Universitetet i Oslo
Institutt for informatikk

NWPT'07/FLACOS'07

Workshop Proceedings

October 9–12, 2007
Oslo, Norway

Einar Broch Johnsen,
Olaf Owe,
Gerardo Schneider
(editors)

Research Report 366
ISBN 82-7368-324-9
ISSN 0806-3036

October 2007

Proceedings of the

19th Nordic Workshop on Programming Theory

10-12 October 2007

and

First Workshop on Formal Languages and Analysis of Contract-Oriented Software

9-10 October 2007

Oslo, Norway

Einar Broch Johnsen, Olaf Owe, Gerardo Schneider (editors)

Foreword

NWPT'07

The objective of the workshop series is to bring together researchers from the Nordic and Baltic countries interested in programming theory, in order to improve mutual contacts and cooperation, especially encouraging young researchers.

The 19th Nordic Workshop on Programming Theory takes place in Oslo, Norway, and is organized by the PMA group at the Department of Informatics, University of Oslo. NWPT'07 is partially supported by the EU project *CREDO*: Modeling and analysis of evolutionary structures for distributed services (IST-33826). The organizing committee consists of Einar Broch Johnsen, Olaf Owe, and Gerardo Schneider.

The workshop attracted 40 submissions, of which 30 were selected. There are 3 invited talks:

Gilles Barthe	INRIA, Sophia-Antipolis	France
Davide Sangiorgi	University of Bologna	Italy
Neelam Soundarajan	Ohio State University	USA

and 52 participants. The programme committee consists of:

Luca Aceto	Reykjavík Univ., Iceland/Aalborg Univ., Denmark
Michael R. Hansen	Techn. U. of Denmark, Denmark
Anna Ingolfsdottir	Reykjavík Univ., Iceland, and Aalborg Univ., Denmark
Einar Broch Johnsen	University of Oslo, Norway (co-chair)
Kim G. Larsen	Aalborg Univ., Denmark
Bengt Nordström	Univ. of Gothenburg, Chalmers Univ. of Tech., Sweden
Olaf Owe	University of Oslo, Norway (co-chair)
Gerardo Schneider	University of Oslo, Norway (co-chair)
Tarmo Uustalu	Inst. of Cybernetics, Estonia
Jüri Vain	Tallinn Technical University, Estonia
Marina Waldén	Åbo Akademi University, Finland
Uwe E. Wolter	Univ. of Bergen, Norway
Wang Yi	Uppsala Univ., Sweden

Further information can be found under the workshop homepage: `http://nwpt07.ifi.uio.no`.

FLACOS'07

The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07) is held in Oslo, Norway. The aim of the workshop is to bring together researchers and practitioners working on language-based solutions to contract-oriented software development. The workshop is partially funded by the Nordunet3 project "COSoDIS" (Contract-Oriented Software Development for Internet Services) and it attracted 32 participants.

The program consists of 5 regular papers and 12 invited participant presentations. The regular papers were selected by the following programme committee:

Pablo Giambiagi	SICS, Sweden
Olaf Owe	University of Oslo, Norway (co-chair)
Anders P. Ravn	Aalborg University, Denmark
Gerardo Schneider	University of Oslo, Norway (co-chair)

Further information can be found at the workshop homepage: `http://www.ifi.uio.no/flacos07`.

Acknowledgments

We thank the Department of Informatics, University of Oslo, for financial support, Mozhdeh Sheibani Harat for help with hotel bookings, and the following PhD students for doing various tasks: Joakim Bjørk, Daniela Lepri, and Cristian Prisacariu.

Welcome to Oslo!
Einar Broch Johnsen, Olaf Owe, Gerardo Schneider

Table of Contents

NWPT - The 19th Nordic Workshop on Programming Theory

organized by sessions

FLACOS - The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software

NWPT 2007

The 19th Nordic Workshop on Programming Theory

Certificate translation

Gilles Barthe
INRIA Sophia Antipolis-Méditerranée
Gilles.Barthe@inria.fr

September 27, 2007

Proof Carrying Code (PCC) provides a means to establish trust in a mobile code infrastructure, by requiring that mobile code is sent along with a formal proof that it adheres to a security policy agreeable by the code consumer. A typical PCC architecture comprises at least the following items: a formalism for specifying policies, a verification condition generator (VCGen) that generates proof obligations from the program and the policy, a formal representation of proofs, known as certificates, and a certificate checker. While PCC does not make any assumption on the way certificates are generated, the prominent approach to certificate generation is *certifying compilation*, which generates certificates automatically for safety policies such as memory safety or type safety. Yet, experience has shown that powerful verification technology, including possibly interactive verification, is often required, both for basic safety policies such as exception safety, and for more advanced security policies such as non-interference and resource control. Thus, there is a need for generating certificates from program verification environments.

However, verification environments target high-level languages whereas code consumers require certificates to bring correctness guarantees for compiled programs. One possible approach to solve the mismatch is to develop verification environments for compiled programs, but the approach has major drawbacks, especially in the context of interactive verification: one looses the benefit of reasoning on a structured language, and the verification effort is needlessly duplicated, as each program must be verified once per target language and compiler. A better solution is to develop methods for transferring evidence from source code to compiled programs, so that verification can be performed as usual with existing tools, and that code is only proved once.

Certificate translation is a mechanism for bringing the benefits of interactive source code verification to code consumers, using a Proof Carrying Code architecture, More formally, the goal of certificate translation is to transform certificates of original programs into certificates of compiled programs. Given a compiler $[\![.]\!]$, a function $[\![.]\!]_{\text{spec}}$ to transform specifications, and certificate checkers (expressed as a ternary relation "c is a certificate that P adheres to ϕ", written $c : P \models \phi$), a certificate translator is a function $[\![.]\!]_{\text{cert}}$ such that for all

programs p, policies ϕ, and certificates c,

$$c : p \models \phi \quad \Longrightarrow [\![c]\!]_{\mathrm{cert}} : [\![p]\!] \models [\![\phi]\!]_{\mathrm{spec}}$$

The talk shall provide sufficient conditions for the existence of certificate translators, and discuss the relation between certification translation, certifying compilation and certified compilation.

The talk is based on joint work with Benjamin Grégoire, César Kunz, Mariela Pavlova, and Tamara Rezk. The work is funded by the EU project Mobius `http://mobius.inria.fr`.

Some historical remarks on bisimulation and coinduction*

Davide Sangiorgi

University of Bologna, Italy

Bisimulation. The classical notion of *bisimulation* is defined on a Labelled Transition System (LTS) thus, where Σ is the set of all states of the LTS:

$$\begin{array}{c} \text{a relation } \mathcal{R} \subseteq \Sigma \times \Sigma \text{ is a } \textit{bisimulation} \text{ if} \\ (P_1, P_2) \in \mathcal{R} \text{ and } P_1 \stackrel{\mu}{\longrightarrow} P_1' \text{ imply:} \\ \text{there is } P_2' \text{ such that } P_2 \stackrel{\mu}{\longrightarrow} P_2' \text{ and } (P_1', P_2') \in \mathcal{R}, \\ \text{and the converse, on the actions from } P_2. \end{array} \quad (1)$$

Bisimilarity is then defined as the union of all bisimulations. When the states of the LTS are processes, bisimilarity can be taken as the definition of behavioural equality for them.

Two important remarks on the definition of bisimilarity are the following:

1. The definition has a strong impredicative flavour, for bisimilarity itself is a bisimulation and is therefore part of the union from which it is defined.
2. The definition immediately suggests a proof technique: To demonstrate that P_1 and P_2 are bisimilar, find a bisimulation relation containing the pair (P_1, P_2) (*bisimulation proof method*).

The definition of bisimilarity is an example of coinductive definition; the bisimulation proof method is an example of coinductive proof method. What makes make the bisimulation proof method practically interesting are two features of the definition of bisimulation:

– the *locality* of the checks;
– the lack of a *hierarchy* on the pairs of the bisimulation.

The checks are local because we only look at the immediate transitions that emanate from the states. An example of a behavioural equality that is non-local is *trace equivalence* (two processes are trace equivalent if they can perform the same sequences of transitions). It is non-local because computing a sequence of transitions starting from a state s may require examining other states, different from s.

There is no hierarchy on the pairs of a bisimulation in that no temporal order on the checks is requires: all pairs have the same status. As a consequence, bisimilarity can be effectively used to reason about infinite objects. This is in

* Sangiorgi's research was partially supported by italian MIUR Project n. 2005015785, "Logical Foundations of Distributed Systems and Mobile Code".

sharp contrast with inductive techniques, that require a hierarchy, and that therefore are best suited for reasoning about finite objects. For instance, here is a definition of equality that is local but inherently inductive:

> $P = Q$ if whenever $P_1 \stackrel{\mu}{\longrightarrow} P_1'$ there is P_2' such that $P_2 \stackrel{\mu}{\longrightarrow} P_2'$ and $P_1' = P_2'$, plus the converse, on the actions from P_2.

