varphi^P = (\neg G_2 \vee T_3) \wedge (\neg G_2 \vee A_2) \wedge (G_2 \vee T_4) \wedge (G_2 \vee A_3)$
4. $\mu^P = G_1 \wedge \neg G_2$	$\varphi^P = \texttt{True}$; $\mathcal{T}$-solver(μ) $= \texttt{Consistent}$

The DPLL algorithm tries to build a model using three main operations: `decide`, `propagate`, and `backtrack` [9]. The operation `decide` heuristically chooses a literal l (which is an unassigned Boolean atom or its negation) for branching. The operation `propagate` then removes all the clauses containing l, and deletes all occurrences of $\neg l$ in the formula; this procedure is also called *Boolean Constraint Propagation* (BCP). If after deleting a literal from a clause, the clause only has only one literal left (*unit clause*), BCP assigns this literal to `True`. If deleting a literal from a clause results in an empty clause, this is called a conflict. In this case, the DPLL procedure must `backtrack` and try a different branch value.

At step 1, G_1 is decided to be the branching literal, and the $\mathcal{T}$-solver validates that $(x_0 > 5)$ is consistent. BCP removes the clause $(G_1 \vee T_2)$, and deletes all occurrences of $\neg G_1$. This results in two unit clauses T_1 and A_1, so they are assigned to `True`, which means $\mu^P = G_1 \wedge T_1 \wedge A_1$. Similarly, at step 2 G_2 is chosen, i.e. $\mu^P = G_1 \wedge T_1 \wedge A_1 \wedge G_2$. The $\mathcal{T}$-solver checks the conjunction: $\mu = (x_0 > 5) \wedge T_1 \wedge (x_1 < 3) \wedge (x_1 = x_0 + 1)$. This is obviously inconsistent, thus DPLL($\mathcal{T}$) backtracks and tries $\neg G_2$, which leads to a consistent model.

Note that DPLL($\mathcal{T}$) refers to various procedures integrating DPLL and a $\mathcal{T}$-solver. There are procedures with an integration schema different from what we have described here. The interested reader is pointed to [15] for further references.

4 Symbolic Execution as DPLL($\mathcal{T}$)

Intuitively, a program can be encoded into a (first-order) formula whose models correspond to program traces. Symbolic Executors explore all program traces w.r.t. the set of program conditions, therefore they can be viewed as SMT solvers that return all (partial) models w.r.t. a set of Boolean atoms.

In this paper we only consider bounded programs, since this is the class of programs that SE can analyse. This means every loop can be unwound into a sequence `if` statements. In order to encode a program into a formula, all program variables are renamed in the manner of Static Single Assignment form [6]: each variable is assigned exactly once, and it is renamed into a new variable when being reassigned. In this way, assignments such as $x = x + 1$ will not be encoded into an unsatisfiable atomic formula. Under these settings, a program P can be modelled by a *Symbolic Transition System* (STS) as follows:

$$P \equiv (S, I, G, A, T)$$

S is the set of program states, $I \subseteq S$ is the set of initial states; each state in the STS models the computer memory at a program point. G is the set of guards and A is the set of actions; guards and actions are first-order formulas. An action models the effect of an instruction on the computer memory. Actions that do not update the computer memory (e.g. conditional jumps) are Boolean atoms, the others are $\mathcal{T}$-atoms. $T \subseteq S \times G \times A \times S$ is the transition function, $t_{ij} = \langle s_i, g_{ij}, a_{ij}, s_j \rangle \in T$ models a transition from state s_i to state s_j by taking action a_{ij} under the guard g_{ij}. After a transition $t_{ij} : s_i \to s_j$, the state s_j is exactly as the state s_i apart from the variable updated by the action a_{ij}.

One way to encode a transition t_{ij} into a first-order formula is to present it in the form: $t_{ij} \equiv g_{ij} \to a_{ij}$, or equally $t_{ij} \equiv \neg g_{ij} \vee a_{ij}$. This encoding expresses that satisfying the guard g_{ij} implies that the action a_{ij} is performed. In this way, a program trace is defined as a sequence of transitions:

$$t_{01} \wedge t_{12} \wedge \cdots \wedge t_{(k-1)k} = (\neg g_{01} \vee a_{01}) \wedge (\neg g_{12} \vee a_{12}) \cdots \wedge (\neg g_{(k-1)k} \vee a_{(k-1)k})$$

The semantics of the program is then defined as the set of all possible traces, or equally the set of all possible transitions, which can be represented as the following formula:

$$\varphi = \bigwedge_{t_{ij} \in T} t_{ij} = \bigwedge_{t_{ij} \in T} (\neg g_{ij} \vee a_{ij}) \tag{4}$$

Fig. 3 depicts a simple example program and its associated STS. Encoding this STS following (4) results in the formula (1) that we have illustrated with DPLL($\mathcal{T}$) in the previous section. We now illustrate this example with SE.

At a high level, a Symbolic Executor can be considered as the integration of two components: a Boolean Executor (BE) to execute the instructions and a $\mathcal{T}$-solver to check the feasibility of path conditions. For example, SPF has a parameter `symbolic.dp` to customize which decision procedure to use. If we set this parameter with the option `no_solver` then SPF solely works on the BE.

A BE can be described as trying to build *all* path conditions using three main operations: `decide`, `update` and `backtrack`. The operation `decide` chooses a literal l, a condition (or its negation) of an `if` statement, for branching, adding it to the path condition. The operation `update` then symbolically executes a block of statements, i.e. no branching statement presents, updating the computer memory. When the BE reaches the end of a symbolic path,

```
void test(int x, int y){
  if(x > 5){
    x++;
    if (x < 3)
      x--;
    else
      y = x + 1;
  }
}
```

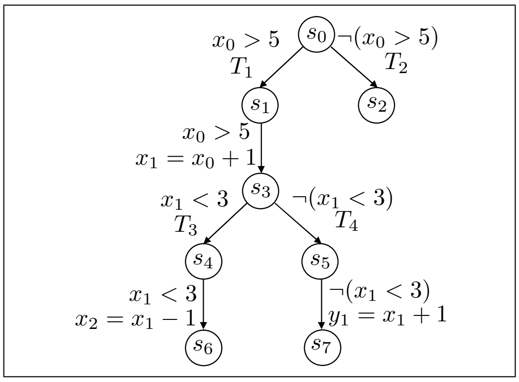


Figure 3 A simple program and its associated STS. The first `if` statement is modelled by two transitions $\langle s_0, (x_0 > 5), T_1, s_1\rangle$ and $\langle s_0, \neg(x_0 > 5), T_2, s_2\rangle$; the assignment `x++` is modelled by $\langle s_1, (x_0 > 5), x_1 = x_0 + 1, s_3\rangle$; similarly for the rest of the program.

it backtracks to explore other paths. A Symbolic Executor, which is the integration of a BE and a $\mathcal{T}$-solver, backtracks if the path condition is not satisfied.

Both DPLL and BE rely on Depth-First Search, they are similar in the way they `decide` and `backtrack`[1]. After choosing a literal, e.g. $(x_0 > 5)$, BE executes the block it guards, i.e. T_1 and $x_1 = x_0 + 1$. This is exactly the same as in DPLL: after choosing g_{ij}, for all the clauses $(\neg g_{ij} \vee a_{ij})$, BCP deletes $\neg g_{ij}$, assigning a_{ij} to `True`. Therefore, the operation `update` does the same work as BCP, we can view a BE as implementing the DPLL algorithm, and SE as DPLL($\mathcal{T}$).

5 A lightweight approach for Symbolic Execution using All-SMT

While both Symbolic Executors and SMT solvers implement DPLL($\mathcal{T}$), the main difference between them is the models they return. Symbolic Executors explore all path conditions, each path condition is identified by the values of the Boolean abstractions of its condition constraints. For instance, although there are many values of x_0 satisfying the path $(x_0 < 5) \wedge \neg(x_1 < 3)$, it is identified by $G_1 =$ `True` and $G_2 =$ `False`. On the other hand, SMT solvers in general stop searching when a model is found. However, the solver MathSAT provides a functionality, called All-SMT [2], that given a formula and a set of Boolean atoms, it returns *all* models of the formula w.r.t. this set. Hence, by asking MathSAT to find all models w.r.t. the set of guards, we can explore all symbolic paths of the program.

We illustrate the approach with our running example in Fig. 3. The formula φ in (1) is expressed in SMT-LIB v2 format [1] as follows (for the limit of space variable declarations are omitted):

```
(assert (= (> x0 5) G1))
(assert (= (< x1 3) G2))
(assert (= (= x1 (+ x0 1)) A1))
(assert (= (= x2 (− x1 1)) A2))
(assert (= (= y1 (+ x1 1)) A2))
(assert (or (not G1) T1))
(assert (or G1 T2))
```

```
(assert (or (not G1) A1))
(assert (or (not G2) T3))
(assert (or (not G2) A2))
(assert (or G2 T4))
(assert (or G2 A3))
(check-allsat (G1 G2))
```

[1] We consider DPLL in its simplest form, without non-chronological backtracking.

Recall that $\varphi = \mathcal{BA} \wedge \varphi^P$ as defined in (2) and (3). Assertions from 1 to 5 are for $\mathcal{BA}$, and each assertion from 6 to 12 represents a clause of φ^P. The final command, `check-allsat`, asks the solver to return all models w.r.t. the set (G_1, G_2). Note that `check-allsat` is not included in standard SMT-LIB, and it is not supported by other solvers, e.g. Z3 [8].

Executing MathSAT on the example above, we obtain three models w.r.t. the set (G_1, G_2) as follows: (`True`,`False`), (`False`,`True`) and (`False`,`False`), which correspond to two feasible symbolic paths: $(x_0 > 5) \wedge \neg(x_1 < 3)$ and $\neg(x_0 > 5)$. As we did not model integer overflow rules, the path (G_1, G_2) as (`True`,`True`), i.e. $(x_0 > 5) \wedge (x_1 < 3)$, is not feasible (this path is feasible when the overflow in the assignment $x_1 = x_0 + 1$ changes a big integer into a small one). The path $\neg(x_0 > 5)$ is listed twice, since in this case the value of G_2, i.e. $(x_1 < 3)$, is irrelevant, but the solver still considered it in two cases `True` and `False`.

6 Discussions and Future Work

In this paper, we show the correspondence between a Symbolic Executor and an SMT solver implementing the DPLL($\mathcal{T}$) framework [15]. Therefore, the claims in this paper do not apply for the bit vector theory, since its solvers do not implement the DPLL($\mathcal{T}$) framework. Bit vector formulas are solved by flattening into propositional formulas, then the resulting formulas are checked by a SAT solver [10].

A common problem for both SE and SMT is that there is a considerable amount of redundancy in the queries to the $\mathcal{T}$-solver. Because the BE and its counterpart in SMT, the SAT component, build the conjunctions of literals in an incremental manner. To address this problem, previous work in SE and SMT used different approaches. In SE, the BE often uses the $\mathcal{T}$-solver as a black box, thus most research on constraint redundancy problem has focused on caching techniques. For example, KLEE [3] has a counterexample cache that maps of sets of constraints to counterexamples, Green [16] can store the constraints offline, reusing them in different runs of the Symbolic Executor. In SMT solver, there is a tight synergy between the SAT component and the $\mathcal{T}$-solver, and both are incremental and backtrackable. Therefore, the $\mathcal{T}$-solver do not start from scratch if the constraint is similar to the previous query.

An immediate direction for investigation is whether the caching techniques in KLEE and Green can improve the efficiency of SMT solvers. Conversely, it would be interesting to investigate if techniques developed for SMT can be exploited for SE, for example to develop a concurrent Symbolic Executor based on previous work in concurrent SMT solver [17].

7 Conclusion

We show the correspondence between Symbolic Execution and Satisfiability Modulo Theories. This correspondence is important, as it enables us to migrate techniques developed in one community to the other. Moreover, it can give an insight for future research. In our previous work, we proposed a DPLL-based algorithm to tackle the quantitative information flow problem, and implemented it using SPF [13, 12].

Based on the correspondence above, we propose a lightweight approach for SE, using the DPLL component of an SMT solver to explore symbolic paths. The main limitation of our approach is path redundancies, which is because the off-the-shelf All-SMT functionality is not tailored for SE. Hence, future work also includes extending the open-source SMT solver Z3 so that it can function as SE.

References

1 Clark Barrett, Aaron Stump, and Cesare Tinelli. The smt-lib standard: Version 2.0. In *SMT Workshop*, 2010.

2 Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The mathsat 4 smt solver. In *Proceedings of the 20th international conference on Computer Aided Verification*, CAV '08, pages 299–303, Berlin, Heidelberg, 2008. Springer-Verlag.

3 Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. OSDI'08, pages 209–224.

4 Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: Preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1066–1071, New York, NY, USA, 2011. ACM.

5 Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, February 2013.

6 R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 25–35, New York, NY, USA, 1989. ACM.

7 Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

8 Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

9 Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, September 2011.

10 Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th international conference on Computer aided verification*, CAV'07, pages 519–531, Berlin, Heidelberg, 2007. Springer-Verlag.

11 James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

12 Quoc-Sang Phan and Pasquale Malacaria. Abstract model counting: A novel approach for quantification of information leaks. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 283–292, New York, NY, USA, 2014. ACM.

13 Quoc-Sang Phan, Pasquale Malacaria, Oksana Tkachuk, and Corina S. Păsăreanu. Symbolic quantitative information flow. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, November 2012.

14 Corina S. Păsăreanu and Neha Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 179–180, New York, NY, USA, 2010. ACM.

15 Roberto Sebastiani. Lazy Satisfiability Modulo Theories. *Journal on Satisfiability, Boolean Modeling and Computation*, 3:141–224, 2007.

16 Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. Green: reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 58:1–58:11, New York, NY, USA, 2012. ACM.

17 Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Mendonça de Moura. A concurrent portfolio approach to smt solving. In *CAV*, pages 715–720, 2009.

Towards a Programming Paradigm for Artificial Intelligence Applications Based On Simulation*

Jörg Pührer

Institute of Computer Science, Leipzig University
Leipzig, Germany
puehrer@informatik.uni-leipzig.de

Abstract

In this work, we propose to use simulation as a key principle for programming AI applications. The approach aims at integrating techniques from different areas of AI and is based on the idea that simulated entities may freely exchange data and behavioural patterns. We introduce basic notions of a simulation-based programming paradigm and show how it can be used for implementing different scenarios.

1998 ACM Subject Classification I.2 Artificial Intelligence

Keywords and phrases Artificial Intelligence, Simulation, Programming Paradigm

Digital Object Identifier 10.4230/OASIcs.ICCSW.2014.66

1 Introduction

We sketch a programming paradigm based on simulation that is targeted towards applications of artificial intelligence (AI) [13]. Simulation has been used in different fields of AI (such as agent-based systems [10, 14] or evolutionary computation [3]) for achieving intelligent behaviour. The rationale is that many aspects of intelligent behaviour are complex and not well understood but emerge when the environment in which they occur is simulated adequately. We propose to use a simulation environment for realising AI applications that offers an easy way to integrate existing formalisms including methods from different areas of AI such as computational intelligence, symbolic AI, or statistical methods. This environment is composed of interacting *entities* that are grouped in different *worlds* and driven by concurrent *processes*. The proposed approach exploits advances in hardware and concurrent computing due to which simulation became feasible for many applications.

