UNIVERSITY OF OSLO
Department of Informatics

TTSS'11—5th Intl. Workshop on Harnessing Theories for Tool Support in Software

Research Report No. 409

Marcel Kyas, Sun Meng, and Volker Stolz

ISBN 82-7368-371-0
ISSN 0806-30360806

September 2011

TTSS'11—5th International Workshop on Harnessing Theories for Tool Support in Software

Marcel Kyas, Sun Meng, and Volker Stolz

This volume contains the proceedings of the 5th International Workshop on Harnessing Theories for Tool Support in Software (TTSS), held on 13. September 2011 at the University of Oslo, Norway. The aim of the workshop is to bring together practitioners and researchers from academia, industry and government to present and discuss ideas about:

- How to deal with the complexity of software projects by multi-view modeling and separation of concerns about the design of functionality, interaction, concurrency, scheduling, and non-functional requirements, and
- How to ensure correctness and dependability of software by integrating formal methods and tools for modeling, design, verification and validation into design and development processes and environments.
- Case studies and experience reports about harnessing static analysis tools such as model checking, theorem proving, testing, as well as runtime monitoring.

Topics of interest include, but are not limited to, the following areas:

- Models, calculi, and tool support for component-based and object-oriented software;
- Mathematical frameworks, methods and tools for model-driven development;
- Models, calculi, and tool support for integrating different scheduling, interaction and concurrency models in highly adaptable systems

The workshop was initiated under the auspices of Prof. Jifeng He (ECNU, China) and Dr. Zhiming Liu (UNU-IIST, Macao). Additionally, Patrick Cousot (ENS, France), Mathai Joseph (TATA, India), Bertrand Meyer (ETH Zurich, Switzerland), and Jim Woodcock (U. York, UK) kindly agreed to lend their advice.

For this edition of the workshop, just like last year, eight papers out of eleven submissions were selected for presentation by the Program Committee. Two of those submissions are in the special session on *Overcoming Challenges of Security and Dependability* organised by Marcel Kyas, Volker Roth, and Katinka Wolter from FU Berlin, Germany (see separate preface). We gratefully acknowledge the contribution of the members of the Program Committee and of their delegate reviewers, listed below. In addition to the regular submission, we are happy to present an invited talk by Dr. Ralf Huuck from NICTA, Australia.

The Program Chairs also thank Violet Pun and the organizers of the 8th International Symposium on Formal Aspects of Component Software (FACS 2011), Peter Ølveczky, and Lucian Bentea for their help in preparing the workshop.

Program Committee

Farhad Arbab	CWI and Leiden University, The Netherlands
Christel Baier	Technical University of Dresden, Germany
Luis Barbosa	Universidade do Minho, Portugal
Manfred Broy	TU München, Germany
Michael Butler	University of Southampton, UK
Dave Clarke	K.U.Leuven, Belgium
Erik De Vink	Technische Universiteit Eindhoven, The Netherlands
Ralf Huuck	NICTA, Australia
Einar Broch Johnsen	University of Oslo, Norway
Joost-Pieter Katoen	RWTH Aachen, Germany
Peter Gorm Larsen	Aarhus School of Engineering, Denmark
Martin Leucker	Uni Lübeck, Germany
Xuandong Li	Nanjing University, China
Laurent Mounier	VERIMAG, France
Jun Pang	University of Luxembourg, Luxembourg
Shengchao Qin	School of Computing, Teesside University, UK
Anders Ravn	Aalborg University, Denmark
Abhik Roychoudhury	National University of Singapore
Wuwei Shen	Western Michigan University, USA
Volker Stolz (co-chair)	United Nations University (UNU-IIST), Macao S.A.R. & University of Oslo, Norway
Meng Sun (co-chair)	Peking University, China
Jaco Van De Pol	University of Twente, The Netherlands
Jim Woodcock	University of York, UK
Jian Zhang	Institute of Software, Chinese Academy of Sciences, China

Volker Stolz is supported by the ARV grant of the Macao Science and Technology Development Fund.

Additional Reviewers

Joao F. Ferreira, Mario Gleirscher, Daniel Thoma, and Hongli Yang.

Workshop on Overcoming Challenges of Security and Dependability

Marcel Kyas, Volker Roth, and Katinka Wolter
Department of Computer Science, Freie Universität Berlin, Germany
{marcel.kyas,volker.roth,katinka.wolter}@fu-berlin.de

The following two contributions were submitted to the Workshop on Overcoming Challenges of Security and Dependability (WOCSD), to be held in August 2011. The aim of the workshop is to bring together practitioners and researchers from academia, industry and government to present and discuss about possible synergy between the research areas of formal methods, quantitative methods and security. The major questions of interest were:

- How to decrease software complexity and specification complexity to increase resilience and security?
- How to ensure correctness, safety, dependability and security of computer systems?
- How to certify software for todays heterogeneous computer platforms?

We have received three submissions of which we accepted two for presentation. But two presentations do not make a workshop. We have thus decided to cancel WOCSD as an individual workshop and organise a session at TTSS instead. The organising committee of WOCSD is grateful to the organisers of TTSS for providing us with the opportunity of our own session. We also thank to our presenters who were flexible and willing to present their work one month later.

Marcel Kyas
Volker Roth
Katinka Wolter
Berlin, September 1, 2011

Program Committee

- Allesandro Aldini, University of Bologna, Italy
- Marcel Kyas, Freie Universität Berlin, Germany
- Mohammad Reza Mousavi, Technical University of Eindhoven, The Netherlands
- Dusko Pavlovic, Royal Holloway University of London, UK
- Volker Roth, Freie Universität Berlin, Germany
- Nigel Thomas, New Castle University, UK
- Katinka Wolter, Freie Universität Berlin, Germany
- Stephen Wolthusen, Royal Holloway University of London, UK
  and Gjøvik University College, Norway

Table of Contents

Real World Model Checking of Millions of Lines of C/C++ Code **7**
Ralf Huuck . . . .

Towards Certifiable Software for Medical Devices: The Pacemaker Case Study Revisited **8**
Michaela Huhn and Sara Bessling . . . .

Admissible adversaries in PRISM for probabilistic security analysis **15**
Alain-Freddy Kiraga and John Mullins . . . .

Monadic Scripting in F# for Computer Games **35**
Giuseppe Maggiore, Michele Bugliesi and Renzo Orsini . . . .

Tool Supported Analysis of Web Services Protocols **50**
Abinoam P. Marques Jr., Anders Ravn, Jiri Srba and Saleem Vighio . . . .

A Formal Approach to Data Validation Constraints in MDE **65**
Alessandro Rossini, Adrian Rutle, Khalid Mughal, Yngve Lamo and Uwe Wolter . . .

Towards rigorous analysis of Open Source Software **77**
Luis Barbosa, Pedro Henriques and Alejandro Sanchez . . . .

Stochastic Reo: a Case Study **90**
Young-Joo Moon, Farhad Arbab, Alexandra Silva, Chretien Verhoef and Andries Stam

A Calculus for a New Component Model in Highly Distributed Environments **106**
Antoine Beugnard and Ali Hassan . . . .

Real World Model Checking of Millions of Lines of C/C++ Code

Ralf Huuck
NICTA, Australia
ralf.huuck@nicta.com.au

Abstract Model checking has a long stigmatized history of being slow and not scalable to large real life systems. In this talk we report on our experiences of using model checking at the core of our C/C++ source code analysis tool Goanna. We present our underlying abstractions, refinement models and auxiliary techniques to obtain a solution that is fast, scalable, and sufficiently precise. Moreover, we report on our experience from routinely uncovering security vulnerabilities and mission critical bugs in real life systems, and the challenges in moving our Goanna software from an academic project to widely used commercial product.

Bio Dr. Ralf Huuck is a senior researcher with NICTA, Australia's national center of excellence for computer science research, a senior lecturer with the Univeristy of New South Wales, Sydney, and the CTO and co-Founder of Red Lizard Software, an enterprise delivering software source code analysis solutions. Ralf obtained his PhD in formal methods from the University of Kiel, Germany, and was holding visiting appointments in France, Hong Kong, Australia and Japan.

Towards Certifiable Software for Medical Devices: The Pacemaker Case Study Revisited

Michaela Huhn Sara Bessling
Department of Informatics, Clausthal University of Technology
38678 Clausthal-Zellerfeld, Germany
email{Michaela.Huhn|Sara.Bessling}@tu-clausthal.de

Design and verification of pacemaker software - as an instance of a highly dependable medical device - has been investigated in numerous works tackling various safety requirements with different formal methods. However, in order to certify a product, a conclusive argument has to be provided that seamless and concerted safety activities starting from the hazard analysis towards the verification of the derived safety requirements yield a dependable product.

We present an approach towards the development of certifiable medical device software using SCADE Suite for the pacemaker case study. For safety analysis we use *Deductive Cause Consequence Analysis* (DCCA) as an enhanced, systematic technique to identify potential hazards and verify the derived the safety requirements. Formal verification is split into a part done in the SCADE Suite and the real-time behavior which is proven using UPPAAL.

1 Introduction

Within the prospering markets for health care, the area of medical devices is thriving as well. Numerous new application areas, e.g. for living assistance or home-based medical support, have been developed based on the emerging possibilities of software-controlled devices. Dependability was an issue for medical devices always, but only 2006 a safety standard, i.e. IEC 62304 [7] that regulates the software life cycles activities for medical devices, was agreed on. IEC 62304 names quality goals as well as core processes and development activities that are well-suited for dependable software. But in difference to other domains like IEC 61508 [2], the standard's recommendations are not underpinned by concrete techniques that are considered appropriate for a certain software integrity level. Thus a commonly agreed or at least scientifically justified line of methods for safety development of medical devices is still missing.

We demonstrate a model-based, formally founded approach to the development of safety-critical software on the case of a cardiac pacemaker. Our approach, as we present it here, provides coordinated hazard analysis, model-based design with automated code generation and formal verification by model checking. We use DCCA [9] as a systematic and formal method to identify hazards and derive safety requirements. These are modeled as observer nodes and reused for verification later on. We decided for SCADE Suite by Esterel Technologies [4] as development framework, since SCADE Suite has been formally qualified as an adequate tool for developing software for safety-critical systems compliant to safety standards (see [4]). Tool qualification is a key argument in a safety case that has to be provided for certification, because in the safety case appropriateness of methods has to be proven.

In a first attempt the design was done fully in SCADE, but efficiency arguments from both, design and verification, turned the decision towards an external handling of timers and events. Consequently, we have to verify the real-time behavior separately from the control logic. For this we extend the time abstraction we proposed in [3] for a case study on a railway level crossing modeled in SCADE as well.

© M. Huhn & S. Bessling
This work is licensed under the
Creative Commons Attribution License.

The contribution of this paper is twofold: (1) The combined verification of the safety requirements using SCADE Design Verifier and UPPAAL is of interest in itself. (2) We give a showcase for a concerted line of methods for safety development on which a safety argument can be built as backbone for certification.

2 Background

2.1 The SCADE Tool Suite

The acronym SCADE stands for Safety-Critical Application Development Environment. The main objectives of the SCADE Suite are (1) to support systematic, model-based development of correct software based on formal methods and (2) to cover the whole development process [4]. The language *Scade* underlying the tool is data-flow oriented. Its formal semantics is based on a synchronous model of computation, i.e. cyclic execution of the model.

The SCADE Suite is an integrated development environment that covers many development activities of a typical process for safety-critical software: modeling, formal verification using the SAT-based SCADE Design Verifier [1], certified automatic code generation producing readable and traceable C-code, requirements tracing down to model elements and code, simulation and testing on the code level and coverage metrics for the test cases with the Model Test Coverage module.

2.2 Deductive Cause Consequence Analysis

A major goal in safety analysis is to determine how faults modes at the component level causally relate to system hazards. Among the various formally founded techniques proposed for this task we have selected *Deductive Cause Consequence Analysis (DCCA)* by Ortmeier et al. [9, 5], because DCCA does not only formalize techniques like FTA (*Fault Tree Analysis*) and an FMEA (*Failure Mode and Effect Analysis*) [6], which are well-established and recommended by the standards. In addition, the identified fault modes and hazards can be reused in safety assurance to formally verify that sufficient measures have been taken to prevent the identified hazards. In DCCA, components faults are modeled as simple fault automata that extend the normal behavior of the component. Hazards are specified as observer nodes that read signals from the control logic and evaluate them according to the *negation* of the hazard predicate. Then the verification process can be performed in order to iteratively determine the so-called *minimal critical sets*, i.e. subsets of faults that may lead to the hazard - in case that they occur in a certain order (for details and formalization see [5, 3]).

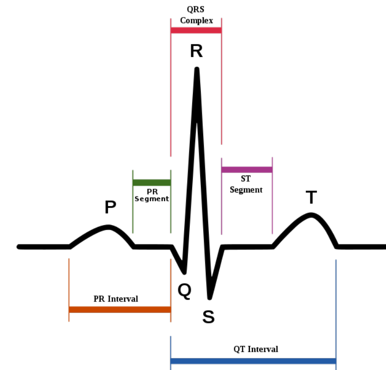


Figure 1: Sinus rhythm of a human heart (sane adult), source: Wikipedia

2.3 Related Work

Since PACEMAKER Formal Methods Challenge was set up by publishing Boston Scientific's Specification of an industrially produced pacemaker [10], pacemakers were investigated intensively within the formal methods community. For brevity, we only refer to two of them: Jee, Lee, and Sokolsky worked on assurance cases of the pacemaker software [8]. The authors focused on the basic VVI mode of a pacemaker, and employed UPPAAL for both, design and verification. Moreover, they

implemented their own code generation from UPPAAL to C-code to generate an executable from their UPPAAL model. In [11] the authors used timed CSP to verify some of the timing constraints for different pacemaker modes. However, only the specification level is considered in this work.

3 The Pacemaker

3.1 The Human Heart

From a bio-mechanical point of view, the human heart is the pump of the circulatory system. It consists of two atria and two ventricles. The contraction of the heart is initiated at the so-called *sinoatrial node (SA node)*, an area of self-excitable cells within the right atrium known as *P wave*. The electrical impulses spread through the atria and ventricles with a dedicated timing characteristics (see the *electrocardiogram (EKG)* and shown in Figure 1), thereby causing the contraction of the chambers.

Normally, the SA node generates electrical impulses with a frequency of 60-100 beats per minute. A too low or sporadically missing pulse generation is called *bradycardia*. In order to support the natural pulse generation notably for bradycardia, artificial cardiac pacemakers are implanted nowadays. Artificial pacemakers have to respect the timing characteristics of the sinus rhythms as it is critical. First and foremost, pulses must not be generated within the refractory intervals after depolarization, as this may cause life-threatening cardiac fibrillation.

3.2 Informal Specification of a Pacemaker

In this paper we mainly describe a modern, atrium-controlled DDD pacemaker [10], although we analyzed a whole family of pacemakers. Beginning with the least complex A00/V00 and D00 pacemakers which stimulate the heart periodically with a fixed time interval, over AAI/VVI pacemakers which sense the chamber's signals and stimulate one of the chamber only when a signal is missing, till the most complex pacemaker DDD monitoring and stimulating both chambers.

DDD means dual pacing, dual sensing, and dual response mode, see the NBG code for details of the configuration. A DDD pacemaker senses both right chambers and can also stimulate them both, but only if no natural pulses are detected. The DDD pacemaker basically uses two timers to monitor time intervals: The *base* interval is the period between two subsequent P waves, natural or artificial, of the atrium. The *AV interval* is the time between a stimulation of the atrium and the consecutive ventricle pulse, the *QRS complex*. If the base interval expires without sensing a natural P wave in the atrium, an artificial impulse is generated in the atrium. Then the ventricle is monitored and only in case of no natural pace within the AV interval, an artificial ventricle pulse is generated. Both timers are reset in case an appropriate natural impulse is sensed. In addition, the base interval is reset if a ventricular extrasystole occurs. Then the base interval is restarted without starting the AV interval timer again. In case the SA node generates a natural pace, the AV interval is adapted (so-called AV hysteresis) in the next base period. For now, we consider the base interval and thereby the pace frequency to be fixed.

4 The Safety Process

As prescribed in the safety standards [7, 2], the system development is complemented by a safety analysis that identifies hazards and traces them back to potential failures. From the identified hazards system safety requirements are derived to eliminate failures or mitigate their effects. In consistency with the

architectural decomposition of the system into components, the safety requirements are split into sub-requirements that are assigned to individual components. The safety analysis and the decomposition of components and safety requirements are iterated until basic components are derived that can be realized and for which evidence can be provided that they fulfill their safety requirements.

4.1 Safety Analysis

The principal architecture of a pacemaker is depicted in Figure 2. In the safety analysis we concentrate on those hazards and the induced safety requirements that refer to the *functional level* of the software control of the pacemaker whereas the mechanics, the electrics, and the deployment are analyzed no further. We performed an FTA and an FMEA [6] and formalized it according to the DCCA approach [5]. The resulting safety requirements for the functional level of the software control are as follows:

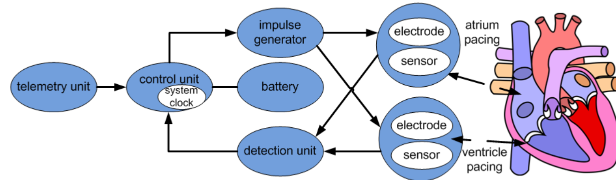


Figure 2: Principal architecture of a cardiac pacemaker

Timed Interrupt: As a pacemaker is limited by its battery and its replacement requires an operation, energy consumption of all components has to be kept as low as possible. *Refractory periods*: Within the refractory periods after the atrium (ARP) and ventricle pace (VRP), detection and impulse generation have to pause in that chamber in order to guarantee that neither an artificial pulse is sensed nor disturbances after depolarization are misinterpreted. *Time intervals BI, AVI + AVH:* The timing constraints as the base interval, the AV interval with the AV hysteresis and their sequencing are respected within specified tolerances. *Pacing:* An artificial atrium pace is triggered if the base interval expires without sensing a natural P wave. An artificial ventricle impulse is generated if the AV timer has been started and expires without sensing a natural pace there. If detection is active it suspends the impulse generation for that chamber and vice versa.

4.2 Safety Design

When using SCADE suite, the C-code generated from the SCADE design model is embedded into a wrapper that is usually periodically executed. Thereby it is ensured that the model reacts synchronously. Within each execution cycle all input signals are read and all output signals are written. For the pacemaker, the control logic should be executed only if a control state or one of the output signals is about to change. In those cycles in which both refractory periods overlap and the software control only waits for the ARP timer to expire, model execution may pause in order to save energy. Thus in order to meet the efficiency request *Timed interrupt* we decided to handle timers and events (from the detection unit) outside the SCADE model and call the inner part of the control modeled in SCADE only to react on timeouts and event occurrences. As a consequence the verification task has to be decomposed: Since the real-time behavior know is realized by both, the SCADE model and the wrapper, it can be proven correct only partially on the SCADE level. The overall reactivity has to be verified separately. For this task we will employ UPPAAL [12], a model checker based on timed automata that is capable to deal with real-time constraints. The safety requirements referring to the inner control logic are proven using SCADE Design Verifier. The inner control is modeled as a state machine on the top level (see Figure 4).

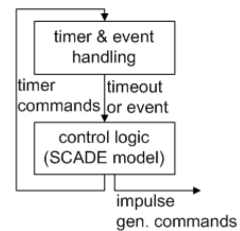


Figure 3: Event handling

In the first glance, the alternative, namely doing the control fully within SCADE seems to have the advantage of a simple and seamless design *and* verification methodology. However, a pure SCADE solution suffers not only from inefficient execution but verifying the timing behavior with SCADE Design Verifier is seriously affected by complexity problems: In a pure SCADE model, the system clock is referred as an external signal that has to be compared with the timeout value (integer) in each execution cycle. From the design perspective this corresponds to polling. When the state space is explored for verification, all intermediate states are traversed which corresponds to successively increasing the timer cycle by cycle. SCADE Design Verifier provides SAT-based model checking performed on the transition graph representing the formal semantics of the design model. The SAT-Solver is enhanced by a number of built-in abstraction techniques, in particular integer linear arithmetics [1]. We had expected that to be an effective tool, in particular for handling the comparisons with timing constants. However, it was not able to cope with this way modeling the timers.

We have shown in [3] that time abstraction is very promising in order to cope with this kind of state explosion caused by a real-time clock.

4.3 Safety Assurance

4.3.1 Verifying the Timing Constraints

The timer and event handling can be modeled in UPPAAL in a straightforward manner: Each relevant interval is monitored by a clock, events like the sensing or stimulating a pace are modeled as communication and the statuses of these timers are represented in variables. The possible statuses of a timer are *inactive, init, active*, and *timeout*, and they can be considered as the input signals for the control logic modeled in SCADE. The simplest way to complete the UPPAAL model is to construct a timed automata for the inner control logic as well by using the model transformation we proposed in [3]. The real-time constraints are expressed using the UPPAAL query language. At this level the real-time constraints are specified stating that within certain real-valued intervals certain timer statuses are set, or certain events must or must not occur. E.g. whenever a ventricle pace has been sensed or stimulated the timer monitoring the ventricle refractory period is set active for a time constant VRP.

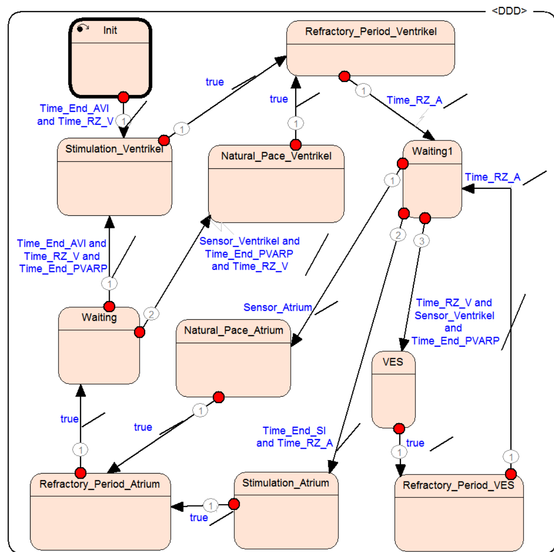


Figure 4: DDD pacemaker: Inner control logic

4.3.2 Verifying the Control Logic

We verified several safety requirements referring to the control logic that we derived in the safety analysis. During the verification process the SCADE Design Verifier tries to find an input configuration which changes the output of the proof obligation from true to false. If the output stays true for each possible input configuration, the output is considered as valid. First pausing of the detection and impulse generation

	Natural pace	**Natural pace atrium**	**Refractory period**	**No VES**	**Hysteresis**	**One pace**
VVI	x (0 s)		x (0 s)			x (3 s)
AAI	x (0 s)		x (0 s)			x (3 s)
DDD	x (1 s)	x (0 s)	x (0 s)	x (1 s)	x (0 s)	x (1 s)

Table 1: The correlation of constraints and pacemaker modes, the runtime is shown in brackets

during refractory periods is verified. For this we argue that it shall always be true that while a timer for a refractory period is running, neither the sensor is active nor a stimulation is triggered. We verified the pacing requirement for each chamber separately. Moreover, the pacing requirement is divided into two parts: The first sub-requirement concerns the case of a natural pace, the second one the artificial stimulation. For the first part we argue similar to the previous constraint that it is not possible that the statuses of the timers allow pace sensing and a pace is sensed as well as a stimulation takes place. We call the constraints *(*natural pace BI) for the base interval and *natural pace AVI* for the AV interval. Furthermore we have a constraint called *refractory period* in which we verify that during the refractory periods no sensing or stimulation takes place. For the second part we argue that if no natural pace is sensed during the corresponding base or AV interval, exactly one artificial pace takes place. This constraint is called *one pace*. To verify this requirement we created an operator in which the natural and artificial paces during one interval are counted. For the timing constraint AVH we verified that after sensing a natural ventricle pace the next AV interval will be prolonged. This is done by saving the ventricle pace and waiting for the beginning of the AV interval. At that point we control the length of the AV interval. Furthermore we verified the correct handling of a ventricular extrasystole (VES). This constraint is called *VES*. In the corresponding proof operator we determined that it shall not be possible that a VES and an atrial pace (natural or artificial) take place together in one base interval. For this we memorize in an operator the paces that occurred within an base interval. As result we received for all six proof obligations to be valid. The runtimes for each verification are shown in table 1. Due to the particular characteristics of the different pacemaker variants, not every constraint is required and consequently guaranteed for every pacemaker. In table 1 we oppose the constraints for the AAI/VVI pacemaker with the ones for the DDD pacemaker.

If we fully integrated the timers into the pacemaker logic, the verification process did not terminate within a reasonable time of two days. We set this boundary out of our experience with the SCADE Design Verifier in combination with the model complexity. If we wait longer for results we experienced no results at all because of memory overflow or buffer overflow. This negative result also justifies the architectural design as depicted in Figure 3.

5 Conclusion

We sketched how to systematically develop a safety-critical embedded system using formal methods. We employed the SCADE suite for modeling and code generation for the inner control logic, as SCADE is qualified to the most relevant safety standards like IEC 61508 and RCTA DO 178-B. Timers and events were handled in an outer control loop. This decomposition was motivated by an efficiency requirement. For verification purposes it can be understood as a time abstraction which turned out to be a useful also for proving the real-time behavior correct. The approach presented here extends the ideas of time

abstraction we presented in [3]. In that work, pure SCADE models were embedded into a standard cyclic wrapper and the time abstraction was performed on the SCADE model. Here timers and simple event handling are transferred to the wrapper on which is an abstraction to UPPAAL is applied.

In the full version of this paper, the safety constraints referring to real-time behavior as well as to the control logic will be detailed. Moreover, an alternative proof strategy based on assume-guarantee style will be explored.

References

[1] Parosh Aziz Abdulla, Johann Deneux, Gunnar Stålmarck, Herman Ågren & Ove Åkerlund (2004): *Designing Safe, Reliable Systems Using Scade*. In Tiziana Margaria & Bernhard Steffen, editors: *ISoLA*, *LNCS* 4313, Springer, pp. 115–129. Available at `http://dx.doi.org/10.1007/11925040_8`.

[2] Intern. Electrotechnical Commission (2010): *IEC 61508-3:2010: Functional safety of electrical/electronic/programmable electronic safety-related systems Part 3: Software requirements*.

[3] Ilays Daskaya, Michaela Huhn & Stefan Milius (2011): *Formal Safety Analysis in Industrial Practice*. In Gwenn Salaün & Bernhard Schätz, editors: *16th Intern. Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, LNCS 6959, Springer, pp. 68–84.

[4] Esterel Technologies (2009): *SCADE Suite KCG 6.1: Safety Case Report of KCG 6.1.2*.

[5] Matthias Güdemann, Frank Ortmeier & Wolfgang Reif (2007): *Using deductive cause-consequence analysis (DCCA) with SCADE*. In: *Proc. 26th Intern. Conference on Computer Safety, Reliability and Security (SAFECOMP)*, *Lecture Notes Comput. Sci.* 4680, Springer, pp. 465–478.

[6] International Electrotechnical Commission (2006): *IEC 60812: Analysis Techniques for System Reliability*.

[7] International Electrotechnical Commission (2006): *IEC62304: Medical device software - Software life-cycle processes*.

[8] Eunkyoung Jee, Insup Lee & Oleg Sokolsky (2010): *Assurance Cases in Model-Driven Development of the Pacemaker Software*. In Tiziana Margaria & Bernhard Steffen, editors: *4th Intern. Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, *LNCS* 6416, Springer, pp. 343–356.

[9] Frank Ortmeier, Wolfgang Reif & Gerhard Schellhorn (2006): *Deductive Cause Consequence Analysis (DCCA)*. In: *Proc. IFAC World Congress*, Elsevier, Amsterdam.

[10] Boston Scientific (2007): *PACEMAKER System Specification*.

[11] Luu A. Tuan, Man C. Zheng & Quan T. Tho (2010): *Modeling and Verification of Safety Critical Systems: A Case Study on Pacemaker*. In: *4th Conf. on Secure Software Integration and Reliability Improvement*, IEEE, pp. 23–32.

[12] (2009): *UPPAAL 4.0: Small Tutorial*. `http://www.it.uu.se/research/group/darts/uppaal/small_tutorial.pdf`. November 16, 2009.

Admissible adversaries in PRISM for probabilistic security analysis

Alain-Freddy Kiraga and John Mullins [1,2]

Dept. of Comp. & Soft. Eng.
École Polytechnique de Montréal
Campus of the Université de Montréal
Montreal (Quebec), Canada

Abstract

In order to resolve the non-determinism when dealing with systems exhibiting both probabilistic and non-deterministic behavior, a device called *scheduler* has been introduced. In the context of security analysis, systems are assumed to run in a hostile environment also it is quite natural to consider the scheduler to be under control of the adversary. However if not constrained, schedulers gives the adversary an unreasonably strong power as they can reveal secret information even for obviously secure systems. In this paper, we propose an automata theoretic model for limiting this power by defining two levels of scheduling. A cooperative scheduler resolves probabilistically (internal) non-determinism over observationally equivalent actions specified my mean of an equivalence relation on protocol's action set while a adversarial or admissible scheduler resolves the (external) remaining non-determinism. We then present efficient implementation techniques for embedding the model into the PRISM model checker by using the symbolic approach based on MTBDDs.

Key words: Security analysis; Markov Decision Processes.

1 Introduction

Formalisms that combine both probabilistic and nondeterministic behavior are very convenient for modeling probabilistic security systems like security protocols and particularly those for anonymity and fair exchange as they usually use randomization to meet their security requirements. Many such formalisms have been proposed in automata theory [18] and in process algebra [1]. See also [19] for comparative overviews.

[1] Author partially supported by the NSERC of Canada under discovery grant No.13321-2011.

[2] Email: john.mullins@polymtl.ca

While it's customary to use schedulers for resolving non-determinism in probabilistic systems, scheduling process must be carefully designed in order to reflect as accurately as possible the intruder's capabilities to control the communication network without controlling the internal reactions of the system.

In this paper we propose an automata theoretic approach to the problem of hiding the outcome of internal random choices and investigate whether this method can be implemented and used in a probabilistic model checker like PRISM. Currently, PRISM considers all schedulers when verifying a specification without the possibility to restrict to a subset of them. In order to realize that, the first issue is to formulate the method in the context of Markov Decision Processes, the formalism used for modeling in PRISM, and verify the soundness of the method. The next issue is to implement efficiently the method by using the symbolic implementation techniques used in PRISM and based on manipulations of MTBDD encodings. Both issues are carefully addressed and proposed solutions are detailed.

Related works. Many works have proposed different methods for calibrating schedulers in order to restrict power of adversaries during the analysis of security systems. The most closely related to ours are [5,10,16,6,12,11]. In [5], the approach is presented in the context of probabilistic I/O automata and is also based on a bi-level scheduling process where the set of actions is partitioned in tasks. The order of execution of tasks is static and performed by a so-called task scheduler. The remaining non-determinism within a task is resolved by a second scheduler modeling the standard deterministic adversarial scheduler. This approach is somewhat orthogonal to ours as in our approach we impose the static resolution of non-determinism within a task by an arbitrary probabilistic internal scheduler while the task scheduling is deterministic. In [10], a class of probabilistic schedulers, called admissible schedulers, is defined in the context of probabilistic automata. An admissible scheduler assigns, in the current state, a probability distribution on the possible non-deterministic next transitions. Unlike our scheduler, it is history-dependent since it defines equiprobable paths and it is not stochastic, and might therefore halt execution at any time. Roughly speaking, an admissible scheduler schedules in the same way any trace equivalent paths leading to bisimilar states. In [16], the authors use an extension of the CCS process algebra with finite replication and probabilistic polynomial-time terms (functions) denoting cryptographic primitives to better take into account the analysis of cryptographic protocols. In order to constrain schedulers, priority is always given to internal actions over external communications and non-determinism between internal actions is resolved uniformly. By contrast, our approach allows any probabilistic resolution of internal non-determinism. Moreover this approach avoids the problem of extra power but can also weaken the scheduler. In [12], it is proposed a variant of [16] intended to reflect in a more accurate way the intruder's real power. In the process algebraic approach proposed in [6], the control on the scheduler can be

specified by terms and hence, may be seen as dual to our semantic approach. It is provided a way to specify the deterministic choice that should be invisible to the adversary rather then a restriction to the class of unconstrained schedulers. Note that we achieve the same goal for more general schedulers with our observation classes of actions. In [11], it is considered several classes of distributed schedulers. Distributed schedulers were proposed in order to avoid considering unrealistic power of unconstrained schedulers. In this setting, roughly speaking, there is a local scheduler for each component hence, the resolution of non-determinism is distributed among the different components. It is proved that randomization adds no extra power to distributed schedulers that is, the subclass of schedulers that are both history-based and Dirac are sufficient to reach supremum probabilities of any measurable set. However non-state-based schedulers are required to reach supremum probabilities in distributed systems. It is also proved that for the class of strongly distributed schedulers that constrains the non-determinism concerning the order in which components execute, as it is the case in [5,12] and in this work, randomized and non-state-based are required to reach supremum probabilities. Recall that for the class of unconstrained schedulers, as the one considered in PRISM, schedulers that are both state-based and Dirac are sufficient to reach these suprema.

Contributions of the paper. The contributions of the paper are the following: A Markov decision process based security model embedding the model of the system together with the model of a system-dependent calibrated probabilistic adversary which generalizes [5,12] to arbitrary probabilistic internal schedulers; An efficient implementation of the security model using the symbolic approach into the PRISM model checker which, to our knowledge, enhances PRISM as the first automated tool for analyzing probabilistic security systems;

Content of the paper. In next section we briefly recall basic notions related to discrete-time Markov chains and Markov decision processes and their encoding using binary decision diagrams and multi-terminal binary decision diagrams. In Section 3, we define in the context of the Markov decision processes, a security model taking in account together a description of the system and what should be considered as observation equivalent to any adversary. The model is then proved to be sound in this context. The generalization of a bi-level scheduling process first proposed in [5,12] for uniform internal schedulers, leads to the definition of a class of admissible system's adversaries. In Section 4, symbolic implementation techniques for embedding the security model into the model checker PRISM are presented, namely, the MTBDD-based algorithms requested for constructing and manipulating the security MDP from the general MDP model currently processed by PRISM. Section 5 concludes the paper.

2 Preliminaries

In this section we recall basic definitions for discrete-time Markov chain, Markov decision process [2] and their representation using symbolic approaches [4,9].

2.1 Discrete-time Markov chains and Markov decision processes

Probability measure. Given a set X, a *σ-field* over X is a set $\mathcal{F} \subseteq 2^X$ that includes X and closed under complement and countable union. We call *measurable space* the pair $(X, \mathcal{F})$ where X is a set and $\mathcal{F}$ is a σ-field over X. A measurable space $(X, \mathcal{F})$ is discrete if $\mathcal{F}= 2^X$. A *discrete probability measure*, called also *probability distribution*, over a discrete measurable space $(X, 2^X)$ is a function $\mu : 2^X \to [0, 1]$ such that $\mu(X) = 1$ and $\mu(\cup_i X_i) = \sum_i \mu(X_i)$ where $\{X_i\}$ is a countable family of pairwise disjoint subsets of X. We denote by $Disc(X)$ the set of all discrete probability distributions over the set X. We denote by $\sum_{i=1}^n p_i\mu_i$ the convex sum of measures μ_i. It can easily be shown that the convex sum of measures is a probability measure. The *Dirac measure* on μ_i, denoted $\delta(\mu_i)$, is the convex sum $\sum_{i=1}^n p_i\mu_i$ with $p_j = 0$ if $j \neq i$. The *uniform* measure on $\mu_1, \mu_2, \ldots, \mu_n$, denoted $\mathcal{U}(\{\mu_i : 1 \leq i \leq n\})$ is the convex sum $\sum_{i=1}^n p_i\mu_i$ with $p_i = \frac{1}{n}$ for $1 \leq i \leq n$.