This definition is ill-founded if processes are infinite, that is, can perform an infinite number of transitions.

The origins. In Computer Science, the standard reference for bisimulation and the bisimulation proof method is David Park's paper "Concurrency on Automata and Infinite Sequences" [Par81a] (one of the most quoted papers in concurrency). While the reference to David Park is fully justified, mentions to that particular paper are sometimes questionable.

David Park has been one the pioneers of theoretical Computer Science. For instance, his contributions to fixed-point theory have been fundamental. David Park has also been the first in Computer Science to formalise the notions of bisimulation and bisimilarity. In doing so, Park has completed a line of studies whose beginning may be dated back in the late 60s (the works by Landin, Manna, Floyd, and others on program correctness) and that then continued through the 70s, most notably through the work by Milner (such as [Mil70,Mil71,Mil80]). Park made the final step precisely guided by fixed-point theory. Park noticed that the inductive notion of equivalence that Milner was using for his CCS processes was based on a monotone functional over a complete lattice. By adapting an example by Milner, Park showed that Milner's equivalence was not a fixed-point for the functional; he then derived bisimilarity as the greatest fixed-point of the functional, and the bisimulation proof method from the theory of greatest fixed-points.

Park's discovery is only partially reported in [Par81a]. The main topic of that paper is a different one, namely omega-regular languages and operators for fair concurrency. Bisimulation only appears as a secondary contribution: a proof technique for trace equivalence on variants of Buchi automata. Further, the bisimulation in [Par81a] is not the classical one – for instance it is not transitive – and, above all, bisimilarity and the coinduction proof method are not mentioned. Indeed, Park never wrote a paper to report on his findings about bisimulation. It is possible that this did not appear to him a contribution important enough to warrant a paper: he considered bisimulation a variant of the earlier notions by Milner [Mil70,Mil71]; and it was not in Park's style to write many papers. The best account I have found of Park's discovery of bisimulation are the summary and the slides of his talk at the 1981 Workshop on the Semantics of Programming Languages [Par81b].

A very important contribution to the discovery and success of bisimulation is also Milner's. I have already pointed out that Park's work was strongly based on earlier results by Milner. Further, Milner immediately and enthusiastically

adopted Park's proposal, and made it the cornerstone of the theory of CCS [Mil89].

Outside Computer Science, bisimulation was independently discovered, roughly at the same time, in two other areas: Philosophical Logic (precisely, Modal Logic) and Set Theory.

In Philosophical Logic, bisimulations, called p-relations, were introduced by van Benthem in his work on correspondence theory (the study of the relationship between modal and classical logics), in 1976 [Ben76]. Precisely, van Benthem introduced bisimulation for his theorem stating that a formula of first-order logic (over Kripke structures) is equivalent to a formula of modal logic iff it is bisimulation invariant (that is, it does not distinguish bisimilar states). Only bisimulation, however, appears in van Benthem's work: he did not introduce bisimilarity or the bisimulation proof method (in other words, there is no coinduction). van Benthem's definition of bisimulation was based on earlier works by de Jongh and Troelstra [JT66], and Segerberg [Seg71].

In Set Theory, bisimulation appears at the beginning of 80's in works devoted to the the foundations of set-theory, in particular non-well-founded sets. The first such work is Forti and Honsell [FH83]. Bisimulations are called f-conservative relations. The bisimulation proof method is also introduced, derived from the theory of fixed points. The method is however rather hidden in these works, whose main goal is to study axioms of non-foundation and prove their consistency (for this the main technique uses f-admissible relations, which are essentially bisimulation equivalences). In Mathematics, bisimulation and non-well-founded sets were made popular by Aczel [Acz88], who was looking for mathematical foundations for infinite objects, such as the processes, that the work of Milner and others had shown to be important in Computer Science. Aczel used the bisimulation proof method to prove equalities between non-well-founded sets, developed the theory of coinduction, in particular he set the basis of the coalgebraic approach to semantics (Final Semantics).

More or less at the same time as Forti and Honsell, and independently from them, bisimulation-like relations are used by Roland Hinnion [Hin80,Hin81] (a related, but later, paper is also [Hin86]). Hinnion however does not formulate axioms of anti-foundation. Thus while imposing the anti-foundation axiom (called AFA) of Forti-Honsell and Aczel makes equality the only possible bisimulation for any structure, Hinnion uses bisimulations to define new structures, via a quotient. Constructions similar to Hinnion's, that is, uses of relations akin to bisimulation to obtain extensional quotient models, also appear in works by Harvey Friedman [Fri73] and Lev Gordeev [Gor82]. In this respect, however, the first appearance of a bisimulation relation I have seen is in a work by Jon Barwise, Robin O. Gandy, and Yiannis N. Moschovakis [BGM71], and used in the main result about the characterisation of the the structure of the next admissible set A^+ over a given set A. (Admissible Sets form a Set Theory weaker than Zermelo-Fraenkel's in the principles of set existence; it was introduced in the mid 60s by Saul Kripke and Richard Platek with the goal of generalising ordinary recursion theory on the integers to ordinals smaller than a given "well-behaved" one.) As

most of the above results, so the Barwise-Gandy-Moschovakis Theorem was inspired by Mostowski's collapse lemma. While the papers [Fri73,Gor82,BGM71] make use of specific bisimulation-like relations, they do not isolate or study the concept, not do they introduce bisimilarity.

Earlier on, in Set Theory we find however constructions that have already a definite bisimulation flavor. A good example of this is Dimitry Mirimanoff's pioneering work on non-well-founded sets (e.g., the notion of set isomorphism introduced in [Mir17]).

Both in Computer Science, and in Philosophical Logic, and in Set Theory, bisimulation has roughly been derived through refinements of the notion of homomorphism between algebraic structures.

Bisimulation and bisimilarity are coinductive notions, and as such intimately related to fixed points. More details on the history of bisimulation, coinduction, and fixed points can be found in [San07].

References

[Acz88] P. Aczel. *Non-well-founded Sets*. CSLI lecture notes, no. 14, 1988.

[Ben76] J. van Benthem. *Modal Correspondence Theory*. PhD thesis, Mathematish Instituut & Instituut voor Grondslagenonderzoek, University of Amsterdam, 1976.

[BGM71] J. Barwise, R. O. Gandy, and Y. N. Moschovakis. The next admissible set. *J. Symb. Log.*, 36:108–120, 1971.

[FH83] M. Forti and F. Honsell. Set theory with free construction principles. *Annali Scuola Normale Superiore, Pisa, Serie IV*, X(3):493–522, 1983.

[Fri73] H. Friedman. The consistency of classical set theory relative to a set theory with intuitionistic logic. *J. Symb. Log.*, 38:315–319, 1973.

[Gor82] L. Gordeev. Constructive models for set theory with extensionality. In A.S. Troelstra and D. van Dalen, editors, *The L.E.J. Brouwer Centenary Symposium*, pages 123–147, 1982.

[Hin80] R. Hinnion. Contraction de structures et application à NFU. *Comptes Rendus Acad. des Sciences de Paris*, 290, Sér. A:677–680, 1980.

[Hin81] R. Hinnion. Extensional quotients of structures and applications to the study of the axiom of extensionality. *Bulletin de la Société Mathmatique de Belgique*, XXXIII (Fas. II, Sér. B):173–206, 1981.

[Hin86] R. Hinnion. Extensionality in Zermelo-Fraenkel set theory. *Zeitschr. Math. Logik und Grundlagen Math.*, 32:51–60, 1986.

[JT66] D.H.J. de Jongh and A.S. Troelstra. On the connection of partially ordered sets with some pseudo-boolean algebras. *Indagationes Mathematicae*, 28:317–329, 1966.

[Mil70] R. Milner. A formal notion of simulation between programs. Memo 14, Computers and Logic Resarch Group, University College of Swansea, U.K., 1970.

[Mil71] R. Milner. An algebraic definition of simulation between programs. In *Proc. 2nd Int. Joint Conferences on Artificial Intelligence*. British Comp. Soc. London, 1971.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mir17] D. Mirimanoff. Les antinomies de Russell et de Burali-Forti et le problème fondamental de la théorie des ensembles. *L'Enseignement Mathématique*, 19:37–52, 1917.

[Par81a] D. Park. Concurrency on automata and infinite sequences. In P. Deussen, editor, *Conf. on Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1981.