The next section introduces the basic notions of a simulation-based programming paradigm and Section 3 discusses how to use it for modelling different scenarios. In Section 4, we give considerations on the user interface, consistency maintenance, and discuss the relation to related techniques. Section 5 concludes the paper and gives an outlook on future work.

2 Simulation-Based Programming

In this section we explain the proposed simulation-based programming paradigm (SBP) on an abstract level. An SBP system deals with different *worlds*, each of which can be seen as a different point of view. The meaning of these worlds is not pre-defined by SBP, e.g., the programmer can decide to take an objectivistic setting and consider one world the designated

[*] A longer version of this paper appeared in the proceedings of ReactKnow 2014 [12]. The German Research Foundation (DFG) partially supported this work under grants BR-1817/7-1 and FOR 1513.

Imperial College Computing Student Workshop (ICCSW'14).
Editors: Rumyana Neykova and Nicholas Ng; pp. 66–73
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

real one or treat all worlds alike. Different worlds allow, e.g., to model the beliefs of different agents as in an agent-based approach. Other applications are hypothetical reasoning or realising different granularities of abstraction for efficiency (see Section 3).

A world contains a set of named *entities* which are the primary artifacts of SBP and represent the subjects of the simulation. Entities may have two sorts of named attributes: *data entries* which correspond to arbitrary data (including references to other entities) and *transition descriptions* which define the behaviour of the entities over time. The name of an entity has to be unique with respect to a world and serves as a means to reference the entity, however the same entity may appear in different worlds with potentially different attributes and attribute values. Transition descriptions can be seen as the main source code elements in the approach and they are, similar to the data entries, subject to change during runtime. This allows for a dynamic setting in which the behaviour of entities can change over time, e.g., new behaviour can be learned, acquired from other entities, or shaped by evolutionary processes. We do not propose a particular language or programming paradigm for specifying transition descriptions. It might, on the contrary, be beneficial to allow for different languages for different transition descriptions within the same simulation. E.g., a transition description implementing sorting can be realised by some efficient standard algorithm, while another transition description that deals with a combinatorial problem with many side constraints uses a declarative knowledge representation approach like answer-set programming (ASP) [8, 11] in which the problem can be easily modelled. We require transition descriptions—in whatever language they are written—to comply to a specific interface that allows us to execute them in asynchronous *processes*. In particular, the output of a transition contains a set of updates to be performed on worlds, entities, data entries, and transition descriptions. When a transition has finished, per entity, these changes are applied in an atomic transaction that should leave the entity in a consistent state (provided that the transition description is well designed). As mentioned, transition descriptions are executed in processes. Each process is associated with an entity and runs a transition description of this entity in a loop. A process can however decide to terminate itself or other processes at any time, initiate other processes, and wait for their results before finishing their own iteration.

We now formally summarise the concepts discussed above. We assume the availability of a set Σ of *semantics* for transitions descriptions. A semantics is a function defining how a piece of source code is interpreted and stands for the formalism used for a particular transition description. Let $\mathcal{N}$ be a set of names. We call a concept c *named* if it has an associated name $n_c \in \mathcal{N}$. A *map* is a set M of pairs $\langle n_v, v\rangle$ where v is a named object and $\langle n, v_1\rangle, \langle n, v_2\rangle \in M$ implies $v_1 = v_2$. With slight abuse of notation we write $v \in M$ for $\langle n_v, v\rangle \in M$.

▶ **Definition 1.**

- A *transition description* is a pair $t = \langle \mathrm{sc}, \sigma\rangle$, where sc is a piece of source code, and $\sigma \in \Sigma$ is a semantics.
- A *process* is a tuple $p = \langle t, \mathtt{t}_b\rangle$, where t is a transition description and $\mathtt{t}_b$ is a timestamp marking the begin of the current transition.
- An *entity* is a tuple $e = \langle D, T, P\rangle$, where D is a map of named data, T is a map of named transition descriptions, and P is a map of named processes. Entries of D,T, and P are called *properties* of e.
- A *world* is a map of named entities.
- An *SBP configuration* is a map of named worlds.

For space reasons we describe the runtime behaviour of an SBP system only on an informal level and refer the interested reader to a longer version of this paper [12]. We assume a pre-specified set Υ of *updates* describing what changes should be made to an SBP configuration together with a fixed *update function* υ that maps an SBP configuration, an entity name, the name of a world, and a set of updates, to a new SBP configuration. Whenever a process has finished, it returns a pair $\langle U, b_c\rangle$ where $U \subseteq \Upsilon$ is a set of updates and b_c is one of the boolean values *true* or *false* that decides whether the process should continue with another transition. Intuitively, $\langle U, b_c\rangle$ is the result of the semantics of the transition description of the process given the current SBP configuration. All updates created at one time instant are collected and used as input for the update function υ that produces a follow-up SBP configuration. Consequently, an SBP system can be started with an initial SBP configuration c^0 whose active processes will trigger a sequence of SBP configurations $c^1, c^2, \ldots$ Note that we do not make assumptions whether time is continuous or discrete.

Using names (for worlds, entities, and properties) allows us to speak about concepts that change over time. E.g., if e_0 is an entity $e_0 = \langle D, T, P\rangle$ in some world at time 0 and some data is added to D for time 1 then, technically, this results in another entity $e_1 = \langle D', T, P\rangle$. As our intention is to consider e_1 an updated version of e_0 we use the same names for both, i.e., $n_{e_0} = n_{e_1}$. In the subsequent work we will sometimes refer to concepts by their names and use a path-like notation using the .-operator for referring to SBP concepts, e.g., we refer to an the entity e in a world w by $n_w.n_e$.

3 Modelling AI Applications in SBP

Next, we sketch how to model different scenarios in SBP. Note that we use high-level pseudo code for expressing transition descriptions and emphasise that in an implementation we suggest to use different high-level programming languages tailored to the specific needs of the task handled by the transition description.

In contrast to objects as in object-oriented programming (OOP) entities are not grouped in a hierarchy of classes. Classes are a valuable tool when clear structures and rigorously defined behaviour are required. However, they also impose a rigid corset on their instances: data and behaviour of objects is limited to what is pre-defined in their class. In the scenarios we target, the nature of entities may change and the focus is on emerging rather than predictable behaviour. E.g., a town may become a city and a caterpillar a butterfly, etc., or, in fictional settings (think of a computer game) a stone could turn into a creature or vice versa. We want to directly support metamorphoses of this kind, letting entities transform completely over time regarding their data as well as their behaviour. Instead of using predefined classes, type membership is expressed by means of properties in SBP, e.g., each entity n_e may have a data entry n_e.*types* that contains a list of types that n_e currently belongs to.

▶ **Example 2.** We deal with a scenario of a two-dimensional area, represented by a single SBP world w, where each entity may have a property *loc* with values of form $\langle X, Y\rangle$ determining the coordinates of the location of the entity. The area is full of chickens running around, each of which is represented by an entity. In the beginning, every chicken *ch* has a transition description *ch*.*mvRand* that allows the chicken to move around with the pseudo code:

```
wait(randomValue(1..5000))
switch{randomValue(1..4)}
 case 1: return {'mv_up'}; case 2: return {'mv_right'};
 case 3: return {'mv_down'}; case 4: return {'mv_left'}
```

The transition first waits for a random amount of time, and randomly chooses a direction for the move, represented by the updates *mv_up*, *mv_right*, $\dots \subseteq \Upsilon$. The semantics of *mvRand* always returns $\langle U, \mathit{true}\rangle$, where U contains the update (the direction to move) and *true* indicates that after the end of the iteration there should be another. When the update function υ is called with one of the *mv* updates it changes the value of *ch.loc*, e.g., if *ch.loc* has value $\langle 3, 5\rangle$ at time $\mathtt{t}$ and the update is *mv_left* then *ch.loc* has value $\langle 2, 5\rangle$ at $\mathtt{t} + 1$.

Besides chickens, the area is sparsely strewn with corn. Corn does not move but it is eaten by chickens. Hence, each chicken *ch* has another transition description *eat*:

```
if there is some entity en in myworld with en.loc = my.loc and
  en.types contains 'corn' then return {'eatCorn(en)'}
```

Using the keyword `my` we can refer to properties of the entity to which the transition description belongs (*ch* in this case). Furthermore, `myworld` refers to the world in which this entity appears. Also here, every iteration of *eat* starts another one. For an update *eatCorn*(*en*), the location entry *en.loc* of the corn is deleted and the corn entity is notified about being eaten by setting the data entry *en.eatenBy* to *ch* (we need the information which chicken ate the corn later) and adding a process to the corn entity with the transition description *en.beenEaten*. For now we assume that it only causes the corn to delete itself.

For an initial SBP configuration $c^0 = \langle w\rangle$, where every chicken entity in w has an active process named *move* for transition description *mvRand* and a process for *eat*, an SBP run simulates chickens that walk around randomly and eat corn on their way.

While Example 2 illustrates how data is changed over time and new processes can be started by means of updates, the next example enriches the scenario with functionality for learning new behaviour.

▶ **Example 3.** We extend the scenario of Example 2 to a fairy tale setting by assuming that among all the corn entities, there is one dedicated corn named *cornOfWisdom* that can make chickens smarter if they eat it. It has a transition description *cornOfWisdom.moveSmart*:

```
wait(randomValue(1..1000))
en is an entity in myworld where en.types contains 'corn' and  there is no
 other entity en' in myworld where en'.types contains 'corn'  and
 distance(en'.loc,my.loc) < distance(en.loc,my.loc)
let my.loc=(myX,myY) and en.loc=(otherX,otherY)
if |otherX - myX| > |otherY - myY| then
  if otherX - myX > 0 then return {'mv_right'} else return {'mv_left'}
 else
  if otherY - myY > 0 then return {'mv_down'} else return {'mv_up'}
```

Intuitively, this transition causes an entity to move towards the closest corn rather than walking randomly as in *mvRand*. Another difference is that *moveSmart* processes have shorter iterations on average as the range of the random duration to wait is smaller. The *cornOfWisdom* does not have active processes for this transition definition itself but can pass it on to everyone who eats it. This is defined in the transition *cornOfWisdom.beenEaten* that differs from the *beenEaten* transition description of other corn by not only deleting the corn but also copying the *moveSmart* transition description from the *cornOfWisdom* to the chicken by which it was eaten. It also changes the *move* process of the chicken to use its new *moveSmart* transition description instead of *mvRand*. Thus, if a chicken happens to eat the *cornOfWisdom* it subsequently has a better than random strategy to catch some corn.

Having means to replace individual behavioural patterns, as in the example allows for modelling evolutionary processes in an easy way. E.g., if the chicken scenario is modified in a way that chicken which do not eat corn regularly will die, a chicken that ate the *cornOfWisdom* has good chances to survive for a long period of time. Further processes could allow chickens to reproduce when they meet such that baby chicken may inherit which transition description to use for moving from one of the parents. Then, most likely, chicken using *moveSmart* will be predominant soon.

Example 3 also illustrated that inheritance in SBP works on the individual level: entities can pass their transition descriptions and data entries to fellow entities. Thus, compared to OOP, inheritance is not organised in a hierarchical way. The underlying motivation is to follow the main idea of simulating real world objects, taking the stance that entities in nature are individuals that are not structured into distinct classes per se.

The next example illustrates the use of worlds for hypothetical reasoning.

▶ **Example 4.** Entity *barker* represents a waiter of a restaurant in a tourist area trying to talk people on the street into having dinner in his restaurant. To this end, *barker* guesses what food they could like and makes offers accordingly. We assume an SBP configuration in which for every entity h that represents a human there is a world w_h that models the view of the world of this person. The transition description *barker.watchPeople* allows *barker* to set the eating habits of passer-by in his world w_{barker} using country stereotypes:

```
wait(randomValue(50))
let en be an entity in myworld where en.loc near my.loc,
  en.types contains ’human’, and en.eatingHabits = unknown
 country = guess most likely home country of en
 prototype = myworld.country.inhPrototype
 return {’setEatingHabits(en,propotype)’,’setPotentialCustomer(en)’}
```

For every country w_{barker} contains a reference *inhPrototype* to an entity representing a typical person from this country. The update *setEatingHabits*($p1, p2$) copies transition descriptions and data properties related to food from person $p2$ to person $p1$. The update *setPotentialCustomer*(p) lets *barker* consider entity p to be a potential customer. In order to choose what to offer a potential customer, the waiter thinks about what kind of food the person would choose (based on his stereotypes). This is modelled by transition *barker.makeOffer*:

```
let cus be a potential customer in myworld
w’ = copy of myworld
w’.cus.availableFood = restaurant.availableFood
w’.cus.hungry = true
intermediate return {addWorld(w’), startProcess(w’.cus.startDinner)}
when process w’.cus.foodSelected is finished
 food = w’.cus.selectedFood
 return {praiseFood(food), deleteWorld(w’)}
```

To allow for hypothetical reasoning by the waiter, a temporary copy w' of world w_{barker} is created. The sole purpose of w' is to simulate the customer dining. We need a temporary world as we use the same transition descriptions that drive the overall simulation. E.g., if we would use w_{barker} instead, it would mean that *barker* thinks that the customer is actually dining, using the world of the customer would mean that she thinks she is dining and so on.

After creating w', the transition description defines that the food available to the version of the customer in w' is exactly the food that is on the menu of the restaurant and the customer is set to be hungry in the imagination of the waiter. Then, w' is added to the SBP configuration and a process for $w'.cus$ is started using the transition description *startDinner* that lets enitity *cus* start dining in w'. Note that the keyword `intermediate return` in the code is a convenience notation that allows for manipulating the SBP configuration during a transition which is strictly speaking not allowed in our framework. The same behaviour could be accomplished in a complient way by splitting *makeOffer* into two transition descriptions used in an alternating scheme. As soon as the customer chooses some food in the simulation, transition *barker.makeOffer* is notified which then reads which food has been chosen in the hypothetical setting. Finally, the update *praiseFood*(*food*) causes the waiter to make an offer for the chosen food, whereas *deleteWorld*(w') deletes the temporary world.

Note that copying worlds as done in Example 4 does not necessarily imply copying all of the resources in this world within an implementation of an SBP runtime engine (cf. Section 5).

▶ **Example 5.** We continue Example 4 by assuming an SBP configuration where a tourist, *Ada*, passes by the waiter. His process for transition *barker.watchPeople* classifies Ada by her looks to be an Englishwoman. After that, the process for *barker.makeOffer* starts hypothetical reasoning about Ada having dinner. Following the stereotypes of *barker* about English eating habits, the process reveals that *Ada* would go for blood pudding which he offers her subsequently. However, Ada is not interested in this dish as she is vegetarian. She explains her eating habits to the waiter, modelled by the transition description *ada.explainEatingHabits*:

```
let pers be current discussion partner in myworld
if pers offers food containing meat then
 let w_pers be the world of pers
 return {'setEatingHabits(w_pers.me, myworld.me)'}
```

Here, update *setEatingHabits* that we know from Example 4 overwrites food related properties of the entity representing Ada in the world of *barker* with her actual eating habits. If *barker* repeats the dining simulation for making another offer the result matches her real choices.