Probabilistic models. We briefly summarize here the models we use for analyzing probabilistic protocols. Let AP be a fixed finite set of atomic propositions and Act a fixed finite set of actions defined on the system. A *discrete-time Markov chain* (DTMC for short) is a tuple $\mathcal{M} = (S, s_{\text{init}}, \Delta, L, Act)$ where S is a finite set of states, s_{init} a initial state, $\Delta : S \to Act \times Disc(S)$ a transition function and $L : S \to 2^{\text{AP}}$ a function which assigns to each state $s \in S$ the set $L(s)$ of atomic propositions valid in state s. DTMC is the simplest model supporting only probabilistic behavior. The transition function Δ can be more conveniently represented as a pair of functions from S to Act and $Disc(S)$ respectively. Also, by abuse of notation we will refer to Δ as the distribution component of this pair while Δ_{Act} will refer to the action component.

A *run* in $\mathcal{M}$ is a sequence of transitions written as: $\rho = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \cdots \xrightarrow{a_n} s_n$. For such a run, $\text{fst}(\rho)$ (resp. $\text{lst}(\rho)$) denotes s_0 (resp. s_n). We will also write $\rho \cdot \rho'$ for the run obtained by concatenating runs ρ and ρ' whenever $\text{lst}(\rho) = \text{fst}(\rho')$. The set of finite runs starting in state s is denoted by $Run_s(\mathcal{M})$ and $Run(\mathcal{M})$ denotes the set of finite runs starting from an initial state. The *trace* of a run $\rho = s_0 \xrightarrow{a_1} s_1 \cdots \xrightarrow{a_n} s_n$ is the word $\text{tr}(\rho) = a_1 \cdots a_n \in Act^*$. The set of traces of runs in $Run(\mathcal{M})$ is noted $Tr(\mathcal{M})$.

Markov decision process (MDP for short) extends DTMC model by allowing both probabilistic and non-deterministic behavior. A MDP is a tuple $\mathcal{C} = (S, s_{\text{init}}, \Delta, L, Act)$ where AP, Act, S, s_{init} and L are defined as in DTMC and a transition function $\Delta : S \times Act \to Disc(S)$. MDP allows non-deterministic choice between distinct actions enabled in a state, each one

associated with a probability distribution. Runs and traces in $\mathcal{C}$, $Run_s(\mathcal{C})$, $Run(\mathcal{C})$and $Tr(\mathcal{C})$ are defined like in DTMC.

A scheduler selects probabilistically a transition among the ones available in $\mathcal{C}$ and it can base its decision on the history of runs. Given an MDP $\mathcal{C} = (S, s_{\text{init}}, \Delta, L, Act)$, a *scheduler* on $\mathcal{C}$ is a function $\xi : Run(\mathcal{C}) \to Disc(Act)$ such that $\xi(\rho)(a) \geq 0$ implies that $\Delta(\text{lst}(\rho), a)$ is defined. We denote by $\mathcal{C}_\xi$, the DTMC generated by ξ scheduling $\mathcal{C}$ and by $Sched(\mathcal{C})$, the set of schedulers of $\mathcal{C}$.

2.2 Decision diagrams

Binary decision diagrams (BDDs) [4] are data structures representing boolean functions $f\colon \mathbb{B}^n \to \mathbb{B}$ that are defined on ordered boolean variables $x_1 < x_2 < \ldots < x_n$. A BDD representation for f is an acyclic rooted directed graph resulting from a folding of the binary decision tree obtained by using repeatedly the Shannon expansion of f for variable x_i, $f = (\neg x_i \wedge f_{|x_i=0}) \vee (x_i \wedge f_{|x_i=1})$, for $i = 1$ to n, such that $f_{|x_i=c}$ is the function f where all instances x_i are evaluated to value c. More formally, given a set $Var = \{x_1, x_2, \ldots, x_n\}$ of boolean variables and an ordering relation $<$ over Var, a BDD is a tuple $(V, V_{\text{I}}, V_{\text{T}}, var, val, v_0)$ where V is a set of vertices, $V_{\text{I}} \subseteq V$ is the set of non-terminal vertices (each vertex $v \in V_{\text{I}}$ has an argument $index(v)$ representing the index of the variable and two children $Then(v), Else(v) \in V$), $V_{\text{T}} \subseteq V$ is the set of terminal vertices, $var : V_{\text{I}} \to Var$ (with $var(v) < var(w)$ for each non-terminal vertex v and its non-terminal child w), $val : V_{\text{T}} \to \mathbb{B}$ and v_0 is the root.

Multi-terminal binary decision diagrams (MTBDDs) [9] are the extension of BDDs by allowing terminals to be labeled with numeric values from an arbitrary set $\mathbb{D}$ (e.g. $[0, 1]$). More formally, given $Var = \{x_1, x_2, \ldots, x_n\}$ a set of boolean variables and $\mathbb{D}$ a domain of numeric values, MTBDDs are representations of the functions $f\colon \mathbb{B}^n \to \mathbb{D}$. Reduction rules and variables ordering defined on BDDs are also used. Specifically, MTBDDs are very convenient for representing sparse matrices in a more compact manner and can be used for representing efficiently transition functions of very large systems.

We summarize here some useful operations for our algorithms. These operations are implemented in the CUDD package [20] or in PRISM [15]. For the following, we assume (MT)BDDs M, M_1 and M_2 defined on boolean variables $x_1, x_2, \ldots, x_n$.

- $Const(c)$ where $c \in \mathbb{R}$, creates a new MTBDD with the constant value c.
- $Abstract(op, (x_1, x_2, \ldots, x_m), \text{M})$ where op is a binary operator over the reals or the boolean, returns the (MT)BDD resulting of abstracting $x_1, x_2, \ldots, x_m$ from M by applying op over all possible values taken by $x_1, x_2, \ldots, x_m$.
- $Apply(op, \text{M}_1, \text{M}_2)$ where op is a binary operator over the reals or the boolean, returns the (MT)BDD representing the function $f_1 \; op \; f_2$ such that f_1 and f_2 are respectively encoded by M_1 and M_2.

- $GreaterThan(\mathrm{M}, c)$ where $c \in \mathbb{R}$, returns the BDD where terminals greater than c in the MTBDD M are set to 1 and remaining terminals to 0.
- $MatrixMultiply(\mathrm{M}_1, \mathrm{M}_2, varsZ)$ where $varsZ$ is the set of summation boolean variables, returns the MTBDD representing the matrix product of matrices encoded by M_1 and M_2.
- $SetMatrixElement(\mathrm{M}, rVars, cVars, line, column, c)$ where M is encoded on row variables set $rVars$ and column variables set $cVars$, inserts the value $c \in \mathbb{R}$ in M at the position indexed by integer values $line$ and $column$.
- $GetMatrixElement(\mathrm{M}, rVars, cVars, line, column)$ where M is encoded on row variables set $rVars$ and column variables set $cVars$, returns the value from the position indexed by integer values $line$ and $column$.

We refer to [17] for detailed Information concerning the methods used in PRISM tool [15] to represent probabilistic systems in memory but by sake of completeness, we summarize in Appendix the very basics restricted to the notions useful for the presentation of the algorithms in Sect. 4.

3 Admissible adversaries

In this section, we define what is a class of adversaries admissible by a given probabilistic internal scheduler by means of a bi-level scheduling process which extends to Markov Decision Processes the scheduling process proposed in [12] in the framework of process algebra extended with cryptographic primitives. In order to do this, we first define a variant of a MDP called security MDP. It is a MDP together with an (static) observation relation over actions specifying the observable of MDP's actions from an adversary point of view.

Definition 3.1 A security MDP is a pair $\mathcal{S} = (\mathcal{C}, \mathcal{O})$ where $\mathcal{C}$ is an MDP and $\mathcal{O}$, an equivalence relation over *Act* called observation relation. An equivalence class o of $\mathcal{O}$ is called an $\mathcal{O}$-observable. The set of observables is denoted $Obs_{\mathcal{O}}$.

We now define a (static) internal scheduler for a security MDP as a scheduler resolving internal non-determinism on the base of any probabilistic static policy depending only on the subsets of observationally equivalent actions available at anytime.

Definition 3.2 Let $\mathcal{S} = (\mathcal{C}, \mathcal{O})$ be a security MDP. A *(static) internal scheduler* for $\mathcal{S}$ is a function $\xi_{int} : Task_{\mathcal{O}} \to Disc(Act)$ where $Task_{\mathcal{O}} = \bigcup_{o \in Obs_{\mathcal{O}}} 2^o$ is called the set of tasks for ξ_{int}. We denote by $Sched_{\mathrm{int}}(\mathcal{S})$ the set of all internal schedulers for $\mathcal{S}$.

We are now ready to define the MDP as observed by an adversary after the enforcement of the obfuscation policy by the internal scheduler.

Definition 3.3 Given a security MDP $\mathcal{S} = (\mathcal{C}, \mathcal{O})$ with MDP $\mathcal{C}$ defined as $(S, s_{\mathrm{init}}, \Delta, \; L, Act)$, the ξ_{int}-observable MDP is the MDP $\mathcal{C}_{\xi_{int}}$ defined as

$(S, s_{\text{init}}, \Delta_{\xi_{int}}, L, Obs_{\mathcal{O}})$ where

$$\Delta_{\xi_{int}}(s, o) = \sum_{\alpha \in o \cap Actin(s)} \xi_{int}(o \cap Actin(s))(\alpha) \cdot \Delta(s, \alpha)$$

with $Actin(s) = \{\alpha \in Act : \Delta(s, \alpha) \text{ is defined}\}$.

The following Proposition states that $\mathcal{C}_{\xi_{int}}$ is well-defined, that is, $\Delta_{\xi_{int}}(s, o)$ is in $Disc(S)$ and hence, is obtained from $\mathcal{C}$ by resolving all internal non-determinism inside each class of observables available in a given state by following the strategy of ξ_{int}.

Proposition 3.4 *$\mathcal{C}_{\xi_{int}}$ is well-defined.*

Proof This follows from the definition of ξ_{int} and also from the fact that $Disc(S)$ is closed under the convex sum and the observation that $o \cap Actin(s)$ specifies exactly the set of transitions of $\mathcal{C}$ available in s and observed as o. □

It follows from Prop. 3.4 that the only remaining non-determinism is external:

Corollary 3.5 *For every state $s \in S$ and observable $o \in \mathcal{O}$, at most one action $\alpha \in o$ is enabled in s.*

We then call scheduler admissible for ξ_{int}, any scheduler resolving the remaining (external) non-determinism in the system as observed by the adversary.

Definition 3.6 Let a security MDP $\mathcal{S} = (\mathcal{C}, \mathcal{O})$ and an internal scheduler for $\mathcal{S}$. An *admissible scheduler* of $\mathcal{S}$ for ξ_{int} is a scheduler of $\mathcal{C}_{\xi_{int}}$.

Hence, we get that, given an admissible scheduler ξ_{adm} of $\mathcal{S}$ for ξ_{int}, $\mathcal{C}_{\xi_{int}\xi_{adm}}$ is generated by a scheduler for $\mathcal{C}$, in the usual sense. That is, ξ_{adm} is a special case of a scheduler for the underlying MDP $\mathcal{C}$.

Proposition 3.7 *Let $\mathcal{S} = (\mathcal{C}, \mathcal{O})$ be a security MDP. For each internal scheduler $\xi_{int} \in Sched_{\text{int}}(\mathcal{S})$ and admissible scheduler $\xi_{adm} \in Sched(\mathcal{C}_{\xi_{int}})$ of $\mathcal{S}$ for ξ_{int} there is a scheduler $\xi \in Sched(\mathcal{C})$ such that $\mathcal{C}_{\xi_{int}\xi_{adm}}$ is $\mathcal{C}_{\xi}$.*

Proof The function $\xi : Run(\mathcal{C}) \to \mathcal{D}(Act)$ defined as

$$\xi(\rho)(\alpha) = \xi_{int}([\alpha]_{\mathcal{O}} \cap Act(\text{lst}(\rho))(\alpha) \cdot \xi_{adm}(\rho)([\alpha]_{\mathcal{O}})$$

is this scheduler. Indeed, suppose that $\xi(\rho)(\alpha) > 0$ then we have that $\xi_{int}([\alpha]_{\mathcal{O}} \cap Act(\text{lst}(\rho))(\alpha) > 0$ and also that $\xi_{adm}(\rho)([\alpha]_{\mathcal{O}}) > 0$ hence, $[\alpha]_{\mathcal{O}} \in Act(\text{lst}(\rho))$ since ξ_{adm} is a scheduler and $\alpha \in Act(\text{lst}(\rho))$. □

The design of the internal scheduler is highly dependent on the modeled security system and, as it is shown in the following example, probabilistic internal schedulers are needed to achieve the minimum information leakage.

Consider the security MDP $\mathcal{S}$ given in Fig. 1 where $\mathcal{O} = \{o_1, o_2\}$ with $o_1 = \{a, b\}$ and $o_2 = \{c\}$ modeling a system sending a's or b's until a successfull transmision. A success occurs in 75% of the cases for a and 25% of the cases for b. At each step, the probability of success for a is three times the probability of success for b. It is easily seen that any internal scheduler of $\mathcal{S}$ has the form $\xi_p(o_1) = p\mu_a + (1-p)\mu_b$ for $0 \leq p \leq 1$ where μ_a and μ_b denote $\Delta(s_{\text{init}}, a)$ and $\Delta(s_{\text{init}}, b)$ respectively, for the transition function Δ depicted in Fig. 1. Also (internally) scheduling a with a probability three times less than b makes the event *a is transmitted successfully given o_2 is observed* equiprobable with the event *b is transmitted successfully given o_2 is observed*. Hence, $\xi_{\frac{1}{4}}$ is required in order to preserve the secrets a and b face to an observer able to infer from statistics on a and b.

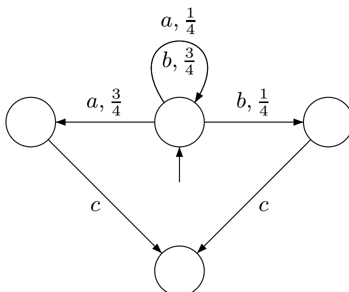


Figure 1. Security MDP

This discussion can be summarized in the following proposition:

Proposition 3.8 *There exist a security MDP $\mathcal{S} = (\mathcal{C}, \mathcal{O})$ with $\mathcal{C}$ defined as $(S, s_{\text{init}}, \Delta, L, Act)$ and a probabilistic scheduler $\xi_{rnd} \in Shed_{int}(\mathcal{S})$ such that for any deterministic scheduler $\xi \in Shed_{int}(\mathcal{S})$, $\mathcal{C}_\xi$ leaks more information than $\mathcal{C}_{\xi_{rnd}}$.*

However, as we will illustrate in Sec B, the case with ξ_{int} defined as $\xi_{int}(t) = \mathcal{U}(t)$ for any $t \in Task_{\mathcal{O}}$ is pretty well suited for the large class of security MDP modeling security systems whose the only secret actions are outcomes of random experiences (e.g. rolling a dice, generating a nonce or tossing a coin) it needs to perform in order to achieve its security goal and then proceeds on the base of the outcome [3]. An adversary trying to control such a system in order to get the secret information which should appear undistinguishable to the environment, is required to never be able to perform better then a random guess among all the possible outcomes. Also, this leads to an admissible scheduler that gives the adversary full control of its own actions but does not allow to control internal actions of the security system. Moreover, the defense

[3] Note that the security MDP depicted in Fig. 1 does not model such a system and that in this case, scheduling a and b uniformly would not preserve the secrecy of a and b.

strategy required here is static as any occurrence of any outcome anytime along the system execution should be obfuscated. In the next section we implement this scheduler.

4 Algorithms and implementation

This section describes the implementation of the method for constructing the security MDP in the model checker PRISM. Fig.2 represents the PRISM architecture extended with the module performing the construction of our security model from the general MDP model as described in Section 3. Both the module and its source code can be downloaded from [14]. Basically, the model

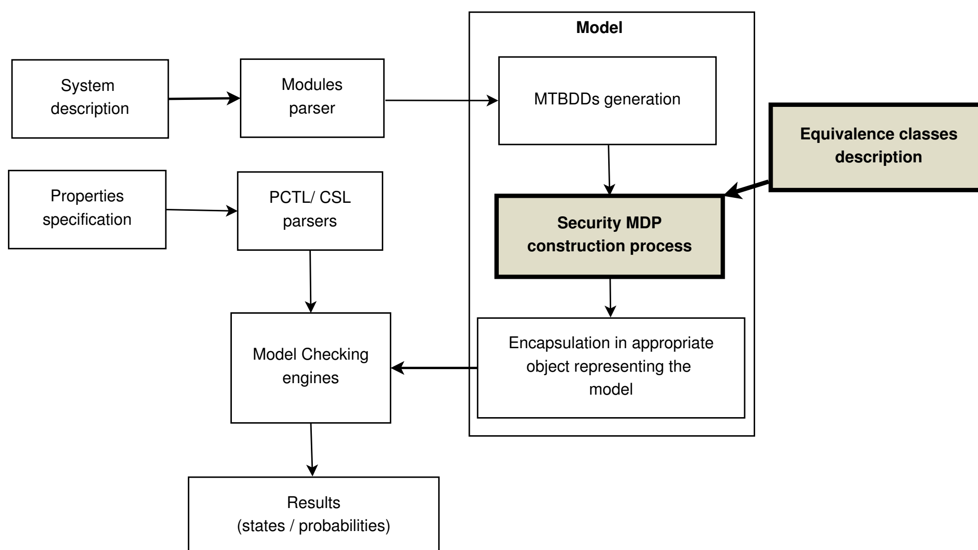


Figure 2. Security MDP construction process in PRISM architecture

checker PRISM parses the system description provided by users and generates different MTBDDs representing the system (transition function, rewards, reachable states, etc.). These data structures are then encapsulated into the object model depending on the type of the model (probabilistic for DTMC model, non-deterministic for MDP model and stochastic for CTMC model which is outside the scope of this paper). Then, by mean of one out of three engines (*MTBDD engine*, *Sparse engine* or *Hybrid engine*), model checking can be performed on the object model and the PCTL [13] or CSL property specification (depending on the type of the model). Results are finally returned as set of states or probabilities.

We have chosen to integrate the building of the ξ_{int}-observable MDP after the MTBDDs generation process has taken place and before their encapsulation in the appropriate model for two reasons. First, we want to integrate it as an option we can enable or disable from user interfaces. If enabled, the method constructs new MTBDDs representing the ξ_{int}-observable MDP ac-

cording to the set $Obs_{\mathcal{O}}$ of observables described in a file loaded by PRISM tool. The second reason is to preserve the same object model structure as used originally in PRISM in order to re-use the built-in PRISM algorithms for model checking the resulting ξ_{int}-observable MDP model.

4.1 Merging directives algorithm

In this section we describe the algorithm used to construct an MTBDD encoding information identifying for each state s and observable o the set of distributions $\{\Delta(s,a) : a \in o\}$ corresponding to the actions $\mathcal{O}$-observable by o in s. This data structure is very useful to the algorithm shown in Section 4.2 which performs the combination in order to compute the relative probability distributions on observables in the ξ_{int}-observable MDP model. Fig. 3 shows the algorithm for constructing this data structure.

The algorithm takes as in input the MTBDD M_{f_Δ} encoding the distribution component function f_Δ of the MDP $\mathcal{C}$, the vector $(\mathrm{M}_{f_\alpha})_{\alpha \in Act}$ of BDDs representing the family $\{f_\alpha : \mathbb{B}^n \times \mathbb{B}^{nd} \rightarrow \mathbb{B}\}_{\alpha \in Act}$ and the set $Obs_{\mathcal{O}}$ generated by the observational relation $\mathcal{O}$.

```
GenerateMerg(M_{f_Δ},(M_{f_α})_{α∈Act},Obs_O)
Begin
result=Const(0)
bddStates=Abstact(∨,colVars,Abstract(∨,GreaterThan(M_{f_Δ},0),ndetVars));
bddClasses=Abstract(∨,rowVars,Abstract(∨,GreaterThan(M_{f_Δ},0),colVars));
For i= 0..nbStates−1
  state=Traverse(i,bddStates)
  For j= 0..nbClasses−1
    class=Traverse(j,bddClasses)
    actLabels(j)=SearchAction((M_{f_α})_{α∈Act},state,class);
  endFor;
  tmp=MergingArr(actLabels,Obs_O)
  For j= 0..nbClasses−1
    result=SetMatrixElement(result,rVars,cVars,i,j,tmp(j));
  endFor;
endFor;
return result;
End.
```

Figure 3. Algorithm encoding merging directives in the MTBDD

After creating and initializing the MTBDD *result*, the algorithm computes the BDD representing all reachable states and the BDD representing the transition classes (i.e locations for actions enabled in each state) by performing abstraction of row and column boolean variables on the BDD's version of M_{f_Δ}. The BDD encoding the reachable states is then traversed to compute the values of PRISM variables representing the state. The $Traverse$

procedure, at each iteration, traverses the BDD from the root to a non-zero terminal and returns a list of values representing the state. For each state, the BDD encoding the transition classes is then traversed to compute the integer value representing each transition class. The PRISM values representing the state are combined with each value of the transition class to compute the corresponding action label from $(\mathrm{M}_{f_\alpha})_{\alpha \in Act}$. The computed string is stored in the vector *actLabels*. While some transition classes in the states are dummy as the maximum of locations could be not attained in some states, we need to deal with such cases. In our algorithm, we use a special one-character string $''*''$ for dummy transition classes in the vector *actLabels*. When all action labels for transition classes in the state have been assigned, the function $MergingArr$ allocates to each transition class of the MDP $\mathcal{C}$ the number indicating the transition class of the corresponding observable in the ξ_{int}-observable MDP (i.e. the transition classes labeled with actions having the same $\mathcal{O}$-observable are assigned to the same number). For special cases of dummy transition classes we use the integer value $''-1''$. After assigning such a number to each transition class, one next deals with the encoding of the vector obtained from $MergingArr$ function in the MTBDD $result$. The list of row boolean variables $rVars$ and column boolean variables $cVars$ are computed from the number of states *nbStates* and the maximum number of transition classes *nbClasses* in the state of the MDP model (i.e. $rVars = \{x'_1, x'_2, \ldots, x'_n\}$ and $cVars = \{y'_1, y'_2, \ldots, y'_m\}$ such that $n = \lceil \log_2(nbStates) \rceil$ and $m = \lceil \log_2(nbClasses) \rceil$). After performing this encoding process for each reachable state, the computation of probability distributions on observables can take place.

4.2 Probability distributions on observables in the security MDP

Internal non-determinism in the security MDP needs to be resolved before evaluating probabilistic properties of the model according to admissible schedulers. The algorithm presented in Fig. 4 performs this uniformly and returns the MTBDD $\mathrm{M}_{f_{\Delta_{\xi_{int}}}}$ encoding the distribution component function $f_{\Delta_{\xi_{int}}}$ of the ξ_{int}-observable MDP model $\mathcal{C}_{\xi_{int}}$.

The algorithm takes as in input the MTBDD M_{f_Δ} encoding the distribution component function f_Δ of the MDP $\mathcal{C}$, boolean variables used in M_{f_Δ} (row, column and non-deterministic) and the MTBDD M_{dir} encoding the merging directives as explained in Section 4.1. First, the algorithm decomposes M_{f_Δ} along non-deterministic boolean variables $ndetVars = \{z_1, z_2, \ldots, z_{nd}\}$. The aim of the procedure *DecomposeRec* is to build a vector of MTBDDs, each rooted by a node v such that $var(v) = x_1$. Intuitively each cell of the vector arr stores distributions corresponding to a given transition class in the MDP model.

As illustrated in Fig. 5, the procedure *DecomposeRec* splits the MTBDD M in two sub-MTBDDs M_1 and M_2, each relating to a transition class. The

```
CombinationTrans(M_{f_△},rowVars,colVars,ndetVars,M_{dir})
Begin
DecomposeRec(M_{f_△},ndetVars,arr);
For i=0..nbNewCl−1 newArr(i)=Const(0);
bddStates=Abstact(∨,colVars,Abstract(∨,GreaterThan(M_{f_△},0),ndetVars));
For i=0..nbStates−1
  state=Traverse(i,bddStates)
  selecL=SetMatrixTabElement(Const(0),rowVars,varsZ,state,state,1);
  For j=0..nbClasses−1
    val=GetMatrixElement(M_{dir},rVars,cVars,i,j);
    if(val≠ −1)
     tmp=MatrixMultiply(selecL,arr(j),varsZ);
     newArr(val)=Apply(+,tmp,newArr(val));
    endif;
  endFor;
endFor;
For j=0..nbNewCl−1
  tmp=Abstract(+,newArr(j),tmp);
  newArr(j)=Apply(÷,newArr(j),tmp);
endFor;
result=Const(0);
For j=0..nbNewCl−1
  tmp=GenerateMTBDDClass(j,newNdetVars);
  tmp=Apply(×,tmp,newArr(i));
  result=Apply(+,result,tmp);
endFor;
return result;
End.
```

Figure 4. Algorithm performing the computation of the MTBDD $M_{f_{\Delta_{\xi_{int}}}}$.

integer value encoding each transition class corresponds to the evaluation of the path from the root labeled with z_1 to the node x_1 of the sub-MTBDD. For readability, we have omitted the paths leading to zero-terminals on the Fig. 5.

Before computing the distributions on observables, one needs to create a vector whose components are distributions corresponding to new transition classes of $\mathcal{C}_{\xi_{int}}$. The length of the vector *newArr* is specified as the maximum number *nbNewCl* of new transition classes possible in the ξ_{int}-observable MDP model. For each state, we need to extract and encapsulate in the same transition class all distributions associated with actions from the same $\mathcal{O}$-observable. The extraction is done by the *MatrixMutliply* operation and the encapsulation by the *Apply_Plus* operation. The *SetMatrixTabElement* operation used to encode the MTBDD *selecL* built for extracting distributions differs slightly from the *SetMatrixElement* operation presented in Section 2.2 as it uses as row and column indices two vectors of integers. After grouping distributions of

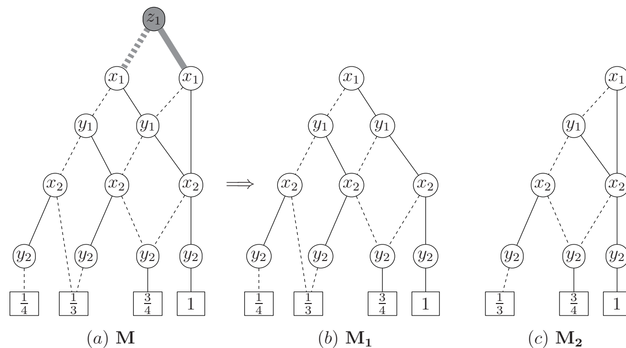


Figure 5. Decomposition of the MTBDD M into MTBDD M_1 and M_2 by transition class

equivalent transitions in the classes numbered by the values extracted from the merging directives MTBDD, these distributions are normalized by distributing $\frac{1}{k}$ to k probability distributions gathered in the same transition class. At this stage, all distributions on the observables in the ξ_{int}-observable MDP model had been correctly computed and stored by transition class in the vector of MTBDDs *newArray*. It remains to build a new MTBDD $M_{f_{\Delta_{\xi_{int}}}}$ encoding the new distribution component function $f_{\Delta_{\xi_{int}}}$ of the ξ_{int}-observable MDP model $\mathcal{C}_{\xi_{int}}$ by assembling MTBDDs stored in vector *newArray* on new non-deterministic boolean variables $newNdetVars = \{z_1, z_2, \ldots, z_{nd'}\}$ such that $nd' = \lceil \log_2(nbNewCl) \rceil$. The assembling process generates for each transition class the MTBDD representing its encoding number and then performs *Apply_Times* and *Apply_Plus* operations to encode the newly computed distributions in the MTBDD *result* representing the distribution component function $f_{\Delta_{\xi_{int}}}$ of the ξ_{int}-observable MDP.

The remaining non-determinism in the ξ_{int}-observable MDP model is resolved during the security analysis by *admissible schedulers* which are considered under control of the adversary and which are nothing more than standard schedulers of the ξ_{int}-observable MDP model. In the next section, we present and compare results obtained by applying our method to the dining cryptographers protocol. Finally, we have validated the extended implementation of the model checker by analyzing the Dining Cryptographers Protocol for anonymity. This case study can be found in Appendix B.

5 Conclusion and Future Work

In this paper, we have introduced a method for calibrating the power of schedulers by defining two levels of schedulers in the context of MDPs . We have also proved the soundness of the method in this context. The method has been implemented in PRISM using the symbolic techniques based on MTBDD, enhancing so PRISM with the possibility to restrict to a subclass of probabilistic schedulers, namely the class of admissible schedulers.

Future work. As future work, we plan to investigate the ability of the class of admissible schedulers to reach the minimum probability of information leakage and the separation in the expressive power of internal scheduler classes obtained by defining more general observation schemes then the static ones used here. In order to improve performances of the implementation and avoid the multiple combinations performed during the translation of modules in the MTBDDs inherent to our semantic approach, we plan to investigate a syntactic approach in order to define a subclass of admissible schedulers.

Acknowledgements. We wish to thank David Parker for helping and guiding our first steps into the architecture of the model checker PRISM.

References

[1] Andova, S., "Probabilistic process algebra," Ph.D. thesis, Technische Universiteit Eindhoven (2002).

[2] Baier, C., "On the Algorithmic Verification of Probabilistic Systems," Habilitation, Universität Mannheim (1998).

[3] Bhargava, M. and C. Palamidessi, *Probabilistic anonymity*, in: M. Abadi and L. de Alfaro, editors, *CONCUR*, Lecture Notes in Computer Science **3653** (2005), pp. 171–185.

[4] Bryant, R. E., *Graph-based algorithms for boolean function manipulation*, IEEE Trans. Comput. **35** (1986), pp. 677–691.
URL http://portal.acm.org/citation.cfm?id=6432.6433

[5] Canetti, R. and al., *Task-structured probabilistic i/o automata*, in: *In Proceedings of the 8th International Workshop on Discrete Event Systems (WODES)*, 2006, pp. 207–214.

[6] Chatzikokolakis, K. and C. Palamidessi, *Making random choices invisible to the scheduler*, Inf. Comput. **208** (2010), pp. 694–715.
URL http://dx.doi.org/10.1016/j.ic.2009.06.006

[7] Chaum, D., *The dining cryptographers problem: unconditional sender and recipient untraceability*, J. Cryptol. **1** (1988), pp. 65–75.
URL http://portal.acm.org/citation.cfm?id=54235.54239

[8] Dolev, D. and A. C.-C. Yao, *On the security of public key protocols*, IEEE Transactions on Information Theory **29** (1983), pp. 198–207.

[9] Fujita, M. and al., *Multi-terminal binary decision diagrams: An efficient data structure for matrix representation*, Formal Methods in System Design **10** (1997), pp. 149–169.

[10] Garcia, F. D., P. van Rossum and A. Sokolova, *Probabilistic anonymity and admissible schedulers*, CoRR **abs/0706.1019** (2007).

[11] Giro, S. and P. R. D'Argenio, *On the expressive power of schedulers in distributed probabilistic systems*, Electr. Notes Theor. Comput. Sci. **253** (2009), pp. 45–71.

[12] Hamadou, S. and J. Mullins, *Calibrating the power of schedulers for probabilistic polynomial-time calculus*, Journal of Computer Security **18** (2010), pp. 265–316.

[13] Hansson, H. and B. Jonsson, *A logic for reasoning about time and reliability*, Formal Asp. Comput. **6** (1994), pp. 512–535.

[14] Kiraga, A.-F., *A prism security module* (2010).
URL http://www.crac.polymtl.ca/psm

[15] Kwiatkowska, M., G. Norman and D. Parker, *Prism: Probabilistic symbolic model checker*, in: T. Field and al., editors, *Computer Performance Evaluation / TOOLS*, Lecture Notes in Computer Science **2324** (2002), pp. 200–204.

[16] Mitchell, J., A. Ramanathan, A. Scedrov and V. Teague, *A probabilistic polynomial-time process calculus for the analysis of cryptographic protocols*, Theoretical Computer Science **353** (2006), pp. 118–164.

[17] Parker, D., "Implementation of Symbolic Model Checking for Probabilistic Systems," Ph.D. thesis, University of Birmingham (2002).

[18] Segala, R., "Modeling and verification of randomized distributed real-time systems," Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, USA (1995), not available from Univ. Microfilms Int.

[19] Sokolova, A. and E. P. D. Vink, *Probabilistic automata: System types, parallel composition and comparison*, in: *In Validation of Stochastic Systems: A Guide to Current Research* (2004), pp. 1–43.

[20] Somenzi, F., *Cudd: Colorado university decision diagram package, release 2.42*, Technical report, University of Colorado at Boulder (2009).
URL http://vlsi.colorado.edu/fabio/CUDD/

A Encoding DTMCs and MDPs using (MT)BDDs

The distribution component of the transition function in DTMC is represented as a function $f_\Delta : \mathbb{B}^n \times \mathbb{B}^n \to [0,1]$ defined on a set $\{x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n\}$ of boolean variables. Row (resp. column) variables in the set $rowVars = \{x_1, x_2, \ldots, x_n\}$ (resp. $colVars = \{y_1, y_2, \ldots, y_n\}$) encode outgoing (resp. ingoing) states for transitions. The number of boolean variables used for each state depends on the variables used in the PRISM model. The number of boolean variables necessary for encoding a variable is determined by the cardinality of its value domain and then the state is encoded using the union of boolean variables corresponding to all PRISM variables in the model. For DTMC models, PRISM use the interleaved variable ordering between row and column variables, such as $x_1 < y_1 < x_2 < y_2 < \cdots < x_n < y_n$. The action component of the transition function in DTMC is represented as a family $\{f_\alpha : \mathbb{B}^n \to \mathbb{B}\}_{\alpha \in Act}$ on boolean variables set $\{x_1, x_2, \ldots, x_n\}$, using variable ordering $x_1 < x_2 < \cdots < x_n$. For each action $\alpha \in Act$, f_α encodes the set $\Delta_{Act}^{-1}(\alpha)$.