[Par81b] D. Park. A new equivalence notion for communicating systems. In G. Maurer, editor, *Bulletin EATCS*, volume 14, pages 78–80, 1981. Abstract of the talk presented at the Second Workshop on the Semantics of Programming Languages, Bad Honnef, March 16–20 1981. Abstracts collected in the Bulletin by B. Mayoh.

[San07] D. Sangiorgi. On the origins of bisimulation, coinduction, and fixed points. Draft, June 2007.

[Seg71] Krister Segerberg. An essay in classical modal logic. Filosofiska Studier, Uppsala, 1971.

Design Refinement

Neelam Soundarajan*
Computer Science & Engineering
Ohio State University, Columbus, OH 43210, USA

September 15, 2007

Refinement has been a central theme of software engineering for many years. Indeed, the field is often considered to have been born at about the same time as when the concept of procedural refinement, or stepwise refinement, was initially developed. The concept of data refinement, developed a few years later, extends the ideas underlying procedural refinement and provides a powerful set of techniques for use by software developers in designing and implementing ADTs as well as object-oriented (OO) systems[1]. Equally important, corresponding to each refinement concept, suitable reasoning techniques or refinement calculi have been developed that the software engineer can use to verify the correctness of the refinement steps he or she has performed in the design and implementation of a given system. The result has been a dramatic improvement in the quality and reliability of software developed using these techniques.

The thesis underlying the work that this talk is based on is that there is another type of refinement, *design refinement*, that corresponds to going from a set of *design patterns* [1, 2] to the design of a system or part thereof. Over the last decade and half, design patterns have indeed fundamentally altered the way we think about the design of large software systems. This is not surprising since the use of design patterns helps system designers exploit the collective wisdom and experience of the community as captured in the patterns. Patterns provide time-tested solutions to recurring problems, solutions that can be tailored –refined– to the needs of the individual system. But there has been relatively little effort to develop techniques that software designers can use to demonstrate that their systems, as implemented, are faithful to the underlying designs, or to check that, as the system evolves over time, that it does so in a manner that is consistent with the original design. Indeed, while patterns play a prominent role in the initial design of the system, their role tends to diminish quickly through the software lifecycle, from implementation to testing to maintenance and system evolution.

The primary reason for this is that patterns are commonly described in an informal, although stylized, manner in various pattern catalogs. While such descriptions are of great value, the ambiguity inherent in them means that different team members of a software team may have very –or worse, subtly– different interpretations of these descriptions which,

*Joint work with Jason Hallstrom, Jason Kirschenbaum, Ben Tyler and others

[1]Interestingly, some of these ideas seem to have been anticipated by the designers of Simula, the original OO language designed at the University of Oslo.

in turn, can lead to bugs in systems designed using the patterns. The main goal of our work then is to develop suitable reasoning techniques that software engineers can appeal to in order to verify that, in their system design, the patterns in question have indeed been used correctly. In the approach we have adopted, we propose a notation for specifying precisely, in the form of a *pattern contract*, the requirements that must be satisfied in using a given pattern P and the resulting behaviors that a system designed using P will exhibit; a notation for specifying, in the form of a *subcontract*, mappings from the system's components to the pattern contract's component showing precisely how P has been specialized or refined for use in this system; and a set of correctness requirements on the relation between the pattern contract and the system subcontract. Having a clear specification of the pattern's requirements, in the form of the pattern contract, and clear documentation, in the form of the subcontract, of how the pattern is specialized in the particular system will not only help the design/implementation team to avoid problems caused by ambiguous descriptions but also help the maintenance team avoid making changes that might compromise the *design integrity* of the system.

But there is also a potential risk in formalizing design patterns. The power of patterns arises in large part from their *flexibility*, i.e., from the ability of system designers to tailor the pattern to the needs of their particular systems. It would seem that formalizing patterns would eliminate, or considerably reduce, this flexibility. As it turns out, however, our approach, using a pattern contract to specify the requirements that must be satisfied by *all* applications of a given pattern and using a subcontract to specify how it is specialized in a *particular application*, allows us to preserve all of the pattern's flexibility. Indeed, it turns out that the very task of formalizing the pattern, i.e., developing its contract, often enables us to identify *additional* dimensions of flexibility that are not included, at least not explicitly, in standard informal descriptions of the pattern.

In this talk, I will summarize our work to date [4, 3] on developing a reasoning system for specifying pattern contracts and subcontracts for systems designed using patterns. We have also proposed [5] development of *runtime monitors* that can be used to check that the requirements of the pattern contracts and subcontracts are not violated during system execution; I will summarize our approach to such runtime monitoring. Finally, I will talk about our current efforts to extend the reasoning system to deal with complexities that arise in dealing with patterns in systems that involve complex interconnections, including cyclic references, among participating objects. Indeed, addressing these problems is essential since such interconnections are often *required* by many patterns.

References

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented software architecture: A system of patterns*. Wiley, 1996.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable OO Software*. Addison-Wesley, 1995.

[3] J.O. Hallstrom, N. Soundarajan, and B. Tyler. Amplifying the benefits of design patterns. In J. Aagedal and L. Baresi, editors, *The 9th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 214–229. Springer, 2006.

[4] N. Soundarajan and J.O. Hallstrom. Responsibilities and rewards: Specifying design patterns. In A. Finkelstein, J. Estublier, and D. Rosenblum, editors, *Proc. of 26th Int. Conf. on Software Engineering (ICSE)*, pages 666–675. IEEE Computer Society, 2004.

[5] B Tyler, J Hallstrom, and N Soundarajan. A comparative study of monitoring tools for pattern-centric behavior. In M Hinchey, editor, *Proc. of 30th IEEE/NASA Sofware Engineering Workshop (SEW-30)*. IEEE-Computer Society, 2006.

Validating QoS Properties in Biomedical Sensor Networks

Simon Tschirner and Wang Yi

Dept. of Information Technology, Uppsala University, Sweden.
Email: {simon.tschirner,yi}@it.uu.se

Abstract In this paper, we present a formal model of a Biomedical Sensor Network whose sensor nodes are constructed based on the IEEE 802.15.4 ZigBee standard for wireless communication. We have used the UPPAAL tool to tune and validate the temporal configuration parameters of the network in order to guarantee the desired QoS properties for a medical application scenario. The case study shows that even though the main feature of UPPAAL is model checking, it is also a promising and competitive tool for efficient simulation.

Background

A wireless sensor network is a wireless communication network containing normally a large number of sensor nodes distributed over a determined area. A sensor node contains basically two parts: a sensor for data collection and a transreceiver for wireless communication. The sensor of a node measures data in the environment with a certain period. Depending on the application, the sensor node may send the measured data immediately to a specified sink node or when it reaches a threshold value. Due to the limited size and energy supply, the range for wireless communication is highly bounded. Thus a message often has to be forwarded by a number of nodes to reach its destination.

A Biomedical sensor network (BSN) is a small area sensor network containing a relatively small number e.g. tens to hundreds of sensor nodes depending on the application. A typical application of BSN is to collect data on humen bodies e.g. in a hospital or an accident site. Fig. 1 illustrates an application scenario of BSN, provided by the national hospital of Norway for the EU Credo project [1]. For instance, in a situation when there are many injured persons, or the site is difficult to access, or the number of available medics at an accident site is limited, a quickly deployed BSN on the accident victims may be used to collect and transmit vital data to a centralized server for analysis so that proper and efficient medical operations can be carried out.

Due to the life-critical application, a BSN has to meet several *Quality of Service* (QoS) requirements. In this work, we study five classes of QoS properties imposed on the BSN in the above application scenario:

– *End-to-end delay*: This characterises the time from the transmission of a message at a node to the reception at the sink node.