The last two examples showed how worlds can be used to express different modalities like individual points of views or hypothetical scenarios. Next, we sketch a setting where different worlds represent the same situation at different granularities.

▶ **Example 6.** Consider a computer game in which the player controls a character in an environment over which different villages are distributed. Whenever the character is close to or in a village the inhabitants of the village should be simulated following their daily routines and interacting with the player. However, as the game environment is huge, simulating all inhabitants of each village at all times is too costly. The problem can be addressed by an SBP configuration with two worlds, $w(v)_{act}$ and $w(v)_{apx}$, for each village v. Intuitively, $w(v)_{act}$ simulates the village and its people in detail but has only active processes while the player is closeby. The world $w(v)_{apx}$ approximates the behaviour of the whole village, e.g., increasing or shrinking of the population, economic output and input, or relations to neighbour villages, based on statistics and it has only active processes when the player is not around. When the player enters a village, a process is started that synchronises the world $w(v)_{act}$ with the current state of the village in $w(v)_{apx}$, e.g., by deleting or adding new inhabitants or shops. It also starts processes in $w(v)_{act}$ and cancels processes in $w(v)_{apx}$. If the player leaves again, another type of process performs an opposite switch from $w(v)_{apx}$ to $w(v)_{apx}$ being active.

4 Discussion And Related Work

For handling interaction of an SBP system with the user or another system, we suggest to use externally controlled processes: A transition description can use a dedicated 'external semantics' where the result returned for every transition is provided externally. By deciding which parts are controlled externally on the level of transition descriptions, the behaviour of an entity can be partially controlled by the user and partially by the system. This way one could replace, e.g., a human player in a computer game by an AI player or vice versa.

A natural question is how concurrent access on data is handled in the approach. Conceptionally, conflicting updates that occur at the same time instant are managed by the update function υ. In practice, however, this function has to be implemented and conflicts have to be addressed. Here, techniques for concurrency control in databases [1] could be useful. Besides technically conflicting updates on data, another issue is semantical inconsistency, i.e., data whose meaning with respect to the modelled problem domain is conflicting. Consider an SBP configuration modelling a banana and two monkeys where each monkey has a transition description for grabbing the banana. Suppose that both monkeys detect the banana their grabbing processes start. Then, it could happen that the system gets into a state where each monkey is believed to have the banana. It is probably sensible to tackle consistency problems of this kind on the level of modelling rather than by the underlying computational framework as there are many types these issues that must be addressed differently and also in the real world two monkeys could believe that they succeeded in getting a banana for a moment.

One goal of SBP is integrating different AI techniques where a main mechanism of integration is the explicit use of existing AI formalisms (e.g., ASP, planning techniques, etc.) for solving emergent subproblems by means of transitions descriptions. This is related to the use of different contexts in recent reactive forms of heterogenous multi-context systems [2, 4].

Evolutionary processes are intrinsic to many types of simulation. In, genetic algorithms [5, 9] the fitness of individuals is typically rated by a dedicated fitness function, whereas the most obvious approach in SBP is simulating natural selection by competition in the simulated environment, as discussed for the chicken scenario after Example 3. Evolving behaviour is related to genetic programming [6] where programs are shaped by evolutionary processes.

Agent-based systems (ABS) [10, 14] share several aspects with SBP like the significance of emerging behaviour when entities are viewed as agents. This view, however, is not adequate for all types of entities, e.g., when they represent objects like stones, collections of entities, or intangible concepts like 'the right to vote' which should not be seen as agents. Moreover, agents interact via explicit acts of communication that can but need not be modelled in an SBP configuration. Thus, we see SBP conceptionally one level below ABS, i.e., SBP languages can be used for implementing ABSs rather than being ABSs themselves. Nonetheless, many techniques from ABS literature could be beneficial for future SBP systems.

The idea of using multiple worlds for different points of view and modalities is loosely related to the possible world semantics of modal logics [7].

5 Conclusion

We proposed initial ideas on an approach for using simulation as a programming paradigm. Its cornerstones are 1. typeless entities, 2. different worlds for different views on reality, 3. behaviour defined by heterogenous concurrent services, and 4. exchange of behavioural patterns and individual inheritance. Clearly, there are many important aspects that were not discussed here that need to be addressed when putting SBP in practice. Thus, a major goal is a prototype of an SBP engine which opens a wide range for further work: E.g., on how to manage resources in SBP systems in which many worlds and entities share slightly different data and processes. Efficiency requirements could necessitate mechanisms for sharing resources, e.g., by only keeping track of differences when a world or entity is cloned.

References

1 Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

2 Gerhard Brewka, Stefan Ellmauthaler, and Jörg Pührer. Multi-context Systems for Reactive Reasoning in Dynamic Environments. In *Proceedings of the 21st European Conference on Artificial Intelligence* (*ECAI 2014*), pages 159–164, 2014.

3 Lawrence J. Fogel, Alvin J. Owens, and Michael J. Walsh. *Artificial Intelligence through Simulated Evolution.* Wiley, Chichester, WS, UK, 1966.

4 Ricardo Gonçalves, Matthias Knorr, and João Leite. Evolving Multi-context Systems. In *Proceedings of the 21st European Conference on Artificial Intelligence* (*ECAI 2014*), pages 375–380, 2014.

5 John H. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, 1975.

6 John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

7 Saul Kripke. A Completeness Theorem in Modal Logic. *Symbolic Logic*, 24(1):1–14, 1959.

8 Victor W. Marek and Mirosław Truszczyński. Stable Models and an Alternative Logic Programming Paradigm. In Krzysztof R. Apt, Victor W. Marek, Mirosław Truszczyński, and David S. Warren, editors, *In The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer, 1999.

9 Melanie Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, 1998.

10 Muaz Niazi and Amir Hussain. Agent-based Computing from Multi-agent Systems to Agent-based Models: a visual survey. *Scientometrics*, 89(2):479–499, 2011.

11 Ilkka Niemelä. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.

12 Jörg Pührer. Towards a Simulation-based Programming Paradigm for AI applications. In *Proceedings of the International Workshop on Reactive Concepts in Knowledge Representation (ReactKnow'14)*, pages 55–61, 2014.

13 Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.).* Pearson Education, 2010.

14 Michael J. Wooldridge. *An Introduction to MultiAgent Systems (2. ed.).* Wiley, 2009.

Defining and Evaluating Learner Experience for Social Adaptive E-Learning

Lei Shi

Department of Computer Science, University of Warwick, Coventry, CV4 7AL, UK
lei.shi@dcs.warwick.ac.uk

Abstract

Social adaptive e-learning combines, threads and balances the amount of social and adaptive features for e-learning in order to achieve high-quality Learner eXperience (LX). Evaluating a social adaptive e-learning system is a difficult task due to its complexity. It is crucial to ensure that appropriate evaluation methods and measures are used. A User-centric approach serves the empirical system evaluation using subjective user feedback on satisfaction and productivity as well as the quality of work and support, so as to verify the quality of product, detect problems, and support decisions. This paper proposes a learner-centric evaluation framework, which applies a user-centric approach, aiming to evaluate LX in social adaptive e-learning from the end-user (learner) point of view, taking into consideration both social and adaptive perspectives.

1998 ACM Subject Classification H.5.2 [Information interfaces and presentation]: User Interfaces - Evaluation/methodology

Keywords and phrases Social adaptive e-learning, user-centric evaluation, learner experience

Digital Object Identifier 10.4230/OASIcs.ICCSW.2014.74

1 Introduction

Social adaptive e-learning [31] combines, threads and balances the amount of social and adaptive features for e-learning in order to achieve high-quality overall learner experience. With social features, an e-learning system connects like-minded learners, so they can achieve learning goals via communication and interaction with each other by sharing knowledge, skills, abilities and materials. With adaptive features, an e-learning system delivers learning contents to learners adaptively, namely, the appropriate contents are delivered to the learners in an appropriate way at an appropriate time based on the learners' needs, knowledge, preferences and other characteristics [32].

Evaluating a social adaptive e-learning system is a difficult task due to its complexity. It is crucial to ensure that appropriate evaluation methods and measures are used. A variety of evaluation methodologies and techniques for adaptive e-learning systems have been proposed. Apart from the traditional prediction accuracy evaluation [15, 20], researchers have reached the agreement on the importance of introducing additional criteria that investigate user experience issues [21] such as trustability [10], enjoyability [13], coverage and serendipity [4], as user satisfaction of a system does not necessarily correlate with the high prediction accuracy. Additionally, since a social dimension has been introduced into adaptive e-learning [3, 29], some evaluation frameworks for social adaptive e-learning have been developed such as [30], which takes into account also social metrics.

Based on existent studies, the study presented in this paper applies a user-centric (learner-centric) approach, aiming to evaluate Learner eXperience (LX) of social adaptive e-learning, from the end-user (learner) point of view, taking into consideration both social and adaptive perspectives. We firstly, in Section 2, review related works including the user-centric

Imperial College Computing Student Workshop (ICCSW'14).
Editors: Rumyana Neykova and Nicholas Ng; pp. 74–82
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

evaluation methodology and existent evaluation frameworks. To propose the learner-centric evaluation framework, we look into general definitions of User eXperience (UX) in Section 3, to simulate the definition of LX and design the evaluation metrics. Section 4 reports a case study using the framework, and finally, Section 5 draws conclusions and future works.

2 Related Work

User-centric evaluation (UCE), as one of the best accepted approaches that identify determinants of User eXperience (UX) issues, serves the empirical system evaluation using subjective user feedback on satisfaction and productivity as well as the quality of work and support [12], aiming at verifying the quality of product, detecting problems and supporting decisions [8]. The nature of UCE makes itself a valuable approach to improve a system and help researchers and engineers implement easier to use and more enjoyable UX, and thus eventually lead to a higher adoption of the system [33].

However, unified frameworks have always been needed to compare the results of different studies. Several recent user-centric evaluation frameworks are proposed to address this issue. Chrysafiadi and Virvou proposed an evaluation framework, called PeRSIVA [7], using Kirkpatrick's model [16] in combination with a layered evaluation approach to access learners? satisfaction, performance, progress, behaviour and states. Knijnenburg, et al. proposed a pragmatic procedure that centralised UX including users? perceptions of a system (perceived system effectiveness and fun), system usage (usage effort and choice difficulty), and outcome of system usage (satisfaction with the chosen items) [17].

In this paper, based on the definition of Learner eXperience (LX) (Section 3) and existent evaluation frameworks and methods including [2, 7, 16, 17, 23, 34], a new learner-centric evaluation framework for social adaptive e-learning is proposed. This framework aims at evaluating LX from the end-user (learner) point of view taking into consideration both social and adaptive perspectives. It also aims at allowing for comparisons of evaluation results between different social adaptive e-learning systems.

3 Definition and Evaluation of Learner Experience

User eXperience (UX) started to attract designers? attention since the early 1990s, when it was first coined and brought to wider knowledge by Donald Norman. UX is associated with a wide range of meanings [11], ranging from classical usability to fuzzy and dynamic concepts such as emotional, affective, experiential, hedonic and aesthetic variables [18], which causes itself an elusive notion with many different definitions. Although UX is well discussed in conferences and symposiums, till now there is still no widely share view of its definition. The following are the definitions made by different organisations, researchers and designers ? just to name a few.

Alben [1] defines UX as *all the aspects of how people use an interactive product: the way it feels in their hands, how well they understand how it works, how they feel about it while they are using it, how well it serves their purposes, and how well it fits into the entire context in which they are using it.*

Hassenzahl and Tractinsky [14] define UX as *a consequence of user's internal state such as predispositions, expectations, needs, motivation, mood, the characteristics of the designed system such as complexity, purpose, usability, functionality, and the context (or the environment) within which the interaction occurs such as social setting, meaningfulness of the activity, voluntariness of use.*

ISO 9241-210 [9] defines UX as *a person's perceptions and responses that result from the use and/or anticipated use of a product, system or service, including all the users' emotions, believes, preferences, perceptions, physical and psychological responses, behaviours and accomplishments that occur before, during and after use, influenced by three factors, i.e., system, user and the context of use.*

As UX includes a lot of dynamic concepts such as emotional, affective, experiential, hedonic and aesthetic variables, and this is still no widely shared view of its definition, it is even harder to define criteria against which it could be evaluated [24]. The metrics settings for the UX evaluation could be various, depending on the perspectives of both system aims and user needs. For example, for gaming systems, UX metrics may concern the more about the fun and enjoyable experience; for banking systems, UX metrics may concern the more about the secure and trustworthy experience; for online shopping systems, UX metrics may concern the more about customers' satisfaction on the payment process; and for e-learning systems, UX metrics may concern the more about learners' engagement of accessing learning materials.

Researchers have been proposing, summarising and classifying the UX metrics (or metrics, criteria, frameworks) in the literature. Van Velsen et al. [33] represented UX metrics such as appreciation, user satisfaction, usability, user performance, intention to use, appropriateness of adaptation, comprehensibility. Morville [22] proposed UX criteria to evaluate if the system is useful, usable, findable, credible, accessible, desirable and valuable. Wu et al.'s UX evaluation framework [34] includes six constructs: flow, perceived technology acceptance, telepresence, performance gains, technology adoption, and exploratory behaviours.

When evaluating UX of more specific types of systems, the metrics are usually tailored to suit the system types' aims, either by specifying the UX metrics, or introducing some other metrics as the complements.

In the adaptive systems and recommender systems area, several elaborate user-centric evaluation frameworks have been proposed. For example, Pu et al. proposed ResQue [23], which defines a set of metrics that are grouped into four high level layers: *perceived system qualities*, *users' beliefs*, *subjective attitudes*, and *behavioural intentions*. For each metrics, ResQue also specifies several questions to ask users. Knijnenburg et al.'s framework consists of five dimensions to evaluate including *objective system aspects*, *subjective system aspects*, *user experience*, *interaction*, and *personal and situational characteristics*.

In the e-learning area, Ardito et al. [2] proposed four dimensions for the evaluation including *presentation*, *hypermediality*, *application proactivity*, and *user activity*. And for each of them, both effectiveness and efficiency are considered as the general principles. Liaw and Huang [19] examined the relationships between *perceived self-efficacy*, *perceived anxiety*, *interactive learning environments*, *perceived satisfaction*, *perceived usefulness*, and *perceived self-regulation*, and proposed *perceived satisfaction*, *perceived usefulness*, and *interactive learning environments* as predictors to self-regulation in e-learning environments.

These definitions and metrics indicate that to evaluate social adaptive e-learning systems it is crucial to understand the changes that have occurred due to the merging of techniques into the innovation. Taking into consideration their potential influence on the learning process, it is crucial to characterise learner experience and define the metrics of its evaluation.