Extra encoding is also needed for taking non-determinism in account. Rows (resp. column) variables $rowVars = \{x_1, x_2, \ldots, x_n\}$ (resp. $colVars = \{y_1, y_2, \ldots, y_n\}$) encode outgoing (resp. ingoing) states as in DTMC. The non-determinism is encoded with boolean variables $ndetVars = \{z_1, z_2, \ldots, z_{nd}\}$ where 2nd is the maximum number of probability distributions from a state of the model aiming to provide locations to multiple actions enabled in a state. The transition function may then be seen as a function of the form $S \times \{0, 1, \ldots, 2^{nd} - 1\} \to Act \times Disc(S)$ and then also be represented as a pair of functions coping respectively with the distribution associated with transitions and the actions labeling transitions. The distribution component may be represented as a function f_Δ: $\mathbb{B}^n \times \mathbb{B}^{nd} \times \mathbb{B}^n \to [0,1]$ on boolean variable set $\{x_1, \ldots, x_n, z_1, \ldots, z_{nd}, y_1, \ldots, y_n\}$. For the MDPs, the variable ordering orders non-deterministic variables first followed by row variables and column boolean variables interleaved as in DTMC. The action component may be represented as a family $\{f_\alpha : \mathbb{B}^n \times \mathbb{B}^{nd} \to \mathbb{B}\}_{\alpha \in Act}$ on boolean variables set $\{x_1, x_2, \ldots, x_n, z_1, z_2, \ldots, z_{nd}\}$ with the variable ordering $z_1 < \cdots < z_{nd} < x_1 < \cdots < x_n$. For each action $\alpha \in Act$, f_α encodes the set $\Delta_{Act}^{-1}(\alpha)$.

B Case Study and Results Analysis

The Dining Cryptographers Problem [7] is described as follows. Three cryptographers have a dinner. They agree that the bill has to be paid by either one of them or by their organization (master) and that the master will decide who pays. The master will also have to inform everyone secretly whether he has to pay or not. The cryptographers would like to find out whether the bill is paid by the master or by one of them. However, in the latter case, they wish to keep anonymous the identity of the payer. The solution of the problem

is described in [7] as follows: each cryptographer tosses a fair coin which is visible to himself and his right neighbor. Each one checks the two adjacent coins and, if he is not the payer, announces *agree* if they are the same and *disagree* if they are not. However, the payer announces the opposite. It has been proved that the number of *disagree* is even if the master is paying the bill.

Modeling DCP. We model this protocol in PRISM language as a MDP. We set first the number N of cryptographers in the model to 3. The global variable `payId` : [0 . . N] models the identity of the payer. The value $i > 0$ indicates `cryptographer`i, for $i \in \{1, 2, 3\}$, as the payer and 0, as the master. The module `cryptographer`i makes use of four PRISM variables: The variable `out`i : [0 . . 2] indicates the outcome of the flipping coin, values 0, 1 and 2 indicating respectively *not yet flipped*, *head* and *tail*; The values 0 and 1 of the variable `announce`i : [0 . . 1] stand respectively for *disagree* and *agree*; To state the order in which cryptographers broadcast, we use the variable `order`i : [0 . . 3]; The last variable `ok`i : [0 . . 1] is used to indicate the end of the actions performed by the `cryptographer`i. The flipping action is specified by the following command:

```
[xi] outi=0 → 1/2:(outi'=1) + 1/2:(outi'=2);
```

The string `x`i stands for the action labeling the transitions generated by the command. After the coin is flipped, the announcement depends on whether he is paying or not. We consider here a non-deterministic master. There are four cases. In each case the announcement position is chosen non-deterministically by considering the positions already occupied by other cryptographers in the model. We have done such to prevent the case where more than one cryptographer broadcast the announcement in the same position. The following commands concern the case where the two coins have the same outcomes and `cryptographer`i is not the payer.

```
[a1i] oki=0 & outi>0 & out(i+1)mod 3>0 & outi=out(i+1)mod 3 & (payId!=i) &
(order(i+1)mod 3!=1) & (order(i+2)mod 3!=1) →(oki'=1) & (announcei'=1) &(orderi'=1);
[a2i] oki=0 & outi>0 & out(i+1)mod 3>0 & outi=out(i+1)mod 3 & (payId!=i) &
(order(i+1)mod 3!=2) & (order(i+2)mod 3!=2) →(oki'=1) & (announcei'=1) &(orderi'=2);
[a3i] oki=0 & outi>0 & out(i+1)mod 3>0 & outi=out(i+1)mod 3 & (payId!=i) &
(order(i+1)mod 3!=3) & (order(i+2)mod 3!=3) →(oki'=1) & (announcei'=1) &(orderi'=3);
```

The case where the two coins show different outcomes and `cryptographer`i is not the payer is resumed by these commands:

```
[b1i] oki=0 & outi>0 & out(i+1)mod 3>0 & outi!=out(i+1)mod 3 & (payId!=i) &
(order(i+1)mod 3!=1) & (order(i+2)mod 3!=1) →(oki'=1) & (announcei'=0) &(orderi'=1);
[b2i] oki=0 & outi>0 & out(i+1)mod 3>0 & outi!=out(i+1)mod 3 & (payId!=i) &
(order(i+1)mod 3!=2) & (order(i+2)mod 3!=2) →(oki'=1) & (announcei'=0) &(orderi'=2);
[b3i] oki=0 & outi>0 & out(i+1)mod 3>0 & outi!=out(i+1)mod 3 & (payId!=i) &
(order(i+1)mod 3!=3) & (order(i+2)mod 3!=3) →(oki'=1) & (announcei'=0) &orderi'=3);
```

The third and forth cases concern the situations where the `cryptographer`i is paying. In each case, three commands describe the behavior of the cryptographer as seen above. We use action labels `c1`i, `c2`i, `c3`i for coins exhibiting the same outcomes and `d1`i, `d2`i, `d3`i otherwise. The last command in the module `cryptographer`i is used to synchronize for termination `[end] ok`i`=1` $\rightarrow$ `true.`

Modeling the adversary. The adversarial model we consider here corresponds to the Dolev-Yao model [8]. In this model, the scheduling process is controlled by the adversary who controls all the communication network and hence, the schedulers. If no constraint is imposed to the schedulers then, as showed in [10,3], the adversary can break the anonymity by forcing the payer if any to make his announcement in the last position (more generally in predetermined position). The role of the observation relation $\mathcal{O}$ is to avoid this. The relation $\mathcal{O}$ defined on the set

$$\mathtt{Act} = \{\mathtt{xi}, \mathtt{a1i}, \mathtt{a2i}, \mathtt{a3i}, \mathtt{b1i}, \mathtt{b2i}, \mathtt{b3i}, \mathtt{c1i}, \mathtt{c2i}, \mathtt{c3i}, \mathtt{d1i}, \mathtt{d2i}, \mathtt{d3i}, \mathtt{fin}\},$$

for $i \in \{1, 2, 3\}$ specifies that any choice of a position of announcement for a given cryptographer should be indistinguishable. We have then 16 $\mathcal{O}$-observables for a model of three cryptographers:

$$o_1 = \{x1\},\ o_2 = \{x2\},\ o_3 = \{x3\},\ o_4 = \{end\},\ o_5 = \{a11, a21, a31\},$$
$$o_6 = \{b11, b21, b31\},\ o_7 = \{c11, c21, c31\},\ o_8 = \{d11, d21, d31\},$$
$$o_9 = \{a12, a22, a32\},\ o_{10} = \{b12, b22, b32\},\ o_{11} = \{c12, c22, c32\},$$
$$o_{12} = \{d12, d22, d32\},\ o_{13} = \{a13, a23, a33\},\ o_{14} = \{b13, b23, b33\},$$
$$o_{15} = \{c13, c23, c33\},\ o_{16} = \{d13, d23, d33\}$$

Table B.1
MDP model $\mathcal{C}$ and ξ_{int}-observable MDP $\mathcal{C}_{\xi_{int}}$ model for 3–6 cryptographers

Crypto	State	Transitions	Choices		Time(sec.)
			MDP $\mathcal{C}$	MDP $\mathcal{C}_{\xi_{int}}$	
3	1,884(4)	4,200	3,948	2,316	5
4	33,365(5)	96,120	91,420	45,100	3.324×10^2
5	667,098(6)	2,391,540	2,290,350	990,030	1.634×10^4
6	14,853,279(7)	64,276,716	61,859,406	28,453,239	1.402×10^5

The PRISM model and the adversarial model can be generalized to an arbitrary number $N > 3$ of cryptographers by using the same principles as in the three cryptographers protocol.

MDP vs ξ_{int}-observable MDP. Tab. B.1 represents for a given number of cryptographers, the number of states (the numbers in parentheses stand for

initial states), the number of transitions, the number of choices respectively for MDP model and ξ_{int}-observable MDP model and the time in seconds used by PRISM to build the ξ_{int}-observable MDP model. The ξ_{int}-observable MDP

Table B.2
Boolean variables used for encoding the functions f_Δ and $f_{\Delta_{\xi_{int}}}$

Crypto	Line variables	Column variables	non-deterministic variables	
			MDP $\mathcal{C}$	MDP $\mathcal{C}_{\xi_{int}}$
3	20	20	15	6
4	31	31	20	7
5	38	38	25	7
6	45	45	30	8

model have less choices than the MDP model since some distributions in the MDP model have been combined. We remark also that the construction time of the ξ_{int}-observable MDP model increases as the number of cryptographers in the model do since in this case, the number of states and transition classes also increase. The boolean variables used in the MDP and ξ_{int}-observable MDP models are shown in Tab. B.2. We observe that the number of non-deterministic variables decreases in the ξ_{int}-observable MDP model. This decreasing is due to the diminution of transition classes caused by the combination of distributions on actions in the same $\mathcal{O}$-observable and by the elimination of the useless non-deterministic variables in the resulted ξ_{int}-observable MDP model. By assuming 20 bytes for each MTBDD node [20], Fig. B.1 shows the memory size in Megabytes occupied by the MTBDD encoding the distribution component functions f_Δ and $f_{\Delta_{\xi_{int}}}$ and the merging directives. For any given number of cryptographers, the size of the MTBDD encoding the function $f_{\Delta_{\xi_{int}}}$ decreases as the non-determinism decreases (as the MTBDD nodes relating to these non-deterministic boolean variables disappears) (Tab. B.2). Furthermore, the merging directives MTBDD M_{dir} occupies more space than MTBDDs encoding distribution component functions f_Δ and $f_{\Delta_{\xi_{int}}}$ because more cryptographers in the model means that they generate more states and transition classes and then more terminal nodes which reduce in turn possibilities of application of the reduction rules.

Schedulers vs admissible schedulers. We have also investigated how the unlimited power of the schedulers can bias the analysis of a proved secure protocol as the Dining Cryptographers Protocol. For three cryptographers protocol, the probability that a payer makes his announcement in the last position is formulated as follows:

```
Pmax=? [true U (ok1=1 & ok2=1 & ok3=1) & ((payId=1 & order1=3) |
(payId=2 & order2=3) | (payId=3 & order3=3)){"init"}{max}]
```

The probability computed with PRISM for 3–6 cryptographers protocol de-

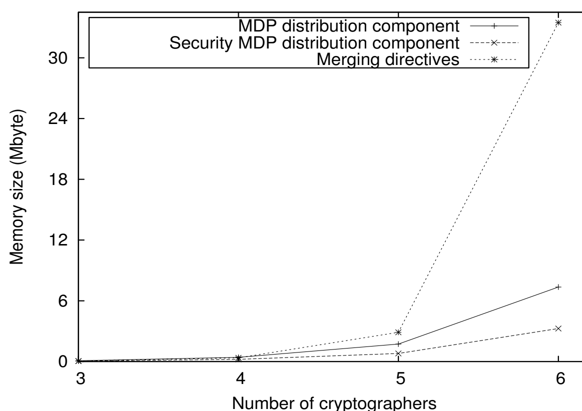


Figure B.1. Memory size occupied by M_{f_Δ}, $\mathrm{M}_{f_{\Delta_{\xi int}}}$ and M_{dir} for 3–6 cryptographers

pends on the considered set of schedulers. By considering all schedulers, the maximum probability is always 1 for any given number of cryptographers. Intuitively, this means that the adversary can always force a payer, if any, to announce in the last position and then compromise the anonymity of the protocol. This flaw is general for any position chosen by the adversary. But assuming only admissible schedulers, the payer can not announce in the last position with a probability exceeding $\frac{1}{n}$ (in n cryptographers protocol) and then the anonymity is preserved.

Monadic Scripting in F# for Computer Games

G. Maggiore, M. Bugliesi, R. Orsini

Università Ca' Foscari Venezia
DAIS - Computer Science
{maggiore,bugliesi,orsini}@dais.unive.it

Abstract

Scripting in video games is a complex challenge as it needs to allow a game designer (usually not a developer) to interact with an extremely complex piece of software: a game engine. Game engines handle AI, physics, rendering, networking and various other functions; scripting languages usually act as the main interface to drive the entities managed by the game engine without exposing too much of the complexity of the engine.
A good scripting language must be very user-friendly because most of its users will not be developers; it must support transparent continuation mechanisms to ease the most common tasks faced when writing scripts and it must be easy to integrate in an existing game engine. Also, since games are very performance sensitive, the faster the scripting system the better.
In this paper we present a monadic framework that elegantly solves these problems and compare it with the most commonly used systems.

Keywords: games, monadic programming, state management, scripting

1 Introduction

Games are the next frontier in entertainment. Game sales in 2010 have reached 10 billion dollars, making games the absolutely preferred means of entertainment of our time. As more and more developers focus on building games, we believe that a contribution can be made to this field with a study of game engine architectures and languages for making games.

The core of a game is its engine. A game engine [12] is a fairly complex piece of software: it encompasses most of the aspects of computer game development, touching areas such as computer graphics, AI, algorithms, networking, and so on. Game engines are difficult to maintain and long to compile. Game designers need a simpler access venue to build the game logic of a game, and for this reason game engines are made *scriptable*, so that their functionality

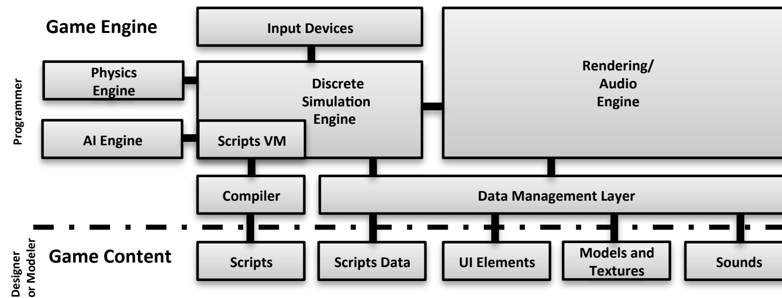


Fig. 1. Data-driven game architecture

related to gameplay can be programmed without direct access to its source and with a simpler language.

A general architecture of scriptable game engines (we report here Figure 2 from [23]) can be seen in Figure 1.

As we can see the `discrete simulation engine` joins together the processing of the game state performed by the AI engine, the physics engine, user input and various scripts. The `discrete simulation engine` is responsible for maintaining an updated state and it is usually written in a language such as C or C++. C# is also emerging as the language of choice for implementing portions of the game engine (see [7]) and for implementing entire independent games ([9] and [11] are widely adopted game frameworks based on C#).

The scripting solutions that games use nowadays are based on either simple, in-house built languages [10] or ready-made scripting languages: among those we find Lua and Python ([5,13]), together with C# which sits somewhere in between a proper scripting language and a game development language.

The main contribution of this work is an improvement over existing scripting languages, most notably LUA (the current state of the art). We use F# in combination with a monadic domain specific language [22] to create a statically typed scripting language that is as succint as LUA but which is, thanks to type safety, more robust. Also, encoding coroutines (one of the major characteristics of scripting languages for games) with monads offers greater flexibility over LUA's approach of wiring coroutines inside the virtual machine itself; this flexibility makes it possible to tailor our scripting system precisely around the requirements of the game, without knowledge about the (complex) internals of a virtual machine. Finally, the runtime overhead of our system is so little that our scripts run faster than LUA's and at least as fast as C# scripts.

Section 2 describes current approaches to scripting and explains the metrics we will use to compare our system with these approaches. Section 3 details the core runtime of our monadic scripting language, while Section 4 develops a library of useful combinators that are built around the core and support a powerful set of customized behaviors: this library of combinators represents

our scripting language. Section 5 illustrates an actual example of how we have used our DSL in the development of a game. Section 6 presents our results and Section 7 concludes the presentation.

2 Scripting in Games

The most important function of scripts is that of modeling the behaviors of characters and other in-game objects; in the remainder of the paper we will focus on this specific aspect. As an example, consider a script that describes the behavior of a prince in an RPG game:

```
prince:
  princess = find_nearest_princess()
  walk_to(princess)
  save(princess)
  take_to_castle(princess)
```

Depending on the size of the game world, there may be up to thousands of scripts running at any time. This means that each script must be interruptible, that is at each discrete step of the simulation engine each script must perform a finite number of transitions transition and then suspend itself; failing to do so would slow down the simulation steps, and the resulting framerate of the game would decrease, thereby reducing the player immersion. For this reason simple scripts are sometimes coded as a state machine (SM). As behaviors grow more complex, the code for an SM becomes difficult to maintain. Also, the designers who write these scripts think in terms of (nested) sequences of character actions and not in terms of SMs. For this reason scripting languages make heavy use of *coroutines* as a mechanism to build state machines implicitly instead of coding them explicitly as seen in the snippet above [6,10,14]. Coroutines are generalization of subroutines that allow multiple entry points for suspending and resuming execution at certain locations. With coroutines the code for a SM is written "linearly" one statement after another, but thanks to the suspension mechanism each action may suspend itself (often called "yield") many times before completing. The resulting code will look more like the pseudo-code for `prince`, where the current state of the state machine is stored implicitly in the current continuation.

We can state the list of requirements that a good scripting system should have:

- support for coroutines
- ease of programming
- speed (games **require** very fast execution)
- extensibility of the coroutine framework (to better adapt it to the game engine)

2.1 Coroutines in action

In the remainder of this section we analyze coroutines in action in Lua. We will also briefly discuss how coroutines are emulated in Python and C# with generators. We will implement the pseudo-code taken from the `prince` sample seen above. For a more detailed discussion of the mechanisms of coroutines in Lua, Python and C# see [18,15,2].

Lua coroutines are based on the three functions `coroutine.yield`, `coroutine.resume` and `coroutine.create` which respectively pause execution of a coroutine, resume execution of a paused coroutine and create a coroutine from a function.

```
function walk_to(self,target)
  return coroutine.create(
    function()
      while(dist(self,target) > self.reach) do
        self.Velocity = towards(self, target)
        coroutine.yield()
      end
    end)
end

...

function prince(self)
  return coroutine.create(
    function()
      princess = bind_co(find_nearest_princess(self))
      bind_co(walk_to(self, princess))
      bind_co(save(self, princess))
      bind_co(take_to_castle(self, princess))
    end)
end
```

Notice that to invoke a coroutine we need to explicitly bind it with the `bind_co` function, which resumes a coroutine until it yields for the last time, when it returns the resulting value:

```
function bind_co(c)
  local s,r,r_old = true,nil,nil
  while(s) do
    r_old = r
    s,r = coroutine.resume(c)
    coroutine.yield()
  end
  return r_old
end
```

A similar mechanism to implement coroutines in Python makes use of generators.

A generator is a special routine that returns a sequence of values. However, instead of building an array containing all the values and returning them all at once, a generator yields the values one at a time; yielding effectively suspends the execution of the generator until the next element of the sequence is requested by the caller. Python generators may appear as a way to return

lazy sequences but they are powerful enough to implement coroutines. We can adopt the convention that a coroutine is actually a generator which yields a sequence of null (None) values until it is ready to return; the returned value will be yielded last.

```
def walk_to(self,target):
  while(dist(self,target) > self.Reach):
    self.Velocity = towards(self,target)
    yield

...

def prince(self):
    for princess in find_nearest_princess(self):
      yield
    for x in walk_to(self,princess):
      yield
    for x in save(self,princess):
      yield
    for x in take_to_castle(self,princess):
      yield
```

As in Python, C# supports generators. Since C# is statically typed, we need to assign a type to our coroutines. We have two alternatives; a coroutine that returns nothing (void) has type IEnumerable, that is it returns a sequence of Objects that are all null (a similar strategy is used by Unity, even though with unsafe casts [6]) and we can type a coroutine that returns a value of type T as IEnumerable<T?>, where T? is either null or an instance of T.

We omit the C# sample for brevity, and also because of its similarity with Python. Moreover, when compared with LUA generators to implement coroutines are quite cumbersome in a scripting language and indeed LUA is by far more used in games.

In the remainder of the paper we will present a different approach to coroutines, namely building a meta-programming abstraction (called *monad*) to implement coroutines in F#. We will discuss how our approach produces code which is faster and shorter than similar implementations in Lua, Python and C#. We will also discuss how our approach is very customizable, thanks to the fact that coroutines are not *wired* into the language runtime but rather we have defined them with our monad. Also, thanks to type inference the resulting scripts require no typing annotations. Finally (see Section 6 for the details), our system offers a good runtime performance and is type safe; this makes it suitable for large and complex scripts.

3 The Script Monad

Monads can be used for many purposes [17,20,19,21,1,16]. Indeed, monads allow us to overload the bind operator, in order to define exactly what happens when we bind an expression to a name.

Monads in F# enjoy syntactic sugar that simplifies their use. Monadic operators are inserted with the specialized keywords `let!` for bind and `return` for return.

For our present purposes, one extremely relevant use we can make of monads is as the basis of coroutines for our scripting system.

The script monad is not the actual scripting language. Rather, it is the runtime framework that we use to transparently support coroutines. In Section 4 we will see how we can define a library of functions which can be seen as additional keywords and operators for the resulting scripting language.

The monad we define, at every bind will *suspend* itself and return its continuation as a lambda. This is one possible, very simple implementation of coroutines which does not feature an explicit `yield` operator. The monad type is `Script`:

```
type Script<'a,'s> = 's -> Step<'a,'s>
 and Step<'a,'s> = Done of 'a
                 | Next of Script<'a,'s>
```

Notice that the signature is very similar to that of the regular state monad, but rather than returning a result of type α it returns either `Done of` α or the continuation `Next of Script<`α, σ`>`. The continuation stores, in its closure, the current state of a suspended script.

Returning a result in this monad is simple: we just wrap it in the `Done` constructor since obtaining this value requires no actual computation steps. Binding together two statements is more complex. We try executing the first statement; if the result is `Done x`, then we return `Next(k x)`, that is we perform the binding and we will continue with the rest of the program with the result of the first statement plugged in it. If the result is `Next p'`, then we cannot yet invoke `k`. This means that we have to bind `p'` to `k`, so that at the next execution step we will continue the execution of `p` from where it stopped.

```
type ScriptBuilder() =
  member this.Bind(p:Script<'a,'s>,
                   k:'a->Script<'b,'s>)
     : Script<'b,'s> =
    fun s ->
      match p s with
      | Done x -> Next(k x)
      | Next p' -> Next(this.Bind(p',k))

  member this.Return(x:'a) : Script<'a,'s>
    = fun s -> Done x

let script = ScriptBuilder()
```

Integrating our monadic runtime for scripts in a game engine loop is simple. We define a game script as an instance of the `Script` datatype where the state (the σ type variable) is instantiated to some type `GameState` which defines the current state of the game. The main loop will now carry around the current

computation of the game script:

```
let rec update (script_step:Script<Unit,GameState>)
               (game_state:GameState) =
  let script_step' =
     match script_step game_state with
     | Done() -> fun _ -> Done()
     | Next k -> k
  let game_state' =
     (** compute new state **)
  in update script_step' game_state'
```

The update function executes a step of the script. If the script has finished, then we create an identity script that will be called indefinitely or we could return that the game is finished and some recap screen must be shown. When an iteration of the update loop is completed, then we call update with the next state of the script as its parameter.

This integration with the main loop can be easily translated to C# and then integrated with the rest of the game engine.

Auxiliary Functions

Existing functions are, of course, not defined in terms of our monad. Often though, we will wish to apply some existing function directly to our scripts rather than bind them to some variables, apply the function to those variables and finally returning the result. For example, consider the case where we have two scripts `s1:Script<bool>` and `s2:Script<bool>` and we wish to compute the logical *and* of their result; currently we would have to write:

```
script{
  let! x = s1
  let! y = s2
  return x && y
}
```

whereas we would prefer to be able to simply write:

```
s1 &&. s2
```

for some appropriate operator `(&&.)`. For this reason we define the lifting functions, very useful functions that lift an operation from the domain of values to the domain of monads; the general shape of the `n`-ary lifting functions is:

```
let lift_n (f : 'a1 -> a2 -> ... -> 'an -> 'b) :
           (Script<'a1> -> Script<'a2> -> ... -> Script<'an> ->
              Script<'b>) =
    fun s1 -> s2 -> ... -> sn ->
      script{
        let! x1 = s1
        let! x2 = s2
        ...
        let! xn = sn
        return f x1 x2 ... xn
      }
```

Unfortunately it is very difficult to define `lift_n` for an arbitrary `n`, so we will define `lift_i` for various values of `i`. As an example application we can define binary operators for scripts:

```
let not_ (s:Script<bool>) : Script<bool> =
  lift_1 not s

let and_ (s1:Script<bool>) (s2:Script<bool>) : Script<bool> =
  lift_2 (&&) s1 s2

let or_ (s1:Script<bool>) (s2:Script<bool>) : Script<bool> =
  lift_2 (||) s1 s2
```

Also, we can define a useful `ignore_` function that discards the result of a script when we do not need it:

```
let ignore_ (s:Script<'a>) : Script<Unit> =
  lift1_ (fun x -> ()) s
```

In general any n-ary function that is not capable of manipulating scripts can be lifted to the domain of scripts with `lift_n`

4 A Library of Reusable Scripts

Here, and throughout we use the standard F# convention that, inside a monad, missing `else` branches correspond to `else` branches with `return ()` as the body. Similarly, we use the infix application operator `|>`, writing `x |> f` as an equivalent for `(f x)`.

The main advantage of using monads rather than hardcoded mechanisms is flexibility. On one hand we can modify the definition of our monad in order to accomodate for different functionalities, such as referential transparency, multi-threading, etc. On the other hand we have an explicit representation of coroutines (values of type `Script`) with which we can easily build libraries that functionally manipulate coroutines in powerful ways. In this section we study one such general purpose library based on the script monad. This library is being used in the development of a commercial strategy game, and as such its usefulness has been put to the test in a practical application (the game is released as open source, and can be found at [4]). It is important to realize that even though what follows is a very general library of combinators (in particular the Calculus of Coroutines presented below) there are many alternative libraries that may better suit a specific kind of games; the monadic system described in Section 3 can be used as the basis for any of those alternative libraries.

A Calculus of Coroutines

The basic combinators we define are a simple calculus of coroutines; this means that with these operators we take one or more coroutines and we return

name	syntax	operation
`parallel`	$s_1 \wedge s_2$	executes two scripts in parallel and returns both results
`concurrent`	$s_1 \vee s_2$	executes two scripts concurrently and returns the result of the first to terminate
`guard`	$s_1 \Rightarrow s_2$	executes and returns the result of a script only when another script evaluates to `true`
`repeat`	$\uparrow s$	keeps executing a script over and over
`atomic`	$\downarrow s$	forces a script to run in a single tick of the *discrete simulation engine*

Table 1
Calculus of Coroutines

another coroutine which can be plugged as a parameter for another one of this operators. The basic building blocks of these operators are instances of the script monad and are listed in Table 1.

We show here the implementation of these combinators with our monadic system:

```
let rec parallel_ (s1:Script<'a>) (s2:Script<'b>) : Script<'a * 'b> =
  fun s ->
    match s1 s,s2 s with
    | Return x, Return y        -> Return (x,y)
    | Continue k1, Continue k2 -> parallel_ k1 k2
    | Continue k1, Return y     -> parallel_ k1 (fun s -> Return y)
    | Return x, Continue k2     -> parallel_ (fun s -> Return x) k2

let rec concurrent (s1:Script<'a>) (s2:Script<'b>)
                   : Script<Either<'a,'b>> =
  fun s ->
    match s1 s,s2 s with
    | Return x, _         -> Return(Left x)
    | _, Return y         -> Return(Right y)
    | Continue k1, Continue k2 -> concurrent_ k1 k2

let rec guard_ (c:Script<bool>) (s:Script<'a>) : Script<'a> =
  script{
    let! x = c
    if x then
      let! res = s
      return s
    else
      let! res = guard_ c s
      return res
  }

let rec repeat_ (s:Script<Unit>) : Script<Unit> =
  script{
    do! s
    do! repeat_ s
  }

let rec atomic_ (p:Script<'a>) : Script<'a> =
```

```
  fun s ->
    match p s with
    | Return x -> Return x
    | Continue k -> atomic_ k s
```

Game Patterns

Thanks to our general combinators we can define a small set of recurring game patterns; by instantiating these game patterns one can build the final game scripts with great ease. The first game pattern is the most general, and for this reason it is called `game_pattern`. This pattern initializes the game in a single tick, then performs a game logic (while the game is not over) and finally it performs the ending operation before returning some result:

```
let game_pattern (init:Script<'a>)
                 (game_over:'a -> Script<'bool>)
                 (logic:'a -> Script<Unit>)
                 (ending:'a -> Script<'c>) : Script<'c> =
  script{
    let! x = init |> atomic_
    let! (Left y) = concurrent_ (guard_ (victory x)
                                        (ending x |> atomic_))
                                (logic x |> repeat_)
    return y
  }
```

A simplified, recurring variation of this game pattern simply does nothing until the game is over:

```
let wait_game_over (game_over:Script<bool>) : Script<Unit> =
  let null_script = script{ return () }
  game_pattern null_script
               (fun () -> game_over)
               (fun () -> null_script)
               (fun () -> null_script)
```

Writing a script with our system will consist of instantiating one game pattern with specialized scripts as its parameters; these scripts will alternate accesses to the specific state of the game with invocations of combinators from the calculus seen above. In the next session we will see an example of this.

5 An Actual Script

We are now ready to discuss an actual script coming from a strategy game. In this game, the players compete to conquer a series of *systems* by sending fleets to reinforce their systems or to conquer the opponent's.

The basic game mode returns the winning player; as long as there is more than one player standing, the script waits. This script computes the union of the set of active fleet owners with the set of system owners:

```
let alive_players_set =
  script{
```

```
    let! fs = get_fleets
    let fleet_owners =
             fs |> Seq.map (fun f -> f.Owner)
             |> Set.ofSeq
    let! ss = get_systems
    let system_owners =
             ss |> Seq.map (fun s -> s.Owner)
             |> Set.ofSeq
    return fleet_owners + system_owners
  }

let game_over =
  script{
    let! alive_players = alive_players_set
    let num_alive_players = alive_players |> Seq.length
    return num_alive_players = 1
  }
```

The main task of our script is to wait until the set of active players has exactly one element; when this happens, that player is returned as the winner:

```
let basic_game_mode = wait_game_over game_over
```

An interesting alternative coding style could be built by lifting a series of useful operators and redefining `alive_players_set` and `game_over` in a very concise form; let us assume that all operators and functions that end with `_` or `.` in the following snippet have been lifted appropriately, for example:

```
let (=.) = lift_2 (=)
let (+.) = lift_2 (+)
...
```

This way, we can write:

```
let alive_players_set =
  (get_fleets  |> Seq.map_ (fun f -> f.Owner) |> Set.ofSeq_) (+.)
  (get_systems |> Seq.map_ (fun s -> s.Owner) |> Set.ofSeq_)

let game_over =
  (alive_players_set |> Seq.length_) =. (script{ return 1 })
```

5.1 Other scripts

Variations of the game are soccer (one system acts as the ball which can be moved around), capture the flag, siege and others.

The siege game mode features a central system which must be conquered and held to obtain bonuses. Holding this system for sixty seconds gives a bonus to its owner, while losing the central system resets its bonuses.

The siege mode is an instance of the most general game pattern:

```
let wait dt =
  script{
    let! t0 = time
    do! guard
          script{
            let! t = time
            return t - t0 > dt
```

```
          }
          script{ return () }
  }

let init : Script<System> = get_system "Center"
let logic (center:System) =
  script{
    let previous_owner = center.Owner
    do! concurrent_
          script{
            do! wait 60.0f
            center.AddBonus()
          }
          (guard (script{ return center.Owner <> previous_owner })
                 (script{
                   center.ResetBonus()
                   return () }))
        |> ignore_
  }

let siege_game_mode =
  game_pattern init (fun center -> game_over)
               logic (fun _ -> script{ return () })
```

We omit a detailed discussion of the other variations for reasons of space; the important thing to realize is that all of these variations have been implemented with the same simplicity of the scripts above, by instancing one game pattern with appropriate scripts which are built with a mix of combinators interspersed with accesses to the game state.

6 Benchmarks

We will focus our comparison mostly on LUA, since it is the most widely adopted scripting language, it fully supports coroutines and is considered the current state of the art. We will include some benchmarking data on Python and C# for completeness, but their poor support for coroutines makes them unsuitable for large scale use as scripting languages.

LUA and F# offer roughly the same ease of programming, given that:

- scripts are approximately as long and as complex
- there are no explicit types, thanks to dynamic typing in LUA and type inference in F#

It is important to notice that, since F# is a statically type language, it offers a relevant feature that LUA does not have: **safety**. This means that more errors will be catched at compile time and correct reuse of modules is made easier.

To measure speed, we have run three benchmarks on a Core 2 Duo 1.86 GHz CPU with 4 GBs of RAM. The tests are two examples of scripts computing large Fibonacci numbers concurrently plus a syntetic game where each script animates a ship moving in a level and then dying. The tests have been made with Windows 7 Ultimate 64 bits. Lua is version 5.1, Python is version

3.2 and .Net is version 4.0. The lines of code of each script are listed in Table 2, while the number of yields per seconds (higher is better) are listed in Table 3. We have measured the number of yields per second in order to assess the relative cost of the yielding architecture; more yields per second implies more scripts per second which in turn implies more scripted game entities and thus a more complex and compelling gameplay.

Language	Fibonacci	Many Fibonacci	Ships
F#	21	21	35
Python	24	29	48
Lua	30	39	52
C#	51	58	59

Table 2
Lines of code

Language	Fibonacci	Many Fibonacci	Ships
F#	7.6	5.8	4.0
C#	7.1	4.2	4.1
Lua	1.5	1.4	0.8
Python	1.1	1.1	0.9

Table 3
Speed test in millions of yields per seconds

It is quite clear that F# offers the best mix of performance and simplicity. Also, it must be noticed that Python and Lua suffer a noticeable performance hit when accessing the state, presumably due to lots of dynamic lookups; this problem can only become more accentuated in actual games, since they have large and complex states that scripts manipulate heavily.

An additional note must be given about architectural convenience. For games where the `discrete simulation engine` is written in C# (either because the entire game is written in C# or because the game is written in C++ while only the game logic is in C#) then using a language such as F# can give a further productivity and runtime performance boost because scripts would be able to share the game logic type definitions given in C#, thereby removing the need for binding tools such as SWIG or the DLR [8,3] (or many others) that enable interfacing C++ or C# with Lua or Python. See Figure 2 for a representation.