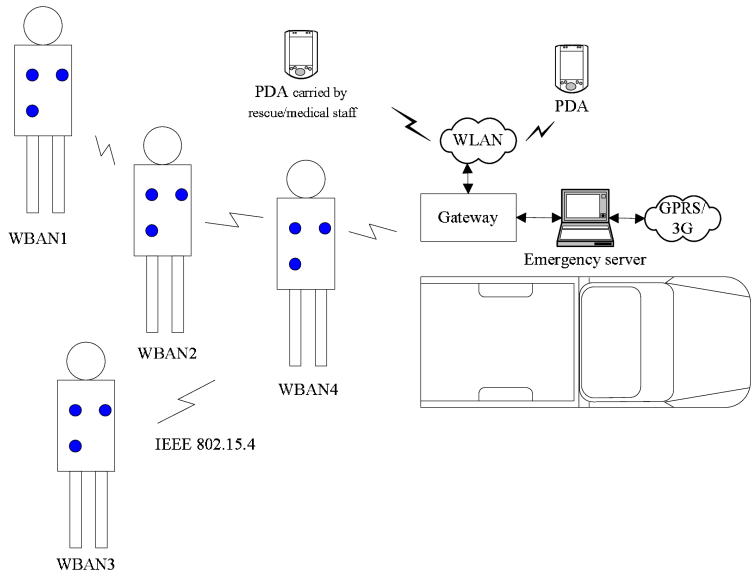


Figure1. Deployment of biomedical sensor networks at the site of an accident.[1]

- *Packet delivery ratio*: Packages in a sensor network can get lost easily, because a node may lose the connection to the sink node occasionally, and also packages may get corrupted e.g. when two nodes with overlapping ranges are transmitting simultaneously. To guarantee the service, the successful transmission of packages must be above a certain ratio.
- *Network throughput*: Some sensor data requires a certain bandwidth. If the network throughput is too low, the service can not be guaranteed.
- *Energy consumption*: The energy consumption of a sensor node has to be low in order to afford a long uptime of the network.
- *Service coverage area*: In general, this indicates the size of the area in which sensor data is available. In our scenario, the coverage area is important in that the sensor data from every injured person should be able to reach the sink.

The Technical Challenge

One separate sensor node does not have a complex behaviour. But as a BSN consists of a large number of nodes, the complexity of its behaviour is increasingly higher. For instance, the topology of the described BSN is very dynamic. The sensor nodes may move, disappear and new nodes may appear from time to time. Thus this kind network needs a suitable communication protocol to

guarantee the QoS properties. For this purpose, the IEEE 802.15.4[1] ZigBee[2] standard for wireless communication has been developed. It offers for instance different modes for communication and algorithms for signal routing if no direct connection to the sink exists. However, the specification of the standard covers only the logical behavioir of a sensor node in wireless communication. Temporal configuration parameters such as the transmission period and standby period of a node must be determined according to the application and the QoS requirements to be satisfied. For example, the application defines how often a given sensor node should transmit measured data and the necessary bandwidth. The duration a node spends in its sleeping (i.e. standby) mode or in a mode for package forwarding can also be carefully set to reduce energy consumption. The technical challenge is to tune and validate the temporal parameters in a BSN network to guarantee the desired QoS properties.

Modelling ZigBee-Based Networks

In the literature, the analysis of sensor networks is mainly based on simulation. There is little work on formal analysis of QoS properties of such networks. The goal of this is to validate the QoS properties of BSN's through formal modeling, simulation and verification with the UPPAAL tool. In particular, we want to explore the power of the UPPAAL symbolic simulator.

We study a BSN with sensor nodes based on the Chipcon CC240 transreceiver which offers wireless communication according to the IEEE 802.15.4 ZigBee standard. We use timed automata to model the radion control logic of the Chipcon CC240 transreceivers as described in [3]. The advantage of using timed automata is that the temporal parameters of the transreceivers can be expressed natuarlly as clock bounds.

To establish a model of the whole network, we also need to model the topology of the network as well as the dynamic changes or reconfigurations of the network topology. For this purpose, we use a matrix to model the routing table provided by the routing layer. The matrix contains the delay for package transmission from one node to another within the transmision area. A package transmitted by a node can be spread to the neighbouring nodes according to this matrix. Nodes that are not connected or in a state that does not allow package reception are marked with reserved values. Thus the matrix is a representation of the network topology. The dynamic reconfigurations of the network are modeled by transitions of individual sensor nodes that may update the matrix e.g. when a node changes its state from idling to sleeping or its position from one transmission area to another.

[1] http://ieee802.org/15/pub/TG4.html
[2] http://www.zigbee.org/

Validating QoS Properties

For networks of moderate size e.g. with tens of nodes, we are able to use the UPPAAL model checker to check safety properties such as deadlock freeness, and QoS properties such as network connectivity. Unfortunately, the UPPAAL querry language is not expressive enough to formalize all the listed QoS properties even though using testing automata and the annotation technique [2], we are able to encode a large class of QoS properties.

The main obstacle of using a model checker in this context is the limited scalability of the technique with respects to the number of nodes in a large network. This lead us to use the symbolic simulator of UPPAAL. Using meta variables in UPPAAL, we are able to collect statistical information about the simulated behaviour of a network. Our experiments show that the symbolic simulation technique scales very well with the network size. We can easily simulate a complete network with hundreds of nodes, and validate all the desired QoS properties of interests in the applciation scenario as listed below:

- *End-to-end delay*: The simulation may provide the average end-to-end delay between two nodes. For example, this is useful for the analysis of routing protocols.
- *Packet delivery ratio*: This kind of statistical data can be obtained with UPPAAL meta-variables. For example, we may count the number of packages sent and received by the sink node, using the meta-variables.
- *Network throughput*: It is similar to the end-to-end delay. For example, the elements in the matrix modeling the network topology can be set and interpreted as the bandwidth between two nodes. While for the calculation of the end-to-end delay the particular values have to be summed up, we need to calculate the minimum in this case.
- *Energy consumption*: For each location or mode in which a node can be, we can count the time it remains there. These times can be mutliplied with specified factors for the energy consumption in that particular state.
- *Service coverage area*: As explained, it is enough to validate that each node can be connected to the sink – at least after a bounded time has past. This property can also be verified in our model.

Note that simulation can – in contrast to verification – not guarantee that a QoS requirement is met definitely. However, we may use the technique to validate and debug the design parameters of a network. In this case study, we have also experienced that even though the main feature of UPPAAL is model checking, it is also a promising and competitive tool for efficient simulation.

References

1. Xuedong Liang, Bjarte M. Østvold, Wolfgang Leister, and Ilangko Balasingham. Credo, case study 2: Biomedical sensor networks, March 2007.

2. Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal Design and Analysis of a Gearbox Controller. *Springer International Journal of Software Tools for Technology Transfer (STTT)*, 3(3):353–368, 2001.
3. Texas Instruments. *2.4 GHz IEEE 802.15.4 / ZigBee-Ready RF Transceiver (Rev. B)*, March 2007.

Task Scheduling in Rebeca

Mohammad Mahdi Jaghoori · Frank S. de Boer · Marjan Sirjani

1 Introduction

Rebeca [3] (reactive objects language) is an actor [2] based language with formal semantics, which can be used at a high level of abstraction for modeling concurrent and distributed reactive systems. Reactive objects (called rebecs) run in parallel and can communicate by asynchronous message passing. Rebecs have no explicit receive statement; instead, incoming messages are queued. A rebec has a message server for each message it can handle. A message sever (also called a method) is defined as a piece of sequential code, which, among others, may include sending messages.

All rebecs must implement a message server '`initial`'. At creation, a rebec has the '`initial`' message in its queue. At each step, each rebec executes (the message server corresponding to) the message at the head of the queue and then removes it from queue (i.e., there is no intra-object concurrency). In this paper, we allow rebecs to define their own scheduling policies (which has been traditionally FIFO in Rebeca). The scheduling policy of each rebec, upon receiving a message, determines where in the queue the message should sit; however, it cannot preempt the currently running method.

Task automata [1] is a new approach for modeling real time systems with non-uniformly recurring computation tasks; where tasks are generated (or triggered) by timed events. Tasks, in this model, are represented by a triple (b, w, d), where b and w are, respectively, the best-case and worst-case execution times, and d is the deadline. A task automaton is said to be schedulable if there exists a scheduling strategy such that all possible sequences of events generated by the automaton are schedulable in the sense that all associated tasks can be computed within their deadlines. It is shown in [1] that, among other cases, with a non-preemptive scheduling strategy, the problem of checking schedulability for task automata is decidable.

In this paper, we add real time constraints to Rebeca and present a compositional approach based on task automata for schedulability analysis of timed Rebeca models. In this approach, instead of just best-case and worst-case execution times, the behavior of each task is given (in terms of timed automata) and used in the schedulability analysis. These timed automata may in turn generate new tasks. Task automata, as introduced in [1], cannot model tasks generated *during* the execution of another task.

2 The Timed Rebeca Model

For each rebec, the message servers are modeled as timed automata, in which actions may include sending messages, either to the same rebec, called *self calls*, or to other rebecs. Since message servers always terminate, every execution of the corresponding automata also stops at a state with no outgoing transition. The modeler also gives an abstract behavior of the environment for each rebec in terms of a timed automaton (called the *driver* automaton). The driver automaton models the (expected) timings for arrival of messages to the rebec, together with their deadlines. The driver automaton is similar to task automata in the sense that receiving a message corresponds to generating a new task.