In this study Learner eXperience (LX) is defined as *a learner's perceptions and responses resulting from the use and/or anticipated use of an e-learning system* - as a simulation of UX, whereas more extensive and going beyond the traditional study of skills and cognitive process of users and their behaviours when interacting with an e-learning system. It involves learners' behaviours, attitudes, believes, sensation, assessment, emotional response obtained and so on throughout the entire time of using an e-learning system. Based on the discussed evaluation frameworks and metrics, the following learner-centric scales are developed. Each of them has several statements in a five-point Likert scale ranging from -2 (strongly disagree) to 2 (strongly agree), except System Usability Scale (SUS) whose scale ranges from 1 to 5.

The Learner Belief Scale (LBS) is designed to test the impact that the learning system had on the overall LX as perceived by learners. It also evaluates the user-system interaction considered as helpful in learning as perceived by learners. It consists of ten statements:

LBS.01 The system helped me to learn more topics.
LBS.02 The system helped me to learn more profoundly.
LBS.03 The system helped me to identify my weak points.
LBS.04 The system helped me to plan my classwork.
LBS.05 The system increased my learning interests.
LBS.06 The system increased my learning confidence.
LBS.07 The system increased my learning outcome.
LBS.08 It was easy to use the system.
LBS.09 It was easy to learn how to use the system.
LBS.10 It was easy to remember how to use the system.

The User Interface Scale (UIS) focuses on the evaluation of learners' overall feelings towards a e-learning system's user interface. The eight UIS statements are:

UIS.01 The user interface is familiar to me.
UIS.02 The user interface is consistent.
UIS.03 The user interface is understandable.
UIS.04 The user interface is enjoyable.
UIS.05 The user interface is attractive.
UIS.06 The user interface is interactive.
UIS.07 The user interface is personalised.
UIS.08 The user interface is efficient.

The Content Quality Scale (CQS) is developed to evaluate learners' opinion on the content provided (recommended) by the e-learning system. Its statements include:

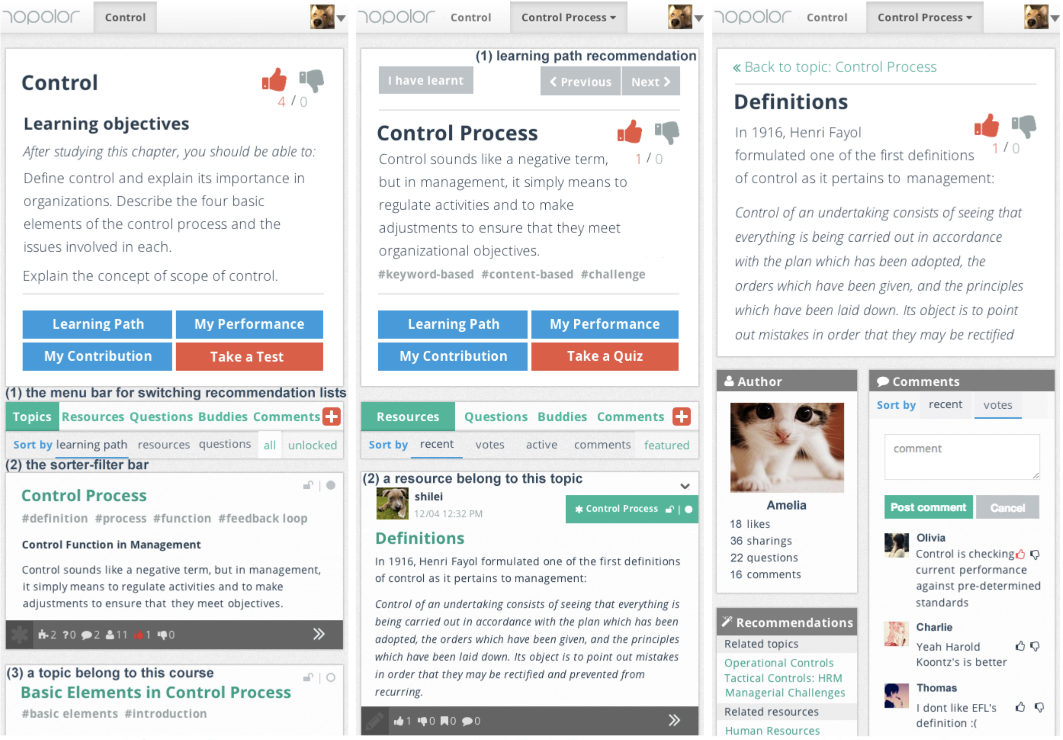
CQS.01 The content recommended by the system was what I wanted to learn.
CQS.02 The content recommended by the system was what I needed to learn.
CQS.03 It was easy to recognise the content recommended by the system.
CQS.04 I was confident that I would like the content recommended to me.
CQS.05 I understood why the system recommended certain content to me.
CQS.06 I understood how the content is organised in the system.
CQS.07 The content provided by the system was understandable.
CQS.08 The content provided by the system was sufficient.
CQS.09 The content provided by the system was useful.

The Socialisation Quality Scale (SQS) is developed to evaluate students' opinion on the e-learning system's social interaction features. The SQS statements are:

SQS.01 It was easy to discuss with the peers.
SQS.02 It was easy to share content with peers.
SQS.03 It was easy to access the content shared by peers.
SQS.04 It was easy to tell peers what I liked/disliked.
SQS.05 The statistic numbers (mine and peers') engaged me to learn more.
SQS.06 The system helped me engaged in interacting with peers.

The Behavioural Intention Scale (BIS) is designed to evaluate if the e-learning system could influence learners' decision to use and reuse the system, as well as to introduce the system to their friends, in order to understand learners' loyalty. The BIS statements are:
BIS.01 I will use the system again.
BIS.02 I will use the system frequently.
BIS.03 I will tell my friends about the system.
The Perceived Motivation Scale (PMS) is designed based on the Self-Determination Theory (STD) [25], to exam learners' feelings on their autonomy (question 1-4), competence (question 5-8) and relatedness (question 9-12). The PMS statements are:
PMS.01 I felt in control of my learning process.
PMS.02 I felt interested in using the system.
PMS.03 I felt confident to use the system.
PMS.04 I felt my learning experience was personalised.
PMS.05 I felt having fun when using the system.
PMS.06 I felt I only needed a few steps to complete tasks.
PMS.07 It was easy to understand why I received recommendations.
PMS.08 It was easy to find the content I need.
PMS.09 It was easy to share content with peers.
PMS.10 It was easy to access the shared resources from peers.
PMS.11 It was easy to tell peers what I like/dislike.
PMS.12 It was easy to discuss with peers.
The System Usability Scale (SUS) [6] is a simple, ten-item Likert scale giving a global view of subjective assessments of system usability. It was developed by Brooke in 1996 as a 'quick and dirty' questionnaire scale of a given product or service. Its merits make is widely accepted and used in both industry and academic: first, the SUS questionnaire is non-proprietary, making it a cost effective tool; second, the SUS is technology agnostic, and thus flexible enough to evaluate various products and services including hardware, software, websites and applications; third, the SUS questionnaire is relatively quick and easy to use by both experimental subjects and researchers; fourth, the SUS questionnaire provides a single score on a scale, which is easy to understand [37]. The ten items (statements) in the SUS questionnaire are as follows:
SUS.01 I think that I would like to use this system frequently.
SUS.02 I found the system unnecessarily complex.
SUS.03 I thought the system was easy to use.
SUS.04 I think I would need support of a technical person to be able to use this system.
SUS.05 I found the various functions in this system were well integrated.
SUS.06 I thought there was too much inconsistency in this system.
SUS.07 I would imagine that most people would learn to use this system very quickly.
SUS.08 I found the system very cumbersome to use.
SUS.09 I felt very confident using the system.
SUS.10 I needed to learn a lot of things before I could get going with this system.

The SUS questions above are all scored on a five-point Likert scale of agreement strength - from strongly disagree to strongly agree. They are alternately positive and negative, so a better system should have higher scores for question 1, 3, 5, 7 and 9, and lower scores for question 2, 4, 6, 8 and 10. The SUS final score is calculated using Equation 1, where U_i presents the score of the i-th question. It ranges from 0 to 100, where the higher the final score the better the usability. However, normally a 'good system' should have a score between 70 and 80, and an 'exceptional system' should have a score greater than 90 [5].



Control

Learning objectives

After studying this chapter, you should be able to:

Define control and explain its importance in organizations. Describe the four basic elements of the control process and the issues involved in each.

Explain the concept of scope of control.

(1) the menu bar for switching recommendation lists

(2) the sorter-filter bar

Control Process

#definition #process #function #feedback loop

Control Function in Management

Control sounds like a negative term, but in management, it simply means to regulate activities and to make adjustments to ensure that they meet objectives.

2 ?0 2 11 1 0

(3) a topic belong to this course

Basic Elements in Control Process

#basic elements #introduction

(a) A Course Page

(1) learning path recommendation

Control Process

Control sounds like a negative term, but in management, it simply means to regulate activities and to make adjustments to ensure that they meet organizational objectives.

#keyword-based #content-based #challenge

(2) a resource belong to this topic

shilei

12/04 12:32 PM

Control Process

Definitions

In 1916, Henri Fayol formulated one of the first definitions of control as it pertains to management:

Control of an undertaking consists of seeing that everything is being carried out in accordance with the plan which has been adopted, the orders which have been given, and the principles which have been laid down. Its object is to point out mistakes in order that they may be rectified and prevented from recurring.

1 0 0 0

(b) A Topic Page

Definitions

In 1916, Henri Fayol formulated one of the first definitions of control as it pertains to management:

Control of an undertaking consists of seeing that everything is being carried out in accordance with the plan which has been adopted, the orders which have been given, and the principles which have been laid down. Its object is to point out mistakes in order that they may be rectified

Amelia

18 likes

36 sharings

22 questions

16 comments

Related topics

Operational Controls

Tactical Controls: HRM

Managerial Challenges

Related resources

Human Resources

Olivia

Control is checking current performance against pre-determined standards

Charlie

Yeah Harold Koontz's is better

Thomas

I dont like EFL's definition :(

(c) A Resource Page

Figure 1 User Interface of Topolor (on smart phone).

In this study, the SUS questionnaire is used as a 'highly established' measure to evaluate the 'overall usability' of the system. It also contributes to the criterion validity [26] - comparison with other evaluation results using the measures defined in this research.

$$SUS_{SCORE} = 2.5 \times \left[\sum_{n=1,i=2n-1}^{n=5} (U_i - 1) + \sum_{n=1,i=2n}^{n=5} (5 - U_i) \right] \tag{1}$$

4 Case Study

A case study was conducted to demonstrate the usage of the proposed learner-centric evaluation framework. In order to collect data, an experiment was carried out at the Department of Economics, Sarajevo School of Science and Technology, Bosnia and Herzegovina, in December 2013. Twenty students, two observers and one course instructor participated in the one and a half hours online learning session - using Topolor (a social adaptive e-learning system [27]; its user interface is shown as Figure 1) to learn a course on 'Control'. After the initial online session, students were encouraged to further use Topolor to revise the covered materials, for two weeks. After that, students were asked to complete an optional online survey. Out of the twenty students who participated in the online course, fifteen completed the online survey. The results of the survey are shown in Table 1.

The mean values rank between 0.53 and 1.60, with standard deviations ranging from 0.35 to 0.74. All the reported mean values are larger than 0 (the neutral response), suggesting students' attitudes to be generally positive. Additionally, the results' *Cronbach's alpha* is

Table 1 Results of LBS, UIS, CQS, BIS and PMS (μ: mean value; σ: standard deviation).

	μ	σ		μ	σ		μ	σ		μ	σ
LBS.01	1.27	0.46	UIS.03	1.00	0.38	CQS.07	1.27	0.46	PMS.01	1.00	0.38
LBS.02	1.47	0.52	UIS.04	1.07	0.46	CQS.08	1.27	0.46	PMS.02	1.27	0.46
LBS.03	0.93	0.46	UIS.05	0.87	0.74	CQS.09	1.33	0.49	PMS.03	1.40	0.51
LBS.04	1.00	0.65	UIS.06	0.80	0.56	SQS.01	0.93	0.46	PMS.04	1.40	0.51
LBS.05	0.87	0.74	UIS.07	0.93	0.46	SQS.02	1.13	0.35	PMS.05	1.20	0.41
LBS.06	1.07	0.70	UIS.08	0.80	0.41	SQS.03	0.93	0.46	PMS.06	1.40	0.51
LBS.07	0.87	0.52	CQS.01	1.13	0.52	SQS.04	1.12	0.41	PMS.07	1.33	0.49
LBS.08	1.00	0.53	CQS.02	0.73	0.59	SQS.05	1.13	0.35	PMS.08	1.27	0.59
LBS.09	0.87	0.35	CQS.03	0.93	0.46	SQS.06	1.60	0.51	PMS.09	1.80	0.41
LBS.10	0.87	0.52	CQS.04	1.60	0.51	BIS.01	1.20	0.41	PMS.10	0.53	0.52
UIS.01	0.87	0.52	CQS.05	1.27	0.46	BIS.02	1.07	0.46	PMS.11	0.93	0.59
UIS.02	0.87	0.52	CQS.06	1.47	0.52	BIS.03	1.60	0.51	PMS.12	1.20	0.41

0.864 (>0.8), suggesting a high level of result reliability. The SUS score is 73.67 out of 100 (σ=9.01, median=72.50), indicating Topolor is a 'good system', according to Section 3.7.

5 Conclusion and Future Work

This study has proposed a new learner-centric framework for evaluating Learner eXperience (LX) in social adaptive e-learning. The new recommended definition of LX - *learner's perceptions and responses resulting from the use and/or anticipated use of an e-learning system*, is a simulation of User eXperience (UX) from Human-Computer Interaction (HCI) research area, which is more extensive and going beyond the traditional study of skills and cognitive process of users and their behaviours when interacting with an e-learning system. Seven learner-centric scales have been defined including *LBS (Learner Belief Scale)*, *UIS (User Interface Scale, introduced)*, *CQS (Content Quality Scale)*, *SQS (Socialisation Quality Scale)*, *BIS (Behavioural Intention Scale)*, *PMS (Perceived Motivation Scale)* and *SUS (System Usability Scale)*. They involve learners' behaviours, attitudes, believes, sensation, assessment, emotional response obtained and so on throughout the entire time of using an e-learning system.

Following the definition of LX and its metrics, a case study was presented to demonstrate the use of the proposed learner-centric evaluation framework. The evaluation results illustrated generally positive students' attitudes of using Topolor, a social adaptive e-learning system. The main limitation of the case study is the low number of participants, although *Cronbach's Alpha* suggests a high level of reliability of the results. However, Topolor has been opened to public (www.topolor.com), with larger student cohorts expected in the near future, allowing for feedback, use data and suggestions collecting, in further studies.