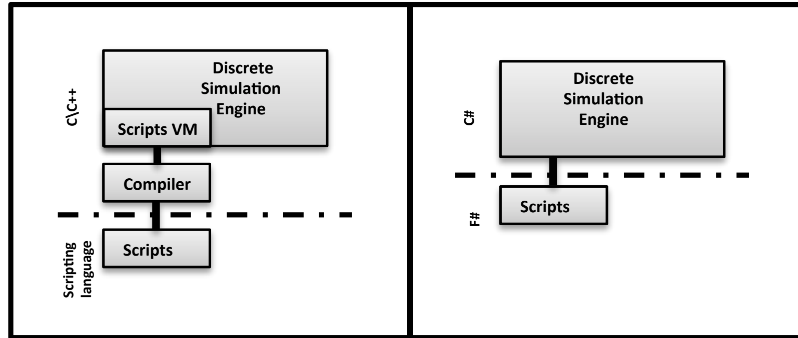


Fig. 2. Native vs managed discrete simulation engine

7 Conclusions

Scripts are an important and pervasive aspect of computer games. Scripts simplify the interaction with computer game engines to the point that a designer or an end-user can easily customize gameplay. Scripting languages must support coroutines because these are a very recurring pattern when creating gameplay modules. Scripts should be fast at runtime because games need to run at interactive framerates. Finally, the scripting runtime should be as modular and as programmable as possible to facilitate its integration in an existing game engine.

In this paper we have shown how to use meta-programming facilities (in particular monads) in the functional language F# to enhance in terms of speed, safety and extensibility the existing scripting systems which are based on Lua, the current state of the art. We have also shown how having a typed representation of coroutines promotes building powerful libraries of combinators that abstract many common patterns found in scripts.

References

[1] C# async and await (reference). http://msdn.microsoft.com/en-us/vstudio/async.aspx.

[2] C# yield (reference). http://msdn.microsoft.com/en-us/library/9k7k7cf0(v=vs.80).aspx.

[3] Entertainment software association. http://www.theesa.com.

[4] Galaxy wars game. http://vsteam2010.codeplex.com/.

[5] Games using lua as a scripting language. http://en.wikipedia.org/wiki/Category:Lua-scripted_video_games.

[6] Scripting in unity. http://unity3d.com/support/documentation/ScriptReference/index.Coroutines_26_Yield.html.

[7] Sims 3 and second life built with mono. http://en.wikipedia.org/wiki/Mono_(software)#Software_developed_with_Mono.

[8] Swig (simplified wrapper and interface generator). http://www.swig.org/.

[9] Unity and mono. http://unity3d.com/support/documentation/Manual/HOWTO-MonoDevelop.html.

[10] Unrealscript documentation. http://unreal.epicgames.com/UnrealScript.htm.

[11] Xna 4 documentation. http://msdn.microsoft.com/en-us/library/bb200104.aspx.

[12] Lars Bishop, Dave Eberly, Turner Whitted, Mark Finch, and Michael Shantz. Designing a pc game engine. *IEEE Comput. Graph. Appl.*, 18:46–53, January 1998.

[13] Bruce Dawson. Game scripting in python. http:// www.gamasutra.com/features/20020821/dawson_ pfv.htm, 2002. Game Developers Conference Proceedings.

[14] L. H. de Figueiredo, W. Celes, and R. Ierusalimschy. Programming advanced control mechanisms with lua coroutines. In *Game Programming Gems 6*, pages 357–369, 2006.

[15] Ana L. de Moura, Noemi Rodriguez, and Roberto Ierusalimschy. Coroutines in Lua. *Journal of Universal Computer Science*, 10(7):910–925, July 2004.

[16] Erik Meijer, Brian Beckman, and Gavin Bierman. Linq: reconciling object, relations and xml in the .net framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 706–706, New York, NY, USA, 2006. ACM.

[17] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1989.

[18] Guido Van Rossum and Phillip Eby J. Pep 342 - coroutines via enhanced generators. http://www.python.org/dev/peps/pep-0342/, 2010.

[19] Philip Wadler. Comprehending monads. In *Mathematical Structures in Computer Science*, pages 61–78, 1992.

[20] Philip Wadler. How to declare an imperative. *ACM Comput. Surv.*, 29(3):240–263, 1997.

[21] Philip Wadler and Peter Thiemann. The marriage of effects and monads, 1998.

[22] Keith Wansbrough, Keith Wansbrough, John Hamer, and John Hamer. A modular monadic action semantics. In *In Conference on Domain-Specific Languages*, pages 157–170, 1997.

[23] Walker White, Alan Demers, Christoph Koch, Johannes Gehrke, and Rajmohan Rajagopalan. Scaling games to epic proportions. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 31–42, New York, NY, USA, 2007. ACM.

Tool Supported Analysis of Web Services Protocols

Abinoam P. Marques Jr.[1]

Brazilian Health Informatics Society
Natal-RN, Brazil

Anders P. Ravn[2] Jiří Srba[2,3] Saleem Vighio[2,4]

Department of Computer Science, Aalborg University
Aalborg, Denmark

Abstract

We describe an abstract protocol model suitable for modelling of web services and other protocols communicating via unreliable, asynchronous communication channels. The model is supported by a tool chain where the first step translates tables with state/transition protocol descriptions, often used e.g. in the design of web services protocols, into an intermediate XML format. We further translate this format into a network of communicating state machines directly suitable for verification in the model checking tool UPPAAL. We introduce two types of communication media abstractions in order to ensure the finiteness of the protocol state-spaces while still being able to verify interesting protocol properties. The translations for different kinds of communication media have been implemented and successfully tested, among others, on agreement protocols from WS-Business Activity.

Keywords: Service Oriented Computing, Communication Channels, Model Checking, Verification

1 Introduction

Service oriented computing is gaining in popularity, mainly because the Internet offers a widespread, cheap and efficient infrastructure. Thus there is an

[1] Email: abinoam@gmail.com
[2] Email: {apr,srba,vighio}@cs.aau.dk
[3] The author is partially supported by Ministry of Education of Czech Republic, MSM 0021622419.
[4] The author is supported by Quaid-e-Awam University of Engineering, Science, and Technology, Nawabshah, Pakistan, and partially by the Nordunet3 project COSoDIS.

incentive to develop applications as clients that dynamically and flexibly connect to available services over the net using for instance the Web Services (WS) protocols. For simple client-server applications this may work without many considerations of error-handling, as errors can be often handled just as exceptions in standard sequential programs. However, when several services, possibly from different organizations, are involved, more sophisticated coordination protocols need to be employed in order to implement distributed transactions supporting roll-back or other compensation mechanisms. Therefore the OMG body (www.omg.org) has put much effort into developing protocol standards to handle these issues. They include protocols for atomic transactions [13], coordination [15] and for general web services business activities [14].

Protocols are distributed algorithms and that makes them hard to analyse as they contain several local-state machines that evolve independently and communicate through message exchanges. A further difficulty is asynchronous, perhaps even unreliable communication media. In the standards, protocols are often described by a combination of state/transition tables for the individual state machines, global communication graphs, and concise English text. This is useful for understanding the intent and purpose of a protocol but may be insufficient for in-depth analysis of the possible behaviours under different communication assumptions. Here formal notations like process algebras, temporal logics and automata-based formalisms are often used [7]. We discuss possible protocol formalizations in Section 2.

We have recently worked on analysing protocols using model checking techniques. The first result was an analysis of the atomic transaction protocol [17], heavily inspired by its corresponding TLA encoding [8]. Among other points, this work demonstrated that message exchange through asynchronous media is hard to model via handshake synchronization between automata. We used the experience in a more detailed study of the BAwCC coordination protocol [18] and found a fault in the protocol design [19].

During this work we have seen that it is far from simple to prepare the analysis; many hours are spent on understanding the protocol and on encoding state/transition tables, messages and communication media into a format accepted by a model checking tool. In particular the encoding part is a tedious and error-prone process, when done manually. For instance, the encoding of the BAwCC protocol into the model checker UPPAAL [21] presented in [18] ends up with 800 lines of C-code and it took at least one person month to do the encoding and check it thoroughly to remove translation bugs. This process can to a large extent be automated, and the main contributions of this paper are a tool chain and abstraction techniques, presented in Section 3, that were developed for that purpose. The components of the tool chain are detailed in Section 4.

As a further contribution, the tool chain is used to analyse the Subser-

vice Termination and Alternating Bit Protocols, the first one being used as a running example. More importantly, we have applied the tool chain to some recently studied web services protocols as well as to the BAwPC protocol [14] that has not been previously verified. The automatic approach showed a large degree of flexibility and the verification results are summarized in Section 5. We observe that it is now a matter of hours to conduct an experiment with a proposed protocol. It should also be feasible for protocol developers to use our tool without any deep knowledge of the particular model checker we employ in our tool chain. This perspective and future work are discussed in Section 6.

Related Work. Reachability analysis is a well-known technique for the analysis of small communication protocols (see e.g. [22,1]). An approach most related to our work was presented in [12]. Here the authors perform a static analysis of three-way handshake connection establishment protocol and the alternating bit protocol via dataflow static analysis using the tool FLAVERS. They model a communication medium as a finite state automaton but consider only limited notions of lossiness, media of fixed sizes and do not suggest any abstraction techniques. In our model checking approach, we are able to argue about correctness also for unbounded communication channels and provide an automatic encoding of the communication medium in a more compact way. Even though the verification problems for unbounded communication buffers are in general undecidable [4], partial decidability results exist for lossy communication channels [6], however with nonprimitive recursive complexity [20] which puts them among the hardest decidable problems. In our approach we provide a practical solution that allows to analyze complex protocols like the ones from WS-Business activity in a matter of seconds. Recently Lohmann [9] surveys possible communication models and divides them into (i) ordered/unordered, (ii) bounded/unbounded and (iii) single/multiple buffer communication. For bounded media different nonblocking sending strategies are discussed as well. In our paper we focus both on ordered and unordered as well as single and multiple buffer communication strategies, but our main goal is to argue about the behaviour of protocols with unbounded communication via the use of model checking techniques that however allow us to verify only bounded media. Moreover we consider unreliable communication policies.

2 Protocol Modelling

Web services protocols are usually described by means of state/transition tables (see e.g. [14]) that specify the behaviours of the protocol roles on *inbound events* (received messages) and on *outbound events* (sent messages). A small example of such a table describing a Subservice Termination Protocol (STP) is presented in Figure 1. The protocol contains three roles A, B and C, all of them are initially in their *Active* states. Once the role A executes the out-

ROLE	A			
	MESSAGES \STATES	Active	AwaitingB	Ended
OUTBOUND	exitB	goto Active		
INBOUND	preparingB	goto AwaitingB	goto AwaitingB	goto Invalid
INBOUND	exitedB		goto Ended	goto Ended
ROLE	B			
	MESSAGES \STATES	Active	AwaitingC	Ended
OUTBOUND	preparingB		goto AwaitingC	
OUTBOUND	exitC		goto AwaitingC	
OUTBOUND	exitedB			goto Ended
INBOUND	exitB	send preparingB goto AwaitingC		
INBOUND	exitedC		send exitedB goto Ended	
ROLE	C			
	MESSAGES \STATES	Active	Ended	
OUTBOUND	exitedC		goto Ended	
INBOUND	exitC	send exitedC goto Ended		

Fig. 1. State/Transition Table of Subservice Termination Protocol (STP)

bound event *exitB*, it waits on confirmation from B that it is preparing for termination and once B is exited, it will reach the *Ended* state. Similarly, the role B waits for the *exitedC* event from role C before it can terminate. Moreover, once the role A receives the message *exitedB* from B and enters its *Ended* state, the arrival of the message *preparingB* will lead to an *Invalid* state as the messages arrived in a wrong order. The protocol moreover compensates for the possibility of messages being lost by repeatedly retransmitting all outbound events.

2.1 Abstract Protocol Model

As our aim is to provide a tool supported analysis of web services protocols like the STP example, we need to formalize the notion of a protocol. We shall use an automata-based approach as it is convenient for our purposes, but as a part of our tool chain we provide a front end that accepts state/transition tables created in a spreadsheet application and translates them into our automata model, essentially a conventional Mealy machine.

Definition 2.1 An *abstract protocol model* is a pair $(\mathit{Msgs}, \mathit{Roles})$ where Msgs is a finite set of *messages* with $\diamond \in \mathit{Msgs}$ being the *empty message* and Roles is a finite set of *roles* such that for every role $A \in \mathit{Roles}$ we have its description $D_A = (S_A, \longrightarrow_A)$ where S_A is a finite set of *states* and $\longrightarrow_A \subseteq S_A \times \mathit{Msgs} \times \mathit{Msgs} \times S_A$ is the set of the transitions of the role A.

Whenever $(s, m, m', s') \in \longrightarrow_A$ we shall write $s \stackrel{m,m'}{\longrightarrow}_A s'$ or simply $s \stackrel{m,m'}{\longrightarrow} s'$ if the role A is clear from the context. The meaning is that the role A in its current state s is ready to receive a message m and after that it sends the message m' and changes its state to s'. The messages m and m' can be empty,

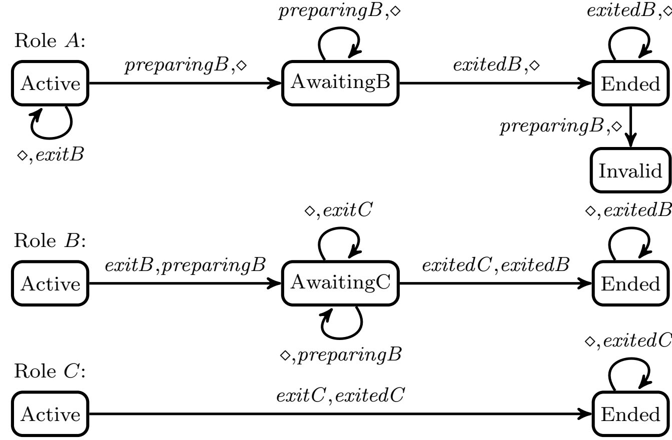


Fig. 2. Formal Specification of the STP Protocol

meaning that the transition can happen either without receiving any message or without sending any message. If both m and m' are empty, this represents an internal transition.

A formal automata-based model of the STP protocol is depicted in Figure 2. One can easily verify that this abstract protocol model describes the same behaviour as the state/transition tables in Figure 1.

2.2 Communication Policy

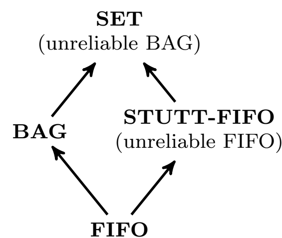
In order to define the semantics of the abstract protocol model, we need to discuss the communication policies. We shall discuss *asynchronous communication* policies with different reliability requirements on the communication medium. We consider e.g. FIFO (First In First Out) communication channels representing a perfect order-preserving communication or, as the other extreme, unreliable (lossy and duplicating) communication policy where messages can be reordered. We can abstract the possible medium implementation by its interface.

Definition 2.2 A *communication medium* is a data structure *Medium* providing the following three operations:

- $send : Medium \times Msgs \to Medium$,
- $available : Medium \times Msgs \to \{true, false\}$, and
- $receive : Medium \times Msgs \to Medium$.

Given a current medium $med \in Medium$ and a message $m \in Msgs$, the operation $med.send(m)$ updates the communication medium with the message m that was sent by one of the roles, $med.available(m)$ answers whether the message m is available for receiving (without modifying the medium) and finally $med.receive(m)$ receives the message m and updates the communication

medium accordingly. The empty message $\diamond$ is always available and sending or receiving the message $\diamond$ has no effect on the medium.



We provide a few examples of possible medium implementations for some of the classical communication policies. For the perfect FIFO policy, we model the medium as a queue. The operation *send*(m) simply enqueues m at the end of the queue, *available*(m) checks whether m is at the head of the queue and *receive*(m) removes the message m from the front of the queue. Similarly for order-preserving but lossy and duplicating policy (called STUTT-FIFO for stuttering FIFO), the call *send*(m) adds the message m at the end of the queue only if it is not already present there (if the last sent message was m then sending m does not change the queue). When a message m is received then an arbitrary number of messages before m can be dequeued (lossiness) but the message itself stays in the queue (duplication). As another example, a perfect medium, which can however reorder messages, can be modelled as a multiset (we call it BAG). Mathematically, the medium can be represented as a function f from the set of messages *Msgs* to nonnegative integers. The operation *send*(m) is then implemented as $f(m) := f(m) + 1$, *available*(m) is simply returning $f(m) > 0$ and *receive*(m) is equivalent to $f(m) := f(m) - 1$. Finally, an unreliable medium with reordering can be represented as a set SET of messages that have been already sent. Now *send*(m) means $\text{SET} := \text{SET} \cup \{m\}$, availability is checking the presence of the message in SET and receive does not modify SET, this models duplication of messages.

2.3 Semantics of Abstract Protocol Model

Let us assume a given communication policy. The semantics of an abstract protocol model $(\mathit{Msgs}, \mathit{Roles})$ where $\mathit{Roles} = \{A_1, A_2, \ldots, A_n\}$ is given as transition system with states (configurations) of the form $(s_1, s_2, \ldots, s_n, \mathit{med})$ where $s_i \in S_{A_i}$ for all i, $1 \leq i \leq n$, are the current states of all roles and $\mathit{med} \in \mathit{Medium}$ represents the current content of the communication medium. The transitions are defined by $(s_1, \ldots, s_i, \ldots, s_n, \mathit{med}) \Rightarrow (s_1, \ldots, s'_i, \ldots, s_n, \mathit{med}')$ whenever $s_i \xrightarrow{m,m'}_{A_i} s'_i$ is a transition of the role A_i such that $\mathit{med}.\mathit{available}(m)$ is true and $\mathit{med}' = (\mathit{med}.\mathit{receive}(m)).\mathit{send}(m')$. By $\Rightarrow^*$ we denote the reflexive and transitive closure of $\Rightarrow$.

Consider now our running example from Figure 2. Clearly, with FIFO policy the communication medium this protocol is unbounded due to the possibility of unbounded message retransmission. Also STUTT-FIFO is unbounded due to e.g. the alternating resubmission of the messages *exitedB* and *exitedC*

in the ended states. Due to the resubmission of messages also BAG makes the medium of our example protocol unbounded. On the other hand, for SET the state-space remains finite and we can construct it algorithmically.

2.4 Analysis of Abstract Protocol Models

In the analysis of protocols we are interested in state reachability problems.
State Reachability Problem: given a target state s of a role A_i and the initial configuration of the protocol $c^0 = (s_1^0, s_2^0, \ldots, s_n^0, med^0))$ where med^0 contains no messages, we ask whether there is a reachable protocol configuration $c = (s_1, \ldots, s_{i-1}, s, s_{i+1}, \ldots, s_n, med)$ such that $c^0 \Rightarrow^* c$ for some states $s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n$ and some medium configuration med.

The state reachability problem can provide safety guarantees about the protocol behaviour. Considering our example from Figure 2 with the SET communication policy, it is possible for role A to reach an invalid state. This can be interpreted as an error in the protocol design as the protocol is not immune to reordering of messages. On the other hand, we may want to verify that for order-preserving communication policies, the protocol is safe (does not enter any invalid state). However, this is impossible to do in a fully automatic way as the model with FIFO communication has a full Turing power [4] and the corresponding transition system cannot be enumerated as it has in general infinitely many reachable configurations. There are similar problems with infinite state-spaces for the media STUTT-FIFO and BAG. For BAG the state reachability problem can be shown equivalent to the EXPSPACE-complete coverability problem on Petri nets (see e.g. [5]). In this paper we shall suggest different techniques that will allow us to efficiently answer the state reachability problem on a practically interesting set of protocol models.

Returning to the hierarchy of communication media, it is easy to see that if we show that a state is not reachable under the SET policy, it will not be reachable in any other policy below it. On the other hand, if a state is reachable in FIFO communication, then it will be reachable (with exactly the same trace) also in any communication policy above it.

We shall conclude this section with the discussion of some instances of the state reachability problem relevant for the verification of WS protocols. We assume that all roles contain at least three states called *Active*, *Ended*, *Invalid* representing the state where each role starts, where it ends (both successfully or with a failure) and finally a state representing inconsistency in the protocol design. Such states are often present in the standard specification documents for web services like WS-BA [14]. The following questions will be of interest.

- **Boundedness:** We ask whether starting with all roles in their *Active* states and the empty medium with a given (finite) capacity, is it the case that for all executions the medium does not exceed its capacity?

- **Correctness:** We ask whether starting with all roles in their *Active* states and the empty communication medium, do all roles avoid entering their *Invalid* states in all possible executions?
- **Termination:** We ask whether starting with all roles in their *Active* states and the empty communication medium, is there an execution where all roles reach their *Ended* states?
- **Deadlock-Freeness:** We ask whether starting with all roles in their *Active* states and the empty medium, is there a possible continuation from any reachable configuration but the one where all roles are in their *Ended* states?

The aim is to design protocols that are correct (cannot reach invalid states), can terminate and have no deadlocks. We call such protocols *safe*. As already discussed, most protocols that communicate over FIFO-like channels are not bounded. In the next section we discuss possible approaches that will allow us to prove that the protocol in question is *safe* even for an unbounded medium.

3 Abstractions of Communication Media

Let us consider a situation where a given protocol is sensitive to the order of message arrivals (and hence cannot be proved safe with the SET medium), but at the same time we wish to automatically establish its safety with respect to order-preserving communication policies like FIFO or STUTT-FIFO. We shall now suggest two abstraction strategies to tackle the problem that FIFO and STUTT-FIFO channels are in general unbounded. Both strategies provide an over-approximation of the communication medium, meaning that if the protocol is proved correct under the abstracted communication policy, it will be correct also under FIFO and STUTT-FIFO.

The main reason for introducing the abstractions is to guarantee that the state-space of the corresponding transition system becomes bounded and automatic analysis can be performed. As the problem is in general undecidable [4] and model checking of protocols communicating over FIFO channels is hence impossible, the proposed abstractions are not universal. On some protocols, the abstractions may not guarantee boundedness of the medium. On others the abstractions may be too coarse and thus they may not allow us to verify safety of the protocol, even though it is actually safe for the perfect FIFO communication. Nevertheless, as we demonstrate in our case studies provided in Section 5, the proposed abstractions are sufficient to establish safety of several well-known WS protocols.

3.1 Multiple Channel Optimization

Under the perfect FIFO or STUTT-FIFO communication policy, one can think of each sent message as being time-stamped. The property of the medium is

that it delivers the messages in the order in which they were time-stamped. In other words, the global order of messages is preserved and the medium can be seen as one universal FIFO or STUTT-FIFO channel.

In our first abstraction, called *multiple channel optimization*, we will relax the global order-preserving requirement and introduce several independent communication channels such that only messages sent via the same channel preserve their relative ordering, but two different channel do not synchronize in any way. We may possibly create a separate channel for each message which would in result give us either a communication policy equivalent to BAG (when applied to FIFO) or SET (when applied to STUTT-FIFO). This will clearly not help us with the automatic analysis as we apply the abstractions only to protocols that are not correct under the BAG or SET communication policies. Hence we instead introduce a more refined multiple channel optimization.

The idea is that for each message m that appears in the protocol description, we will compute the function $\mathit{recipients}(m)$ which contains all roles that can possibly receive the message m. On our running STP protocol example from Figure 2 we get the following: $\mathit{recipients}(\mathit{exitB}) = \{B\}$, $\mathit{recipients}(\mathit{preparingB}) = \{A\}$, $\mathit{recipients}(\mathit{exitC}) = \{C\}$, $\mathit{recipients}(\mathit{exitedB}) = \{A\}$, $\mathit{recipients}(\mathit{exitedC}) = \{B\}$. In the STP protocol each message has exactly one recipient, hence the sets are singletons. In general scenarios that include broadcast or multi-party communication, a message can have several recipients.

Formally, for a given protocol $(\mathit{Msgs}, \mathit{Roles})$ where each role $A \in \mathit{Roles}$ has its description $(S_A, \longrightarrow_A)$ we define for each $m \in \mathit{Msgs} \setminus \{\diamond\}$ its recipients: $\mathit{recipients}(m) = \{A \in \mathit{Roles} \mid (s, m, m', s') \in\longrightarrow_A,\ s, s' \in S_A,\ m' \in \mathit{Msgs}\}$. Let $\mathit{channels} = \{\mathit{recipients}(m) \mid m \in \mathit{Msgs}\}$ serve as names of newly introduced communication channels. Now every time a message m is sent, it arrives to the channel $\mathit{recipients}(m)$ and whenever a role checks the availability of a message, it does so on the channel $\mathit{recipients}(m)$. As a result, messages that arrive to the same channel preserve their relative order but messages in two different channels are unordered.

The multiple channel optimization process described above can be run in a fully automatic way (as implemented in our tool) and it has proved particularly useful to verify protocols like Business Agreement with Coordinator Completion protocol from the WS-Business Activity coordination framework [14].

3.2 Unordered Messages

We shall now discuss another abstraction technique, motivated by the fact that the multiple channel optimization may not be sufficient to achieve boundedness of the communication medium as it can be seen e.g. in our STP protocol from Figure 2. The reason here is that the role B can receive the messages

exitB and *exitedC* and both these messages can be repeatedly retransmitted in an arbitrary order, causing the unboundedness of the FIFO and STUTT-FIFO medium, even with multiple channel optimization.

Nevertheless, the protocol is still correct (as also formally verified in Section 5) in the sense that there are no invalid states reachable as long as messages cannot be reordered. In order to show this, we may notice that for example the ordering of the message *exitB* relative to the other messages does not seem to be relevant. We will mark it (using the symbol * in our tool implementation) as a message where it is not necessary to preserve its ordering. Formally, this means that we introduce an additional communication channel behaving as a SET medium such that all marked messages are sent/received to/from this channel. If all messages get marked then we get the SET communication medium. As showed later, marking the single message *exitB* is sufficient to prove the boundedness of the medium, while at the same time allowing us to prove that no invalid states can be reached.

One issue with this abstraction is the selection of messages for marking, as it is up to the designer of the protocol to mark unordered messages. In principle one may automate the process by exploring all combinations of marked/unmarked messages, however, for larger protocols this may not be computationally feasible due to exponentially many such combinations. The development of possible heuristics so that the markings of messages that are more likely to work (provide a bounded medium but still avoid the reachability of invalid states) are enumerated first is left for future research.

4 Automatic Analysis and Tool Support

We shall now outline a solution to the state reachability problem for communication protocols. The answer to this problem is provided by automatic translation of the state/transition tables into an intermediate XML format (denoted as part (i) in Figure 3), followed by a translation to networks of timed automata suitable for a direct verification in the model checker UPPAAL [3,21] (denoted as part (ii) in Figure 3).

As the reader can see, we created an intermediate XML representation of the state tables. The main motivation is that the translation (i) from state/transition tables to its XML representation can be replaced by another front end allowing us for example to describe a protocol with some domain-specific language and to translate it automatically into the XML format. This provides a better modularity of our proposed tool chain. The translation (i) is to a large extent a syntactic reformulation of the tables with added explicit definitions of states and messages that allow us to check for typos in the state/transition tables. This has proved useful during the creation of state/tables of nontrivial size in our applications.

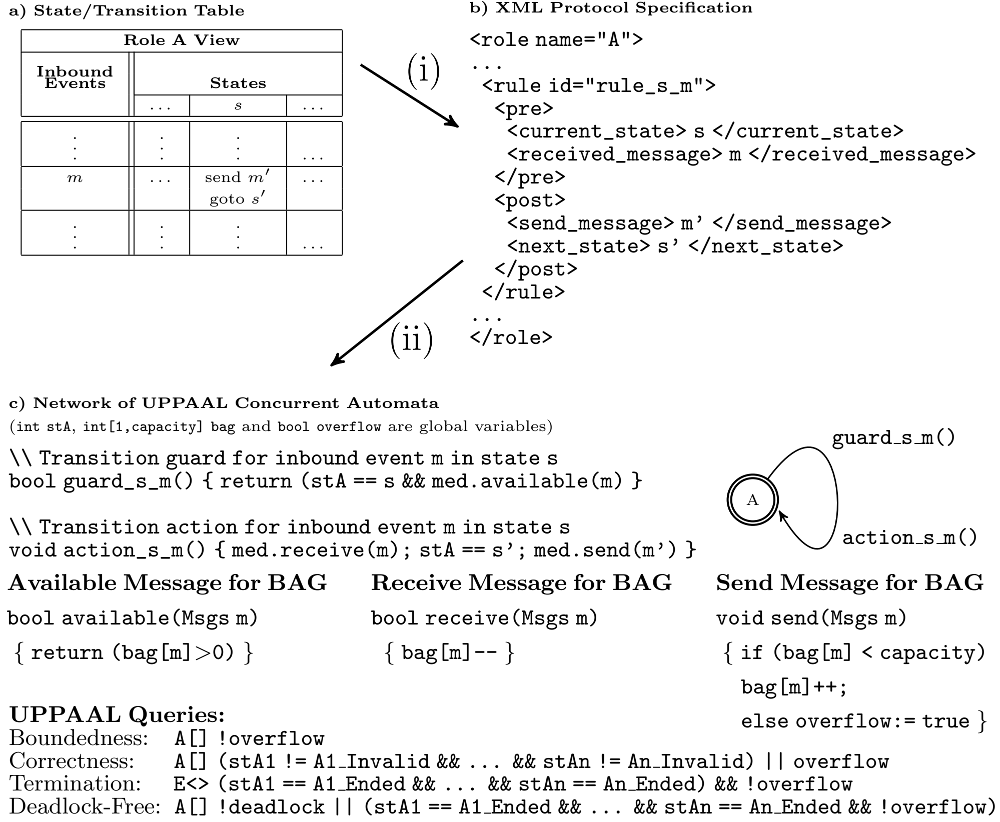


Fig. 3. Process of Automatic Analysis of WS-Protocols with the Medium BAG

In the second part of the translation, the XML description is further encoded into networks of communicating finite automata in the UPPAAL [21] style. UPPAAL is a tool for modelling, simulation and verification of networks of finite automata communicating via handshake and via shared variables. It allows the user to include a restricted C-like syntax for describing guards and updates of transitions and one of the main UPPAAL features is also the explicit treatment of timing information by using real-valued clocks. In the present paper we do not exploit the timing aspects but this will be considered in our future work. We refer the reader to [3] for a detailed introduction into UPPAAL modelling language.

In part (ii) of our translation each `pre` tag is converted into a transition guard (written as a function in C-like syntax accepted by UPPAAL) and each `post` tag is translated to an action (again represented in C-like syntax) that is performed should the transition be executed. The transition forms a loop for every rule of the given role as the data is stored in global variables. The states and message names are declared as global constants and global variables keep track of the states of each role (variable `stA` for role A in our example) as well as of the current content of the communication medium. In our example we

chose to demonstrate the obvious implementation of the communication policy BAG. Note that we assume a given capacity of the `bag` data structure (limited by the constant `capacity`). Should the protocol require more messages in transit, the global flag `overflow` is set to true. As UPPAAL allows a large set of C-constructs in its syntax, the more advanced media like FIFO and STUTT-FIFO (including the abstractions) are implemented in the expected way as in any other imperative programming language. The details can be found in our publicly available tool chain.

Finally, in Figure 3, we describe how our protocol related questions of boundedness, correctness, termination and deadlock-freeness are formulated in the UPPAAL query language (assuming the role names `A1, A2, ..., An`). The queries are formulated in a subset of CTL logic used in UPPAAL (for more info see [3]). Intuitively, the path quantification `A[]` stands for "for all reachable configurations holds that" and `E<>` stands for "there is a reachable configuration such that".

Tool Details

Translations (i) and (ii) from Figure 3 are implemented in the open source tool `csv2uppaal` available at [11]. The input state/transition tables are created in standard spreadsheet editors like OpenOffice and saved as csv files (textual representation of the tables). The csv files are then parsed using an awk script that generates the intermediate protocol description in the XML format with elements representing the messages, roles and their states and transition rules with pre and post conditions. The final part of the tool-chain is written in Ruby and generates files directly readable by UPPAAL (concurrent automata descriptions and a query file). Finally, in command line mode, the tool calls the UPPAAL verification engine to verify the properties of boundedness, correctness, termination and deadlock-freeness. For Mac OS we provide additionally also a graphical user interface. The outcome of the verification is the statistics with details about the protocol, medium, roles and messages, the verification results and possibly execution traces if relevant for the verified properties. The traces are printed in a human readable form. The use of the tool chain requires no expertise with the model checker UPPAAL and is accessible to WS protocol designers without any particular training in formal verification. On the other hand the advanced users may open the generated files in the UPPAAL GUI, experiment with simulating the protocol and ask more advanced queries that are protocol specific. For example in our running STP protocol from Figure 2 we verified an additional property saying that role A can enter the state *Ended* only after the roles B and C already reached their *Ended* states. This query is formulated as

```
A[] (stA!=A_Ended || (stB==B_Ended && stC==C_Ended))
```

Buffer Type	Properties	BAwCC		BAwPC		STP	ABP
		Org.	**Enh.**	**Org.**	**Enh.**		
BAG	Boundedness	no	no	no	no	no	no
	Correctness	NO	NO	NO	yes	NO	NO
SET	Boundedness	YES	YES	YES	**YES**	YES	YES
	Correctness	NO	NO	NO	**YES**	NO	NO
FIFO	Boundedness	no	no	no	no	no	no
	Correctness	yes?	yes	yes?	yes	yes	yes
STUTT-FIFO	Boundedness	no	no	no	no	no	no
	Correctness	NO	yes	NO	yes	yes	yes
multiple channel STUTT-FIFO	Boundedness	no	**YES**	no	YES	no	**YES**
	Correctness	NO	**YES**	NO	YES	yes	**YES**
multiple channel reorder STUTT-FIFO	Boundedness	YES	YES	YES	YES	**YES**	YES
	Correctness	NO	YES	NO	YES	**YES**	YES

Fig. 4. Summary of Verification Results (Org. means original, Enh. means enhanced)

and UPPAAL confirms that it holds.

5 Applications

In order to investigate the applicability of our proposed framework, we carried out experiments on case studies ranging from well-known academic examples like Alternating Bit Protocol (ABP) [2,10] and Subservice Termination Protocol (STP) described in this paper, to larger-size protocols from WS-Business Activity specification [14], namely Business Agreement with Coordination Completion (BAwCC) and Business Agreement with Participant Completion (BAwPC).

State/transition tables were described in OpenOffice as spreadsheets and then automatically verified using our `csv2uppaal` tool. The verification results are presented in Figure 4. As all considered protocols were deadlock-free and terminating (apart from ABP where termination is not desirable), we list only the answers for boundedness of the medium and correctness of the protocols (absence of invalid states). The answers "YES" and "NO" in capital letters mean that the tool returned a conclusive answer on the instances in question. The answer "yes" stands for the fact that even though the tool on this concrete medium was not bounded, we were able to conclude the answer using our abstraction techniques (in bold font are marked the prominent positive results that imply correctness for all other less-abstract media). Finally, the answer "no" on boundedness stands for the fact that for any chosen

medium capacity (where the verification terminated within a reasonable time limit) the answer was negative. As boundedness is an undecidable problem, a more precise answer cannot be obtained automatically in general. However, a manual examination of error traces revealed that for all our instances the medium was really unbounded.