M. M. Jaghoori · F.S. de Boer
CWI, Amsterdam, The Netherlands; E-mail: {jaghouri, f.s.de.boer}@cwi.nl

M. Sirjani
University of Tehran and IPM, Tehran, Iran; E-mail: msirjani@ut.ac.ir

$$((s,s'),[B_1,\ldots,B_l]) \xrightarrow[c\ ,\ X]{?m(d)}_H ((u,s'),sched([B_1,\ldots,B_l],m(d))) \quad [recieve]$$
$$if\ s \xrightarrow[c\ ,\ X]{?m(d)}_T u$$

$$((s,s'),[B_1,\ldots,B_l]) \xrightarrow[c\ ,\ X]{m(d)}_H ((s,u),sched([B_1,\ldots,B_l],m(d))) \quad [self-call]$$
$$if\ (s' \xrightarrow[c\ ,\ X]{!m(d)}_{B_1} u)\ and\ (m \in M)$$

$$((s,s'),[B_1,\ldots,B_l]) \xrightarrow[c\ ,\ X]{!m(d)}_H ((s,u),[B_1,\ldots,B_l]) \quad [send]$$
$$if\ (s' \xrightarrow[c\ ,\ X]{!m(d)}_{B_1} u)\ and\ (m \notin M)$$

$$((s,s'),[B_1,\ldots,B_l]) \rightarrow_H error \quad [overflow]$$
$$if\ (l=q)\ and\ ((s \xrightarrow[c\ ,\ X]{?m(d)}_T u)\ or\ (s' \xrightarrow[c\ ,\ X]{!m(d)}_{B_1} u\ and\ m \in M))$$

$$((s,s'),[B_1,B_2,\ldots,B_l]) \rightarrow_H ((s,start(B_2)),[B_2,\ldots,B_l]) \quad [context-switch]$$
$$if\ s' \in final(B_1)$$

Fig. 1 Calculating the edges of the behavior automata

A timed automaton is identified by a finite set of locations N (including an initial location n_0); a set of actions Σ; a set of clocks C; location invariants $I : N \rightarrow \mathcal{B}(C)$; and, the set of edges $\rightarrow \subseteq N \times \mathcal{B}(C) \times \Sigma \times 2^C \times N$, where $\mathcal{B}(C)$ is the set of all clock constraints. An edge written as $s \xrightarrow[c\ ,\ X]{a} s'$ means that action a may change state s to s' by resetting the clocks in X, if clock constraints in c hold. In the sequel, assume that the sets of messages handled by different rebecs are disjoint and their union is $\mathcal{M}$.

Definition 1 A rebec R is formally defined as $[(m_1 : A_1, \ldots, m_n : A_n), T, \mathcal{C}]$, where

- $M = \{m_1, \ldots, m_n\} \subseteq \mathcal{M}$ is the set of messages handled by R;
- $A_i = (N_i, \rightarrow_{A_i}, \Sigma, C_i, I_i, n_{0_i})$ is a timed automaton representing the message server handling m_i.
- $T = (N_T, \rightarrow_T, \Sigma_T, C_T, I_T, n_T)$ is a timed automaton modeling the rebec's environment (the driver).
- $\mathcal{C}$ is a set of clocks shared by all A_i and T (called the global clocks).

The action set of A_i is defined to be $\Sigma = \{!m | m \in M\} \cup \{!m(d) | m \in \mathcal{M} \wedge d \in I\!N\}$; and, the action set of the driver automaton is $\Sigma_T = \{?m(d) | m \in M \wedge d \in I\!N\}$. Intuitively, the driver automaton is similar to task automata in the sense that executing an action in the driver (i.e., receiving a message from another rebec) creates a new task. However, a rebec may send messages to itself (self calls), which also result in new (internal) tasks being generated. According to the definition of Σ, internal tasks are not necessarily assigned deadlines. Internal tasks without an explicit deadline (called *delegation*) inherit the (remaining) deadline of the task that generates them (parent task).

Delegation implies that the internal task (say t') is in fact the continuation of the parent task (say t). Notice that unconstrained loops in delegations result in nonschedulability, because deadline becomes smaller every time. To bound delegation loops, one can use the global clocks $\mathcal{C}$. A common scenario for delegation happens when a task t creates an instance of t' to continue the computation, after another task (say y) is executed. In such cases, if t' is scheduled before y is executed, it would need to create another instance of itself (t'). This results in a loop in calling t'.

As mentioned above, the driver automaton has the same syntax as a task automata, but it models only the messages sent by other rebecs (does not include internal tasks). Therefore, analyzing the driver alone is not enough for determining schedulability of the rebec. Instead, schedulability analysis should be performed on the automaton obtained by executing the abstract behavior of the message servers as controlled by the driver automaton.

Definition 2 (Behavior Automaton) The behavior automaton for a rebec R (cf. Definition 1) is a timed automaton $H = (S_H, \rightarrow_H, \Sigma_H, C_H, I_H, s_H)$ where

- $S_H = error \cup \big(N_T \times (\bigcup_{i \in [1..n]} N_i) \times (M \cup \{emp\})^q\big)$, where N_T and N_i are the sets of locations of T and A_i, respectively, and q is a statically computable bound on the length of schedulable queues.

- $\Sigma_H = \{!m(d) | m \notin M\} \cup \{?m(d) | m \in M\} \cup \{m(d) | m \in M\}$, where $d \in \mathbb{N}$ denotes the deadline.
- $C_H = C_T \cup \big(\bigcup_{i \in [1..n]} C_i\big) \cup \mathcal{C}$, where C_i and C_T are the sets of clocks for A_i and T, respectively.
- For each state $u = (s_T, s, Q)$, if $s \in N_i$ then $I_H(u) = I_T(s_T) \wedge I_i(s)$.
- The initial state s_H is $\big((n_T, start(A_1)), [A_1]\big)$, where n_T is the initial location of T; and, A_1 is the automaton corresponding to the '`initial`' message server.
- The edges $\rightarrow_H$ are defined with the rules in Figure 1.

In Figure 1, functions $start(A)$ and $final(A)$, respectively, give the initial location of A, and the set of locations in A with no outgoing transitions. Function *sched* puts the given message in the queue based on the scheduling policy of rebec R. Each state of the behavior automaton is written as $((s, s'), [B_1, \ldots, B_l])$, where $B_1, \ldots, B_l$ show the automata corresponding to the messages in the queue (empty queue elements are not written); s shows the current state in the driver; and, s' shows the current state in B_1. Notice that self calls are modeled as *internal* actions, while send and receive operations to/from other rebecs are *visible* actions. As discussed in the next section, sends and receives of different rebecs must match.

Assume that b_{min} is the smallest best-case execution time of the automata A_i representing the message servers in R; and, d_{max} is the longest deadline for the tasks that may be triggered on R. In Definition 2, one can statically compute $q = d_{max}/b_{min}$, as the bound on the length of schedulable queues. It means that the behavior automaton for each rebec is finite state and computable.

The schedulability analysis can be performed in a way similar to task automata. Schedulability can be verified by resetting a fresh clock (say x_i) whenever a new task (with deadline d_i) in scheduled into the queue. From every state, if $x_i \geq d_i$ for some task in the queue, the behavior automata should move to the *error* state. Consequently, the schedulability problem reduces to the reachability of the *error* state.

As a timed automaton, the semantics of the behavior automaton can be defined in terms of a timed transition system. The states of this transition system are pairs (S_H, u) where S_H is a location of the behavior automata and u is a clock assignment. Considering the delay transitions, the semantics of the behavior automaton is related to the semantics of the automata in the definition of a rebec:

$$((s_1, s_2), [B_1, \ldots, B_l], u) \xrightarrow{\delta} ((s_1', s_2'), [B_1, \ldots, B_l], u + \delta) \text{ iff } \begin{cases} (s_1, u_T) \xrightarrow{\delta} (s_1', u_T + \delta); \text{ and,} \\ (s_2, u_B) \xrightarrow{\delta} (s_2', u_B + \delta) \end{cases}$$

where, $((s_1, s_2), [B_1, \ldots, B_m])$ is a state of the behavior automaton; u_T and u_B represent the projection of u on the clocks of T and B_1, respectively; and, $\delta \in \mathbb{R}_+$ is a positive real valued number.

3 Compatibility checking

After performing schedulability analysis for each rebec separately, one should check if the driver automaton for each rebec correctly models the messages sent to that rebec. Notice that due to the schedulability of all rebecs, an action $?m(d)$ implies that m can be finished within d time units. Therefore, an action $!m(d')$ (requiring that m should finish within d' time units) can match $?m(d)$ only if $d \leq d'$.