Whilst subjective and objective measures could be substitutes for each other, both empirical and analytical studies have suggested that the two types of measurement are equally important and thus both of them should be considered. Therefore, in the follow-up work, objective measures will be integrated into the proposed evaluation framework. Learners' usage data will be collected using data-logging techniques and analysed using various data mining and visualisation tools. In fact, this work has been already initialised, detailed in [28].

References

1 L. Alben Quality of Experience. *Interactions*, 3(3):11-15 (1996).

2 C. Ardito, M.F. Costabile, M. De Marsico, R. Lanzilotti, S. Levialdi, T. Roselli, and V. Rossano An approach to usability evaluation of e-learning applications. *Universal access in the information society*, 4(3):270-283, 2006.

3 A.I. Cristea, and F. Ghali Towards adaptation in e-learning 2.0. *New Review of Hypermedia and Multimedia*, 17(2):199-238, 2011.

4 M. Csikszentmihalyi Beyond boredom and anxiety *Jossey-Bas*, 2000

5 A. Bangor, P.T. Kortum, and J.T. Miller An empirical evaluation of the system usability scale. *International Journal of Human?Computer Interaction*, 24(6):574-594, 2008.

6 J. Brooke SUS-A quick and dirty usability scale. *Usability evaluation in industry*, 189:194, 1996.

7 K. Chrysafiadi, and M. Virvou PeRSIVA: An empirical evaluation method of a student model of an intelligent e-learning environment for computer programming. *Computers & Education*, 68:322-333, 2013.

8 M. DeJong, and P.J. Schellens Reader-Focused Text Evaluation An Overview of Goals and Methods. *Journal of Business and Technical Communication*, 11(4):402-432, 1997.

9 DIS, I9241-210: 2010 Ergonomics of human system interaction-Part 210: Human-centred design for interactive systems. *International Standardization Organization (ISO)*, 2009.

10 J. O?Donovan, and B. Smyth Trust in recommender systems. In *Proc. of the 10th international conference on Intelligent user interfaces*, pages 167-174, 2005.

11 J. Forlizzi, and K. Battarbee Understanding experience in interactive systems. In *Proc. of the 5th conference on Designing interactive systems: processes, practices, methods, and techniques*, pages 261-268, 2004.

12 J.D. Gould, and C. Lewis. Designing for usability: key principles and what designers think. *Communications of the ACM*, 28(3):300-311, 1985.

13 U. Gretzel, and D.R. Fesenmaier Persuasion in recommender systems. *International Journal of Electronic Commerce*, 11(2):81-100, 2006.

14 M. Hassenzahl, and N. Tractinsky User experience-a research agenda. *Behaviour & Information Technology*, 25(2):91-97, 2006.

15 J.L. Herlocker, J.A. Konstan, L.G. Terveen, and J.T. Riedl Evaluating collaborative filtering recommender systems. *ACM Trans. on Information Systems*, 22(1):5-53, 2004.

16 D.L. Kirkpatrick Techniques for Evaluating Training. *Training & Development Journal*, 33(6):78-92, 1979.

17 B.P. Knijnenburg, M.C. Willemsen, and A. Kobsa A pragmatic procedure to support the user-centric evaluation of recommender systems. In *Proc. of the fifth ACM conference on Recommender systems*, pages 321-324, 2011.

18 E. Law, V. Roto, A.P. Vermeeren, J. Kort, and M. Hassenzahl Towards a shared definition of user experience. In *CHI?08 extended abstracts on Human factors in computing systems*, pages 2395-2398, 2008.

19 S.-S. Liaw, and H.-M. Huang Perceived satisfaction, perceived usefulness and interactive learning environments as predictors to self-regulation in e-learning environments. *Computers & Education*, 60(1):14-24, 2013.

20 M.R. McLaughlin, and J.L. Herlocker A collaborative filtering algorithm and evaluation metric that accurately model the user experience. In *Proc. of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 329-336, 2004.

21 S.M. McNee, J. Riedl, and J.A. Konstan Being accurate is not enough: how accuracy metrics have hurt recommender systems. In *CHI'06 extended abstracts on Human factors in computing systems*, pages 1097-1101, 2006.

22 Peter User Experience Design *http://semanticstudios.com/publications/semantics/000029.php*

23 P. Pu, L. Chen, R. Hu A user-centric evaluation framework for recommender systems. In *Proc. of the 5th ACM conference on Recommender systems*, pages 157-164, 2011.

24 V. Roto, H. Rantavuo, and K. Vaananen-Vainio-Mattila Evaluating user experience of early product concepts. In *Proc. of the International Conference on Designing Pleasurable Products and Interfaces DPPI09*, pages 1-10, 2009.

25 R.M. Ryan, and E.L. Deci Self-determination theory and the facilitation of intrinsic motivation, social development, and well-being. *American psychologist*, 55(1):68, 2000.

26 W.R. Schumm, L.A. Paff-Bergen, R.C. Hatch, F.C. Obiorah, J.M. Copeland, L.D. Meens, and M.A. Bugaighis Concurrent and discriminant validity of the Kansas Marital Satisfaction Scale. *Journal of Marriage and the Family*, pages 381-387, 1986.

27 L. Shi, and A.I. Cristea Making It Game-Like: Topolor 2 and Gamified Social E-Learning. In *Proc. of the 22nd Conference on User Modeling, Adaptation and Personalization*, pages 61-64, 2014.

28 L. Shi, A.I. Cristea, M.S. Awan, C. Stewart, and M. Hendrix Towards understanding learning behavior patterns in social adaptive personalized e-learning systems. In *Proc. of the 19th Americas Conference on Information Systems*, pages 1-10, 2013.

29 L. Shi, D. Al-Qudah, and A.I. Cristea Social E-Learning in Topolor: A Case Study. In *Proc. of IADIS e-Learning conference*, pages 57-64, 2013.

30 L. Shi, M. Awan, and A.I. Cristea Evaluation of Social Personalized Adaptive E-Learning Environments: From End User Point of View. In *Proc. of the 3th Imperial College Computing Student Workshop*, pages 103-110, 2013.

31 L. Shi, D. Al-Qudah, and A.I. Cristea Designing social personalized adaptive e-learning. In *Proc. of the 18th ACM conference on Innovation and technology in computer science education*, pages 341-341, 2013.

32 L. Shi, A.I. Cristea, J.G.K. Foss, D. Al-Qudah, and A. Qaffas A social personalized adaptive e-learning environment: a case study in Topolor. *IADIS International Journal on WWW/Internet*, 11(13):13-34, 2013.

33 L. Van Velsen, T. Van Der Geest, R. Klaassen, and M. Steehouder User-centered evaluation of adaptive and adaptable systems: a literature review. *The knowledge engineering review*, 23(03):261-281, 2008.

34 W. Wu, A. Arefin, R. Rivas, K. Nahrstedt, R. Sheppard, and Z. Yang Quality of experience in distributed interactive multimedia environments: toward a theoretical framework. In *Proc. of the 17th ACM international conference on Multimedia*, pages 481-490, 2009.

On Recent Advances in Key Derivation via the Leftover Hash Lemma

Maciej Skorski

Cryptology and Data Security Group, University of Warsaw
maciej.skorski@gmail.com

Abstract

Barak et al. showed how to significantly reduce the entropy loss, which is necessary in general, in the use of the Leftover Hash Lemma (LHL) to derive a secure key for many important cryptographic applications. If one wants this key to be secure against any additional *short* leakage, then the min-entropy of the source used with the LHL must be appropriately bigger (roughly by the length of the supposed leakage). Recently, Berens came up with a notion of collision entropy that is much weaker than min-entropy and allows proving a version of the LHL with leakage robustness but without any entropy saving. We combine both approaches and extend the results of Barak et. al to Beren's collision entropy. Summarizing, we obtain a version of the LHL with optimized entropy loss, leakage robustness and weak entropy requirements.

1998 ACM Subject Classification F.1.3 Complexity Measures and Classes, K.6.5 Security and Protection

Keywords and phrases Key derivation, Leftover Hash Lemma, leakage resilient cryptography

Digital Object Identifier 10.4230/OASIcs.ICCSW.2014.83

1 Introduction

1.1 Randomness extractors

Analyzing security of cryptographic primitives one usually assumes an access to *perfect randomness*. However in practice we are limited to imperfect sources, which have a lot of randomness (measured by entropy[1]) but exhibit some patterns, bias or correlations between particular bits in generated sequences. As an entropy source one can use biometric data (like fingerprints), data collected from user-application interaction (mouse movements, typing the keyboard, timing events and others) and even physical sources (thermal noise, nuclear decay). Having collected "noisy" data with enough entropy, in the post-processing phase one "extracts" pure randomness by dedicated procedures called *randomness extractors*. One needs to stress that in general these procedures, in addition to a high-entropy source, require some amount of pure randomness used as a "catalyst"[2], referred to as the *seed*.

1.2 Leftover Hash Lemma

The famous Leftover Hash Lemma [4] states that universal hash functions are good extractors: if an n-bit source X has at least $\ell = k + 2\log(1/\epsilon)$ bits of min-entropy[3] then the distribution

[1] In information theory, the most popular notion is Shannon Entropy. In the context of extracting randomness one typically uses min-entropy or collision entropy.

[2] Since obtaining true random bits might be hard and expensive, a lot of research is focused on deterministic extractors, which don't need to be seeded. However, they work only for very limited class of sources.

[3] Which means that no adversary can guess the output of X with probability better than $2^{-\ell}$.

$H(X), H$, where H is a randomly chosen universal hash function from n to k bits (the seed), is ϵ-close to the k-bit uniform distribution, even if H is published.

Entropy loss and RT bound. Note that the Leftover Hash Lemma requires the significantly bigger entropy on its input comparing to what it extracts. More specifically, we sacrifice $L = 2\log(1/\epsilon)$ bits of entropy in order to obtain an output of quality ϵ. The result of Radhakrishnan and Ta-Shma [5], called the RT-bound, shows that this loss is necessary for any extractor. In this sense, the LHL as an extractor achieves the optimal entropy loss.

Importance of optimizing the LHL. Although the loss of $2\log(1/\epsilon)$ bits might seem to be negligible from an asymptotic point of view, however in some important applications it affects the efficiency (for instance the Diffie-Hellmann key exchange). Thus, minimizing it is important for efficiency. Similarly, shorter seeds than that one of the LHL are desired.

1.3 Key derivation: ideal and real settings

For any cryptographic primitive (like an encryption scheme or a signature), which uses randomness R to derive its secure key, we compare two different settings:

(a) ideal: R is *perfectly* uniform and independent of any attacker's side information Z

(b) real settings: the key owner has only an *imperfect* entropy source X and uses the key extracted (in our case: by hashing) from X as R. In addition, an attacker may have some *side information* Z correlated with X.

The security of the primitive is parametrized by ϵ, which is the success probability or the advantage of an attacker with certain resources. The LHL implies that if the security is ϵ for uniform R, then the same application keyed with a random hash of X is ϵ'-secure, where

$$\epsilon' \leqslant \epsilon + \sqrt{2^{-L}} \tag{1}$$

and the entropy loss L is the difference between the entropy of X given Z (suitably defined) and the length of the hashing output (the length of the extracted key).

1.4 Side leakage and chain rules

In the context of leakage, we want the guaranteed security not to degrade much when some extra but short information is revealed to an attacker. For an entropy notion $\mathbf{H}$ and two correlated random variables X, Z, the chain rule is an inequality of the following form

$$\mathbf{H}(X|Z) \geqslant \mathbf{H}(X) - C \cdot |Z| \tag{2}$$

where $|Z| = \log|\text{supp}(Z)|$ is the length of Z and C is some constant (ideally $C = 1$).

Applications of chain rules. Many information-theoretic notions of entropy satisfies the property (2). In fact, it is what one expects from a "good" notion of entropy and is often used in security proofs. Typically, high entropy corresponds to high security. Thus, Equation (2) is often used to prove security in the presence of leakage.

Example: Guessing Probability. The min-entropy of a random variable X given Z is defined as the negative logarithm of the maximal probability that an attacker can guess the output of X given only Z. It is known that it satisfies the inequality (2) with $C = 1$. In particular, if X is 128-bit uniform key, then any adversary given 10 bits of extra information, can guess X correctly with probability at most 2^{-118}, comparing to original 2^{-128}.

Example: leakage-robustness of the LHL. From Equation (1) it follows that to derive a secure key of required length we need to guarantee that $\mathbf{H}(X|Z)$, where $\mathbf{H}$ is the min-entropy, is big enough. Let us say that we have an application whose security for the uniform k-bit key is ϵ and that we want to derive a k-bit key which guarantees the security (with no extra side information) 2ϵ. For this we need $\mathbf{H}(X) \geqslant k + 2\log(1/\epsilon)$. Suppose now that we look at the security against *any* leakage Z of length $m = 20$. The chain rule for min-entropy (2) implies that the entropy loss $L = \mathbf{H}(X|Z) - k$ is *not smaller* than $\mathbf{H}(X) - m - k$. Combining this with (1) we see that the security is now $\epsilon + \epsilon \cdot 2^{m/2} \approx 2^{10}\epsilon$. Alternatively, we could start with $m = 20$ bits of entropy more and achieve the security 2ϵ. The chain rule is *necessary* to derive keys which remain secure against any bounded leakage.

1.5 Improvements in key derivation by the LHL

Reducing the entropy loss . The RT bound shows that it is impossible to avoid losing $2\log(1/\epsilon)$ bits of entropy if we want the extracted output to be ϵ-indistinguishable from uniform (i.e. ϵ-close) by *all* statistical tests. However, in cryptographic applications we are interested only in *very special* tests, corresponding to the definition of the security of the application. This suggests that one can overcome the RT bound in specific situations. Indeed, Barak et al. proved in [1] the stronger version of LHL, which essentially says that for many scenarios the “ideal” security ϵ and “real” security ϵ' are related by

$$\epsilon' \leqslant \epsilon + \sqrt{\epsilon 2^{-L}} \tag{3}$$

where L is the entropy loss. This shows that one obtains roughly the same level of security with L reduced by half up to $L = \log(1/\epsilon)$. The result is valid for the broad class of cryptographic applications, including unpredictability applications (for example, one-way functions) and some indistinguishability applications (the so called squared-friendly applications [1, 3], like weak pseudo-random functions or stateless chosen plaintext attack secure encryption).

Reducing the seed length. It is known that seed for LHL must grow linearly with the number of extracted bits [7]. In [1] the authors also observed that in some cases the length of the seed in the LHL can be reduced. The natural idea of expanding a shorter seed works either for small number of bits or in minicrypt [1].

Relaxing the entropy assumptions. The assumptions in the Leftover Hash Lemma are typically formulated in terms of min-entropy. However, it gives the same security guarantees when the upper bound on the probability of guessing X (which corresponds to min-entropy) is replaced by the same bound on the *collision probability* of X [4]. Since the collision probability is typically much bigger than the guessing probability, this means that from many sources we can *extract more* bits than it is guaranteed by the min-entropy bounds[5]. This result can be extended into the conditional case (when there is side information Z correlated with X):

(a) The generalized LHL of Barak et al. [1]. The min-entropy requirement can be replaced by the upper bound on the collision probability of X given Z, as observed by the authors. Unfortunately, the latter does not guarantee the leakage-robustness.