The ABP was proved correct for order-preserving communication media by considering the STUTT-FIFO with multiple channel abstraction. For unordered asynchronous communication the protocol is (as expected) not correct. Similarly the STP was proved correct for all order-preserving communications by marking the message *exitB* as unordered and using multiple channel STUTT-FIFO communication.

Both in BAwCC and BAwPC protocols we discovered an error for all considered communication media except for perfect FIFO, where the correctness seems to be valid, though we were not able to prove it in automatic way, hence the answers "yes?". Example of a trace leading to an invalid state has been communicated to the OASIS body. The main reason for the problems is a confusion on retransmission of messages in the ended states. We suggested fixes to the protocols and designed enhanced versions of both protocols. The enhanced BAwPC protocol now turned out to be correct for the most general SET communication (and hence also for any less abstract one) while the enhanced version of BAwCC still contains traces leading to invalid states. The issue here is more subtle and it has been announced on the OASIS discussion forum [16] and a correction is currently under development. Using our automatic analysis we were able to identify that the issue is connected with reordering of messages and not with the lossiness of the medium (the protocol is incorrect even for BAG).

6 Conclusion

We presented an automatic, tool-supported framework for modelling and analysis of communication protocols with a particular focus on web services protocols. A particular strength of our solution is the possibility to choose different models of communication media and various abstractions in order to prove protocol correctness. The approach was successfully tested on e.g. protocols from WS-BA, where in one case we confirmed the presence of a fundamental problem in the protocol design and in the other one we suggested an improvement in the specification sufficient to automatically validate its correctness.

Our tool chain is modular as the state/transition tables are first translated to an intermediate XML format that is further translated to UPPAAL automata. In the future work we plan to provide different front ends that will accept other popular formats for describing protocols, including parameterization, simple data structures, and timing aspects.

References

[1] Barghouti, N., N. Nounou and Y. Yemini, *An integrated protocol development environment*, in: *Protocol Specification Testing and Verification VI* (1987), pp. 63–69.

[2] Bartlett, K. A., R. A. Scantlebury and P. T. Wilkinson, *A note on reliable full-duplex transmission over half-duplex links*, Commun. ACM **12** (1969), pp. 260–261.

[3] Behrmann, G., A. David and K. Larsen, *A tutorial on UPPAAL*, in: *Proc. of SFM-RT'04*, number 3185 in LNCS (2004), pp. 200–236.

[4] Brand, D. and P. Zafiropulo, *On communicating finite-state machines*, J. ACM **30** (1983), pp. 323–342.

[5] Esparza, J., *Decidability and complexity of Petri net problems — An introduction*, in: W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, LNCS **1491**, Springer Berlin / Heidelberg, 1998 pp. 374–428.

[6] Finkel, A., *Decidability of the termination problem for completely specified protocols*, Distributed Computation **7** (1994), pp. 129–135.

[7] Holzmann, G., "Design and Validation of Computer Protocols," Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[8] Johnson, J., D. E. Langworthy, L. Lamport and F. H. Vogt, *Formal specification of a web services protocol*, Journal of Logic and Algebraic Programming **70** (2007), pp. 34–52.

[9] Lohmann, N., *Communication models for services*, in: *Proc. of ZEUS'10*, CEUR Workshop Proceedings **563** (2010), pp. 9–16.

[10] Lynch, W. C., *Computer systems: Reliable full-duplex file transmission over half-duplex telephone line*, Commun. ACM **11** (1968), pp. 407–410.

[11] Marques, A., A. Ravn, J. Srba and S. Vighio, *The tool csv2uppaal*, available at http://www.cs.aau.dk/~srba/csv2uppaal.zip.

[12] Naumovich, G., L. Clarke and L. Osterweil, *Verification of communication protocols using data flow analysis*, SIGSOFT Softw. Eng. Notes **21** (1996), pp. 93–105.

[13] Newcomer, E. and I. R. (chairs), *Web services atomic transaction version 1.2* (2009), http://docs.oasis-open.org/ws-tx/wstx-wsat-1.2-spec.html.

[14] Newcomer, E. and I. R. (chairs), *Web services business activity version 1.2* (2009), http://docs.oasis-open.org/ws-tx/wstx-wsba-1.2-spec-os/wstx-wsba-1.2-spec-os.html.

[15] Newcomer, E. and I. R. (chairs), *Web services coordination version 1.2* (2009), http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec-os/wstx-wscoor-1.2-spec-os.html.

[16] OASIS, *Discussion forum, report on error trace in BAwCC* (2011), http://markmail.org/message/xgnyonkihfif5vz2.

[17] Ravn, A., J. Srba and S. Vighio, *A formal analysis of the web services atomic transaction protocol with UPPAAL*, in: *Proc. of ISOLA'10)*, LNCS **6416** (2010), pp. 579–593.

[18] Ravn, A., J. Srba and S. Vighio, *Modelling and verification of web services business activity protocol*, in: P. Abdulla and K. Leino, editors, *Proc. of TACAS'11*, LNCS **6605** (2011), pp. 357–371.

[19] Robinson, I., *Answer in WS-BA discussion forum, July 14th, 2010* (2010), http://markmail.org/message/wriewgkboaaxw66z.

[20] Schnoebelen, P., *Verifying lossy channel systems has nonprimitive recursive complexity*, Information Processing Letters **83** (2002), pp. 251–261.

[21] UPPAAL, http://www.uppaal.com.

[22] Vuong, S. T., D. D. Hui and D. D. Cowan, *Valira — a tool for protocol validation via reachability analysis*, in: *Protocol Specification, Testing and Verification VI* (1987), pp. 35–41.

A Formal Approach to Data Validation Constraints in MDE

Alessandro Rossini, Khalid A. Mughal, Uwe Wolter

Department of Informatics
University of Bergen
Bergen, Norway

Adrian Rutle, Yngve Lamo

Department of Computer Engineering
Bergen University College
Bergen, Norway

Abstract

Software security encompasses the measures taken to ensure confidentiality, integrity and availability in software systems. In present-day software development, security is often an afterthought rather than part of the software development life-cycle. In order to reveal potential security flaws before a software system is actually implemented, security aspects should be taken into account starting from the early phases of the development. With model-driven engineering (MDE) gaining momentum in both academia and industry, an interesting challenge is the specification of security constraints within software models. In this paper we focus on data validation – the process of ensuring that a system operates on correct and meaningful data – in the context of MDE. Our contribution is a formal approach to the specification of data validation constraints which involve multiple structural properties. In addition, constraints specified at model level are mapped to Java annotations which are then transformed to executable tests by an existing data validation framework.

Keywords: data validation; model-driven engineering; category theory; Diagram Predicate Framework; SHIP Validator

1 Introduction

Software systems are nowadays widespread in all walks of society. Violating the confidentiality, integrity and availability of these systems can therefore lead to a negative impact on the economy and health. Software security aims at ensuring that these properties are not compromised. In present-day software development, security is often neglected because of lack of skills and budget, and time-to-market

constraints. Typically, security concerns are considered far too late when the system is already nearing deployment. This is clearly insufficient since security aspects should be taken into account starting from the early phases of the development [8,11] in order to reveal potential security flaws before a software system is actually implemented.

Model-driven engineering (MDE) is a branch of software engineering which aims at improving productivity, quality, and cost-effectiveness of software by shifting the paradigm from code-centric to model-centric. MDE promotes models and modelling languages as the main artefacts of the development process and model transformation as the primary technique to generate (parts of) software systems out of models. Models enable developers to reason at a higher level of abstraction while model transformation alleviates developers from repetitive and error-prone tasks such as coding.

In this regard, an interesting challenge is the specification of security constraints within models. In this paper we focus on data validation – the process of ensuring that a system operates on correct and meaningful data – in the context of MDE. The lack of proper data validation is listed as the most prevalent cause of software vulnerabilities by the OWASP [14].

In the state-of-the-art of MDE, models are typically specified by means of modelling languages such as the Unified Modeling Language (UML) [13]. These modelling languages are diagrammatic and allow for the specification of constraints on single structural properties, e.g., a data validation constraint on a single input field. However, the specification of complex constraints on multiple structural properties, e.g., data validation constraint on multiple input fields, requires textual constraint languages such as the Object Constraint Language (OCL) [12].

It is the authors' experience that a completely diagrammatic approach to the specification of data validation constraints in MDE would be desirable [17]. The contribution of this paper is a formal approach to the specification of data validation constraints which can involve multiple, interdependent structural properties. The underpinning of the proposed approach is the Diagram Predicate Framework (DPF) [15,16,17,18] which provides a formalisation of (meta)modelling and model transformation based category theory [1] and graph transformation [5]. The paper also shows how data validation constraints specified at model level are mapped to Java annotations. These annotations are in turn transformed to executable tests at run-time by the SHIP Validator [7,10], a Java based framework which enables the validation of multiple interdependent properties of Java objects.

The remainder of the paper is structured as follows. Section 2 presents DPF. Section 3 introduces the formal approach to data validation by means of a running example. In Section 4, the current research in security within MDE is summarised. Finally, in Section 5, some concluding remarks and ideas for future work are outlined.

2 Diagram Predicate Framework

Before introducing DPF, the terminology adopted in this paper is clarified. The term *model* has different meanings in different contexts. In software engineering, a model denotes “an abstraction of a (real or language-based) system allowing predictions or inferences to be made” [9]. Models in software engineering are typically diagrammatic.

The term *diagram* has also different meanings in different contexts. In software engineering, a diagram denotes a structure which is based on graphs, i.e., a collection of nodes together with a collection of arrows between nodes. Graphs are a well-known and well-understood means to represent structural and behavioural properties of a software system [5], e.g., Entity-Relationship (ER) diagrams and UML diagrams [13].

Since graph-based structures are often visualised in a natural way, the terms *diagrammatic* and *visual* and are often treated as synonyms. In this paper, however, visualisation and diagrammatic syntax are clearly distinguished; i.e., this work focuses on syntax and semantics of diagrammatic models independent of their visualisation.

In DPF, a model is represented by a *specification* $\mathfrak{S}$. A specification $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$ consists of an *underlying graph* S together with a set of *atomic constraints* $C^{\mathfrak{S}}$ which are specified by a *signature* Σ. A signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$ consists of a collection of *predicates* $\pi \in \Pi^{\Sigma}$, each having an arity (or shape graph) $\alpha^{\Sigma}(\pi)$, a proposed visualisation and a semantic interpretation. An atomic constraint (π, δ) consists of a predicate $\pi \in \Pi^{\Sigma}$ together with a graph homomorphism $\delta : \alpha^{\Sigma}(\pi) \to S$ from the arity of the predicate to the underlying graph of the specification.

Definition 2.1 [Signature] A signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$ consists of a collection of predicate symbols Π^{Σ} and a map α^{Σ} which assigns a graph to each predicate symbol $\pi \in \Pi^{\Sigma}$. $\alpha^{\Sigma}(\pi)$ is called the *arity* of the predicate symbol π.

Definition 2.2 [Atomic constraint] Given a signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$, an atomic constraint (π, δ) on a graph S consists of a predicate symbol $\pi \in \Pi^{\Sigma}$ and a graph homomorphism $\delta : \alpha^{\Sigma}(\pi) \to S$.

Definition 2.3 [Specification] Given a signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$, a specification $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$ consists of a graph S and a set $C^{\mathfrak{S}}$ of atomic constraints (π, δ) on S with $\pi \in \Pi^{\Sigma}$.

The semantics of nodes and arrows of the underlying graph of a specification has to be chosen in a way which is appropriate for the corresponding modelling environment [17]. In object-oriented structural modelling, each object may be related to a set of other objects. Hence, it is appropriate to interpret nodes as sets and arrows $\mathsf{X} \xrightarrow{\mathsf{f}} \mathsf{Y}$ as multi-valued functions $f : X \to \wp(Y)$. The powerset $\wp(Y)$ of

Y is the set of all subsets of Y, i.e., $\wp(Y) = \{A \mid A \subseteq Y\}$. Moreover, the composition of two multi-valued functions $f : X \to \wp(Y)$, $g : Y \to \wp(Z)$ is defined by $(f;g)(x) := \bigcup\{g(y) \mid y \in f(x)\}$.

Example 2.4 [Signature and specification] Let us consider an information system for the management of students and universities. The information system has the following requirements:

(i) A student studies at *one to four* universities.

(ii) A university educates *none to many* students.

Table 1 shows a signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$ which is suitable for object-oriented structural modelling.

Table 1
A signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$

π	$\alpha^{\Sigma}(\pi)$	**Proposed vis.**	**Semantic interpretation**
`[mult(`m, n`)]`	$1 \xrightarrow{a} 2$	X —f [m..n]→ Y	$\forall x \in X : m \leq \lvert f(x) \rvert \leq n$, with $0 \leq m \leq n$ and $n \geq 1$
`[injective]`	$1 \xrightarrow{a} 2$	X —f [inj]→ Y	$\forall x, x' \in X : f(x) = f(x')$ implies $x = x'$
`[surjective]`	$1 \xrightarrow{a} 2$	X —f [surj]→ Y	$\forall y \in Y\ \exists x \in X : y \in f(x)$
`[inverse]`	$1 \underset{b}{\overset{a}{\rightleftarrows}} 2$	X —f→ Y, Y —g→ X [inv]	$\forall x \in X$, $\forall y \in Y$: $y \in f(x)$ iff $x \in g(y)$

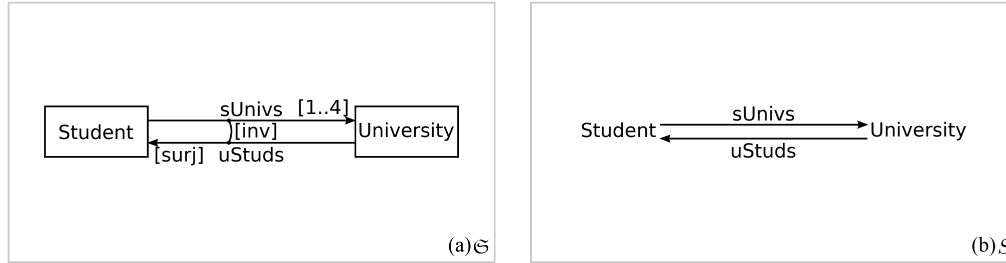


Figure 1. A specification $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$ and its underlying graph S

Fig. 1(a) shows a specification $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$ which is compliant with the requirements above. Fig. 1(b) shows the underlying graph S of $\mathfrak{S}$, i.e., the graph of $\mathfrak{S}$ without any atomic constraints.

In $\mathfrak{S}$, the nodes Student and University are interpreted as sets $Student$ and $University$, and the arrows sUnivs and uStuds are interpreted as multi-valued functions $sUnivs : Student \to \wp(University)$ and $uStuds : University \to \wp(Student)$, respectively.

Based on the requirement i, the function $sUnivs$ has cardinality between one and four. This is enforced by the atomic constraint $(\texttt{[mult(}1,4\texttt{)]}, \delta_1)$ on the arrow sUnivs. Moreover, the function $uStuds$ is *surjective*. This is enforced by the atomic constraint $(\texttt{[surjective]}, \delta_3)$ on the arrow uStuds. Finally, the functions $sUnivs$ and $uStuds$ are *inverse* of each other, i.e., $\forall s \in Student$ and $\forall u \in University : s \in uStuds(u)$ iff $u \in sUnivs(s)$. This is enforced by the atomic constraint $(\texttt{[inverse]}, \delta_2)$ on sUnivs and uStuds. The graph homomorphisms δ_1, δ_2 and δ_3 are defined as follows (see Table 2):

$$\delta_1(1) = \text{Student}, \quad \delta_1(2) = \text{University}, \quad \delta_1(a) = \text{sUnivs}$$
$$\delta_2(1) = \text{Student}, \quad \delta_2(2) = \text{University}, \quad \delta_2(a) = \text{sUnivs}, \quad \delta_2(b) = \text{uStuds}$$
$$\delta_3(1) = \text{University}, \quad \delta_3(2) = \text{Student}, \quad \delta_3(a) = \text{uStuds}$$

Table 2
The atomic constraints $(\pi, \delta) \in C^{\mathfrak{S}}$ and their graph homomorphisms

(π, δ)	$\alpha^{\Sigma}(\pi)$	$\delta(\alpha^{\Sigma}(\pi))$
$(\texttt{[mult(}1,4\texttt{)]}, \delta_1)$	$1 \xrightarrow{a} 2$	Student $\xrightarrow{\text{sUnivs}}$ University
$(\texttt{[inverse]}, \delta_2)$	$1 \underset{b}{\overset{a}{\rightleftarrows}} 2$	Student $\underset{\text{uStuds}}{\overset{\text{sUnivs}}{\rightleftarrows}}$ University
$(\texttt{[surjective]}, \delta_3)$	$1 \xrightarrow{a} 2$	University $\xrightarrow{\text{uStuds}}$ Student

Remark 2.5 [Predicate symbols] Some of the predicate symbols in Σ (see Table 1) refer to single predicates, e.g., `[surjective]`, while some others refer to a family of predicates, e.g., $\texttt{[mult(}m,n\texttt{)]}$. In the case of $\texttt{[mult(}m,n\texttt{)]}$, the predicate is parametrised by the (non-negative) integers m and n, which represent the lower and upper bounds, respectively, of the cardinality of the function which is constrained by this predicate.

The semantics of predicates of the signature Σ (see Table 1) is described using the mathematical language of set theory. In an implementation, the semantics of a predicate is typically given by the code of a corresponding validator where both the mathematical and the validator semantics should coincide. However, it is not necessary to choose between the above mentioned possibilities; it is sufficient to know that any of these possibilities defines valid instances of predicates.

Definition 2.6 [Semantics of predicates] Given a signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$, a semantic interpretation $[\![..]\!]^{\Sigma}$ of Σ consists of a mapping that assigns to each predicate symbol $\pi \in \Pi^{\Sigma}$ a set $[\![\pi]\!]^{\Sigma}$ of graph homomorphisms $\iota : O \to \alpha^{\Sigma}(\pi)$, called valid instances of π, where O may vary over all graphs. $[\![\pi]\!]^{\Sigma}$ is assumed to be closed under isomorphisms.

The semantics of a specification is defined in the so-called *fibred* way [4,20]; i.e., the semantics of a specification is given by the set of its instances. An instance

(I, ι) of a specification $\mathfrak{S}$ consists of a graph I together with a graph homomorphism $\iota : I \to S$ which satisfies the set of atomic constraints $C^{\mathfrak{S}}$.

To check that an atomic constraint is satisfied in a given instance of a specification $\mathfrak{S}$, it is enough to inspect only the part of $\mathfrak{S}$ which is affected by the atomic constraint. This kind of restriction to a subpart is obtained by the pullback construction [1], which can be regarded as a generalisation of the inverse image construction.

Definition 2.7 [Instance of specification] Given a specification $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$, an instance (I, ι) of $\mathfrak{S}$ consists of a graph I and a graph homomorphism $\iota : I \to S$ such that for each atomic constraint $(\pi, \delta) \in C^{\mathfrak{S}}$ we have $\iota^* \in [\![\pi]\!]^{\Sigma}$, where the graph homomorphism $\iota^* : O^* \to \alpha^{\Sigma}(\pi)$ is given by the following pullback:

$$\begin{array}{ccc} \alpha^{\Sigma}(\pi) & \xrightarrow{\delta} & S \\ \uparrow{\scriptstyle \iota^*} & P.B. & \uparrow{\scriptstyle \iota} \\ O^* & \xrightarrow[\delta^*]{} & I \end{array}$$

3 Data Validation

A running example based on [7,10] is adopted to show how the formal approach to (meta)modelling can be applied to the problem of data validation. Note that the example is kept intentionally simple, retaining only the details which are relevant for the discussion.

Example 3.1 [International money transfers] Let us consider international money transfers. *IBAN* (International Bank Account Number) is the standard for identifying bank accounts internationally. Some countries have not adopted this standard and, for money transfer to these countries, a special *clearing code* is needed in combination with the plain *account number*. *BIC* (Bank Identifier Code) is the standard for identifying banks globally. Therefore, a form for international money transfers should contain (at least) the input fields bic, iban, account and clearingCode. Moreover, supposing that the currency is Euro, the form should also contain the input fields amountEuros and amountCents. In addition, the transfer system should satisfy the following requirements:

(i) The BIC code of the beneficiary's bank is required.

(ii) Either the IBAN or both clearing code and account number are required.

(iii) The amount to transfer must be between 0.01 and 100000.00 Euros.

Table 3 shows a signature $\Phi = (\Pi^{\Phi}, \alpha^{\Phi})$ which contains predicates used to specify data validation constraints.

Note that in the semantic interpretation of the `[cross-range]` predicate we denote lexicographical order by $\leq$.

Table 3
The data validation signature Φ

π	$\alpha^{\Phi}(\pi)$	**Proposed vis.**	**Semantic interpretation**
`[required]`	$1 \xrightarrow{a} 2$	X —f→ Y	$\forall x \in X : f(x)$ defined
`[exactly-one-null]`	$1 \xrightarrow{a} 2$, $1 \xrightarrow{b} 3$	X —f→ Y, X —g→ Z, [eon]	$\forall x \in X : (f(x)$ defined and $g(x)$ undefined) or ($f(x)$ undefined and $g(x)$ defined)
`[all-or-none-null]`	$1 \xrightarrow{a} 2$, $1 \xrightarrow{b} 3$	X —f→ Y, X —g→ Z, [aonn]	$\forall x \in X : (f(x)$ defined and $g(x)$ defined) or ($f(x)$ undefined and $g(x)$ undefined)
`[cross-range-` `(`$(m_1,n_1),(m_2,n_2)$`)]`	$1 \xrightarrow{a} 2$, $1 \xrightarrow{b} 2$	X —f, g→ Int, $[m_1.n_1 - m_2.n_2]$	$\forall x \in X : (m_1,n_1) \leq (f(x),g(x)) \leq (m_2,n_2)$
`[range(`m,n`)]`	$1 \xrightarrow{a} 2$	X —f→ Int, [m−n]	$\forall x \in X : m \leq f(x) \leq n$

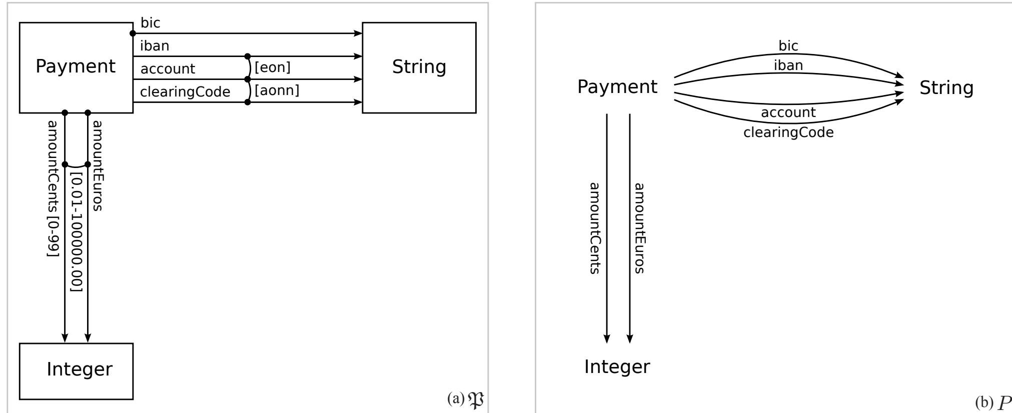


Figure 2. The specification $\mathfrak{P} = (P, C^{\mathfrak{P}} : \Phi)$ and its underlying graph P

Fig. 2(a) shows a specification $\mathfrak{P} = (P, C^{\mathfrak{P}} : \Phi)$ which is compliant with the requirements above. The form is represented by the node Payment while the input fields are represented by the arrows bic, iban, account, clearingCode, amountEuros and amountCents. Fig. 2(b) presents the underlying graph P of $\mathfrak{P}$, i.e., the graph of $\mathfrak{P}$ without any atomic constraints.

In $\mathfrak{P}$, the requirement i is enforced by the atomic constraint (`[required]`, δ_1) on the arrow bic, i.e., $\delta_1 : (1 \xrightarrow{a} 2) \mapsto (\text{Payment} \xrightarrow{\text{bic}} \text{String})$. This atomic constraint ensures that the user provides a value in the input field bic. Moreover, the requirement ii is enforced in $\mathfrak{P}$ by two atomic constraints: (`[exactly-one-null]`,

δ_2) on the arrows iban and account together with ([all-or-none-null], δ_3) on the arrows account and clearingCode. These atomic constraints ensure that a user provides values in either the input field iban or both the input fields account and clearingCode. Furthermore, the requirement iii is enforced in $\mathfrak{P}$ by the atomic constraint ([cross-range$((0,1),(100000,0))$], δ_4) on the arrows amountEuros and amountCents. This atomic constraint ensures that the user provides values in the input fields amountEuros and amountCents which sum up to a value within the range 0.01 to 100000.00. In addition, the atomic constraint ([range$(0,99)$], δ_5) on the arrow amountCents ensures that a user provides a value in the input field amountCents within the range 0 to 99.

Fig. 3(b) shows a valid instance I of the specification $\mathfrak{P} = (P, C^{\mathfrak{P}}\colon \Phi)$. Fig. 3 also shows the mappings of the graph homomorphism $\iota : I \to P$ as dashed, grey arrows.

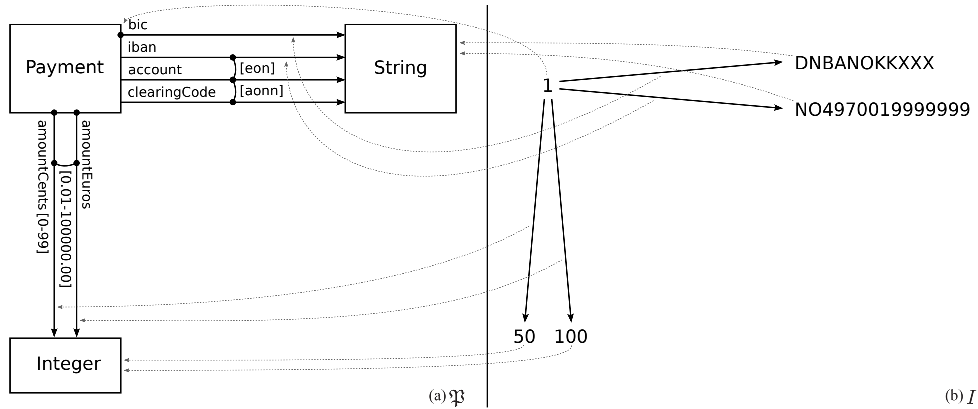


Figure 3. The specification $\mathfrak{P} = (P, C^{\mathfrak{P}}\colon \Phi)$ and a possible instance I

As mentioned, in an implementation, the semantics of a predicate is typically given by the code of a corresponding validator where both the mathematical and the validator semantics should coincide. In this paper, we have chosen to base the implementation of each predicate on the SHIP Validator [7,10]. The XMI serialisation (see Listing 1) of the specification $\mathfrak{P} = (P, C^{\mathfrak{P}} : \Phi)$ specifying the form in Example 3.1 can be transformed to a Java class (see Listing 2) tagged by Java annotations compatible with the SHIP Validator. For each atomic constraint $(\pi, \delta) \in C^{\mathfrak{P}}$ a corresponding Java annotation is attached to the getter methods of the Java class. Note that an atomic constraint on a single arrow, e.g., ([required], δ_1) on the arrow bic, translates to a single Java annotation, e.g., `@Required` on the method `getBic()`. Likewise, an atomic constraint on multiple arrows, e.g., ([exactly-one-null], δ_2) on the arrows iban and account, translates to multiple Java annotations, e.g., `@ExactlyOneNull` on the methods `getIban()` and `getAccount()`. The interested reader can download a proof-of-concept implementation of a code generator from [2].

Listing 1: XMI serialisation of the specification $\mathfrak{P} = (P, C^{\mathfrak{P}}{:}\Phi)$

```
1  <?xml version="1.0" encoding="ASCII"?>
2  <no.hib.dpf.metamodel:Specification
3  xmlns:no.hib.dpf.metamodel="http://no.hib.dpf.metamodel"
4  id="9090a2ec-0e36-4fcc-8f04-3a0226f0a938" name="P">
5
6   <node id="525d2a64-66e1-42f8-aec9-9f186379a77b" name="Payment"/>
7   <node id="d3ae4964-d091-41d7-9127-09856b3ce316" name="String"/>
8   <node id="0cac0671-a7e0-4d99-8216-14d24f186375" name="Integer"/>
9
10  <arrow id="b5a45cda-3ee0-42a0-a568-81f9e92d7e25" name="bic" source="//@node.0"
        target="//@node.1"/>
11  <arrow id="ad030229-b66c-40b5-8f7f-59f1a25e24a8" name="iban" source="//@node.0"
        target="//@node.1"/>
12  <arrow id="1d54b8c6-a51b-4858-ade9-0a66522b80eb" name="account" source="//@node
        .0" target="//@node.1"/>
13  <arrow id="2c4b8f89-dc27-44e6-bdb4-a0e298c26f85" name="clearingCode" source="//
        @node.0" target="//@node.1"/>
14  <arrow id="07a4001b-4c8e-461f-a845-4ac985b0c36d" name="amountEuros" source="//
        @node.0" target="//@node.2"/>
15  <arrow id="7559cb35-863a-49dd-a2b3-3e9e893c1356" name="amountCents" source="//
        @node.0" target="//@node.2"/>
16
17  <constraints id="33003eb9-d287-4bd8-9a28-ccf6d3ea9ee0" type="[required]">
18    <arrow source="//@arrow.0" />
19  </constraints>
20
21  <constraints id="33003eb6-7987-4558-ba28-aaf693349ee0" type="[not-required]">
22    <arrow source="//@arrow.1" />
23    <arrow source="//@arrow.2" />
24    <arrow source="//@arrow.3" />
25    <arrow source="//@arrow.4" />
26    <arrow source="//@arrow.5" />
27  </constraints>
28
29  <constraints id="e0661dc3-0620-44e6-af54-07bf14875c16" type="[exactly-one-null]">
30    <arrow source="//@arrow.1" />
31    <arrow source="//@arrow.2" />
32  </constraints>
33
34  <constraints id="1160e483-b701-4c23-9641-7e73909de528" type="[all-or-none-null]">
35    <arrow source="//@arrow.2" />
36    <arrow source="//@arrow.3" />
37  </constraints>
38
39  <constraints id="e1f2bab1-b58c-4273-97bb-d0cdd14abe45" type="[cross-range]">
40    <param name="m1" value="0" />
41    <param name="n1" value="01" />
42    <param name="m2" value="10000" />
43    <param name="n2" value="00" />
44    <arrow source="//@arrow.4" />
45    <arrow source="//@arrow.5" />
46  </constraints>
47
48  <constraints id="9132c6e8-7af9-4fc6-8b67-afac0471b13b" type="[range]">
49    <param name="min" value="0" />
50    <param name="max" value="99" />
51    <arrow source="//@arrow.5" />
52  </constraints>
53
54 </no.hib.dpf.metamodel:Specification>
```

Listing 2: Java class generated by transformation

```
public class Payment {

  private String bic;
  private String iban;
  private String account;
  private String clearingCode;

  private int amountEuros;
  private int amountCents;

  @Required
  public String getBic() {
    return bic;
  }

  @ExactlyOneNull
  @NotRequired
  public String getIban() {
    return iban;
  }

  @ExactlyOneNull
  @AllOrNoneNull
  @NotRequired
  public String getAccount() {
    return account;
  }

  @AllOrNoneNull
  @NotRequired
  public String getClearingCode() {
    return clearingCode;
  }

  @IntRange(min=0,max=100000)
  @CrossRange
  public int getAmountEuros(){
    return this.amountEuros;
  }

  @IntRange(min=0,max=99)
  @CrossRange
  public int getAmountCents(){
    return this.amountCents;
  }

}
```

These Java annotations are in turn transformed into executable tests by the SHIP Validator. The interested reader can consult [7,10] for details about the implementation and execution of these tests. Note that the idea of using annotations to hide the actual validation code and, at the same time, tag the properties to be tested, allow the constraints to be easily integrated into existing code. Besides, the validation aspects of the system remain well separated from the application aspects. This separation of concerns facilitates the transformation of the diagrammatic constraints into actual existing working code.

4 Related Work

In [6], an approach to integrate input validation constraints into UML diagrams using OCL is presented. In particular, this approach targets four different UML diagrams, i.e., use case diagram, class diagram, sequence diagram and activity diagram. This solution enables the specification of input validation constraints on behavioural models while our approach targets structural models only. However, it adopts a textual constraint language such as OCL while our approach is completely diagrammatic.

In [8], the author illustrates an approach to enrich UML models with security requirements such as secrecy, integrity and authenticity. The approach exploits UML extension mechanisms such as keywords, tags and constraints. In particular, keywords are used together with tags to specify security requirements on the system, while constraints give criteria to determine if these requirements are satisfied by the UML model. However, keywords and tags can be attached only to single model elements, thus these mechanism are not sufficient to express data validation constraints involving multiple structural properties at the model level. On the contrary, data validation constraints involving multiple structural properties can be expressed in our approach in a diagrammatic fashion.

5 Conclusion and Future Work

In this paper, we have illustrated some of the key aspects of data validation in MDE. We have adopted DPF to define an approach to the specification of data validation constraints in models. Moreover, we have shown how these constraints can be mapped to Java annotations which are transformed to executable tests. The diagrammatic and formal nature of the proposed approach constitutes the main contribution and novelty of this work.

In a future work, we will introduce a reasoning system for the analysis of predicate dependencies and a logic for this analysis. This extension will enable users of the proposed approach to detect possible inconsistencies between data validation constraints. Moreover, we will integrate the code generator, which transforms the constraints at model level to Java annotations, in the DPF Editor [3], a diagrammatic (meta)modelling tool based on DPF and Eclipse Modeling Framework (EMF) [19].

Acknowledgement

The authors would like to thank Øyvind Bech and Dag Viggo Lokøen for the proof-of-concept implementation of the code generator, and Federico Mancini for the support with the SHIP Validator.

References

[1] Barr, M. and C. Wells, "Category Theory for Computing Science (2nd Edition)," Prentice Hall International Ltd., Hertfordshire, UK, 1995.

[2] Bech, Ø. and D. V. Lokøen, "DPF to SHIP Validator Proof-of-Concept Transformation Engine," http://dpf.hib.no/code/transformation/dpf_to_shipvalidator.py.

[3] Bergen University College and University of Bergen, "Diagram Predicate Framework (DPF) Web Site," http://dpf.hib.no/.

[4] Diskin, Z. and U. Wolter, *A Diagrammatic Logic for Object-Oriented Visual Modeling*, in: *Proceedings of ACCAT 2007: 2nd Workshop on Applied and Computational Category Theory*, Electronic Notes in Theoretical Computer Science **203/6** (2008), pp. 19–41.

[5] Ehrig, H., K. Ehrig, U. Prange and G. Taentzer, "Fundamentals of Algebraic Graph Transformation," Springer, 2006.

[6] Hayati, P., N. Jafari, S. M. Rezaei, S. Sarencheh and V. Potdar, *Modeling Input Validation in UML*, in: *Proceedings of ASWEC 2008: 19th Australian Software Engineering Conference* (2008), pp. 663–672.