To check the compatibility of the driver automata with the definition of the rebecs in the model, one can compute the synchronous product of the behavior automata of all rebecs. When computing the synchronous product of these automata, $?m(d)$ and $!m(d')$ can synchronize and become an internal action only if $d \leq d'$ (besides matching the timing constraints). The behavior automata of all rebecs are compatible if every send action can be matched by a corresponding receive.

Before computing the synchronous product, the information in the states of the behavior automata (the contents of the queue, etc.) can be abstracted away. Different internal actions (of the general form $m(d)$) can also be treated as one internal action τ.

References

1. Elena Fersman, Pavel Krcal, Paul Pettersson, and Wang Yi. Task automata: Schedulability, decidability and undecidability. 2007. Accepted for publication in Information and Computation (to appear).
2. Carl Hewitt. Procedural embedding of knowledge in planner. In *Proc. the 2nd International Joint Conference on Artificial Intelligence*, pages 167–184, 1971.
3. Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S. de Boer. Modeling and verification of reactive systems using Rebeca. *Fundamamenta Informaticae*, 63(4):385–410, 2004.

A Tool for Compositional Analysis of Timed Systems by Abstraction

Pavel Krčál, Leonid Mokrushin and Wang Yi

Dept. of Information Technology, Uppsala University, Sweden.
Email: {pavelk,leom,yi}@it.uu.se.

Abstract We present a tool for compositional timing and performance analysis of real-time systems modeled using timed automata and the real-time calculus [5]. It is based on an (over-) approximation technique in which a timed automaton is abstracted as a transducer of abstract streams described by arrival curves from network calculus [2]. As the main feature, the tool can be used to check the schedulability of a system and to estimate the best and worst case response times of its computation tasks. The tool is available for evaluation at www.timestool.com/cats.

1 Introduction

Real-time systems are often constructed based on a set of real-time tasks. These tasks may be scheduled and executed according to given release patterns. There have been a number of methods and tools developed for timing analysis to estimate the worst case (and also the best case) response times of computation tasks for such systems, e.g., Rate-Monotonic Analysis [4] for periodic tasks, Real-Time Calculus (RTC) [5] for tasks described using arrival curves [2], and Times [3] using Timed Automata (TA) [1]. Some of these techniques, e.g., implemented in Times can deal with systems with complex release patterns, but do not scale well with system size and complexity; the others are scalable but can not handle systems with complex structures. Our goal is to take the advantages of these existing techniques and develop a tool, that is scalable and also capable of handling complex systems.

We model the architecture of a system using data-flow networks in the style of the RTC, where the nodes stand for the building blocks or components of the system and edges for the communication links between the nodes. In the RTC, nodes represent either tasks or functions on arrival and service curves. To enhance the expressiveness of the task model, we also allow TA nodes as release patterns. The essential idea of our analysis technique is to abstract the TA nodes using arrival curves, which can be done modularly for each node, and to compose the analysis results in order to perform the system level analysis.

2 The Model

The basic concepts of our model are tasks, task arrival patterns and computational resources. Tasks are abstractions of programs that execute on a processing

unit and thus consume computational resources. The parameters of a task are its best and worst case execution times on a reference processor. A task arrival pattern describes the moments in time at which tasks are released for execution. The execution of a task is scheduled on a processing unit according to a preemptive fixed priority scheduling strategy. The capacity and availability of a processing unit form the model of a computational resource. We use arrival curves [5] and TA [1] to model task arrival patterns, and service curves [5] to model computational resources.

We shall introduce a notion of an abstract stream as a set of non-decreasing diverging sequences of timestamps ranging over positive reals. Each timestamp denotes an occurrence of an event. An abstract stream defined by a pair of upper and lower arrival curves is the greatest abstract stream such that all the sequences of events of this abstract stream comply with the constraints induced by the pair of arrival curves as in [5]. A timestamp with a name assigned to it is called an action. We define a timed trace to be a sequence of actions with non-decreasing diverging timestamps. A set of timed traces forms a timed language.

A timed language where action names are taken from a bounded (by the resource capacity) subset of non-negative rational numbers is called an abstract resource. These numbers represent the amount of computational resources in reference processor units until the next action. An abstract resource defined by a pair of upper and lower service curves is the greatest abstract resource such that all the sequences of actions of this abstract resource comply with the constraints induced by the pair of service curves as in [5].

The model analysed by the tool is a finite network of nodes interconnected with links. Nodes may have ports and links are directed edges connecting ports of the nodes. Each port has three parameters: name, direction (input or output) and type (*event* or *resource*). We distinguish the following types of nodes:

- **Task node**, a node with a task assigned to it; it has two input ports: *release* (an arrival pattern of the task) and *demand* (computational resource available for the task execution) and two output ports: *finish* and *rest* representing the pattern of task finishing times and the remaining computational resource respectively; the ports *release* and *finish* are of the *event* type whereas *demand* and *rest* are ports of the *resource* type,
- **Task arrival pattern node**, a node with either a pair of upper and lower arrival curves or a TA assigned to it; in the first case the node has only one output port and no input ports, and in the second case – the input and output ports corresponds to the input and output letters of the TA; all the ports are of the *event* type,
- **Resource node**, a node with one output port of the *resource* type and a pair of upper and lower service curves assigned to it,
- **Function node**, a node containing a function of the real-time calculus [6]; the input ports correspond to the parameters of the function and there is only one output port; all the ports have the same type – *event* or *resource*.

The links between the node ports must always connect an output port to an input port and loops are not allowed. Moreover, it is only possible to connect

ports of the same type. Intuitively, the links model the control flow and the flow of computational resources. Task priorities are defined by the order of the task nodes in the flow of computational resources.

Semantically, every time an event arrives to the *release* port of a task node, the task associated with the node is released for execution. During its execution a task consumes computational resources entering the *demand* port of a task node. The task completes its execution after being computed for a period of time between the best and the worst case times specified in task parameters and issues an event to the *finish* port. The resources not used by the task are passed through to the *rest* port. Task arrival pattern nodes and resource nodes with assigned pair of curves generate events triggering task releases and computational resources respectively. Task arrival pattern nodes with an associated TA transform input abstract streams into the output abstract streams as follows. First, the input abstract streams are converted into a timed language. Then, the TA interpreted as a transducer computes the output timed language. Finally, this language is approximated by an abstract stream for each output port of the node. Function nodes transform abstract streams or resources from incoming ports according to the real-time calculus functions assigned to them.

3 Tool Architecture and Features

The tool implementation (as shown in Fig. 1) consists of two parts: model construction and model analysis. The first part contains internal system model representation and the editors, which allow to define the topology of a system, the timed automata assigned to TA nodes, the functions assigned to the RTC nodes, and pairs of arrival and service curves.

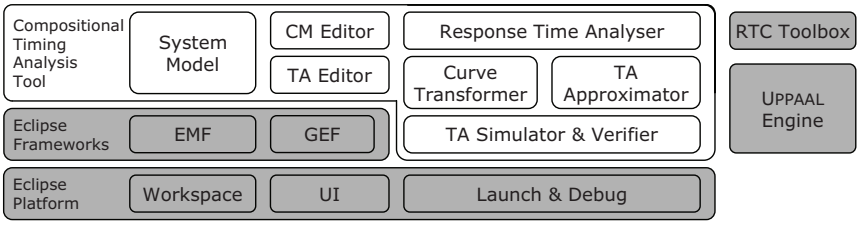


Figure1. The architecture of the tool.

The tool computes the best and the worst case response times of every task for a given task release pattern and a set of computational resources. It does this by analysing abstract streams and resources appearing on the links of the model network. Depending on the type of the node the tool assembles an appropriate operation at the evaluation time. For example, for a TA node an evaluation operation consists of the encoding of the input abstract streams into TA, computing

the output of the TA node using UPPAAL verification engine, and decoding the results back into the form of the abstract streams.

The screenshot of the tool is shown in Fig. 2. The tool is implemented as a set of plugins built on top of the Eclipse Development Platform. The implementations of the internal models are based on the Eclipse Modeling Framework. A script language is used for specification of the model together with a dedicated text editor. The graphical editor assists designers in TA modelling and is built on top of the Eclipse Graphical Editing Framework. The runtime part of the tool extends Eclipse Launch&Debug functionality and provides a set of specialized views for monitoring and interpreting the results. We use Real-Time Calculus Toolbox [6] to evaluate RTC functions.