[4] This fact is actually intuitive, as the hash functions are, by definition, collision resistant and one can expect that the entropy notions based on the collision probability fit well to that settings

[5] Note that this does not contradict to the RT-bound, because that counterexample is a flat distribution for which the collision and guessing probability coincide (and thus, is a very special case)

Table 1 Improvements of the Leftover Hash Lemma

Statement	Standard			Generalized		
Assumptions	min-entropy	collision prob.	collision entropy	min-entropy	collision prob.	collision entropy
Reduced entropy loss	No	No	No	Yes	Yes	?
Reduced seed length	No	No	No	Yes	Yes	?
Side leakage robustness	Yes	No	Yes	Yes	No	?
Minimal entropy requirements	No	No	?	No	?	?

(b) The standard formulation of the LHL given for the appropriate notion of conditional collision entropy [2]. This notion has two big advantages: is the weakest among other notions proposed for the collision entropy (weaker than conditional collision probability!) and it satisfies the chain rule, guaranteeing leakage robustness.

The ideal version of the LHL - issues. On the one hand the generalized LHL allows reducing the entropy loss, the seed length and also handling side information, provided that we quantify randomness by min-entropy. On the other hand, we know how to minimize the entropy requirements for the standard LHL. This landscape is summarized in Table 1 below.

The discussion leads to the natural question about the existence of the "ideal" LHL:

> Question: *Does there exist a variant of the LHL which simultaneously captures the following advantages: reducing the entropy loss, reducing the seed length, leakage robustness and possibly minimal entropy assumptions?*

1.6 Our results

Summary. We answer this question affirmatively. As a first contribution we show that that the generalized LHL is not guaranteed to be leakage-robust with low-collision-probability sources. Second, we show that this can be fixed with a "correct" notion of collision entropy.

Conditional collision probability is not leakage-robust. We show (Section 4) that the generalized LHL [1] does not guarantee security for the key derived from a source of low collision probability against leakages of even one extra bit!

The generalized LHL works with the conditional collision entropy. We extend the generalized LHL [1] to work with Beren's entropy [2] (Section 5). It gives more security (because of the weaker entropy notion) and leakage robustness (because of the chain rule).

Applications - saving entropy. For a *low-collision-probability* source X, from which we want to derive a key ϵ-secure and secure against *arbitrary* one-bit leakage Z we save even $\log(1/\epsilon) - \mathcal{O}(1)$ bits of entropy comparing to what follows from [1]. Indeed, the only way to apply the statement of Barak et al. would be to convert first the "entropy" in X into min-entropy (because there is no chain rule for collision probability). By "entropy smoothing" [6], this can be achieved with loss of $\log(1/\epsilon)$ bits. In our approach this is not necessary.

2 Preliminaries

Notation. By U_k we denote the uniform distribution over $\{0,1\}^k$ and, more generally, by U_S we denote the distribution uniform over a set S. By $\mathrm{supp}(X)$ we denote the support of the distribution X.

Distinguishing Advantage and Statistical Distance. For two distributions X, Y defined on the same set we define $\Delta_{\mathrm{D}}(X;Y) = \mathbf{E}\,\mathrm{D}(X) - \mathbf{E}\,\mathrm{D}(Y)$ as the advantage of a function (attacker) D in distinguishing X and Y. The statistical distance is defined by $\mathrm{SD}(X;Y) = \frac{1}{2}\sum_x |\Pr(X=x) - \Pr(Y=x)|$, we also denote for shortness $\mathrm{SD}(X;Y|Z) = \mathrm{SD}(X,Z;Y,Z)$ and $\Delta_{\mathrm{D}}(X;Y|Z) = \Delta_{\mathrm{D}}(X,Z;Y,Z)$. We have $\mathrm{SD}(X,Y) = \max_{\mathrm{D}} \Delta_{\mathrm{D}}(X,Y)$, where the maximum is taken over all boolean functions D (possibly probabilistic).

Entropy notions and hash functions . Here we provide necessary entropy definitions.

▶ **Definition 1** (Min-entropy). The min-entropy of X given Z is defined as

$$\widetilde{\mathbf{H}}_\infty(X|Z) = -\log\left(\mathop{\mathbf{E}}_{z\leftarrow Z}\left[\max_x \Pr(X=x|Z=z)\right]\right). \tag{4}$$

▶ **Definition 2** (Collision-probability). The collision probability of X given Z is given by

$$\mathrm{CP}(X|Z) = \mathop{\mathbf{E}}_{z\leftarrow Z}\left(\sum_x \Pr(X=x|Z=z)^2\right) = \mathop{\mathbf{E}}_{z\leftarrow Z}\mathrm{CP}(X|_{Z=z}). \tag{5}$$

▶ **Definition 3** (Collision-entropy [2]). The collision entropy of X given Z is equal to

$$\mathbf{H}_2(X|Z) = -\log\left(\mathop{\mathbf{E}}_{z\leftarrow Z}\sqrt{\mathrm{CP}(X|_{Z=z})}\right)^2. \tag{6}$$

All these three definitions are related as follows:

▶ **Lemma 4.** *For any joint distribution X, Z we have*

$$\mathbf{H}_2(X|Z) \geqslant -\log \mathrm{CP}(X|Z) \geqslant \widetilde{\mathbf{H}}_\infty(X|Z) \tag{7}$$

▶ **Definition 5** (Almost universal families). A family $\mathcal{H}$ of functions $h : \mathcal{X} \to \{0,1\}^k$ is called γ-universal hash family, if for any $x_1, x_2 \in \mathcal{X}$, $x_1 \neq x_2$ we have $\Pr_{h\leftarrow\mathcal{H}}[h(x_1) = h(x_2)] \leqslant \gamma$. If $\gamma = 2^{-k}$ then we say that $\mathcal{H}$ is universal.

3 Leftover Hash Lemma

3.1 Standard LHL

Below we formulate the Leftover Hash Lemma for the conditional min-entropy.

▶ **Theorem 6** (The LHL [4]). *Let (X,Z) be a joint distribution on $\mathcal{X}\times\mathcal{Z}$, let $\mathcal{H} = \{h : \mathcal{X} \to \{0,1\}^k\}$ be a $\frac{1+\gamma}{2^k}$-universal family and let H be a random member of $\mathcal{H}$. Then we have*

$$\mathrm{SD}(H(X); U_k|H,Z) \leqslant \frac{1}{2}\cdot\sqrt{2^{-L}+\gamma}, \tag{8}$$

where $L = \widetilde{\mathbf{H}}_\infty(X|Z) - k$ is the entropy loss.

For universal hashing we need $L \approx 2\log(1/\epsilon)$ to make the statistical distance smaller than ϵ.

Entropy requirements for the standard LHL. It is well known that in Theorem 6 one can use collision probability instead of min-entropy. More precisely, we have

$$\mathrm{SD}(H(X); U_k|H, Z) \leqslant \frac{1}{2} \cdot \sqrt{2^k \mathrm{CP}(X|Z) + \gamma}. \tag{9}$$

Since $\mathrm{CP}(X|Z) \leqslant 2^{-\widetilde{\mathbf{H}}_\infty(X|Z)}$ this implies the bound in Equation (8). Berens [2] observed that even weaker assumption is enough. Namely, we have

$$\mathrm{SD}(H(X); U_k|H, Z) \leqslant \frac{1}{2} \cdot \sqrt{2^{k-\mathbf{H}_2(X|Z)} + \gamma}. \tag{10}$$

By Theorem 4 this implies both Equation (9) and Equation (8).

3.2 Generalized LHL

We start with the inequality that can be thought of as an abstract formulation of the LHL:

▶ **Lemma 7.** *[1] Let (Y, Z) be a joint distribution on $\mathcal{Y} \times \mathcal{Z}$ and let U be independent and uniform on $\mathcal{Y}$. Then for all real-valued functions D on $\mathcal{Y} \times \mathcal{Z}$, we have*

$$\mathbf{E}\,\mathrm{D}(Y, Z) - \mathbf{E}\,\mathrm{D}(U, Z) \leqslant \sqrt{\mathrm{Var}\,\mathrm{D}(U_{\mathcal{Y}}, Z)} \cdot \sqrt{|\mathcal{Y}| \cdot \mathrm{CP}(Y|Z) - 1}. \tag{11}$$

The generalized Leftover Hash Lemma is a special case of this result, where $Y = (H(X), H)$ for a randomly chosen hash function H. We need only one simple fact (we omit the proof):

▶ **Lemma 8.** *Let $\mathcal{H} = \{h : \mathcal{X} \to \{0,1\}^k\}$ be a $\frac{1+\gamma}{2^k}$-universal family and let H be a random member of $\mathcal{H}$. Then $\mathrm{CP}(H(X), H) \leqslant \left(\mathrm{CP}(X) + 2^{-k}(1+\gamma)\right)/|\mathcal{H}|$.*

By Theorem 7 and Theorem 8 one obtains the following generalization of Theorem 6:

▶ **Theorem 9** (Generalized LHL [1]). *Let (X, Z) be a joint distribution on $\mathcal{X} \times \mathcal{Z}$, let $\mathcal{H} = \{h : \mathcal{X} \to \{0,1\}^k\}$ be $\frac{1+\gamma}{2^k}$-universal and H be a random member of $\mathcal{H}$. Then*

$$\mathbf{E}\,\mathrm{D}(H(X), H, Z) - \mathbf{E}\,\mathrm{D}(U_k, H, Z) \leqslant \sqrt{\mathrm{Var}\,\mathrm{D}(U_k, U_{\mathcal{H}}, Z)} \cdot \sqrt{2^k \mathrm{CP}(X|Z) + \gamma}, \tag{12}$$

for any real valued function D on $\{0,1\}^k \times \mathcal{H} \times \mathcal{Z}$. In particular, if $L = \widetilde{\mathbf{H}}_\infty(X|Z) - k$ is the entropy loss then we obtain

$$\mathbf{E}\,\mathrm{D}(H(X), H, Z) - \mathbf{E}\,\mathrm{D}(U, H, Z) \leqslant \sqrt{\mathrm{Var}\,\mathrm{D}(U_k, U_{\mathcal{H}}, Z)} \cdot \sqrt{2^{-L} + \gamma}. \tag{13}$$

▶ Remark. Note that we recover the standard LHL in Equation (8) by applying the above theorem to $\mathrm{D}' = \mathrm{D} - \frac{1}{2}$ where D is $[0,1]$-valued and taking the maximum over D.

3.3 Applications of the generalized LHL

The bound in Theorem 9 introduces the factor $\mathrm{Var}\,\mathrm{D}(U_k, U_{\mathcal{H}}, Z)$ depending only on the distinguisher D. If we only want the extracted key $H(X)$ to be (almost) as good as the uniform key U for a *restricted* class of distinguishers[6] D then Equation (13) might offer a significant improvement over Equation (8).

[6] For example, in secure unpredictability applications we assume that D almost always outputs 0.

Security games. In the security game an attacker tries to win against a challenger which uses a key r. To cover the case when the key is extracted by a hash function h and there is some side information z, we assume that the attacker is given also h and z. By $\mathrm{Win}_{\mathcal{A}}(r,h,z)$ we denote the probability that the attacker $\mathcal{A}$ wins the game given h,z and challenged on r.

Unpredictability vs indistinguishability applications. The security requires the advantage of the winning probability over the "trivial strategy"[7] to be small.

▶ **Definition 10** (Security of applications, [1])**.** Let $c=0$ for an unpredictability and $c=\frac{1}{2}$ for indistinugishability application. The application is (T,ϵ)-secure in the ideal model if $\mathbf{E}\,\mathrm{Win}_{\mathcal{A}}(U,H,Z)\leqslant c+\epsilon$ for all attackers $\mathcal{A}$ with resources (running time, number of oracle questions...) less than T. The application is (T,ϵ')-secure in the real model if $\mathbf{E}\,\mathrm{Win}_{\mathcal{A}}(H(X),H,Z)\leqslant c+\epsilon'$ for all attackers $\mathcal{A}$ with resources less that T.

▶ Remark. In the "ideal" scenario the values of h and z do not help the attacker. That is, for the security in the ideal model it is enough to assume that the winning probability is smaller than $c+\epsilon$ for any $\mathcal{A}$ challenged on r which does not know h and z.

Define $\mathrm{D}(r,h,z)$ to 1 if $\mathcal{A}(h,z)$ wins when challenged on r, and 0 otherwise. Clearly we have $\mathrm{Win}_{\mathcal{A}}(H(X),H,Z)-c-(\mathrm{Win}_{\mathcal{A}}(U,H,Z)-c)=\Delta_{\mathrm{D}}(H(X);U|H,Z)$. Theorem 9 implies

$$\epsilon'\leqslant\epsilon+\sqrt{\mathrm{Var}\left(\mathrm{Win}_{\mathcal{A}}(U_k,U_{\mathcal{H}},Z)\right)}\cdot\sqrt{2^{-L}+\gamma}.$$

If the variance term is not bigger than ϵ, we see that the use of the extracted key and the use of the ideally random key are (roughly) equally secure when $L=\log(1/\epsilon)$.

Reducing the entropy loss by bounding the variance term. For the variance term to be small, the adversary's winning probability in the ideal setting must be concentrated around its mean. This is always the case of unpredictability because then we have $\mathrm{Var}\left(\mathrm{Win}_{\mathcal{A}}(U,H,Z)\right)\leqslant\mathbf{E}\,\mathrm{Win}_{\mathcal{A}}(U,H,Z)\leqslant\epsilon$. However, the case of indistinguishability is more subtle. For example, it might happen that the adversary always wins on one half of the keys and always fails on the second half. Then the variance is equal to $\frac{1}{2}$ and we do not get any improvement in the entropy loss. For more details we refer the reader to [1].

4 The generalized LHL and collision probability: no robustness

We show that the "natural" use of the collision probability to define the conditional collision entropy leads to the notion for which any reasonable chain rule fails.

▶ **Lemma 11** (Chain Rule fails for collision probability)**.** *For every k there exist random variables $X\in\{0,1\}^{k+1}$ and $Z\in\{0,1\}$ such that $\mathrm{CP}(X)=2^{-k}$ but $\mathrm{CP}(X|Z)\geqslant 2^{-\frac{k+1}{2}}$. Thus, if we define $\mathbf{H}'_2(X|Z)=-\log\mathrm{CP}(X|Z)$, then for some X and a bit Z we have $\mathbf{H}'_2(X)=k$ but $\mathbf{H}'_2(X|Z)\approx\frac{k}{2}$.*

Proof. Let $p=2^{-\frac{k+1}{2}}$ and $q=1-p$. Fix a point $x_0\in\{0,1\}^n$ and let $S\subset\{0,1\}^n$ be a set such that $|S|=2^{k+1}q^2$ and does not contain x_0. Let $Z=0$ with probability p and 1 with probability q, and let $X|_{Z=0}$ be the point mass at x_0 and let $X|_{Z=1}$ be uniform on S. Then $\mathrm{CP}(X)=p^2+q^2/\left(2^{k+1}q^2\right)=2^{-k}$ and $\mathrm{CP}(X|Z)=p\cdot 1+\frac{q}{(2^{k+1}q^2)}=\frac{1}{2^{\frac{k+1}{2}}-1}\geqslant 2^{-\frac{k+1}{2}}$. ◀

[7] In unpredictability games, an adversary needs to guess a long string so the trivial guess succeeds with the probability close to $c=0$. In indistinguishability games he needs to guess a bit, thus the trivial guess wins with probability $c=\frac{1}{2}$.