[7] Hovland, D., F. Mancini and K. Mughal, *The SHIP Validator: An Annotation-Based Content-Validation Framework for Java Applications*, Technical Report 389, Department of Informatics, University of Bergen, Norway (2009).

[8] Jürjens, J., "Secure Systems Development with UML," Springer, 2005.

[9] Kühne, T., *Matters of (Meta-)Modeling*, Software and System Modeling **5** (2006), pp. 369–385.

[10] Mancini, F., D. Hovland and K. Mughal, *Investigating the Limitations of Java Annotations for Input Validation*, in: *Proceedings of ARES 2010: 4th International Workshop on Secure Software Engineering* (2010).

[11] McGraw, G., "Software Security: Building Security in," Addison-Wesley, 2006.

[12] Object Management Group, "Object Constraint Language Specification," (2010), http://www.omg.org/spec/OCL/2.2/.

[13] Object Management Group, "Unified Modeling Language Specification," (2010), http://www.omg.org/spec/UML/2.3/.

[14] OWASP, "Top Ten Project," http://www.owasp.org.

[15] Rossini, A., A. Rutle, Y. Lamo and U. Wolter, *A Formalisation of the Copy-Modify-Merge Approach to Version Control in MDE*, Journal of Logic and Algebraic Programming **79** (2010), pp. 636–658.

[16] Rutle, A., "Diagram Predicate Framework: A Formal Approach to MDE," Ph.D. thesis, Department of Informatics, University of Bergen, Norway (2010).

[17] Rutle, A., A. Rossini, Y. Lamo and U. Wolter, *A Diagrammatic Formalisation of MOF-Based Modelling Languages*, in: M. Oriol and B. Meyer, editors, *Proceedings of TOOLS 2009: 47th International Conference on Objects, Components, Models and Patterns*, Lecture Notes in Business Information Processing **33** (2009), pp. 37–56.

[18] Rutle, A., A. Rossini, Y. Lamo and U. Wolter, *A Formal Approach to the Specification and Transformation of Constraints in MDE*, Journal of Logic and Algebraic Programming (To appear).

[19] Steinberg, D., F. Budinsky, M. Paternostro and E. Merks, "EMF: Eclipse Modeling Framework 2.0 (2nd Edition)," Addison-Wesley Professional, 2008.

[20] Wolter, U. and Z. Diskin, *From Indexed to Fibred Semantics – The Generalized Sketch File*, Technical Report 361, Department of Informatics, University of Bergen, Norway (2007).

Towards rigorous analysis of Open Source Software

Luis S. Barbosa [1]

Departamento de Informática (HASLab)
Universidade do Minho
Braga, Portugal

Pedro R. Henriques [2]

Departamento de Informática (CCTC)
Universidade do Minho
Braga, Portugal

Alejandro Sanchez [3]

Departamento de Informática
Universidad Nacional de San Luis
San Luis, Argentina

Abstract

This paper discusses the (often hidden) potential of Open Source Software development to resort to, benefit from and cross-fertilize formal engineering methods, whose role is indisputable in the production of trustworthy software components. A strategy addressing the incorporation of formal verification methods in the Open Source Software lifecycle, in a somewhat less conventional way — that of assisting the re-engineering process of running code — is proposed.

Key words: Open Source Software, formal methods, program analysis.

1 Introduction

The impact of Open Source Software on the way software applications and software-based services are currently developed, distributed and deployed, is

[1] Email: `lsb@di.uminho.pt`
[2] Email: `pedrorangelhenriques@gmail.com`
[3] Email: `asanchez@unsl.edu.ar`

indisputable. Usually acknowledged key benefits include rapid code turnover, extensive testing, supported maintenance and low development costs. Linux distributions, Apache and MYSQL, serve as paradigmatic examples of its success and resilience.

Open Source Software is being increasingly adopted by industry, also for mission and safety-critical applications. In general, experience has shown that many open source software products are reliable and have achieved adequate functionality and scalability. For example, an extensive study carried on a few years ago showed that an active, mature open source initiative may have fewer defects than similar commercial projects [4]. Similarly, reference [Aea02] reports on a study of 100 open source applications concluding that structural code quality was higher than expected and comparable with commercially developed software.

This does not mean that Open Source Software is immune to the sort of correctness problems and vulnerabilities affecting software in general. Software development *de facto* standards are still pre-scienctific in their lack of sound mathematical foundations to provide an effective basis to predict and certify programs behaviour. Open source communities are no exception, even if failure is definitely not advertised:

> We tend not to hear very much about the failures. Only successful projects attract attention, and there are so many free software projects in total that even though only a small percentage succeed, the result is still a lot of visible projects. We also don't hear about the failures because failure is not an event. There is no single moment when a project ceases to be viable; people just sort of drift away and stop working on it. There is not even a clear definition of when a project is expired. Is it when it hasn't been actively worked on for six months? When its user base stops growing, without having exceeded the developer base? What if the developers of one project abandon it because they realized they were duplicating the work of another–and what if they join that other project, then expand it to include much of their earlier effort? Did the first project end, or just change homes? Because of such complexities, it's impossible to put a precise number on the failure rate. But anecdotal evidence from over a decade in open source, some casting around on SourceForge.net, and a little Googling all point to the same conclusion: the rate is extremely high, probably on the order of 90 to 95%.
>
> K. Fogel, in [Fog05]

[4] The study, *How open source and commercial software compare: A quantitative analysis of TCP/IP implementations in commercial software and in the Linux kernel*, 2003, is available from `www.reasoning.com/downloads/opensource.html`.

Certifying software with respect to precise specifications of their behaviour and/or given levels of performance and security, constitutes the overall agenda of the so-called *formal* methods. Qualifier *formal* stresses that such a certification is not a matter of opinion (i.e., a legal argument), but has a similar status to that of a mathematical proof, in the sense that precise mathematical techniques are used either to build and compose the software, or to guide a systematic verification procedure. Formal methods in Software Engineering is no longer an esoteric issue, but essential to obtaining the highest degrees of assurance required by trustworthy systems. And industry is becoming more and more aware of this fact. On the other hand, the maturity of current tools to support formal development, analysis and verification is now much more adequate for industrial use than it has been in the past, when it was extremely hard for non-specialists to use such methods.

Open Source Software, however, by the very nature of its open and unconventional development model, in which coding and debugging efforts are shared among a distributed, heterogeneous community, with decentralized control mechanisms, makes software quality assessment, let alone full certification, particularly hard to achieve. On the other hand code is exposed, freely available, often complemented with heavy volumes of informal documentation (in the form of source code comments, wiki notes, forum threads, ...), offering an enormous potential for verification and analysis.

The certification problem for Open Source Software raises specific challenges and opportunities, both from the technical/methodological and the managerial points of view (see, *e.g.*, [DAI09] for an extensive review on the security dimension). Not by chance the discussion on how formal development methods can be brought to Open Source Software practice, has been the focus of a series of workshops promoted by the United Nations University, with the acronym OpenCert since 2007 (see `opencert.iist.unu.edu/` and [BCS10] for the latest proceedings).

This paper aims at contributing to this debate: to what extent, and in which ways, may research in formal methods and accompanying tools become meaningful and usable for open source development and certification? There is certainly no single answer. In the sequel we argue for a lightweight , 'backward' approach: rather than insisting on the effective introduction of formal methods in the development process, we suggest the dissemination of rigorous program understanding and analysis techniques suitably integrated in an open infrastructure where open source code can be registered and analyzed in a number of different ways.

Paper outline.

Section 2 discusses the dichotomy *Formal Methods* vs *Open Source Software*, pointing out which characteristics of Open Source Software one may

build on to introduce such methods without disturbing its peculiar, but successful development cycle. Then, section 3 describes our proposal of a certification infrastructure for Open Source Software. Sections 4 and 5 make such a proposal more concrete through a brief summary of two tools developed within the authors' research team to be part of the envisaged infrastructure. Finally, section 6 concludes and gives some pointers for future research.

2 Quality, Formal Methods and Open Source Software

A standard approach to reduce risks in using an artifact is to establish an independent certification process. However, no certification standards exists that could be used to assess or classify the quality of Open Source Software. Such, a standard would certainly have to include the maturity of the development process, but open source reality has long been ignored in academia and its study largely reduced to a social phenomenon. Empirical studies exist (see e.g. [Aea02,MFH02,MHP05]) but are still insufficient. Moreover, they tend to focus on the *context* of software production, i.e, on the factors that determine the development conditions and, thus, are expected to influence its final quality, rather than on the *product* itself. Technical, or *product oriented* quality, on the other hand, deals with factors directly influencing maintainability, reliability and portability, which are extremely relevant for industry integrating Open Source Software in their own solutions.

That is precisely the focus of formal or rigorous engineering methods. Despite the complex, decentralized nature of Open Source Software development process, it is possible to identify a number of its characteristics which, in our opinion, favor a fruitful interaction with such methods. Our claim is that any proposal for incorporating this sort of methods in Open Source Software development should build on the following:

- *High code modularity*, leading to and stimulating separate development, without a need to change or understand the core system, or interfere with each developer's progress. This not only reduces the risk of propagating defects, but is also the key for a successful introduction of tight control cycles based on rigorous methods, which are, in their majority, compositional. A popular study of the Linux kernel development [LC03] concluded that modularity let multiple developers work on the same solution, often in competition, increasing the probability of timely, high-quality solutions.
- *Rapid release cycles* which keep code reviewers and developers interested and motivated, quickly resulting in systematic and high quality extensions. This also makes possible similarly rapid verification or analysis cycles and the suitable feedback of their results into the development process.
- *Independent and active* code review, typically lead by people outside the project team. A publicly visible bug and issue tracking tool is used by nearly all big open source projects. Users post bugs and enhancement requests.

Each such post becomes, in effect, a tiny public mailing list focused solely on that issue. The introduction of formal analysis methods, simply adds to this already present critic ability.

- *Large, sustainable communities* to develop, test and debug code effectively. An investigation of open source projects evolution cited in [Abe07] found that a large base of voluntary contributing members was one of the most important success factors. Rigorous analysis methods, timely applied, help the coevolution of a product and its community, and reinforce positive feedback as well as the reward- and-recognition culture which facilitates internal cohesion.
- *Traceable pedigree.* Unlike closed software, where the identity of the real supplier is often hidden, the lineage of an open source product can easily be traced: it is easier to determine exactly who did what, and who has modification rights. This provides a sound basis on top of which composition mismatches and errors detected during analysis can be traced to their origins and easily corrected.
- *Tool-mediated communication* is extensively used. Actually, a ubiquitous trait of open source development is that tool mediation is the norm. This enables leaders to shift the burden of policy enforcement from people to tools, which support authentication, regulation of commit privileges, audit and notification. Again, the introduction of analysis and vertification, corrective steps in the development cycle can easily benefit from this tool-mediated communication.
- Last but not the least, and contrary to a widespread belief, Open Source Software development, being distributed and multi-centered is far from being anarchic. Typically, composition, configuration, and information flow in and out of the project's server is somehow (but effectively) controlled. Project initiators and main contributors often exercise tight control over the engineering practices, not by limiting the developers behavior in their own personal space, but by limiting the kinds of transactions developers can make upon the persistent project state on the server. This may provide the needed infrastructure for enforcing quality checks based on formal technics.

If formal methods offer a valuable contribution to assess and promote Open Source Software reliability, an almost reverse claim can also be made: the relevance of Open Source Software for the formal methods community cannot be underestimated. Actually, open source licenses, allowing others to study, use, improve, and release new versions, are an essential ingredient to promote and disseminate tools supporting formal development methods — a first-class vehicle for making continued research possible. Sadly, many such tools have completely disappeared because they were not released under open source licenses. The absence of an open source license for ESC/JAVA, for example, created a difficult situation for many people and companies depending on this popular tool, once COMPAQ/HP decided to abandon its maintenance.

A key benefit of using Open Source Software is that the code can be compiled freely. As technology advances fast, the ability to recompile the source code becomes more and more important. Although a general remark, this applies indeed to support tools for formal methods: Open Source Software remains the key. Similar remarks apply to their long term survival.

Another argument (made mostly in the context of mathematical proofs) stems from the scientific validity and acceptance of computer generated, or computer assisted proofs. For such proofs to be included as standard material, the software system used to arrive at the result must also be available to researchers, e.g., to independently check the proof for its correctness.

3 The reverse perspective

The considerations above lead the authors' current research towards addressing the incorporation of formal verification methods in the Open Source Software lifecycle in a very peculiar perspective: that of assisting the re-engineering process of running code.

Typically, formal methods are designed to be applied during the development phase, preferably from very early design stages. Difficulties and strategies for proceeding this way are discussed elsewhere [BCPS10]. Our starting point here is the fact that, faced with a high risk dependence on open source components, often to be embedded on their own software, industry is more and more prepared to spend resources to increase confidence in (the level of understanding of) their code. From this point of view, the same principles and calculi used for (formal) program development can be applied in the reverse direction, from concrete to abstract models, for understanding and documenting implementations. More precisely, we seek

- Developing *program understanding and analysis techniques* and combine them for quality assessment of open source code. As Open Source Software offers full access to source code this enables the effective application of approaches and tools entirely targeting code analysis. The nature of Open Source Software entails the need for integration of techniques spanning the "micro" to the "macro" levels (e.g., from slicing to architectural recovery) and with different levels of formality (e.g. from statistical analysis based on code metrics to the identification and formal verification of hidden invariants). Sections 4 and 5 details two such techniques developed in this context.
- Catering for their smooth integration into the peculiar development process of Open Source Software without disturbing its collaborative, distributed and heterogeneous character. This amounts to establish feedback loops in open source development, making publicly available a number of interrelated analysis tools, to enhance the overall software reliability.

Our proposal to achieve the latter objective is through an *online, open*

infrastructure in which independently developed analysis tools (with different levels of sophistication) are inserted to monitor, assess and, at a later stage, certify open source products. Ideally, such an infrastructure would allow for the registration of open source projects, their source code visualization and analysis at different levels, as well as the rendering of analysis results in suitable, flexible formats to both Open Source Software developers and users. It will not only provide support for open source software analysis, but also make the evolution of open source software projects clearly visible to the open source software community. In the long run, one may expect that feedback loops will have an effective impact on the overall quality of Open Source Software products, with none or minimal intrusion on their life-cycle.

Such a certification infrastructure, currently under development at Universidade do Minho, adopts an open architecture, in the sense that new analysis or visualization components can be easily added relying on an open, general format for data/code representation. In the very spirit of Open Source Software, this will allow separate use and distribution of the framework, such that third parties can use it and plug-in their own analysis and visualization components.

The following two sections outline two of such plug-in tools — Gamma and CoordPat already developed in this project for source code analysis.

4 Gamma: A plug-in for assertion-based slicing

The Gamma toolkit [dCHP10,BdCHP10] is an *assertion-based slicer* equipped with a *verification* component, to generate verification conditions, and a program visualization functionality. Its purpose is to extract slices from code through a number of different families of slicing algorithms (precondition, postcondition, and contract-based).

This plug-in is intended to operate over source code suitably annotated with *contracts* in the sense of the *design by contract* paradigm — an approach that advocates specifying the behavior of program routines through the use of annotations, and checking them individually, either statically or dynamically, to obtain globally correct programs.

Of course this may sound strange with respect to its main application target – Open Source Software. Actually, even if annotated Open Source code may soon emerge as part of a code documentation effort whose need the community is increasingly aware of, such is clearly not dominant today. However, recent advances in automatic inference of annotations provide other tools which act as pre-processors for Gamma. For example, a component of Frama-C [CCPS09], a popular open source framework based on static analysis, automatically infers the preconditions for a given procedure.

Gamma, whose implementation includes a new, very efficient slicing algorithm [BdCHP10], is not only useful for code analysis, but also to assist automatic code adaptation. This may involve elimination of code redundancy, but

can also go much further. For example, suppose there is a library containing a procedure that implements a traversal of some data structure, and collects a substantial amount of information in that traversal. Now suppose this library is to be reused dropping the requirement that all the information collected in the traversal should be used. In this case the procedure respects a weaker specification, and thus it makes sense to produce a specialized, corresponding version of the library. This is crucial for software reuse, and open source development heavily depends on reuse.

Specializations of assertion-based slicing, for example to focus exclusively on post-condition annotations, may be used to study when a property is valid in a specific section of a program (for example inside a critical region to be executed on a specific thread) and false elsewhere. Therefore, it may be used to study the correct behaviour of code with respect to that section. Similarly, the property may correspond to some invariant on a data structure, which ought to be maintained. The toolkit also generates, in a step-by-step fashion, a set of verification conditions given as input to automatic SMT provers, which allows to establish the initial correctness of the code with respect to their contracts.

5 CoordPat: A plug-in for architectural analysis

If Gamma addresses code *micro* level, i.e., the level of procedures and statements, CoordPat is oriented toward code analysis *in the large*. Basically, it may be regarded as a tool for reverse architectural analysis, providing a systematic way to encode and identify coordination patterns in source code. It aims at uncovering, registering and classifying architectural decisions often left undocumented and hardwired in the application code. Moreover, through the systematic, tool-supported discovery of architectural decisions, it is expected to entail the reconstruction of the corresponding specifications.

Actually, current software systems rely more and more on non trivial coordination logic for combining autonomous services often running on different platforms. Open Source Software is no exception. As a rule, however, in typical, non trivial software systems, such a coordination layer is strongly weaved, at the source code level, with the application. Therefore, its precise identification becomes a major methodological (and technical) problem which cannot be overestimated and to which this tool aims at contributing. Not seldom open source applications emerge by composition of multi-source, heterogeneous and previously unrelated pieces of code, which makes architectural recovery processes both useful and challenging. Moreover, there is a need, particularly critical in open source contexts, to control architectural drifts, i.e., the accumulation of architectural inconsistencies resulting from successive code modifications.

CoordPat implements a rigorous methodology [RB10,RB08] to extract, from source code, its *coordination layer*, i.e. the architectural layer which captures system's behaviour with respect to its network of interactions. The qual-

ifier is borrowed from research on *coordination* models and languages [GC92], which emerged in the nineties to exploit the full potential of parallel systems, concurrency and cooperation of heterogeneous, loosely-coupled components.

The extraction methodology combines suitable slicing techniques to build a family of *dependence graphs* by pruning a *system dependence graph* [HRB88] first derived from source code. After the extraction stage, the tool exploits such graphs to identify and combine instances of *coordination patterns* and then reconstruct the original specification of the system's coordination layer. The word *pattern* is used here with the usual meaning: a way to describe and reuse standard solutions for recurrent problems. Thus, CoordPat maintains an incrementally-built repository of patterns used to guide the analysis process.

Coordination patterns are described in a formal, graph-based language for which a relational semantics was introduced in [ORHB10]. A pattern repository is integrated in the tool and dynamically populated by the users. The tool also provides features for (*i*) basic editing of coordination patterns, (*ii*) their syntactic and semantic validation, (*iii*) graph rendering for their visualisation (see e.g. Fig. 1) and (*iv*) pattern discovery in a dependence graph previously extracted.

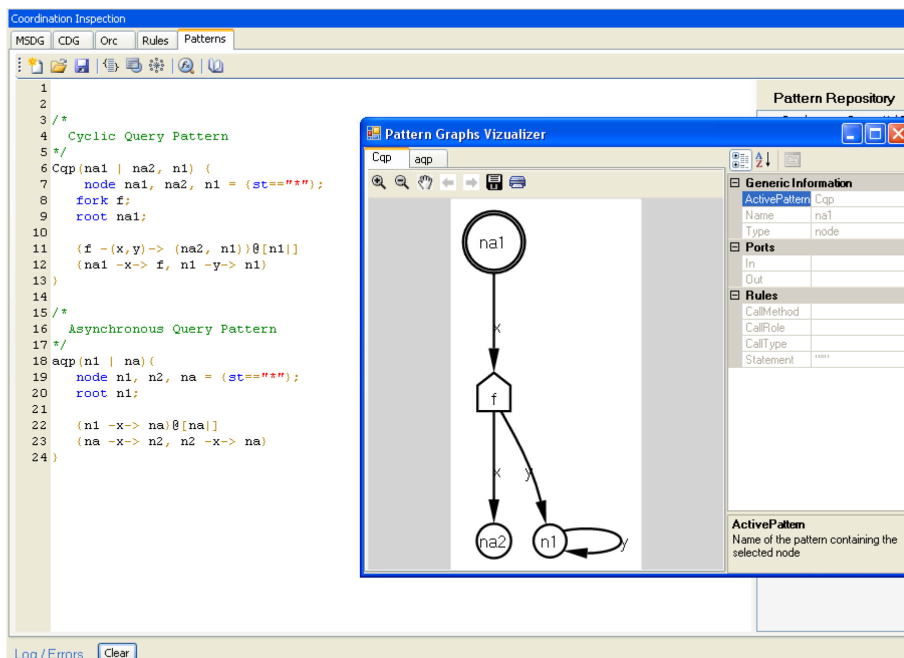


Fig. 1. The *Cyclic Query Pattern* and its graphical representation

6 Conclusions and future work

Open Source Software is software whose license gives users the freedom to run it for any purpose, to study and modify, and to redistribute copies of either

the original or the modified program, without having to pay royalties to previous developers. Companies are becoming aware that integrating Open Source Software into commercial products (made available by liberal open source licenses) reduce development costs while offering high-quality, extensively tested components. Furthermore, governments are getting worried with the growing dependence on proprietary formats and software in their administration, and regard Open Source Software as a warranty of technological independence. This turns out to a strategic advantage, mainly in the developing world.

Strengthening the role of Open Source Software in the global IT sector is, therefore, a strategic aim and, so we believe, a condition for increased, democratic citizenship in our information-led societies. However Open Source Software quality can be very hard to measure and to compare [Spi11]. This could be substantially improved if there were appropriate standards, supported by analysis tools, for certifying such software. Developing such tools, making them widely available for the open source community, and, in the long term, contributing to the creation of an international certification authority for open source software, is the path to which we would like to contribute.

This paper summarizes current research at Minho University, Portugal, on a possible strategy leading to the establishment of an independent certification process, with potential for a long-term impact on the integration of trustworthy, open source components, in large, complex systems. In short, we made a case for formal methods use in the 'reverse' direction, i.e., to guide code based analysis of open source components with potential impact in their improvement and reuse.

As related work, Alitheia Core [GS09] must be cited. This is an extensible platform designed specifically for performing large-scale software quality evaluation through the extraction and combination of a number of metrics on open source projects, resorting both to white-box test and code analysis. A central issue in this project is scalability to huge volumes of data, which entails the need for complex mirroring schemes and multicore execution. Several other projects exist proposing solutions for Open Source Software testing and evaluation. For example, Qsos (`www.qsos.org/`) is a methodology to assess, select and compare, open source components in an objective, traceable way. Project Osstmm `www.isecom.org/osstmm/` developed a peer- reviewed methodology for performing security tests on Open Source Software. What distinguishes our own proposal is the explicit aim of incorporating formal methods in addressing Open Source Software certification.

But, of course, a lot of questions remain to be answered. To mention just one we have not addressed so far: *security*. Security requires a specific analysis, since open source development does not usually follow the best security practices. As [DAI09] notices, in a recent book on security certification of Open Source Software, *the lower number of security events involving Open Source Software may be ascribed to its smaller market share rather than to its robustness*. Tools for security analysis must definitively be pluged-in into the

certification infrastructure suggested above.

Another main issue, requiring further experimental research, is the study of the potential impact of such an infrastructure in the concrete open source communities to which it is directed. In any case such an integration, or synergy, needs to be *non disturbing* of the community principles and (best) practices.

Acknowledgements. This research was partially supported by the Cross project, under contract PTDC/EIA-CCO/108995/2008 with Fct, the Portuguese Foundation for Science and Technology. Several ideas discussed in this paper and pursued in the Cross project benefited from discussions with Antonio Cerone, Bernhard Aichernig and Siraj Shaikh on possible roles for formal methods in Open Source Software certification, namely in the context of the OpenCert workshops. Collaboration with Daniela da Cruz, Jorge Sousa Pinto and José Barros in the development of the Gamma toolkit, as well as with Nuno Oliveira and Nuno Rodrigues in the design of CoordPat, is greatly acknowledged.

References

[Abe07] M. Aberdour. Achieving quality in open source software. *IEEE Software*, pages 58–64, 2007.

[Aea02] L. Angelis and et al. Code quality analysis in open source software development. *Information Systems Journ.*, pages 43–60, 2002.

[BCPS10] L. S. Barbosa, A. Cerone, A. K. Petrenko, and S. A. Shaikh. Certification of open-source software: A role for formal methods? *International Journal of Computer Systems Science and Engineering*, (4):273–281, 2010.

[BCS10] L. S. Barbosa, A. Cerone, and S. Shaikh, editors. *Foundations and Techniques for Open Source Software Certification, Proc. OpenCert 2010, Pisa, September, 2009.* Electronic Communications of the EASST, volume 33, 2010.

[BdCHP10] J. Bernardo Barros, Daniela da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. Assertion-based slicing and slice graphs. In *SEFM'10 — 8th IEEE International Conference on Software Engineering and Formal Methods*, pages 93–102. IEEE Computer Society, Conference Publishing Services (CPS), Sept 2010.

[CCPS09] L. Correnson, P. Cuoq, A. Puccetti, and J. Signoles. Frama-C User Manual. `http://frama-c.cea.fr/download/user-manual-Beryllium-20090902.pdf.`, November 2009.

[DAI09] Ernesto Damiani, Claudio Agostino Ardagna, and Nabil El Ioini. *Open Source Systems Security Certification*. Springer, 2009.

[dCHP10] Daniela da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. Gamaslicer: an online laboratory for program verification and analysis. In *Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications - LDTA '10*, pages 3:1–3:8. ACM, 2010.

[Fog05] Karl Fogel. *Producing open source software - how to run a successful free software project*. O'Reilly, 2005.

[GC92] D. Gelernter and N. Carrier. Coordination languages and their significance. *Communication of the ACM*, 2(35):97–107, February 1992.

[GS09] G. Gousios and D. Spinellis. Alitheia core: An extensible software quality monitoring platform. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 579–582. IEEE, 2009.

[HRB88] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conf. on Programming Usage, Design and Implementation*, pages 35–46. ACM Press, 1988.

[LC03] Gwendolyn K. Lee and Robert E. Cole. From a firm-based to a community-based model of knowledge creation: The case of the linux kernel development. *Organization Science*, 14:633–649, November 2003.

[MFH02] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, July 2002.

[MHP05] Martin Michlmayr, Francis Hunt, and David Probert. Quality practices and problems in free software projects. In Marco Scotto and Giancarlo Succi, editors, *Proceedings of the First International Conference on Open Source Systems*, pages 24–28, Genova, Italy, 2005.

[ORHB10] Nuno Oliveira, Nuno Rodrigues, Pedro Rangel Henriques, and Lus Soares Barbosa. A pattern language for architectural analysis. In *SBLP 2010 14th Brazilian Symposium in Programming Languages*, volume 2, pages 167–180. SBC — Brazilian Computer Society (ISSN: 2175-5922), 2010.

[RB08] N. F. Rodrigues and L. S. Barbosa. Coordinspector: a tool for extracting coordination data from legacy code. In *Proc. IEEE 8th Inter. Working Conference on Source Code Analysis and Manipulation (SCAM'08), Beijing, 2008*. IEEE Computer Society, 2008.

[RB10] N. F. Rodrigues and L. S. Barbosa. Slicing for architectural analysis. *Sci. Comput. Program.*, 75(10):828–847, 2010.

[Spi11] D. Spinellis. Choosing and using open source components. *IEEE Software*, 28(3):96, 2011.

Stochastic Reo: a Case Study

Y.-J. Moon[1], F. Arbab, A. Silva, C. Verhoef A. Stam

CWI, Amsterdam & Almende BV, Rotterdam, The Netherlands

Abstract

QoS analysis of coordinated distributed autonomous services is currently of interest in the area of service-oriented computing and calls for new technologies and supporting tools. In previous work, the first three authors have proposed a compositional automata model to provide semantics for stochastic Reo, a channel based coordination language that supports the specification of QoS values (such as request arrivals or processing rates). Furthermore, translations from this automata model into stochastic models, such as continuous-time Markov chains (CTMCs) and interactive Markov chains (IMCs) have also been presented.
Based on those results, we describe in this paper a case study of QoS analysis. We analyze a certain instance of the ASK system, an industrial software system for connecting people offering professional services to clients requiring those services. We develop a model of the ASK system using stochastic Reo. The distributions used in this model were obtained by applying statistical analysis techniques on the raw values that we obtained from the real logs of an actual running ASK system. These distributions are used for the derived CTMC model for the ASK system to analyze and to improve the performance of the system, under the assumption that the distributions are exponentially distributed. In practice, this is not always the case. Thus, we also carry out a simulation-based analysis by a Reo simulator that can deal with non-exponential distributions. Compared to the analysis on the derived CTMC model, the simulation is approximation-based analysis, but it reveals valuable insight in the behavior of the system. The outcome of both analyses helps both the developers and the installations of the ASK system to improve the performance of the system.

Keywords: Stochastic Reo, QoS analysis, case study, Extensible Coordination Tools

1 Introduction

The increasing complexity of software has motivated much research in order to develop techniques for the modular development of systems. Component-based software engineering and service-oriented computing aim at the development of reusable software components and/or services as building blocks that can be composed to build different applications. Research on software composition plays a key role in this quest, as it offers flexible ways of plugging components together. Connector based-languages, where channels or connectors are used to compose components and services into a system play a prominent role in the world of software composition. One of such languages is

[1] Email: yjm@cwi.nl

Reo [2,3], which offers a model of component and service coordination, wherein complex connectors are constructed by composing various types of primitive connectors called channels.

QoS analysis of composed software (intensive) systems has become popular in the last few years, with the goal of evaluating and improving performance and resource allocation in service-oriented applications.

Stochastic Reo [13] is an extension of Reo which allows for the specification of stochastic values for the channels (e.g., arrival and processing rates). A compositional automata model of Stochastic Reo was proposed in [13] and translations from this automata model to stochastic models such as CTMCs and IMCs were presented.

In this paper we show how the theory developed in previous papers, implemented as tools, can be used to model a part of a real industrial system, perform QoS analysis, and help the developers get an insight into the system behavior, which enables to improve the performance of the system. We model and analyze the ASK system, a software system developed by the Dutch company Almende, which provides efficient matching between service providers and clients. An example of the application of the ASK system consists of a service-based system running in a call center that matches calling clients with the appropriate representatives that can provide them with the specialized customer service that they need.

One challenge that arises when installing particular instances of the ASK system is how to allocate resources, which are typically scarce or expensive. For instance, in the particular example above, the call center wants to have an optimal distribution of its operators' schedules in order to reduce waiting time for the customers without increasing enormously its personnel costs. A stochastic model of the ASK system can be used to perform analysis and provide advice to solve such problems.

The main contributions of this paper are the following:

(i) a stochastic Reo model of the ASK system [2]. The distributions in this model were obtained by statistical analysis of real values filtered out of the logs of an actual running ASK system.

(ii) analysis of several interesting properties using the probabilistic model checker PRISM [12,15] which allowed to produce suggestions for the performance improvement of the ASK system. This analysis is done on a CTMC obtained from the Reo model.

(iii) analysis of the system using a simulator which enables the study of properties involving non-exponential distributions (CTMCs can deal only with exponential distributions).

[2] Details available at http://reo.project.cwi.nl/cgi-bin/trac.cgi/reo/wiki/CaseStudies/SimulatoronASK/Reception.

2 Preliminaries

Overview of Reo

Reo is a channel-based coordination model wherein so-called *connectors* are used to coordinate (i.e., control the communication among) components or services *exogenously* (from outside of those components and services). In Reo, complex connectors are compositionally built out of primitive channels. Channels are atomic connectors with exactly two ends, which can be either *source* or *sink* ends. Source ends accept data into, and sink ends dispense data out of their respective channels. Reo allows channels to be undirected, i.e., to have respectively two source or two sink ends.

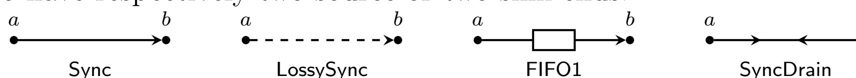


Fig. 1. Some basic Reo channels

Fig. 1 shows the graphical representations of some basic channel types. The Sync channel is a directed, unbuffered channel that synchronously reads data items from its source end and writes them to its sink end. The LossySync channel behaves similarly, except that it does not block if the party at the sink end is not ready to receive data. Instead, it just loses the data item. FIFO1 is an asynchronous channel with a buffer of size one. The SyncDrain channel differs from the other channels in that it has two source ends (and no sink end). If there is data available at both ends, this channel consumes (and loses) both data items synchronously.

Channels can be joined together using nodes. A node can have one of three types: *source*, *sink* or *mixed* node, depending on whether all ends that coincide on the node are source ends, sink ends or a combination of both. Source and sink nodes, collectively called *boundary nodes*, form the boundary of a connector, allowing interaction with its environment. Source nodes act as synchronous replicators, and sink nodes as mergers. A mixed node combines both behaviors by atomically consuming a data item from one sink end and replicating it to all of its source ends.

An example connector is depicted in Fig. 2. It reads a data item from a, buffers it in a FIFO1 and writes it to c. The connector loses data items from a if and only if the FIFO1 buffer is already full. This construct, therefore, behaves as a connector called (overflow) LossyFIFO1.

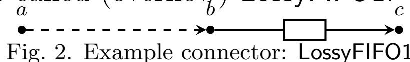


Fig. 2. Example connector: LossyFIFO1

Stochastic Reo

Stochastic Reo is an extension of Reo where channel ends and channels are annotated with stochastic values for *data arrival rates* at channel ends and *processing delay rates* on channels. Such rates are, e.g., non-negative real values that describe how the probability that an event occurs varies with time. Fig. 3 shows the stochastic versions of the primitive Reo channels in Fig. 1.

Here and throughout, for simplicity, we omit the node names, since they can be inferred from the names of their respective arrival rates: for instance, γa is the arrival rate of node a.

Fig. 3. Stochastic Reo channels corresponding to the channels in Fig. 1

A processing delay rate represents how long it takes for a channel to perform a certain activity, such as data-flow. For instance, a LossySync has two associated rates γab and γaL for, respectively, successful data-flow from node a to node b, and losing the data item from node a. In a FIFO1 γaF represents the delay for data-flow from its source a into the buffer, and γFb for sending the data from the buffer to the sink b.

Arrival rates describe the time between consecutive arrivals of I/O requests at the source and sink nodes of Reo connectors. For instance, γa and γb in Fig. 3 are the associated arrival rates of write/take requests at the nodes a and b.

Since arrival rates on nodes model their interaction with the environment only, mixed nodes have no associated arrival rates. This is justified by the fact that a mixed node delivers data items instantaneously to the source end(s) of its connected channel(s). Hence, when joining a source with a sink node into a mixed node, their arrival rates are discarded. A more precise description of Stochastic Reo appears in [4,13]. A stochastic version of the LossyFIFO1 is depicted in Fig. 4, including its arrival and processing delay rates.