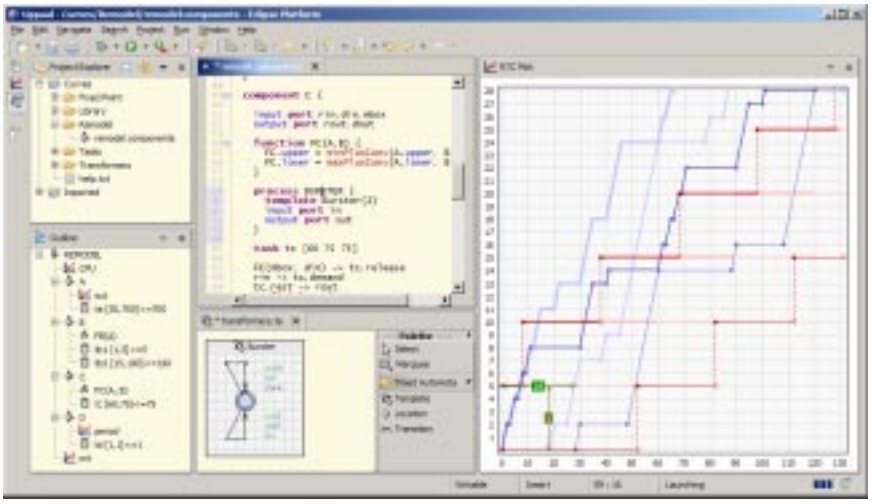


Figure2. The screenshot of the tool.

References

1. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
2. R. Cruz. A calculus for network delay. *Proc. of IEEE Trans. Information Theory*, 37(1):114–141, 1991.
3. E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Schedulability analysis of fixed-priority systems using timed automata. *Theoretical Computer Science*, 354:301–317, March 2006.
4. M. Joseph and P. Pandya. Finding response times in a real-time system. *BSC Computer Journal*, 29(5):390–395, October 1986.
5. L. Thiele, S. Chakraborty, M. Gries, A. Maxiaguine, and J. Greutert. Embedded software in network processors - models and algorithms. In *Proc. of EMSOFT'01*, pages 416–434. Springer-Verlag, 2001.
6. E. Wandeler and L. Thiele. Real-Time Calculus (RTC) Toolbox. `http://www.mpa.ethz.ch/Rtctoolbox`, 2006.

Abstract interface behavior of an object-oriented language with futures and promises

3. September 2007

Erika Ábrahám[1], and Immo Grabe[2], and Andreas Grüner[2], and Martin Steffen[3]

[1] Albert-Ludwigs-University Freiburg, Germany
[2] Christian-Albrechts-University Kiel, Germany
[3] University of Oslo, Norway

1 Motivation

How to marry concurrency and object-orientation has been a long-standing issue; see e.g., [2] for an early discussion of different design choices. Recently, the thread-based model of concurrency, prominently represented by languages like *Java* and *C#* has been criticized, especially in the context of *component-based* software development. As the word indicates, components are (software) artifacts intended for composition, i.e., *open* systems, interacting with a surrounding environment. To compare different concurrency models on a solid mathematical basis, a semantical description of the interface behavior is needed, and this is what we do in this work. We present an *open semantics* for a core of the *Creol* language [4,7], an object-oriented, concurrent language, featuring in particular asynchronous method calls and (since recently [5]) *future*-based concurrency.

Futures and promises A *future,* very generally, represents a result yet to be computed. It acts as a proxy for, or reference to, the delayed result from a given sequential piece of code (e.g., a method or a function body in an object-oriented, resp. a functional setting). As the client of the delayed result can proceed its own execution until it actually *needs* the result, futures provide a natural, lightweight, and (in a functional setting) transparent mechanism to introduce parallelism into a language. Since its introduction in *Multilisp* [6][3], futures have been used in various languages, including Alice ML, E, the ASP-calculus, Creol, and others more. A *promise* is a generalization insofar that the reference to the result on the one hand, and the code responsible to calculate the result on the other, are not created at the same time; instead, a promise can be created independently and only later, after possibly passing it around, the promise is bound to the code (one also says, the promise is *fulfilled*).

Interface behavior An *open* program interacts with its environment via message exchange. The interface behaviour of such an open program C can be characterized by the set of all those message sequences (= traces) t, for which there *exists* an environment E such that C and E can exchange the messages recorded in t. Thus the interface behaviour abstracts away of any concrete environment. However, it only considers such environments, that are compliant to the language restrictions (syntax, type system, etc.).

Consequently, interactions do not consist of arbitrary message sequences $C \stackrel{t}{\Longrightarrow}$; instead we consider the behavior $C \parallel E \stackrel{t}{\underset{\bar{t}}{\Longrightarrow}} \acute{C} \parallel \acute{E}$ where E is an *arbitrary* but *realizable* environment and $\bar{t}$ complementary to t.

To account for the existentially abstracted environment ("there exists an E s.t. ..."), the open semantics is given in an *assumption-commitment* way:

$$\Delta \vdash C : \Theta \stackrel{t}{\Longrightarrow} \acute{\Delta} \vdash \acute{C} : \acute{\Theta}$$

where Δ contains (as an abstract version of E) the *assumptions* about the environment, and dually Θ the *commitments* of the component. Abstracting away also from C gives a language characterization by the set of all possible traces between any component and any environment.

Such a behavioral interface description is relevant and useful for the following reasons. 1) The set of possible traces is more restricted than the one obtained when ignoring the environments. I.e., when *reasoning* about the trace-based behavior of a component, e.g., in compositional verification, with more precise characterization one can carry out stronger arguments. 2) When using the trace description for black-box testing, one can describe test cases in terms of the interface traces and then synthesize appropriate test drivers from it. Clearly, it makes no sense to specify impossible interface behavior, as in this case one cannot generate a corresponding tester. 3) A representation-independent behavior of open programs paves the way for a compositional semantics and allows furthermore optimization of components: only if two components show the same external behavior, one can replace one for the other without changing the interaction with any environment. 4) The formulation gives *insight* into the semantical nature of the language, here, the observable consequences of futures and promises. This helps to compare alternatives, for instance the Creol concurrency model with Java-like threading.

2 Results

We formalize the abstract interface behavior for concurrent object-oriented class-based languages with futures and promises. The long version of the submission includes the following results:

Concurrent object calculus with futures and promises We formalize a class-based concurrent language featuring futures and promises, capturing the core aspects of the *Creol*-language. The formalization is given as a typed, imperative object calculus in the style of [1] resp. one of its concurrent extensions. We present the semantics in a way that facilitates comparison with *Java*'s multi-threading concurrency model, i.e., the operational semantics is formulated so that the multi-threaded concurrency as (for instance) in *Java* and the one based on futures here are represented similarly.

Linear type system for promises Featuring promises, the calculus extends the semantic basis of *Creol* as given for example in [5] (only futures). Promises can refer to a computation with code bound to it later. It is important, that the binding is done at most

once. To guarantee such a *write-once* policy when passing around promises, we refine the type system introducing two type constructors

$$[T]^{+-} \quad \text{and} \quad [T]^{+}$$

representing a reference to a promise that can still be written (and read, and with result type T), resp. a reference with read-only permission. The write-permission constitutes a resource which is consumed when the promise is fulfilled. The resource-aware type system is therefore formulated in a *linear* manner wrt. the write permissions and resembles in intention the one in [8] for a functional calculus with references. Our work is more general, in that it tackles the problem in an object-oriented setting (which, however, conceptually does not pose much complications). It is in addition more general in that we do not give a type system for a closed system, but for an open component. Also this aspect of openness is not dealt with in [5]. Additionally, the type system presented here is simpler as the one in [8], as it avoids the representation of the promise-concept by so-called *handled futures.*

Soundness of the abstractions We show soundness of the abstractions, which includes

- *subject reduction,* i.e., preservation of well-typedness under reduction. Subject reduction is not just proven for a closed system (as usual), but for an open program interacting with its environment. Subject reduction implies
- *absence of run-time errors* such as "message-not-understood", again also for open systems.
- A proof that the characterization of the interface behavior is *sound,* i.e., all interaction behavior which is possible by an actual, concrete environment is included in the abstract interface behavior description.
- for promises: absence of *write-errors,* i.e. the attempt to fulfill a promise twice.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.
2. P. America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.
3. H. Baker and C. Hewitt. The incremental garbage collection of processes. *ACM Sigplan Notices*, 12:55–59, 1977.
4. The Creol language. `http:www.ifi.uio.no/~creol`, 2007.
5. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *Proc. 18th European Symposium on Programming (ESOP'07)*, 2007. To appear in Springer's LNCS series.
6. R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, Oct. 1985.
7. E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, Nov. 2006.
8. J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda-calculus with futures. *Theoretical Computer Science*, 2006. Preprint submitted to TCS.

What is a sort function?