5 Generalized LHL works with conditional collision entropy

▶ **Theorem 12.** *Let X be an n-bit random variable, $\mathcal{H}$ be a $\frac{1+\gamma}{2^k}$-universal family from n to k bits and H be a random member of $\mathcal{H}$. Suppose that we have an application which is (T,ϵ)-secure in the ideal model. Then the same application is (T',ϵ')-secure in the real model where $\epsilon' \leqslant \sqrt{\epsilon 2^{k-\mathbf{H}_2(X|Z)}} + \sqrt{\epsilon\gamma}$ and $T' \approx T$ (against non-uniform attackers).*

Proof. Define $\mathrm{D}(r,h,z) = \mathrm{Win}_A(r,h,z) - c$. For every fixed z, by Theorem 7 and Theorem 8

$$\mathbf{E}\,\mathrm{D}(H(X|_{Z=z}, H, z) - \mathbf{E}\,\mathrm{D}(U,H,z) \leqslant \sqrt{\mathrm{Var}\,\mathrm{D}(U,H,z)} \cdot \sqrt{2^k \mathrm{CP}\left(X|_{Z=z}\right) + \gamma} \tag{14}$$

Since H is independent of X we have $\mathrm{Var}\,\mathrm{D}(U,H,z) \leqslant \max_h \mathrm{Var}\,\mathrm{D}(U,h,z)$ which is smaller by ϵ by the assumptions (applied to D with fixed h and z). Hence,

$$\begin{aligned}\mathbf{E}\,\mathrm{D}(H(X|_{Z=z}, H, z) - \mathbf{E}\,\mathrm{D}(U,H,z) &\leqslant \sqrt{\epsilon} \cdot \sqrt{2^k \mathrm{CP}\left(X|_{Z=z}\right) + \gamma} \\ &\leqslant \sqrt{\epsilon} \cdot \sqrt{2^k \mathrm{CP}\left(X|_{Z=z}\right)} + \sqrt{\epsilon\gamma}. \end{aligned} \tag{15}$$

Since $\mathbf{E}_{z \leftarrow Z} \sqrt{\mathrm{CP}(X|_{Z=z})} = 2^{-\frac{1}{2}\mathbf{H}_2(X|Z)}$, the result follows. ◀

6 Conclusion

Our result extend all the results of [1] for the case of collision entropy, which is much less restrictive and still provides the leakage robustness because of the chain rule. Moreover, our results strongly support the belief that this notion of collision entropy is the “right” one.

References

1 Boaz Barak, Yevgeniy Dodis, Hugo Krawczyk, Olivier Pereira, Krzysztof Pietrzak, Francois-Xavier Standaert, and Yu Yu. Leftover hash lemma, revisited. Cryptology ePrint Archive, Report 2011/088, 2011. `http://eprint.iacr.org/`.

2 Stefan Berens. Conditional renyi entropy. Master’s thesis, Mathematisch Instituut, Universiteit Leiden, 2013.

3 Yevgeniy Dodis and Yu Yu. Overcoming weak expectations. In Amit Sahai, editor, *Theory of Cryptography*, volume 7785 of *Lecture Notes in Computer Science*, pages 1–22. Springer Berlin Heidelberg, 2013.

4 Johan Hastad, Russell Impagliazzo, Leonid A. Levin, and Michael Luby. A pseudorandom generator from any one-way function. *SIAM J. Comput.*, 28(4):1364–1396, 1999.

5 Jaikumar Radhakrishnan and Amnon Ta-Shma. Bounds for dispersers, extractors, and depth-two superconcentrators. *SIAM Journal on Discrete Mathematics*, 13:2000, 2000.

6 R. Renner and S. Wolf. Smooth Renyi entropy and applications. In *International Symposium on Information Theory, 2004. ISIT 2004. Proceedings.*, page 232. IEEE, 2004.

7 D.R. Stinson. Universal hashing and authentication codes. *Designs, Codes and Cryptography*, 4(3):369–380, 1994.

Axiom of Choice, Maximal Independent Sets, Argumentation and Dialogue Games*

Christof Spanring[1,2]

1 Department of Computer Science, University of Liverpool
http://cgi.csc.liv.ac.uk/~christof
c.spanring@liverpool.ac.uk

2 Institute of Information Systems, Vienna University of Technology

Abstract

In this work we investigate infinite structures. We discuss the importance, meaning and temptation of the axiom of choice and equivalent formulations with respect to graph theory, abstract argumentation and dialogue games. Emphasis is put on maximal independent sets in graph theory as well as preferred semantics in abstract argumentation.

1998 ACM Subject Classification G.2.2 Graph Theory, F.4.1 Mathematical Logic, I.2 Artificial Intelligence

Keywords and phrases axiom of choice, graph theory, maximal independent sets, abstract argumentation, dialogue games

Digital Object Identifier 10.4230/OASIcs.ICCSW.2014.91

1 Introduction

In everyday life we hardly ever think of dealing with actually infinite structures. Our time on earth may be complicated but it appears to be strictly finite, we deal with finite space, finite distances and finite numbers. Computer scientists in particular tend to prefer working with finite structures, algorithms are supposed to terminate in a finite amount of time. Often enough infinity introduces odd behaviour and exceptions to the languages we use, and grew to love. Therefore in everyday life infinity seems to be ignored.

However infinite structures actually are important even in our everyday life, see [18] for a fabulous overview in that matter. It might be pointed out that our concepts of economy and wealth build upon the idea of possibly infinite growth. Leibniz and Newton independently introduced infinitesimal calculus to a wider field of mathematics which nowadays plays major roles in almost every application of calculus to everyday life. On a more abstract level the failure of the halting problem is strongly related to a possible infinity. On the one hand it might seem disappointing that we will never be able to list all the perfect algorithms, on the other hand it comes as a relieve that we can always improve. Finally for security issues and problems, often enough specific finite limits play a major role.

If for one reason or another we agree to consider arbitrarily infinite structures then at some point we will also have to decide on what concept of infinity we take into account. In this paper we work with Zermelo-Fraenkel Set Theory [7, 14] and focus on the most controversial concept therein so far: The axiom of choice [13], the axiomatic existence of a choice function selecting exactly one element from an arbitrary set of arbitrary sets.

[*] This research has been supported by FWF (project I1102).

Imperial College Computing Student Workshop (ICCSW'14).
Editors: Rumyana Neykova and Nicholas Ng; pp. 91–98
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Gödel proved consistency of choice with axiomatic set theory in [11]. However a few years later Cohen [6] showed that also its negation is relatively consistent. Which leaves us with its independence and thus the painful choice of accepting it or not.

Focus of this paper is on abstract argumentation as introduced by Dung in his seminal paper [8]. An abstract argumentation framework can be visualized as a graph where nodes reflect arguments and directed edges reflect conflicts between arguments. An argumentation semantics is a set of rules to declare sets of arguments as acceptable. Dialogue games are often motivated by real-life arguments and thus naturally play an important role in the motivation of some argumentation semantics.

In Section 2 we will introduce some basic concepts of set theory and infinity, and discuss the axiom of choice, alternate forms and motivation. In Section 3 we will proceed to discuss choice in the context of infinite graph theory [19]. It is known that the axiom of choice proves equivalent to the existence of a spanning tree for connected graphs and also to the existence of a maximal independent set [9]. We elaborate on the later result, and extend it to cover also preferred semantics of abstract argumentation [8] in Section 4. Finally in Section 5 we will introduce the axiom of determinacy, as used in dialogue games, we will discuss conflicts between the introduced axioms, draw connections and point out difficulties.

2 Set Theory and the Axiom of Choice

In this section we present notions and conventions. As mathematical basis we choose Zermelo-Fraenkel set theory [7, 14]. Observe that we make use of sets of sets, a differentiation between sets and elements by the use of uppercase and lowercase letters is therefore not possible. We will thus use the same lowercase letters to refer to sets as well as to elements.

▶ **Definition 1** (ZF-Axioms)**.** This is merely a selection of the axioms of interest. As an extended introduction into the matter we can recommend [7].

- Axiom schema of restricted comprehension (COMP): we can construct subsets of sets obeying some appropriate formula.
- Axiom of union (UN): The union over the elements of a set is a set.
- Axiom schema of replacement (REP): The image of a set under any definable function will also fall inside a set.
- Axiom of power set (POW): For any set x, there is a set y containing every subset of x.

While there are about (depending on actual definitions) nine axioms in ZF we only gave the above four, necessary here. Also some formalisms try to avoid one or the other of these axioms, as the less axioms a theory needs the stronger it becomes.

Axiomatic set theory proves especially helpful when speaking about infinity, as concepts that seem straightforward in the finite case often turn out to be of paradoxic nature in the infinite case. Since [10] we know that incompleteness necessarily is an inherent feature of reasonably strong formal systems, and thus independence [15] often enough the closest statement to consistency we can actually achieve.

▶ **Definition 2** (Choice)**.** We present the one axiom in many costumes that gave rise to the most objections, and is thus not a mandatory member of ZF. We choose two equivalent definitions. In the literature ZF together with any of these is called ZFC, Zermelo-Fraenkel set theory with choice.

- Axiom of Choice (AC): For any set x of nonempty sets, there exists a choice function f defined on x, i.e. $\forall y \in x : f(y) \in y$.
- Zorn's Lemma (ZL): In a partially ordered set where every chain has an upper bound there is at least one maximal element.

It is a well-known fact that in ZF we have that the extending assumptions (AC) and (ZL) are equivalent, in that each implies the other. When dealing with discrete structures (ZL) often comes in handy, (AC) on the other hand comes most often into play when there is no intuitive understanding of underlying structures. However in ZF these two (and countless others) can be used interchangeably. We would like to point out that countable choice (x contains only countably many sets) and also finite choice (x contains only finite sets) are substantially weaker than choice. An intuitive understanding of some structure might be misleading as uncountable structures seem to lie beyond human comprehension.

3 Graph Theory

Graph Theory [3, 23] lies at the core of discrete mathematics, with a wide range of applications such as shortest paths, and also with simple and clean definitions. Textbooks on algorithms vastly refer to some graph like structure as their standard model.

▶ **Definition 3** (Graph Theory). A *graph* G is a pair $G = (V, E)$ where the set of *vertices* V is an arbitrary set and the set of *edges* E is a collection of two-element subsets of V. For any two vertices $v_1, v_2 \in V$ with $\{v_1, v_2\} \in E$ we say that v_1 and v_2 are *adjacent*. If on the other hand $\{v_1, v_2\} \notin E$ we say that v_1 and v_2 are *independent*. A set of vertices $W \subseteq V$ is called *independent* iff it does not contain any adjacent vertices, in other words iff all vertices in W are pairwise independent. A set $W \subseteq V$ is called a *maximal independent set* of vertices in G iff it is independent and one of the following equivalent conditions holds:

- W is adjacent to any vertices not being member of W, i.e. for $v \in V \setminus W$ there is some $w \in W$ such that $\{v, w\} \in E$.
- W is maximal, i.e. there is no independent set W' such that $W \subsetneq W'$.

3.1 Choice in Graph Theory

We now introduce the statement about the existence of maximal independent sets, which will be shown to be another equivalent definition to (AC), the axiom of choice.[1]

▶ **Definition 4** (Existence of Maximal Independent Sets (MIS)).

- For any graph G there exists a maximal independent set M_G.

▶ **Theorem 5** ((AC) $\implies$ (MIS)). *Assuming Zermelo-Fraenkel Set Theory with Choice (ZFC) every graph has a maximal independent set.*

We take some graph $G = (V, E)$ as given. Now the objective is to construct some maximal independent set $M \subseteq V$ such that

1. M is independent, i.e. there are no $v, w \in M$ such that $\{v, w\} \in E$ and even
2. M is maximal independent, i.e. for any $v \in V \setminus M$ there is some $m \in M$ such that $\{v, m\} \in E$.

▶ Remark (Proof using (ZL)). We observe that sets of nodes represent a partially ordered set using the subset relation. Now for chains of independent sets $(M_i)_i$, where $M_i \subseteq M_j$ for $i \leq j$, we have that $\bigcup_i M_i$ is a natural upper bound for M_i. Using (ZL) we conclude that there has to be a maximum, i.e. a maximal independent set in G. We still present the following proof to highlight respective constructiveness.

[1] As pointed out by an anonymous reviewer this result is not exactly novel. In fact the proof for Theorem 2.4 in [9] follows the same ideas, although it uses an alternate form of (AC) and it provides one *not* too many, i.e. x, y should be adjacent iff they are R-equivalent.

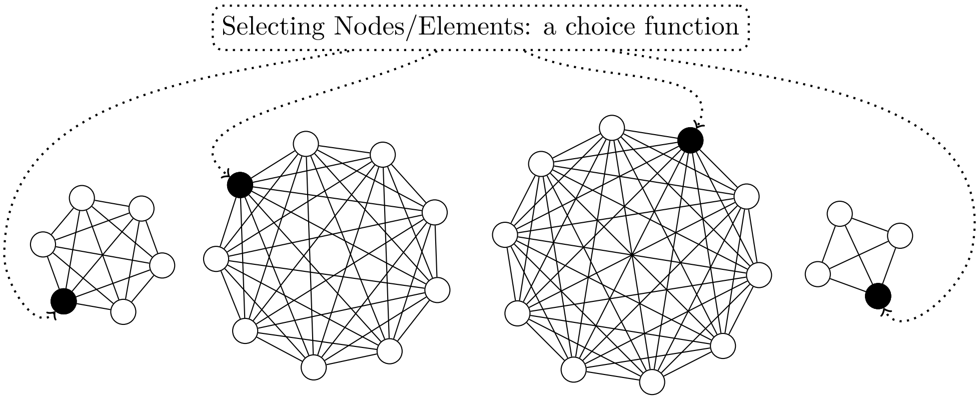


Figure 1 Illustration of the construction used for (MIS) ⇒ (AC) as hinted to in Example 6.

▶ Remark (Cardinality and Ordinal Numbers). In axiomatic set theory ordinal numbers are an extension of natural numbers, a tool for counting infinities. For a given set x we call the smallest ordinal α for which there is a bijection between x and α its cardinality, $|x| = \alpha$. Observe that cardinality actually is strongly linked to (AC).

Proof. Observe that the empty set $\emptyset \subseteq V$ is an independent set of G. We use transfinite recursion to construct a maximal independent set and thus start with $M_0 = \emptyset$.