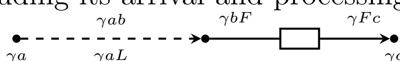


Fig. 4. Stochastic LossyFIFO1

3 ASK system

The “Access Society’s Knowledge” (ASK) system [16] is an industrial software developed by the Dutch company Almende [1], and marketed by their daughter company ASK Community Systems [5]. The ASK system is a communication software product that acts as a mediator between service consumers and service providers, for instance, connecting rescue institutions (e.g., fire departments) and professional volunteers. The connection established by the ASK system is provided by mechanisms for matching users requiring information or services with potential suppliers. For this purpose, the matching mechanisms use the profiles and availability offered by people who provide or require services.

The main goal of the ASK system is to do the matching in an efficient way. To achieve that, the system collects feedback on the quality of services after the connection. Such feedback is used to decide better connections for the subsequent requests of the same type. In addition, the system uses self-learning and self-organizing mechanisms by continuously updating to users’ preferences and available resources. Moreover, the ASK system enables users to inform

others about their status, their availability, and how they can be contacted best. This information is used to select the right people for a communication session as well as the feedback.

To offer efficient connections, the ASK system considers:

- human knowledge and skills of service providers
- time schedules of the provision of services
- communication media such as telephones, SMS, and emails

When people request a certain service from specialists or service providers, the ASK system attempts to select the best possible service provider. This selection is based on the rating of the knowledge and the skills of service providers who are available at that moment. This rating, in turn, is based on the feedback on the quality of services offered by the service providers. The occurrences of events can follow either regular schedules or ad-hoc schedules. The ASK system deals with both of these situations while satisfying the constraints and the purposes of users' requests.

The ASK system generally considers the telephone as a primary communication medium, but other means of communication, such as email or SMS, are also supported. These types of media must be considered according to the reachability and the preferences of the users. For example, people can have more than one email address and telephone number, with different associated usage constraints and user preferences. Such information must be indicated in the system to allow for efficient connections.

The ASK system acts as an agent that connects service providers and service consumers in an *efficient* way, handling multitudes of such connections simultaneously at any given time. The ASK system has a hierarchical modular architecture, i.e., it consists of a number of high-level components, which in turn consist of lower-level components, etc., running as threads. In order to handle massive numbers of connections concurrently, the components need to utilize multiple threads that provide the same functionalities. In this setting, allocation of system resources, e.g. the number of threads, to various components plays a critical role in the performance and responsiveness of an installed system in its actual deployment environment (e.g., properties of servers, available telephone lines, call traffic, available human operators, etc.), but determining the proper resource allocations to provide a good performance is far from trivial. Deriving and analyzing a stochastic model for an installed ASK system provides valuable input and insight for improving its performance. Among other possibilities, such a model allows system architects and installation operators to play what-if games by changing various resource and demand parameters and discover how a deployed system would perform under such scenarios, in order to adjust and fine-tune the system for cost-effective, optimal performance.

Various methods for performance evaluation have been suggested. Rigorous methods require mathematical models of a system involving variables that represent the parameters relevant to its behavior. Stochastic variables describe random system behavior, leading to more realistic models of behavior than their deterministic counterparts. CTMCs are frequently used to model such systems and their features and efficient closed-form and numerical techniques [18] exists for their analysis. Traditionally, such models are constructed by human experts whose experience and insight constitute the only link between the actual system and the resulting models.

Ideally, mathematical models for the analysis of the behavior of a system should be derived from the same (hopefully, verified correct) models used for its design and construction. Such automation makes the derivation of these models less error-prone, and ensures that a derived analytical model corresponds to its respective implemented system. An expressive modeling formalism that simultaneously reflects structural, functional, and QoS properties of a modeled system constitutes a prerequisite for this automation. Reo serves as an example of such a formalism: (1) it provides structural model elements whose composition reflects the composition of their counterpart system components with architectural fidelity; (2) it allows formal verification of functional and behavioral properties of a modeled system; (3) it supports derivation of executable code form its models; and (4) it supports derivation of mathematical models for the analysis of the QoS properties of systems.

A Reo model of the ASK System was developed as a case study [8] within the context of the EU project Credo [7] for verification of its functional properties. In the work we report in this paper, we refined and augmented this Reo model with stochastic delays extracted from actual system logs to derive a Stochastic Reo model for the ASK System. Together with Almende, we use this model to analyze and study the QoS properties of the ASK system in various settings. For instance, using the approach in [13], we derive CTMC models from the Stochastic Reo model of the interesting parts of the ASK System, and feed them into CTMC analysis tools, which enables us to do model checking of the stochastic behavior of the system. We will show the analysis of several such properties using PRISM in Section 5. The following sections describe the architecture of the ASK system in some detail. The figures and the descriptions we use here are based on [17].

3.1 Overview of the ASK system

The top-level architecture of the ASK System is shown in Fig. 5. Every component in this architecture has its own internal architecture, with several levels of hierarchical nesting. At its top-level, the ASK system consists of three parts: a *web front-end*, a *database*, and a *contact engine*. The *web front-end*

deals with typical domain data, such as users, groups, phone numbers, mail address, and so on. The *database* stores typical domain data, together with the feedback from users and knowledge from past experience. The *contact engine* handles the communication between the system and the outside world (e.g., by responding to or initiating telephone calls, SMS, emails, etc.) and provides appropriate matching and scheduling functionalities.

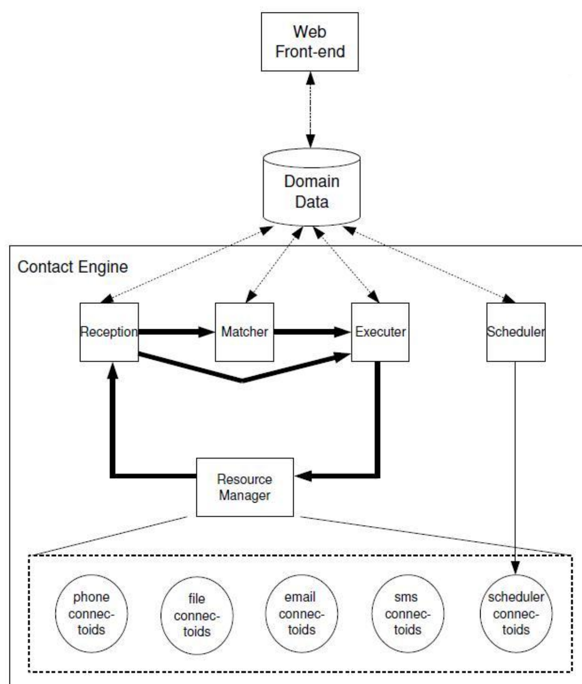


Fig. 5. Overview of ASK system

As mentioned above, the ASK system connects service providers and consumers for incoming requests. A connection is made when appropriate participants for a certain request are found. Until its proper connection is established, an incoming request loops through the system repeatedly as (sub-)tasks. This feature is called *Request loop* and it is represented by **thick arrows** in the contact engine in Fig. 5.

The contact engine consists of five components: *Reception*, *Matcher*, *Executer*, *ResourceManager*, and *Scheduler*. The *Reception* component determines which steps must be taken by the ASK system to fulfill a request. The *Matcher* component determines proper participants for fulfilling a request. The *Executer* component determines the best means of connection between the participants. The *Resource Manager* component either uses the Request loop for complicated requests or establishes direct connections between users for trivial requests. The *Scheduler* component, separated from the components within the request loop, schedules requests based on the time constraints of the requests in the database.

4 Modeling the ASK system

In this section, we consider the contact engine, which contains the Request loop, and focus specifically on the Reception component. The components in the contact engine have very similar architectures, thus, the analysis carried out here for the Reception component can be used for the other ones, as well.

4.1 The Reception component

The Reception component consists of multiple threads, the so-called *ReceptionMonks (RM)*, which handle incoming requests using two different functions:

- **HostessTask (HT)** which converts incoming requests into tasks that will be put into the task queue.outside of the Reception component;
- **HandleRequestTask (HRT)** which takes care of the communication flow, interacts with the database, and possibly generates new requests which are dealt with by the Matcher or the Executer component. For example, given an incoming request, HRT may ask questions from users by playing pre-recorded messages, obtain information such as menu item choices, account number, etc., punched in by the users, and store this information into the database. During this communication, new requests can be generated and sent to other components.

Each thread runs one of these two different functions/tasks exclusively. That is, if an RM thread runs the HT function, then it is forbidden to run the HRT function. This implies that the Reception component needs to have at least two threads, one for HT and the other for HRT. In general, HRT takes more time than HT, since it actually deals with incoming tasks. Thus, the Reception component needs more threads running HRT. For simplicity of modeling, we assume that every thread in the Reception component has only one function, e.g., either HT or HRT. Reflecting this simplification, Fig. 6 shows the Reception model drawn in the Eclipse Coordination Tools (ECT) [9]. This figure shows a Reception component with three RM threads, one with only HT and the other two with only HRT.

The RMHT and the indexed RMHRTs in Fig. 6 correspond to RM threads for a HT and HRT functions, respectively. Incoming requests are converted into tasks by the RMHT, and the converted tasks are stored in the task queue which is represented as a FIFO1 laid between RMHT and the indexed RMHRTs. The converted tasks are selected and handled by the RMHRTs. We model task selection as a non-deterministic choice at the *TQOut* node in Fig. 6, which will turn into a random process once we associate the distributions of the stochastic variables that describe the actual task mix of a running system, as extracted from its logs.

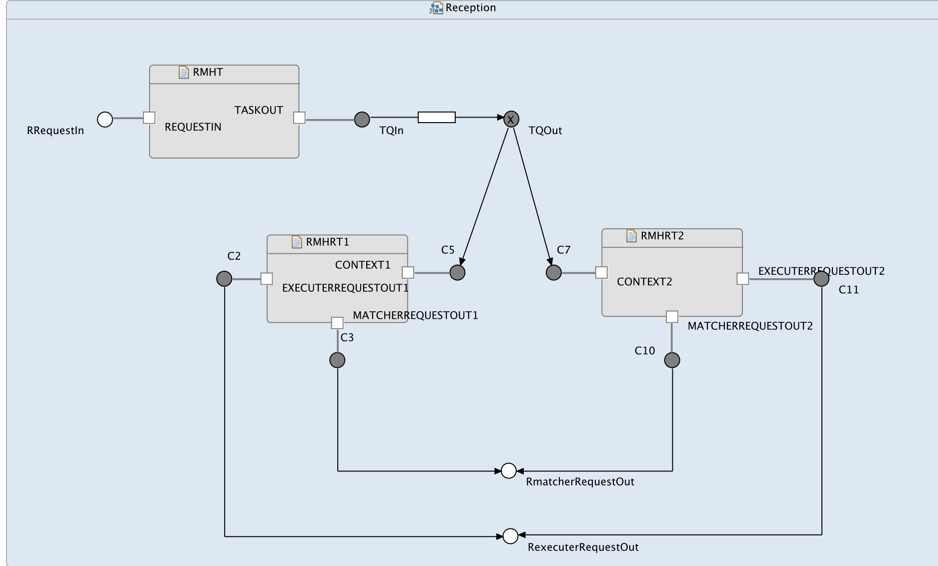


Fig. 6. Reception component in ECT

The graphical notation ⊗ used for *TQOut* in Fig. 6 is an abbreviation for an *exclusive router* [3] whose Reo circuit is depicted on the right. This circuit delivers an incoming data item at node a to either node b or node c, whichever one can accept it, and non-deterministically selects one when both can. The non-deterministic choice is actually conducted by the merger d. Thus, the rates for the random selection apply to the merger d.

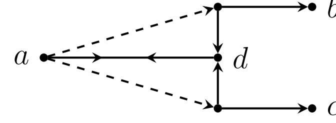


Fig. 6 serves as a basic template model for the Reception component. Depending on the specific properties of interest in each analysis, we adapt and vary this basic template slightly. For example, for the analysis of the properties of the task queue, we may substitute a LossyFIFO1 connector for the FIFO1 channel, as shown below.

4.2 Extracting distributions from logs

A stochastic model of the ASK system requires the distributions for all activities in the system. To obtain these distributions, we applied statistical data-analysis techniques on the raw values extracted from the real logs of a running ASK system. The logs contained the data of 100 incoming calls. Those calls simultaneously resulted in 369 requests sent to the Reception component. The trace holds exact timings of all actions performed related to each process.

We need to determine the rates for request arrivals (RRequestIn) and processing delay at the Reception component, reading request arrivals from the Matcher (RmatcherRequestOut) and the Executer (Rexecuter-RequestOut). For this purpose, after a cleanup of the raw data by removing outliers and erroneous data, we determined the appropriate distributions, using statistical

tests (like the chi-square goodness-of-fit test).

For the Reo model, it is not important which type of distributions we obtain. However, to perform analysis using PRISM, which takes a CTMC as input, only exponential distributions can be used. In the case of request arrival rates, we may indeed assume that the inter-arrival times of the requests are exponentially distributed. This is reasonable since incoming calls to the ASK system are independent from each other, and the inter-arrival times are memoryless. However, in the case of processing delay rates, we were not able to conclude that the rates are exponentially distributed. The statistical tests showed that we may assume that the processing times follow a log-normal distribution.

5 QoS analysis

In this section, we show how to analyze the ASK system using both the CTMC and Reo Simulator approach. As mentioned in the previous section, the arrival or service times for some activities are not exponentially distributed. This is one of the reasons to analyze the Reo model with the Reo Simulator (see Section 5.2). The simulator was also used when we could not obtain any proper distribution from the logs at all. In this case, we used bootstrapping [14] in the simulator with the original data as special inputs in the simulator for the rates.

5.1 Analysis on the derived CTMC

In this section, we analyze the ASK system to reveal some of its interesting properties in order to both evaluate and obtain clues for improving its performance. We carry out our analysis on the CTMC model derived from the Stochastic Reo model of the ASK system. We then feed the derived CTMC model as input to PRISM. In PRISM, properties of models are expressed using operations such as P, S, and R operators: the P operator is used to reason about the probability of the occurrence of a certain event; the S operator is used to reason about the steady-state behavior of a model; the R operator is used to analyze reward-based properties. In addition, labels are used to concisely express the formulas representing the properties of a model. Specifically, we use the following labels to express some properties later.

- `num_dataLoss` represents the number of task-loss in the task queue.
- `run` represents the running status of the RMHRT thread.

In general, resources are neither infinite nor free. Thus, one needs to balance cost-effective resource utilization against most efficient performance, i.e., obtaining the best performance taking into account the limited resource. In the Reception component in Fig. 6, the resources of interest include:

(i) the minimum capacity of the task queue

(ii) the utilization and/or the performance of the RMHRT threads that handle tasks

5.1.1 Task queue

As mentioned above, RMHT merely converts incoming requests into tasks, but it does not actually handles the requests. In general, the conversion into tasks does not take long, whereas handling a request may take considerable time. Thus, if the task queue has a small capacity, then RMHT frequently waits as it is blocked until task queue capacity becomes available. On the other hand, if the task queue has a large capacity, RMHT remains idle most of the time and some queue capacity goes to waste. Therefore, we want to determine a reasonable size for the task queue to make the ASK system efficient. We can check the probability of RMHT blocking by iteratively increasing the queue capacity by 1 in subsequent runs, but this laborious approach is too time consuming. Alternatively, we can assume that the task queue has infinite capacity and try to find how much of it is actually used. With this task queue, we obtained the long-run expected number of task-loss due to unavailable buffer capacity or the unbalanced performance of RMHT and RMHRT threads. For this purpose, we use the following PRISM property `R{"num_dataLoss"}=?[S]`. The result is shown in Fig. 5.1.2.

To mimic an infinite queue, we use a LossySync channel feeding into a queue with a fixed capacity. This construct always accepts arriving tasks, but arriving tasks are lost when the queue is full. We can approximate the minimum required queue capacity out of the expected number of losing tasks

```
Terminal
File  Edit  View  Search  Terminal  Help
Model checking: R{"num_dataLoss"}=? [ S ]

SCCs: 1, BSCCs: 1, non-BSCC states: 0
BSCC sizes: 1:14528

Computing steady state probabilities for BSCC 1

Building hybrid MTBDD matrix... [levels=14, nodes=57678] [2.6 MB]
Splitting into blocks... [levels=5, n=29, nnz=419, compact] [1.8 KB]
Adding explicit sparse matrices... [levels=9, num=418, compact] [557.0 KB]
Creating vector for diagonals... [dist=445, compact] [31.9 KB]
Allocating iteration vectors... [113.5 KB + 4.0 KB = 117.5 KB]
TOTAL: [3.3 MB]

Starting iterations...

Gauss-Seidel: 78 iterations in 2.44 seconds (average 0.001962, setup 2.29)

BSCC 1 Reward: 0.018521245454519365

All states are in a BSCC (so no reachability probabilities computed)

Time for model checking: 3.29 seconds.

Result (expected num_dataloss): 0.018521245454519365
```

Fig. 7. Expected number of task-loss in the task queue

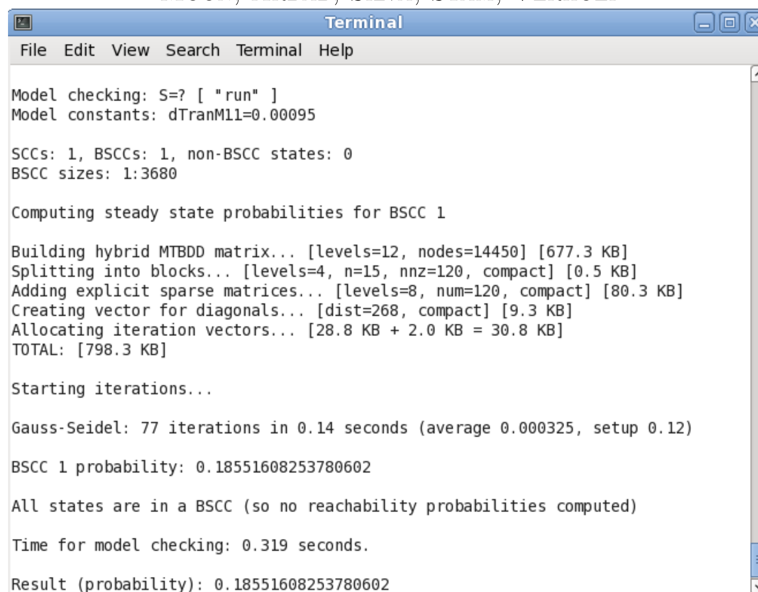


Fig. 8. steady-state probability of thread in use

by this construct. Replacing the FIFO1 queue in Fig. 6 by the LossyFIFO1 connector in Fig. 4 provides such a pseudo-infinite task queue for this analysis. According to this result, around 18.5 [3] requests are lost per second in front of the task queue. From this result, we can conclude that the minimum capacity of the task queue needs to be 20 to guarantee no task-loss.

5.1.2 Functions

The RMHRT threads are the primary task handling processes. Thus, the performance of the Reception component depends on the collective performance of its RMHRT threads. It is interesting to learn how many RMHRT threads are required to handle a task load, or what is the reasonable performance of RMHRT threads that can provide a satisfactory QoS. Instead of changing the number of RMHRT threads, here we fix their number at 2 and vary their performance by changing their processing delay rates. These two threads have the same architecture with the same performance, thus, the analysis on the utilization is carried out on the RMHRT1 thread, the result of which can be used for the other RMHRT thread. We first find the steady-state probability that the RMHRT1 thread is running, expressed as `S=?["run"]` in PRISM. The result, shown in Fig. 8, implies that the utilization of the RMHRT1 is 18%.

In a series of analysis experiments on this property, we varied the processing delay rates for the RMHRT1 thread. However, the gaps between the experiment results are not significant. For example, when we considered the activity of the RMHRT1 as an immediate activity by setting its rate as 2,147,483,687, the steady-state probability `S=?[“run”]` from this rate value

[3] The result 0.0185 was derived with millisecond as time unit.

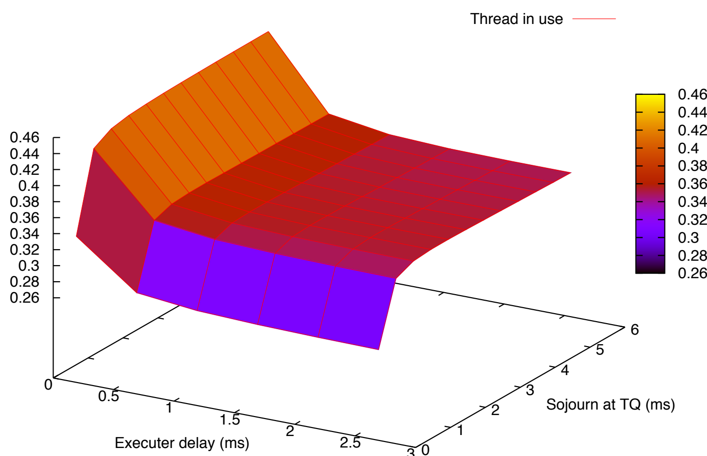


Fig. 9. Steady-state probability `S=?["run"]`

was 14%. Compared to the huge differences between these two values, their resulting probabilities are barely changed. This implies that improving the performance of the RMHRT1 thread does not influence the overall performance of the Reception component that much, which suggests the presence of some bottlenecks in this system.

In order to figure out the bottlenecks, we experimented with the model by varying the rates relevant to other activities in the system. Fig. 9 shows the probability results of these experiments. The label `Sojourn at TQ` presents the exit rate from the task queue. As this rate decreases, incoming requests stay longer in the task queue, and the RMHRT threads become more idle, i.e., the probability of the thread utilization decreases, since the request arrive at the thread less frequently. The graph in Fig. 9 shows this tendency when one projects this graph onto the (`Prob.`, `Soj.`) plane. This implies that increasing `Sojourn at TQ` value generates higher utilization of the thread.

The label `Executer delay` represents the frequency that the Executer component takes the output from the Reception component. As this rate decreases, the threads in the Reception component need to keep their results waiting longer and block incoming tasks. Thus, the thread becomes less idle, i.e., the utilization of the thread increases, but their throughput becomes low since the thread just waits without doing anything. This tendency is also observable in the graph in Fig. 9 when one projects this graph onto the (`Prob.`, `Exe.`) plane. To obtain meaningful utilization, we must increase `Executer delay`.

Based on the graph in Fig. 9, we now determine bottlenecks in this system. In general, a small change in a bottleneck causes significant differences for the overall performance. The graph in Fig. 9 shows an instance of this: variations in the rates in the interval [0.1, 0.6] for both `Executer delay` and `Sojourn at TQ` induce a big variation on the probability of utilization of the thread (represented in the vertical axis). Thus, these two rates can be assumed to

be bottlenecks, which limit the overall performance. In order to mitigate these bottlenecks, we need to increase both rates at least above 0.6. However, we cannot increase these rates enormously since their relevant resources are neither infinite nor free. As a criterion for this increase, we can consider the convergent disposition of this graph. Above the value 1.3 of the respective rates, the utilization of the thread converges. Thus, we can choose the third values of the respective rates for the best cost-effective utilization of the thread in this system.

5.2 Simulation-based analysis

The Stochastic Reo Simulator [10,19] supports performance evaluation of Reo models through simulation. It allows arbitrary distributions for describing stochastic properties of channels and components. The method used by this tool combines simulation techniques and specific stochastic automata models to conduct automated performance analysis of both steady-state and transient properties of the model. The Stochastic Reo Simulator tool uses the coloring semantics [6] of Reo to properly model context-dependent behavior, i.e. to express the availability of requests. The tool is developed as a plug-in within the Eclipse Coordination Tools (ECT) [9,11]. Through the GUI editor of the ECT, one can develop a model of a system as a Reo circuit in an intuitive way, annotate the circuit with rates, and then use the simulator to get insight into the behavior of the model.

The simulator provides information about (1) the average waiting times of I/O requests at boundary nodes, (2) buffer utilization, (3) end-to-end delays, and (4) channel utilization. Using this simulator on the Reception component, we used the distributions extracted in Section 4.2. Due to space limit, we do not show details of the use of the simulator here. We show a more detailed description on the ECT web-page[4]. As a few examples, the properties/facts we learned about the Reception component from the simulation include: the task queue is used 57% of the time; it takes, on the average, 6.5 milliseconds to handle a request; and the waiting time of I/O requests at the `RRequestIn` node in Fig. 6 is, on the average, 1.7 milliseconds.

6 Discussion

In this paper, we have presented a stochastic analysis of (a deployed installation of) the ASK system. We modeled the system using Stochastic Reo, from which we generated the CTMCs corresponding to some of the modules of the system. This enabled us to use the probabilistic model checker PRISM to verify some properties of interest, using the concrete data extracted from the logs

[4] http://reo.project.cwi.nl/cgi-bin/trac.cgi/reo/wiki/CaseStudies/SimulatoronASK

of the running ASK installation. The results of this verification allowed us to draw conclusions about resource allocation and how the system installation can be adapted in order to improve its performance. CTMC models have the limitation of supporting only exponential distributions. To overcome this limitation, we also used a simulator. Even though the result from the simulation is approximation-based analysis, we can gain insight into the aspects of the behavior of the system that involve non-exponential distributions.

We have focused our analysis in this paper only on the Reception component of the ASK system. However, the other components have very similar architectures and, thus, all the techniques used in this paper can be easily applied to them as well.

The distributions used in this case study were obtained by statistical analysis based on the real logs of an actual running ASK system. Our analysis revealed exponential distributions for the arrivals and I/O requests. However, rates for the processing/service times of some components were not exponentially distributed. This made it necessary to do simulation for additional analysis. We used the Reo simulator [10,19], an integrated ECT tool, which enables the use of arbitrary distributions and predefined probabilistic behaviors. Using this simulator we can study a model which, for instance, has exponentially distributed data arrivals and log-normal distributed processing rates in some components.

In this analysis, we found two bottlenecks that were caused by (1) the low availability of the Executer component and (2) the long sojourn time at the task queue. In what concerns (1), we observe that we are modeling the connections between the Reception and other components (Executer and Matcher) synchronously (that is, using Sync channels), and that the observation that the consumption rates of the other two components become bottlenecks is not surprising. We have experimented with replacing the Sync channels with FIFOs to decouple the components and remove these bottlenecks. In the process of these experiments, we identified another bottleneck internal to the Executer component itself. In what concerns (2), the bottleneck is caused by congestion between the task queue and the threads. Thus, we can widen the bandwidth of this connection to obtain better performance for the system.

In earlier initiatives to improve the performance of the ASK system, the focus has been primarily on improving the execution times of request handling tasks, through extensive profiling. The work presented in this paper confirms and explains the observations from small experiments with ASK components in isolation, carried out by Almende last year. As a consequence of this, Almende decided to put additional effort into the optimization of queue sizes and bandwidth between the task queue and the threads in each of the ASK components. First attempts in this direction yield promising results.

Acknowledgments.

The authors are thankful to Christian Krause and Oscar Kanters for their help in using the Reo simulator.

References

[1] Almende website. http://www.almende.com.

[2] F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[3] F. Arbab. Abstract Behavior Types: a foundation model for components and their composition. *Science of Computer Programming*, 55(1-3):3–52, 2005.

[4] F. Arbab, T. Chothia, R. van der Mei, S. Meng, Y.-J. Moon, and C. Verhoef. From Coordination to Stochastic Models of QoS. In *COORDINATION*, volume 5521 of *Lecture Notes in Computer Science*, pages 268–287. Springer, 2009.

[5] ASK community systems website. http://www.ask-cs.com.

[6] D. Clarke, D. Costa, and F. Arbab. Connector colouring I: Synchronisation and context dependency. *Science of Computer Programming*, 66(3):205–225, 2007.

[7] Credo project. http://projects.cwi.nl/credo/.

[8] F. S. de Boer, I. Grabe, M. M. Jaghoori, A. Stam, and W. Yi. Modeling and Analysis of Thread-Pools in an Industrial Communication Platform. In *Proc. ICFEM'09*, volume 5885 of *Lecture Notes in Computer Science*, pages 367–386. Springer, 2009.

[9] Eclipse Coordination Tools. http://reo.project.cwi.nl/.

[10] O. Kanters. QoS analysis by simulation in Reo. Master's thesis, Vrije Universiteit, Amsterdam, The Netherlands, 2010.

[11] C. Krause. *Reconfigurable Component Connectors*. PhD thesis, Universiteit Leiden, 2011.

[12] M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic Symbolic Model Checker. In *Computer Performance Evaluation/TOOLS*, pages 200–204, 2002.

[13] Y.-J. Moon, A. Silva, C. Krause, and F. Arbab. A Compositional Semantics for Stochastic Reo Connectors. In *FOCLASA*, volume 30 of *EPTCS*, pages 93–107, 2010.

[14] C. Z. Mooney and R. D. Duval. *Bootstrapping: a nonparametric approach to statistical inference*. Sage Publications, 1993.

[15] PRISM website. http://www.prismmodelchecker.org/.

[16] A. Stam. The ASK System and the Challenge of Distributed Knowledge Discovery. In *ISoLA*, volume 17 of *Communications in Computer and Information Science*, pages 663–668. Springer, 2008.

[17] A. Stam, S. Klüppelholz, T. Blechmann, and J. Klein. ReASK Final Models. Technical Report To appear, Almende, The Netherlands and Technical University of Dresden, Germany, 2009.

[18] W. J. Stewart. *Introduction to the numerical solution of Markov chains*. Princeton University Press, 1994.

[19] C. Verhoef, C. Krause, O. Kanters, and R. van der Mei. Simulation-based Performance Analysis of Channel-based Coordination Models. In *COORDINATION 2011*, volume 6721 of *Lecture Notes in Computer Science*, pages 187–201. Springer-Verlag, 2011.

A Calculus for a New Component Model in Highly Distributed Environments

Antoine Beugnard[1]

Computer Science Department
Telecom Bretagne
Brest, France

Ali Hassan[2]

Computer Science Department
Telecom Bretagne
Brest, France

Abstract

The current software systems and their corresponding deployment environments are highly complex and demanding. Multiple and unstable network technologies, resource-restricted devices, and mobility, are few examples of these complexities. In this paper we propose a new component model, called Cloud Component (CC), that copes with the challenges posed by mobile and pervasive environments. Traditional distributed applications are based on distribution transparency, where a middleware layer is expected to handle and hide all remote communication. Cloud component model is the result of a paradigm shift from distribution transparency to localization acknowledgment, where all details of the deployment environment including networks and communication, mobile devices, constrained devices, and sensors, are considered a first class concern. The cloud component model is presented informally and formally with a mathematical notation. The informal notation allows for faster comprehension of the general concepts. While the formal notation opens the door for a wide range of theoretical topics and provides a precise language to describe details. We also propose an assembly model to build large systems using CCs as building blocks. This assembly model is presented formally and fully implemented for the designer to be able to automatically check if his/her design conforms to the CC assembly model.

Keywords: Software Components, Formal Models, Component Assembly, Automatic Design Checker, Mobile and Pervasive Computing

[1] Email: Antoine.Beugnard@telecom-bretagne.eu
[2] Email: ali.hassan@telecom-bretagne.eu

1 Introduction

During the last years new distributed platforms have emerged, often qualified as highly distributed environments (HDE). HDEs still include powerful and robust machines but they are rather composed of resource-constrained and mobile devices such as laptops, personal digital assistants (or PDAs), smartphones, GPS devices, sensors, etc [7]. Moreover, these devices communicate using a variety of dependable and undependable fixed and wireless networks.

This fundamental change in the deployment environment was not accompanied by a theoretical software model that provides deep understanding and systematic solutions to build compatible software systems [1].

As Malek *et al.* [10] have noticed "transparency (i.e. hiding distribution, location, and interaction of distributed objects) is considered a fundamental concept of engineering distributed software systems, as it allows for the management of complexity associated with the development of such systems". This is usually achieved through the utilization of a middleware layer that has as a main function (among others) to make remote calls appear as local calls. That is correct for stable distributed systems, however, this same concept, distribution transparency, has been shown to suffer from major shortcomings when applied extensively in HDEs [10].

That leaves us in the following situation: there is excessive and increasing need to build complex mobile and pervasive systems for entertainment and professional uses. And at the same time, the fundamental engineering techniques available are inherited from stable distributed environments, and suffer from several drawbacks and weaknesses when utilized in these new environments. The only available answer currently is applying ad-hoc techniques to overcome these drawbacks and weaknesses.

This work is a direct response to the above mentioned challenge. First we propose a paradigm shift from *remote communication transparency* to *localization* being the first class concern. In other words, we no more hide or abstract location, on the contrary, we acknowledge all aspects related to location including the specification of devices, the networking paradigms they use, the different network specifications available, security features, and all related characteristics of the deployment environment. We discuss the limitations of current component models, the paradigm shift needed, and selected set of related work in section 2.

To achieve the above mentioned objective, we propose in section 3 a novel component model called *cloud component* (CC). This model includes the expected deployment environment in its definition, i.e. we raise the importance of deployment environment to be equal to the functionality required from the component. The other important feature of this novel model is that it is *fundamentally distributed*. A single CC is usually distributed over many dis-

tant hosts, the specification of these hosts are considered and fundamentally acknowledged during the development process of this CC, and all aspects related to communication, coordination, and quality of service are migrated to be internal to the border of the CC.

A software component can be thought of as unit of assembly [3] . This is true for all component models including cloud component model. In this paper we propose a new approach to assemble CCs using systematic methodology that maintains the properties of CC model. CC assembly is a tool to build large systems using CCs as building blocks. Moreover, we present a technique to automatically check the validity of this assembly. Cloud component assembly and checking are presented in section 4.

The cloud component model and CC assembly are presented informally and formally with a mathematical notation. The informal notation allows for faster comprehension of the general concepts. While the formal notation opens the door for a wide range of theoretical topics including component type inference, subtypes, etc, and provides a precise language to describe details. In addition, formal methods allow the designer to produce machine readable designs where automated tools can verify specific properties at design time, which in turn, increases the level of confidence in the correctness of design. We conclude with a brief summary of our proposals and some future work.

2 Highly Distributed Environments - HDE

The emergence of mobile devices such as portable notebook computers, tablet computers, PDAs, and mobile phones, and the advent of various wireless networking solutions make computation possible anywhere [14,13,12]. In this paper, we define highly distributed environments as a target platform of our work. These networks include distributed systems with laptops and wireless networks, mobile systems, and pervasive systems. These networks violate many familiar assumptions about the behavior of distributed environments, and demand new techniques to build compatible and optimized software [1], especially at the architectural level of the software development process.

The characteristics that the HDE infrastructure imposes include [12,1,4,5]: 1- Mobility of hardware, data, and code. 2- Heterogeneity of software and devices. 3- Volatility of hardware and software components. 4- Small devices, highly constrained resources, dynamic resources. 5- Connectivity failure are not rare; disconnected operations. 6- High bandwidth and low latency are no more available in continuous and dependable manner. 7- Software components communicate using a variety of interaction paradigms (e.g.,

[3] In this article we prefer to use the word *assembly* rather than *composition* since the output of this operation (assembly) is not a software component.

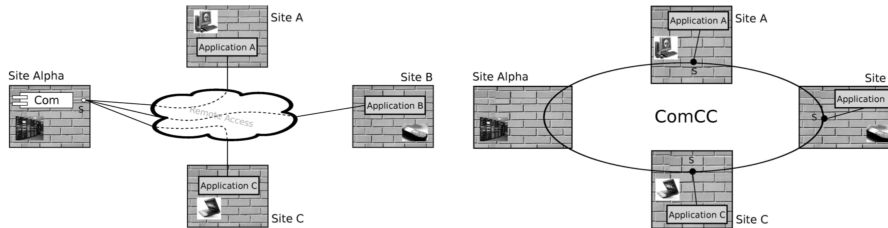


Fig. 1. Left: Distributed component model (CCM, EJB, .Net, etc) - remote access; no control over the underneath infrastructure. Right: Cloud component model - local access; the component is responsible for remote communication.

SOAP messaging, media streaming).

In spite of the above challenges that permeate the entire traditional software development life cycle, software in these systems are expected to obey the following constraints [4,14,5,10]: 1- Customized implementation: the implemented software need to be efficient, customized, and can be deployed on resource constrained devices. 2- Correctly respond to runtime changes in the environment. 3- Preserve dependability and quality of service in this highly dynamic environment.

2.1 Current component models limitations

The concept of software components has been widely adopted because of its attractive and powerful encapsulation attributes [3,9]. Lau *et al.* noted: "Encapsulation has the potential to counter *complexity*" [9].

After analyzing several component technologies such as CCM and EJB for industry and Fractal and SOFA for academia, we found that they follow a common paradigm. These component models rely on strong assumptions, and they emulate local call on top of distributed networks, and finally they consider any deviation from their implicit or explicit assumption as exceptions. All of these points are considered limitations with regards to HDEs. For more detailed discussion on these limitations, please refer to appendix A.

2.2 Paradigm shift with cloud components

To overcome these limitations, we propose a new component model called *Cloud Component* (CC) which is a novel extension to the 'Medium' concept proposed by Beugnard et al [2,11]. CC encompasses all features provided by currently existing component models and, moreover, is especially designed to be used in the above mentioned complex environments. CC model provides the capability of the instantiation of its interface(s) on each host that potentially needs to access the service provided by this CC [2]. This will make the service access explicitly local. In other words, if we want our component to be accessed

at some host, we need to deploy an interface instance at that host. It is evident that this instance will have some sort of remote communication with other entities inside the component, but this is internal with respect to the component border as explained in figure 1.

Migrating the communication to be internal inside the CC border has significant contribution to the overall architecture of distributed applications. In figure 1 (left) the server provides a service S which is accessible in sites A, B, and C. If the resources at site A are not enough, or the connectivity at site C is not adequate, or simply the configuration of site B is not compatible, the service S of component Com is not accessible, i.e. useless.

With CC, figure 1 (right), this is not the case. Cloud component comCC has its interface S instantiated on sites A, B, and C. Using it is simply a local access of a locally available service. S at the three sites provides the same (or similar) functionality, however, it is possible (and highly probable in this case) to be implemented using completely different approaches. For example, in site A special arrangements should be carried out to handle the extremely limited resources and the mobile networking. In site C, constraints of site A are relaxed, as result, different implementation technologies are utilized. The same argument holds for site B, where there are stable fixed networking and power supply, and rather advanced resources. At this site, there is no need for the implementation to be prepared to handle complexities that arise in sites such as A or C.

In other words, we allow several implementations of the same functionality to exist side by side. The variation of implementation is driven by the variation of deployment environment, i.e. the characteristics of the hosts where interfaces will be instantiated, and the characteristics of connectivity available for each host. For the interface S to be instantiated on site A, there need to be implementation variant that is compatible with the characteristics of site A. This compatibility is checked statically before the instantiation of the system and each of its roles but is out of the scope of this article.

2.3 Related Work

Didier Hoarau *et al.* deal with the challenges of HDEs [5,6,7]. However, their solution has different scope from ours. First, they expand the already existing component model Fractal. Second, they only model and handle the disconnection of network connection among all characteristics of HDE.

Marija Mikic-Rakic provides a sophisticated response to one challenge proposed by HDEs, which is the discontinuity of services where the system needs to continue functioning in the near absence of the network [12]. This work proposes a redeployment solution as part of a middleware called Prism-MW.

Finally, and the most related work, Sam Malek *et. al.* propose a frame-

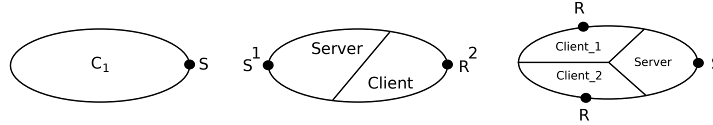


Fig. 2. Left: CC style with a single interface S. Middle: CC with two roles, cardinality, and location. Right: CC *com* with two roles and three hosts.

work and tools to support the complete software engineering life-cycle for the development of HDE applications [10]. While their tools help overcome the challenges posed by HDEs, these challenges become natural details in our novel component model, where they can be handled systematically.

3 Cloud Component Model

We chose the name Cloud Component since our component model encompasses physical borders and hence hides the technologies, implementation variants, and architecture choices used to conform to the physical topology of the underneath infrastructure. Our approach in presenting cloud component model is based on two different notations: the informal notation and the mathematical/formal notation. The informal notation is easier to understand and is highly dependent on figures, while the formal notation is more compact, more precise, and less ambiguous. The formal notation allows us to communicate precise details easier, and allows us to easily present statements and proofs.

3.1 The definition of cloud component

Definitions 1 to 5 collectively form the definition of cloud component.

3.1.1 Definition 1: Roles

Let C_1 be a cloud component with single interface S as illustrated in figure 2 (left). S is defined through an Interface Definition Language - IDL. We assume S defines the signature of provided and required functions of C_1. The contract of this interface could be more sophisticated, but we restrict its definition for the sake of simplicity.

A cloud component can have several interfaces : P, Q, R, etc. We call these interfaces 'roles' because their identification (set of functions they gather) is guided by the way the component can be used through this interface. The cloud component in figure 2 (middle) has two roles: S and R.

3.1.2 Definition 2: Cardinality

Each cloud component can have several roles. In addition, the role is allowed to have several instances, i.e., several carbon copies of the same IDL. The total number of instances of a role in a running version of the component is called: the cardinality of the role. In figure 2 (middle) the role S has cardinality one and the role R has cardinality two. Combined with location property (explained later), this approach will encapsulate the communication and all its details and semantics inside the component.

3.1.3 Definition 3: Connection

Once the component border is defined, the connection rules can be defined. In order to suppress ambiguity of 1-to-many or many-to-many connections identified in [11] we allow a role to connect to only one role of another cloud component in a one-to-one connection.

This rule applies at the instance level, when cloud components are actually implemented. In order to allow a 1-to-many or many-to-many connectivity, we use 'role cardinality'.

3.1.4 Definition 4: Multiplicity

Cardinality is a number $k \in N$. We can allow more complex structure by not specifying k (at some point of the design). Instead, we put constraints on k called multiplicity. For example a role R can have multiplicity [1..5], [1..$*$], or simply $*$.
At this level of definition, we are not bounded by decidability features but only consider constraints definition.

3.1.5 Definition 5: Location

Each role is assigned a location to run on. The location in the most basic form is a computing host/device. In figure 2 (middle) a cloud component has two different roles, S and R. Role S has one instance that is located at the host $Server$. The role R has two instances that are located at the host $Client$. Figure 2 (right) presents a cloud component that has two different roles, S and R. Role S has one instance that is located at the host $Server$. The role R has two instances one of them is located at the host $Client_1$ and the other is located at host $Client_2$.

One should not mix our definition of location with the *geographic location*. Our model does not define or recognize geographic location, rather, we acknowledge location as a computing/electronic device that might be mobile or not. It is fundamental to assert that location is integral part to the CC definition, in other words, without location specification the cloud component

definition will not be complete. Finally, and at design and implementation stages, the collection of all locations are called *'the expected deployment environment.'*

3.2 Formal definition of cloud component

A single cloud component is defined using the following four-tuple:

(i) A finite set of roles $\overline{\Lambda}$.
(ii) A finite set of multiplicities for these roles $\overline{\mu}$.
(iii) A set of possible deployment environments $\overline{L}$. Each L is either a finite set of hosts $\overline{H}$, or a finite set of host types $\overline{T}$.
(iv) A function Z that maps roles to location types or hosts.

$$\Omega \equiv (\overline{\Lambda},\ \overline{\mu},\ \overline{L},\ Z)$$

The following formally defines the cloud component *com* in figure 2 (right):

$$\begin{aligned} &\Omega com \equiv (\overline{\Lambda},\ \overline{\mu},\ \overline{L},\ Z) \ \text{ where:}\\ &\quad \overline{\Lambda} \ = \ \{\Lambda S,\ \Lambda R\}\\ &\quad \overline{\mu} \ = \ \{(\Lambda S,\ 1),\ (\Lambda R,\ 2)\}\\ &\quad \overline{L} \ = \ \{\{TServer,\ TClient_1,\ TClient_2\}\}\\ &\quad Z: \ \ \Lambda S \downarrow TServer,\ \ \Lambda R \downarrow TClient_1,\ \ \Lambda R \downarrow TClient_2 \end{aligned}$$

The formal definition is read as follows: the CC *com* is defined using its four-tuple. The set of roles contains two roles: role type S and role type R. Role type S has multiplicity 1, and role type R has multiplicity 2. The set of expected deployment environments has only one set, which contains three host types: host type $Server$, host type $Client_1$, and host type $Client_2$. Finally Z can be read as: The role S is localized at host of type $Server$. Role R has two instances, one is localized at host of type $Client_1$ and the other is localized at a host of type $Client_2$. Symbols used to construct the formal notation are summarized in table B.1 in appendix B.

3.3 Formal definition of cloud component based system

Generally, a software component can be thought of as 'unit of composition'. This is true for all component models including cloud component model. In CC model, roles are the only access points of the component. A role can serve as a connection port where component C_1 connects to other component C_2 as in figure 3 (left). We choose to assembly components in specific architecture to achieve our desired system specifications. *As result we have σ the set of assembly rules that includes the dependency rules between CCs, and all role connections.* Cloud component assembly will be discussed in detail in section 4.

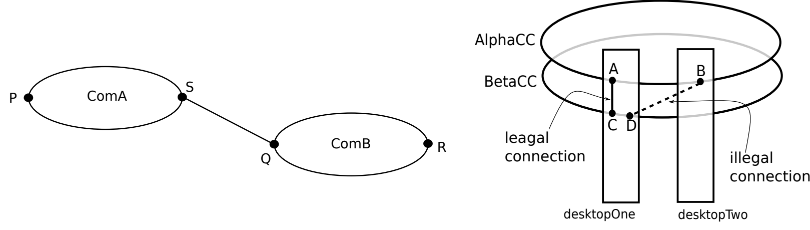


Fig. 3. Left: Two CCs are composed using roles S and Q. Right: Two CCs $AlphaCC$ has two role instances A and B, and $BetaCC$ has two role instances C and D. A, C, and D are hosted by $desktopOne$, while B is hosted by $desktopTwo$. Therefore, the connection between A and C is legal, whereas the connection between B and D is not permitted.

A system built using cloud components consists of:

(i) A finite set of cloud components $\overline{\Omega}$.
(ii) A finite set of multiplicities for these cloud components $\overline{M}$.
(iii) A set of assembly rules σ.
(iv) A set of possible deployment environments $\overline{L}$.

As result, the system type is fully defined using the "four-tuple" notation:

$$S \equiv (\overline{\Omega},\ \overline{M},\ \sigma,\ \overline{L})$$

Finally we define the system instance $\hat{S}$. Let S be a CC based system that is defined as above. $\hat{S}$ is an instance of that system and is defined using the following five-tuple:

(i) The system type S that we want to instantiate.
(ii) The function τ that takes a cloud component as a parameter and returns the number of instances of it.
(iii) The function K that takes a role as a parameter and returns its cardinality, i.e. number of instances.
(iv) The deployment environment L which is a finite set of hosts $\overline{H}$.
(v) The function Z that maps $\overline{\Gamma}$[4] to L.

$$\hat{S} \equiv (S, \tau, K, L, Z)$$

4 Cloud Component Assembly

4.1 Assembly Constraints

In CC model, roles are the only access points of the component. A role can serve as a connection port where component $ComA$ connects to other component $ComB$ as in figure 3 (left).

[4] Γ is defined in appendix B.

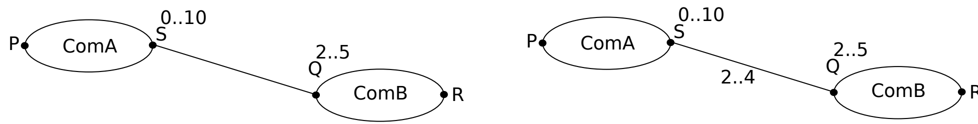


Fig. 4. The importance of the 'connection multiplicity'. Left: No information. Right: The multiplicity of the connection is defined: [2..4].

4.1.1 First constraint - one-to-one

An instance of role S can connect to one instance only of any other role at any time instance. We raised the importance of this constraint from being a recommended design choice to be a fundamental model constraint for several reasons. One of these reasons is to remove ambiguities in the connections. Another and important reason is to control the design precisely, and to be able using this control to ensure the delivery of the expected non-functional properties. As an example, let us take the role S in figure C.1 from the banking example in appendix C. And suppose S is hosted by some regular desktop. If S is expected to have 10 connections, i.e. 10 clients that want to use the video service, is completely different from S is expected to have 10^6 connections at the same time. The difference exists in the design, implementation, and the deployment host (probably a normal desktop will not be able to serve 10^6 connections). This difference should be recognized from the very early stages in the design, and this is done in CC model by setting the multiplicity (or cardinality) constraints over roles.

4.1.2 Second constraint - local connections only

Two instances of two roles can connect to each other only if both of them are instantiated at the same host as in figure 3 (right). If they are instantiated at different hosts they simply can not connect to each other. This is a direct result of the paradigm shift discussed in section 2.2. It is fundamental in our model to migrate all remote communications to be internal to the border of the CC itself. This migration means that these remote communications are designed and implemented using the special software development process [5] of the CC model, and more important, passed all checks necessary to ensure the quality of service expected.

4.1.3 Third constraint - Connection multiplicity

When there is a connection between two roles, that does not mean that all instances of these two roles should connect to each other. Figure 4 (left) is

[5] We propose a novel software development process to build CCs and CC based systems. The description of this process is out of the scope of this article and we will propose it in future publication.

an update of figure 3 (left) by adding multiplicities to roles S and Q. To understand the connection in this figure we need to see the uncertainty that exist at this phase of design. During runtime, there might be one instance of S and five instance of Q, or nine instances of S and two instance of Q. So how many connections we have at runtime between S and Q? To answer this question we need to remember that the final responsibility of the design is held by the designer himself, we only provide an advanced model and accompanied tools and checkers. To facilitate the assembly design we add the *connection multiplicity*, which is a range $[min..max]$, where min is the minimum number of connections that must exist at runtime, and max is the maximum number of connections that might exist at runtime, as in figure 4 (right). Usually these numbers reflect the need of either of the roles, or both. For example if I have a role W that connects an ATM machine (CC ATM) to the bank system (role S of CC $Agent$), I can expect W to need only one connection at runtime, i.e. [1..1]. On the other hand I expect S to allow zero or more connections at runtime, i.e. $[0..*]$. Please see figure C.1 in appendix C.

4.2 Formal definition of cloud component assembly

CC assembly is based on the connection operator $\otimes$, which is a binary operator that takes two CC roles and returns true if the designer explicitly listed those two roles to be connected (this is done in σ as described later), otherwise it returns false. The set of assembly rules is called σ and is defined using the following context free grammar:

$E \rightarrow \{\ I\ \}$

$I \rightarrow IJ, \ \mid \ IJ \ \mid \ \epsilon$

$J \rightarrow (\Omega var.\Lambda var \otimes \Omega var.\Lambda var, \ \ int, \ \ int)$

Where var and int are terminals such that: var represents any string of characters and int represents a positive integer. This grammar will recognize the following syntax:

$\sigma = \{(\Omega name.\Lambda name \otimes \Omega name.\Lambda name, min, max), \quad (\Omega name.\Lambda name \otimes \Omega name.\Lambda name, min, max), ..., (\Omega name.\Lambda name \otimes \Omega name.\Lambda name, min, max)\}$.

The following shows the assembly in figure 4 (right) using formal notation:

$\sigma = \{(\Omega ComA.\Lambda S \otimes \Omega ComB.\Lambda Q, \ \ 2, \ \ 4)\}$

In general we can write:

$\sigma = \{(\Omega ComA.\Lambda S \otimes \Omega ComB.\Lambda Q, \ \ m, \ \ n)\}$

This connection has the following semantics: at least m instance of S connect to m instance of Q, and at most n instance of S connect to n instance of Q. *This is correct for one and only one instance of each cloud component.*

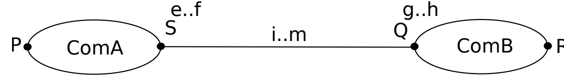


Fig. 5. CC assembly normal form A. Ranges are always consistent (i.e. $min \leqslant max$).

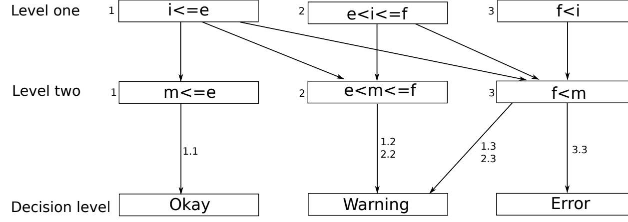


Fig. 6. The relation between the two ranges $[e..f]$ and $[i..m]$ in figure 5. We start with level one, and depending on the value of i we move to level two where we inspect the value of m. The label(s) on the arrows leading to the decision level indicate the decisions made on the upper two levels.

4.3 Remark

The connection operator $\otimes$ is symmetric:
$\Omega ComA.\Lambda S \otimes \Omega ComB.\Lambda Q \Longleftrightarrow \Omega ComB.\Lambda Q \otimes \Omega ComA.\Lambda S$
The above statement is read as follows: role S is connected to role Q if and only if role Q is connected to role S.

4.4 Assembly checking algorithm

Figure 5 presents the general case of assembly, which is defined as normal form A assuming there is a single instance of the CC. The connection here has the following semantics (as mentioned in the definition): at least i instance of $S(Q)$ connect to i instance of $Q(S)$, and at most m instance of $S(Q)$ connect to m instance of $Q(S)$. This is correct for one and only one instance of each cloud component $ComA$ and $ComB$.
The two ranges $[e..f]$ and $[g..h]$ are not related in any way since cloud components may have been designed independently. On the other hand, the two ranges $[e..f]$ and $[i..m]$ are related as in figure 6. Cases presented in figure 6 can be reduced to the following four cases:

(i) $i \leq e$ & $m \leq e \Rightarrow$ Valid
(ii) $i \leq e$ & $e \leq m \Rightarrow$ Warning
(iii) $e < i \leq f \Rightarrow$ Warning
(iv) $f < i \Rightarrow$ Error

The same argument holds for the two ranges $[g..h]$ and $[i..m]$ in figure 5. Depending on the numbers, we have three cases:

(i) Valid: in this case we do not have a chance of connection problems at runtime if the instantiation of roles respected the design.

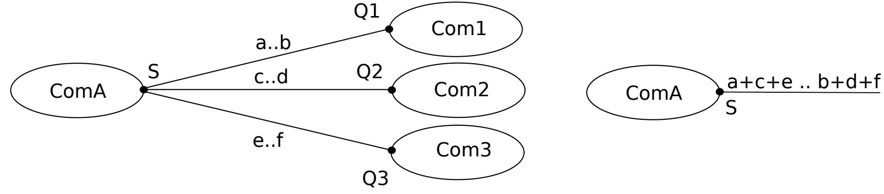


Fig. 7. Left: CC assembly normal form B. Multiple connections - role S is connected to three roles $Q1$, $Q2$, and $Q3$. Right: Role S after assembly reduction - phase one.

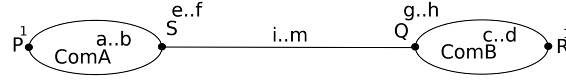


Fig. 8. CC assembly normal form C. Other CCs can connect to Q, S, etc. Omitted for space.

(ii) Warning: in this case the designer need to be careful because even if the instantiation respected the minimum requirements, we might face invalid situations. For example, if $e < i \leq f$, and at runtime we have only e instances of S (legal situation), and we need i connections to S. This situation will produce runtime error. As result, before asking for i connections to S, the application must instantiate at least i instances of S (possible because $i \leq f$).

(iii) Error: here we do not have enough instances of the role to satisfy the minimum connections need.

A role (specifically, role type) is not limited to be connected to only one other role, rather, this number is unlimited. At runtime, this role is expected to have several instances, where each instance is connected to one other role. This is assembly normal form B and presented in figure 7 (left). To check this assembly we need to get it back to normal form A in figure 5. We call this conversion from the form normal form B to normal form A: assembly reduction - phase one. For role S in figure 7 this is accomplished as in figure 7 (right). Formally: Let $\sigma = \{(S \otimes Q_1, a_1, b_1), (S \otimes Q_2, a_2, b_2), \cdots, (S \otimes Q_n, a_n, b_2)\}$. After assembly reduction - phase one, we get: $\sigma = \{(S, Q, a, b)\}$ such that: $a = \sum_{i=1}^{n} a_i$, $b = \sum_{i=1}^{n} b_i$, and Q is virtual role for checking only. This is for role S only and must be done for all other roles that have connections to more than one role.

Figure 8 presents CC assembly normal form C. The connection here has the following semantics: at least i instance of S(resp.Q) connect to i instance of Q(resp.S), and at most m instance of S(resp.Q) connect to m instance of Q(resp.S). This is correct for one and only one instance of each cloud component. More over, CCs $ComA$ and $ComB$ have multiplicities $[a..b]$ and $[c..d]$ respectively.

Because of the multiplicities of the CCs, we are unable to use the checking procedure used for normal form A directly on normal form C. To be able to check this assembly, we will follow several assembly reductions starting

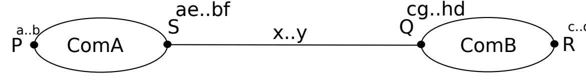


Fig. 9. The result after reduction phase two and three on figure 8- CC multiplicities are completely removed.

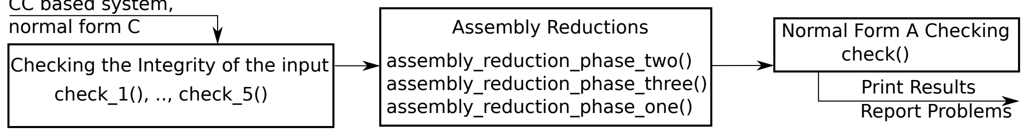


Fig. 10. Inclusive checking algorithm. The integrity checks, namely, check1() through check5(), insure that the input is not corrupted with respect to normal form C.

from this general model. Assembly reduction phase two reduces the multiplicity of the CC to be incorporated (inserted) into the multiplicities of its roles. Formally, let: $\Omega ComA \equiv (\{\Lambda P, \Lambda S\}, \{(\Lambda P, 1), (\Lambda S, e, f)\}, \overline{L}, Z)$ and $\Omega ComB \equiv (\{\Lambda Q, \Lambda R\}, \{(\Lambda Q, g, h), (\Lambda R, 1)\}, \overline{L}, Z)$. $\sigma = \{(\Omega ComA.\Lambda S \otimes \Omega ComB.\Lambda Q, i, m)\}$. Now let $S \equiv (\{\Omega ComA, \Omega ComB\}, \{(\Omega ComA, a, b), (\Omega ComB, c, d)\}, \sigma, \overline{L})$. Assembly reduction phase two produces the new multiplicities for all roles: $\overline{\mu_{ComA}} = \{(\Lambda P, a, b), (\Lambda S, ae, bf)\}$ and $\overline{\mu_{ComB}} = \{(\Lambda Q, cg, hd), (\Lambda R, c, d)\}$.

Assembly reduction phase three is trickier. The multiplicity of the connection $[i..m]$ is affected by both CC's multiplicities, namely $[a..b]$ and $[c..d]$. The objective of this phase is to end up with the connection multiplicity $[x..y]$ using the rule: 'for x we choose the max of the mins, and for y we choose the max of the maxs'. Formally: $x = max\{ia,\ ic\}$, and $y = max\{mb,\ md\}$. By the end of this phase we will get back to normal form A that can be checked directly as in figure 9.

The algorithm in figure 10 is fully implemented using C programming language, and used to check the banking system example in appendix C.

5 Conclusion and Future Work

Highly distributed environments pose a number of challenges for software development process. In this paper we propose a novel component model, the cloud component, that converts the above mentioned challenges into regular and systematic software development details and tasks. Moreover, we proposed a formal notation to describes our component model. This formal notation is more compact, more precise, and less ambiguous. The formal notation allows us to communicate precise details easier, and allows us to easily present statements and proofs. Finally, we developed a formal model to build large systems using CCs as building blocks, and developed an algorithm to check the validity of the assembly, and implemented an automatic assembly checker based on that algorithm.

Several remaining challenges form the scope of our future work. The most important challenge is related to the deployment environment modeling, and designing an effective algorithm to be the basis of an automatic deployment checker that checks the compatibility between cloud components and the actual environment where we are trying to deploy. We have investigated techniques to accomplish this task, these techniques include Ontology and F-Logic (Object Logic) for modeling. Moreover, we investigated several algorithms that depend on reasoning based queries for automatic deployment checker.

References

[1] Cardelli, L., *Abstractions for mobile computation*, Secure Internet Programming, Security Issues for Mobile and Distributed Objects - Lecture Notes in Computer Science **1603** (1999).

[2] Cariou, E., A. Beugnard and J.-M. Jézéquel, *An architecture and a process for implementing distributed collaborations*, in: *6th International Enterprise Distributed Object Computing Conference EDOC 2002, Lausanne, Switzerland, Proceedings* (2002), pp. 132–143.

[3] Crnkovic, I., A. Vulgarakis and M. Chaudron, *A classification framework for software component models*, Software Engineering, IEEE Transactions on (2010).

[4] France, R. and B. Rumpe, *Model-driven development of complex software: A research roadmap*, in: *International Conference on Software Engineering, ISCE, Workshop on the Future of Software Engineering, FOSE, Minneapolis, USA*, IEEE Computer Society, 2007, pp. 37–54.

[5] Hoareau, D., "Composants ubiquitaires pour reseaux dynamiques," Ph.D. thesis, Universite de Bretagne Sud (2007).

[6] Hoareau, D. and Y. Mahéo, *Distribution of a hierarchical component in a non-connected environment*, in: *31st EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA), Porto, Portugal*, 2005.

[7] Hoareau, D. and Y. Mahéo, *Middleware support for the deployment of ubiquitous software components*, Personal and Ubiquitous Computing **12** (2008), pp. 167–178.

[8] Hourdin, V., J. Tigli, S. Lavirotte, G. Rey and M. Riveill, *SLCA, composite services for ubiquitous computing*, in: *Proceedings of the International Conference on Mobile Technology, Applications, and Systems* (2008).

[9] Lau, K. and Z. Wang;, *Software component models*, Software Engineering, IEEE Transactions on **33** (2007), pp. 709–724.

[10] Malek, S., G. Edwards, Y. Brun, H. Tajalli, J. Garcia, I. Krka, N. Medvidovic, M. Mikic-Rakic and G. Sukhatme, *An architecture-driven software mobility framework*, Journal of Systems and Software **83** (2010), pp. 972–989.

[11] Matougui, S. and A. Beugnard, *Two ways of implementing software connections among distributed components*, in: *On the Move to Meaningful Internet Systems, OTM Confederated International Conferences, Agia Napa, Cyprus, Proceedings, Part II*, 2005.

[12] Mikic-Rakic, M., "Software architectural support for disconnected operation in distributed environments," Phd dissertation, University of Southern California (2004).

[13] Mikic-Rakic, M. and N. Medvidovic, *Architecture-level support for software component deployment in resource constrained environments*, Component Deployment (2002), pp. 493–502.

[14] Mikic-Rakic, M. and N. Medvidovic, *Software architectural support for disconnected operation in highly distributed environments*, Component-Based Software Engineering (2004), pp. 23–39.

[15] Tibermacine, C., D. Hoareau and R. Kadri, *Enforcing architecture and deployment constraints of distributed component-based software*, Fundamental Approaches to Software Engineering (2007), pp. 140–154.

A Current component models limitations

After analyzing several component technologies, we found that they follow a common paradigm. These component models rely on strong assumptions, and they emulate local call on top of distributed networks, and finally they consider any deviation from their implicit or explicit assumption as exceptions. All of these points are considered limitations with regards to HDEs as we explain in the following:

Rely on strong assumption A common way to distribute a component-based application consists of installing each component instance on a host; the distribution then refers to the fact that a component can make distant invocations to the services implemented by another component [6,11]. This type of architecture usually relies on rather strong assumptions [6]:

(i) The stability of the execution platforms (the component server is highly available - usually with backup recovery system)

(ii) All hosts have sufficient resources which include processing power, memory, and power supply.

(iii) The connectivity is reliable and has good characteristics (low latency, enough bandwidth, stable, no disconnections, etc.).

In general, an application designed using this architecture can not be installed and executed on deployment environments with hosts that are potentially volatile and limited in resources, especially when disconnected network operation and weak consistency of the characteristics of the connectivity are possible or frequent, which is the case in HDEs [6,7].

Emulation The distributed component models mentioned above share a common goal: making aspects related to the distribution transparent to both the application programmer and the users. They hide distribution by making remote call appears to the caller as local call, but to some ad-hoc and limited exceptions (see next point). However, by hiding distribution, these mechanisms do not incorporate aspects related to disconnections, mobility, and all other complexities mentioned in the previous section [8]. In general, distributed applications are designed using the same techniques as a centralized application [5,15].

Exceptions Most common component technologies were not originally designed for HDE. Therefore, they consider any deviation from the strong assumptions mentioned above such as inaccessibility of a machine or the unavailability of certain resources as exceptions. The treatment of the various changes that may occur within the network is usually done by adding code to adapt to these new events. This code will increase the complexity of applications [5,15] with specific and ad-hoc extensions and poor methodological guidelines.

Typical HDE applications are highly distributed, decentralized, and mobile. Therefore, they are *highly dependent on the underlying network* [14,13,12]. We believe that the successful paradigm in stable distributed networks *'remote communication'* or *'distribution transparency'* is no more dependable in highly distributed environments HDE. There is fundamental need to move from *hiding* the underlying network into *acknowledging* all its aspects and details. It is possible to achieve that by introducing the concept *'location'*. We call this a *paradigm shift from 'distribution transparency' to 'localization'*.

By location we mean the physical actual computing device where software runs, and that ranges from simple mobile phone to large super computing machine. Modeling location at early design stages will better reflect communication, mobility, and heterogeneity of devices.

Instead of delaying the distribution of software components over the computing devices until the deployment phase, we propose integrating this concept to the very early stages of software development process, especially architecture. Mapping software components to the deployment environment is called *'localization'*. Several models and techniques are proposed in this work to facilitate this approach. The localization of components is revised and refined during the development process until we reach the final deployment plan of the whole application.

It is clear that when we attach location property to a software component, we – either explicitly or implicitly – attach information related to the resources available in this location, the communication paradigms available, the power supply type, and the security features, etc. This information will help (guide) the design and implementation of the system itself.

Acknowledging the properties of the target infrastructure at the very early stages of software development process will help us develop customized software for that infrastructure. This software will utilize the resources to the maximum or near maximum, and at the same time will tolerate the weaknesses and treat the previously considered exceptions as survivable expected events.

B Symbols

Cloud Components and CC based systems can be described using formal/mathematical notation. In this appendix we present the elements and symbols of this notation/language in table B.1.

C Example - Banking System

In this appendix we presnt a simple banking system to explain the algoritm proposed in section 4. The banking example is presented in figure C.1. The $*$

Concept	Symbol	Comments
Start Symbol	S	Usually the complete system we want to model
Cloud Component	Ω	ΩA is read 'cloud component A'
Roles	Λ	Type ; ΛR is read 'role R"
	Γ	Instance ; ΓR is read 'instantiated role R'
Cardinality	K	
Role Multiplicity	μ	
CC Multiplicity	M	
Location - Type	T	
Location - Host	H	
Localized at	$\downarrow$	
CC Assembly	σ	The set of assembly rules
Connect Operator	$\otimes$	A binding between two roles
Set of	$\overline{symbol}$	Set of CCs $\overline{\Omega}$, set of roles $\overline{\Lambda}$, etc.
Define	$\equiv$	

Table B.1
Symbols used to construct the formal notation.

symbol can be reduced to $[0..MAXINT]$ for computations. In this example the system is built using three CCs. The *Bank* CC is responsible for all database systems, security, transactions, and accounts. It is basically the backbone of the system. The *Agent* CC is the filter that any access to *Bank* will pass through. In other words, nobody can directly access *Bank* CC. *ATM* CC is installed over all ATM machines to allow customers to access their accounts, and perform bank transactions. Similarly, *Internet* CC is installed on the customers devices to allow them to access their accounts using internet banking.

We encoded this example using the formal language presented in this paper, and used the automatic assembly checker to check the design. The assembly checker generated the output presented in figure C.2.

The checker reports expected warnings and errors. For instance the error reported (figure C.2 - right) on the system described figure C.1 (bottom) is due to the too many instantiations of the ATM and Internet CCs.

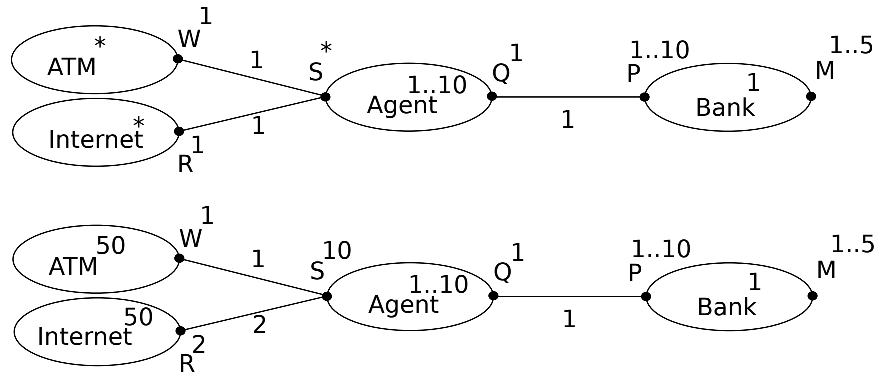


Fig. C.1. The banking system in normal form C. Up: Enterprise Edition. Down: Limited Edition

Warning!!!!
Problem type 1.2
Conncetion to P.
This connection is not safe because it is dependent on the max kardinality.
Warning!!!!
Problem type 1.2
Conncetion to Q.
This connection is not safe because it is dependent on the max kardinality.

The design has potential problems. Please read messages.

Warning!!!!
Problem type 1.2
Conncetion to P.
This connection is not safe because it is dependent on the max kardinality.
Warning!!!!
Problem type 1.2
Conncetion to Q.
This connection is not safe because it is dependent on the max kardinality.
Error!!!! Conncetion to S.
This connection is illegal because it is exceeds the max kardinality.

The design has major errors. Please read messages.

Fig. C.2. The output generated by the assembly checker. Up: For Enterprise Edition. Down: For Limited Edition