For any ordinal i either M_i is already maximal independent or there is some $v \in V \setminus M_i$ with $M_i \cup \{v\}$ still being independent, we then use $M_{i+1} = M_i \cup \{v\}$.

For limit ordinals α we use $M_\alpha = \bigcup_{i\in\alpha} M_i$. Observe that for this step we might need the choice function for the possibly infinitely many choices being necessary to receive M_α.

Since the Axiom of Choice holds and V is a set (using (AC) every set has a cardinality, i.e. transfinite recursion has to come to an end) this process will eventually stop, i.e. there is some $M = \bigcup_i M_i$. We have that $M \subseteq V$ and therefore M is a set. Furthermore since dependence is a finite condition M has to be independent. ◀

We can thus use any independent set and any node to start the construction of a maximal independent set. But what about the other direction. Assume (MIS), does this tell us something about (AC)?

▶ **Example 6** (Graph-Theoretical Motivation). We interpret a set of sets as an independent collection of cliques (subgraphs where any two vertices are adjacent). In other words for a given set of sets X we have that any $y \in Y \in X$ represents a distinct vertex and any two vertices are adjacent iff they originate from the same set $Y \in X$. Now (MIS) delivers a maximal independent set M where exactly one vertex for each clique and thus exactly one member for each set is fixed. An illustration of the resulting graph can be found in Figure 1.

▶ **Theorem 7** ((MIS) $\implies$ (AC)). *Assuming Zermelo-Fraenkel Set Theory without Choice (ZF) and the existence of a maximal independent set for arbitrary graphs (MIS) we can derive the axiom of choice (AC).*

We take some set of nonempty sets X as given. Now the objective is to find a choice function $f : X \to \bigcup X$ such that for any set $Y \in X$ we have $f(Y) \in Y$. Observe that in mathematical terms a function actually is just a specific kind of relation and can thus be defined as a set of pairs.

Proof. We start by constructing a graph $G = (V, E)$, where V consists of all the pairs (Y, y) such that $Y \in X$ and $y \in Y$, and adjacency in E is defined by belonging to the same set Y and being different, i.e. $\{(Y_1, y_1), (Y_2, y_2)\} \in E$ iff $Y_1 = Y_2$ and $y_1 \neq y_2$.

The above steps in more detail:

- We use (REP) to create a set X' such that $Y' \in X'$ iff any $y' \in Y'$ is of the form (Y, y) where $Y \in X$ and $y \in Y$.
- We use (UN) to create $V = \bigcup Y'$.
- We use (REP), (POW), (COMP) and (UN) to create E. I.e. E is constructed by replacing each member of Y' with its size two suitable subsets $\{\{x, y\} \,|x \neq y \in Y'\} \subseteq \mathcal{P}(Y')$ and finally we collect all of those in one single set E by using (UN).

Now assuming (MIS) we conclude that G provides a maximal independent set $M \subseteq V$. By construction M consists of pairs of the form (Y, y) where $y \in Y \in X$. It remains to show that M already serves the purpose of being a choice function for X, mapping Y to y.

We observe that for $v = (Y_1, y_1)$ and $w = (Y_2, y_2)$ such that $v \neq w \in V$ we have v being adjacent to w iff $Y_1 = Y_2$ and v being independent from w iff $Y_1 \neq Y_2$.

M is independent. For each $Y \in X$ there is thus at most one $y \in Y$ such that $(Y, y) \in M$, since M is maximal independent there is furthermore at least and thus exactly one $y \in Y$ such that $(Y, y) \in M$. ◀

▶ Remark (Directed Graphs). A *directed graph* is a pair $D = (V, E)$, where V is an arbitrary set of vertices but this time the set of edges $E \subseteq V \times V$ is a set of pairs. Observe that this allows edges also from one node to itself. The definition of independent sets and proofs for equivalence with axiom of choice carry on also for the directed case.

4 Abstract Argumentation

Abstract argumentation can be seen as applied directed graph theory. It was introduced by Dung in [8] and motivated by philosophical works [12, 20] and non-monotonic logic [4]. In this work we also want to highlight the interplay of argumentation and dialogue games [21].

▶ **Definition 8.** An *argumentation framework (AF)* is an ordered pair $F = (A, R)$ where A is an arbitrary set of *arguments* and $R \subseteq A \times A$ is called the *attack* or *conflict* relation. For $(a, b) \in R$ we also write $a \rightarrowtail^R b$ and say that a attacks b in R. Furthermore for $(a, b), (b, c) \in R$ we say that a defends c against b in R.

If clear from context we might omit the referencing R or other redundant information. If not otherwise stated we will also assume some AF $F = (A, R)$ as given.

Furthermore for $B \subseteq A$ and $a \in A$ we say that $a \rightarrowtail B$ or $B \rightarrowtail a$ iff for some $b \in B$ we have $a \rightarrowtail b$ or $b \rightarrowtail a$ respectively. We extend this notion also for $B, C \subseteq A$ accordingly.

▶ Remark (Directed Graphs and Argumentation Frameworks). Observe that so far abstract argumentation frameworks and directed graphs are formally equivalent, except for names. The main difference being one of intended meaning. In graph theory we think of nodes being near to each other if they are connected by an edge, or even connected via a path of several nodes and edges. Furthermore if two nodes do not cooperate we could reflect this only indirectly by not having any connection between them. In argumentation theory on the other hand nearness is expressed only indirectly via the notion of defense, while an attack represents a conflict between two arguments.

What follows is the notion of semantics as used in abstract argumentation, an attempt of solving the question of acceptability. Investigating some arbitrary AF we might want to consider some sets of arguments, for instance an argument line of defense. A good line of defense of course is a line of attack. Truth in the argumentation sense is thus a notion of acceptability. For a comprehensive introduction into argumentation semantics see [1].

▶ **Definition 9.** An *argumentation semantics* is a mapping from AFs to sets of arguments where for any AF $F = (A, R)$ and semantics σ we have $\sigma(F) \subseteq \mathcal{P}(A)$. The members of $\sigma(F)$ are then called *σ-extensions* of F. The other way around we will define semantics by stating properties an extension has to fulfill. We sometimes call an argument that is member of every σ-extension *sceptically accepted* and an argument that is member of some σ-extension *credulously accepted.*

A set of arguments $E \subseteq A$ is called

- *conflict-free* (*cf*) or a conflict-free extension iff there is no conflict in E, $(E \times E) \cap R = \emptyset$,
- *admissible* (*adm*) or an admissible extension iff it is *cf* and defends itself against any attacks, $E \in cf(F)$ and for any $a \rightarrowtail E$ we also have $E \rightarrowtail a$,
- a *preferred* (*prf*) extension iff it is maximal admissible, $E \in adm(F)$ and there is no $E' \in adm(F)$ with $E \subsetneq E'$.

4.1 Choice in Argumentation

The remainder of this section is a brief summary of implied (AC)-results for abstract argumentation. The first part of this theorem has already been discussed in the literature (see [8, 5]) although with use of (ZL). The second part however is a novel result and might still come as a surprise to some. Preferred Extensions are one of the most basic concepts in abstract argumentation and the necessary definitions seem to be rather intuitive and straightforward. It is not obvious that this concept of discrete structures carries over to e.g. existence of a base in the vector space of all continuous functions.

▶ **Definition 10** (Existence of Preferred Extensions (PE)).

- For any AF F there exists a preferred extension $prf(F) \neq \emptyset$.

▶ **Theorem 11** ((AC) $\Longleftrightarrow$ (PE)). *In ZF, the axiom of choice is equivalent to the existence of preferred extensions in arbitrary AFs.*

Proof Part 1, $\Longrightarrow$: Very similar to (MIS) we use transfinite recursion and observe that the empty set is an admissible extension. If some admissible set is not maximal then there is a bigger one. And finally for any chain of admissible sets $E_1 \subsetneq E_2 \subsetneq \dots$ it holds that $E = \bigcup_i E_i$ also is an admissible set. Conflict is a finite condition and any argument $a \in E$ that is not defended by E was first introduced for some E_i and thus already E_i would not have been able to defend a. ◀

Proof Part 2, $\Longleftarrow$: Observe that the construction from the proof for Theorem 7 can be seen as an argumentation framework by the natural transformation from undirected to directed graphs, where for the graph $G = (V, E)$ we use an AF $F = (V, E')$ where $E' = \{(a, b) \mid \{a, b\} \in E\}$. Then obviously X is a preferred extension of F iff X is a maximal independent set of G iff X is a choice function for the originating set of sets. ◀

5 Discussion and Dialogue Games

When thinking about AFs we think about some mode of argumentation, with every argument being some sort of statement and the conflict relation being naturally induced by the origin of the arguments. We might acquire arguments (and implicit conflicts) from logical formulas or programs or from some natural language dialogue. A major source of motivation for abstract argumentation also are dialogue games [16, 22].

From the very beginning of argumentation it seems that dialogue games played an important motivational role, see [2, 8]. For a nice introduction into the use of dialogue games to reason about credulous and skeptical preferred acceptability also see [21]. We sometimes think of argumentation as an interpretation of dialogue games on an attack graph. A move can be seen as the act of claiming an argument, where Opponent selects attacking arguments and Proponent selects defending arguments. Observe that credulous acceptance of an argument for preferred semantics can be implemented with the idea of Proponent indefinitely being able to defend his selected arguments.

▶ **Definition 12** (Dialogue Games)**.** A *dialogue game* is a two-player game, with alternating moves. In the case of *perfect information* we assume that consequences of any move are known in advance by both players and that both players are aware of all possible states. A *winning strategy* for one of the players is a function mapping game states into a set of moves such that the resulting sequence indicates a win by the respective player. We call the player to start the game *Proponent* and the player to answer first *Opponent*.

When digging into game theory and reflecting the axiom of choice one sooner or later stumbles upon determinacy, the question of whether winning strategies do exist and whether a game is decided before it actually started. For games on fixed AFs we would expect determinacy and we would also hope for a one-to-one relation between abstract argumentation and dialogue games. However sometimes what we hope for is not what we get.

▶ **Example 13** (The number game $\mathbb{S}$)**.** Take some arbitrary set $\mathbb{S} \subseteq \mathbb{N}^{\mathbb{N}}$. We define a dialogue game of length $|\mathbb{N}|$ where moves are natural numbers, i.e. possible game sequences are of the form $(n_1, n_2, \cdots)$. Proponent wins iff the played sequence is an element of $\mathbb{S}$.

▶ **Definition 14** (Axiom of Determinacy (AD))**.**

▬ Every number game $\mathbb{S}$ is predetermined, i.e. one of the players has a winning strategy.

We now want to highlight a quite interesting result [17]: the axioms of choice and of determinacy are incompatible, i.e. (AC) implies not (AD), in other words choice attacks determinacy. The proof idea here is a classical diagonal argument. Now take into account the above result that existence of a preferred extension and the axiom of choice are equivalent. It follows that the idea of determinacy, allowing winning strategies in perfect information games and existence of preferred extensions for arbitrary AFs are incompatible as well.

Once more we point out that most of computer science takes place in the finite or countably infinite case. As (AD) implies countable choice the stated conflict in most cases will not be of immediate relevance. From time to time we should just remind ourselves that a vast generalization of techniques and results might not be possible. In this context the really interesting question will be which axiom to trust in the arbitrarily infinite case. Do we prefer every graph to contain a maximal clique and every argumentation framework to contain at least one preferred extension, or do we prefer every dialogue game with perfect information to be predetermined? What other results can we gain distinguishing these cases? Are there for instance argumentation semantics that rely on existence of winning strategies? For which AFs and which games can we find transformations/mappings such that winning strategies and some semantics correlate?

Acknowledgements. We would like to thank the anonymous reviewers for their helpful and constructive comments.

References

1 Pietro Baroni and Massimiliano Giacomin. Semantics of abstract argument systems. In Iyad Rahwan and Guillermo Ricardo Simari, editors, *Argumentation in Artificial Intelligence*, chapter 2, pages 25–44. Springer, 2009.

2 Trevor J. M. Bench-Capon, Sylvie Doutre, and Paul E. Dunne. Asking the right question: forcing commitment in examination dialogues. In Philippe Besnard, Sylvie Doutre, and Anthony Hunter, editors, *Proceedings of the 2nd Conference on Computational Models of Argument* (*COMMA 2008*), volume 172, pages 49–60. IOS Press, 2008.

3 Béla Bollobás. *Modern graph theory*, volume 184. Springer, 1998.

4 Martin Caminada and Dov M. Gabbay. A logical account of formal argumentation. *Studia Logica*, 93(2):109–145, 2009.

5 Martin Caminada and Bart Verheij. On the existence of semi-stable extensions. In *Proceedings of the 22nd Benelux Conference on Artificial Intelligence*, 2010.

6 Paul J Cohen. The independence of the continuum hypothesis. *Proceedings of the National Academy of Sciences of the United States of America*, 50(6):1143, 1963.

7 Keith Devlin. *The Joy of Sets: Fundamentals of Contemporary Set Theory.* Undergraduate Texts in Mathematics. Springer, Springer-Verlag 175 Fifth Avenue, New York, New York 10010, U.S.A., 2nd edition, 1994.

8 Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artif. Intell.*, 77(2):321–358, 1995.

9 Harvey M. Friedman. Invariant maximal cliques and incompleteness, 2011.

10 Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatsh. Math. Phys.*, 38(1):173–198, 1931.

11 Kurt Gödel and George William Brown. *The consistency of the axiom of choice and of the generalized continuum-hypothesis with the axioms of set theory.* Princeton University Press, 1940.

12 Charles Leonard Hamblin. *Fallacies.* Methuen London, 1970.

13 Thomas Jech. About the axiom of choice. *Handbook of mathematical logic*, 90:345–370, 1977.

14 Thomas Jech. *Set Theory.* Springer, 3rd edition, 2006.

15 Kenneth Kunen. *Set Theory An Introduction To Independence Proofs (Studies in Logic and the Foundations of Mathematics).* North Holland, 1983.

16 Peter McBurney and Simon Parsons. Dialogue games for agent argumentation. In *Argumentation in artificial intelligence*, pages 261–280. Springer, 2009.

17 Jan Mycielski. On the axiom of determinacy. *Fund. Math*, 53:205–224II, 1964.

18 Rudy Rucker. *Infinity and the Mind: The Science and Philosophy of the Infinite (Princeton Science Library).* Princeton University Press, 2004.

19 Lajos Soukup. Infinite combinatorics: from finite to infinite. In *Horizons of combinatorics*, pages 189–213. Springer, 2008.

20 Stephen Toulmin. *The Uses of Argument.* Cambridge University Press, 2003.

21 Gerard AW Vreeswik and Henry Prakken. Credulous and sceptical argument games for preferred semantics. In *Logics in Artificial Intelligence*, pages 239–253. Springer, 2000.

22 Douglas N Walton. *Logical Dialogue-Games.* University Press of America, Lanham, Maryland, 1984.

23 Douglas Brent West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.