Fritz Henglein
Department of Computer Science, University of Copenhagen (DIKU)
henglein@diku.dk

September 22nd, 2007

Total preorders Sorting is one of the most-studied subjects of computer science. So it seems silly to ask what a sort function is.[1] Obviously a sort function is a function that permutes sequences such that the elements are in order. This, however, begs the next question: What does it mean to be "in order"? An obvious answer is that the output must respect the ordering relation of a *given* total order.[2] So this suggests the following definition: Given total order $(S, \leq_S)$, a sort function is a permutation function $sort : S^* \to S^*$ such that if $y_1 \dots y_n = sort(x_1 \dots x_n)$ then $y_i \leq_S y_{i+1}$ for $1 \leq i \leq n$. We can quickly see, however, that expecting the sort function to output according to a *total order* is too much to expect of it; for example when sorting tuples "according to their i-th component" as a subroutine in radix-sort, this step permutes the whole tuples (records), but according to a total order on a particular component (key), not the whole tuple. Expecting keys to be totally ordered is still too much: If the keys are strings we may want to sort the records according to their keys, but *ignoring* their case. This means that the output may contain both $["fReD", 15), ("FrEd", 18)]$ and $[("FrEd", 18), ("fReD", 15)]$, even though this is clearly incompatible with requiring the key domain to be totally ordered. It is the anti-symmetry requirement of total orders that is too strong. Since reflexivity, transitivity and totality seem reasonable requirements, we drop antisymmetry:

A *total preorder* or (here) just *order* (S, R) is a set S together with a binary relation $R \subseteq S \times S$ that is *reflexive*, *transitive*, and *total* $(\forall x, y \in S : (x, y) \in R \vee (y, x) \in R)$. We say R is an *ordering relation* on S and (S, R) is an *order on S*.

Note that an order (S, R) canonically induces an equivalence relation $(S, \cong_R)$: $x \cong_R y \Leftrightarrow R(x, y) \wedge R(y, x)$.

We can now define a sort function to be a function on an order that permutes its input such that the output respects the ordering relation:

A function $sort : S^* \to S^*$ is a *sort function for order* (S, R) if for all $x_1 \dots x_n \in S^*$ and $y_1 \dots y_m = sort(x_1 \dots x_n)$ we have: $y_1 \dots y_m$ is a permutation of $x_1 \dots x_n$ (and so $n = m$); and $y_1 \dots y_m$ is R-ordered: $R(y_i, y_{i+1})$ for all $1 \leq i \leq m$.

Comparators Now we know what a sort function is for a *given* order. But that was not the question: there was no mention of a given order. So what if somebody hands you a function and claims it is a sort function—without giving you an order?

An obvious answer is: It has to be a sort function for *some* order. But then follow-up questions beg themselves: Is there any order at all? If so, what if there are several? Does it *define* an order in some sense? The answer to the first question is trivial: Each permutation function $f : S^* \to S^*$ is a sort function for the trivial order $(S, S \times S)$, which relates all elements to each other. It is least informative, however, since it is least discriminative

[1] The use of *function* is intended to convey that we are considering the mathematical transformation on the data, whether executed in-place, out-of-place and independent of data structure representation.

[2] Since "sorting" in ordinary usage only implies collecting related—equivalent— elements, essentially *partitioning*, a more accurate term would have been "ordering" for "sorting". See Knuth [Knu98] for a humorous discussion of this. Note also the use in this sense more accurate usage of *sort* in multi-sorted alegebra.

amongst candidate orders for which f is a sort function: all elements of S are equivalent to each other under ordering relation $S \times S$.

Consider f applied to two arguments: If $f[x_1, x_2] = [x_1, x_2]$ we can conclude that $x_1 \leq x_2$ for any ordering relation $\leq$ *consistent* with f as a sort function. If $f[x_1, x_2] = [x_2, x_1]$ we can conclude $x_2 \leq x_1$, but we do not know whether or not $x_1 \leq x_2$ holds. Clearly, however, the unique *smallest* and thus most informative *possible* ordering relation $\leq^{min}$ consistent with f is the one that lets us conclude $x_1 \not\leq^{min} x_2$ whenever $f[x_1, x_2] = [x_2, x_1]$ or, equivalently, $x_1 \leq^{min} x_2 \Leftrightarrow f[x_1, x_2] = [x_1, x_2]$, but only *if* this defines an *ordering* relation. It turns out that it does if and only if f as a function of two arguments is *permutative*, *transitive* and *idempotent*.

A *comparator structure* $(S, comp)$ is a set S together with a function $comp : S \times S \to S \times S$ that is *permutative*: $comp(x, y) = (x, y) \vee comp(x, y) = (y, x)$; *transitive*: $comp(x, y) = (x, y) \wedge comp(y, z) = (y, z) \Longrightarrow comp(x, z) = (x, z)$; and *idempotent*: $comp(x, y) = (y, x) \Longrightarrow comp(y, x) = (y, x)$ for all $x, y, z \in S$. We call *comp* a *comparator (function)* on S.

Theorem 1: Consider $f : S \times S \to S \times S$ and define binary relation R on S by $R(x_1, x_2) \Leftrightarrow f(x_1, x_2) = (x_1, x_2)$. Then R is an ordering relation on S if and only if f is a comparator on S.

Sort functions A sort function $sort : S^* \to S^*$ for order (S, R) is *stable* if R-equivalent elements occur in the same order in the output as in the input: $sort(\vec{x})|_{[z]_{\cong_R}} = \vec{x}|_{[z]_{\cong_R}}$ for all $z \in S$ and $\vec{x} \in S^*$, where $[z]_{\cong_R}$ denotes the set of R-equivalent S-elements of z.[3] Each order has exactly one stable sort function since stability fixes the order in which equivalent elements must be output.

Note that stipulating that the ordering relation we are after satisfy $x_1 \leq x_2 \Leftrightarrow f[x_1, x_2] = [x_1, x_2]$, as we did for comparators, is tantamount to insisting that f be a *stable* sort function for that order, at least when applied to two elements. So insisting that one's proclaimed sort function is *stable* (for arbitrary length inputs) is a way of identifying the most informative order consistent with f. It is easy to see that, if f is a stable sort function for any ordering relation at all, then that ordering relation is unique. As it turns out, f is a stable sort function for some order if and only if it is *consistently permutative*.

Theorem 2: Let $f : S^* \to S^*$. There exists an ordering relation R on S such that f is a stable sort function for (S, R) if and only if f is *consistently permutative*: For each sequence $x_1 \ldots x_n \in S^*$ there exists permutation $\pi \in S_{|\vec{x}|}$ such that

- $f(x_1 \ldots x_n) = x_{\pi(1)} \ldots x_{\pi(n)}$ (permutativity);
- $\forall i, j \in [1 \ldots n] : f(x_i x_j) = x_i x_j \Leftrightarrow \pi^{-1}(i) \leq_\omega \pi^{-1}(j)$ (consistency).

Furthermore, if R exists, it is uniquely determined by f: $R(x_1, x_2) \Leftrightarrow f[x_1, x_2] = [x_1, x_2]$.

The permutation π in the definition of consistent permutativity can be thought of as mapping the *rank* of an (occurrence of an) element—where it occurs in the output of *sort*—to its *index*—where it occurs in the input. The inverse permutation π^{-1} thus maps the index of an element occurrence to its rank. Consistency expresses that the relative order of two elements in the output of *sort* must always be the same.

Now we know what a sort function is: any consistently permutative function! Finding such an *intrinsic* characterization of when a function is a (stable) sort function for some order is surprisingly tricky. Several plausible candidates turn out to be almost correct, but only almost [Hen07].

Isomorphisms Theorems 1 and 2 show that the categories of orders, comparator structures and sort structures are isomorphic where the morphisms are all the set-theoretic functions

[3] We write $\vec{x}|_P$ for the subsequence of $\vec{x}$ made up of all elements from P. Defining stability is notoriously tricky to do correctly.

between the underlying sets; that is, the morphisms ignore the structure. The question then is: Are they isomorphic under some notion of structure-preserving morphisms; and if so, under which notion?

Loosely speaking, monotonic and order-mapping functions on orders correspond to *sort-preserving* functions on sort structures. See Henglein [Hen07].

Observing orders Imagine we want to implement an abstract data type and export a function that makes an ordering relation $\leq$ on its element set S observable, but nothing else.[4]

The most obvious representation is by exporting a *comparison (function)*, the characteristic function $lte : S \times S \to Bool$ of $(S, \leq)$: $\forall x, y \in S : lte(x, y) = true \Leftrightarrow x \leq y$. There are alternatives, however. Theorems 1 and 2 show that an ordering relation can be made observable (and nothing else about the data type) by exporting a comparator or sort function. Each of the three possible exported functions *defines* an order once their respective intrinsic characteristics are verified, and, given any one, the other ones are parametrically polymorphically definable [Hen07].

So does it matter, whether comparison, comparator or sort function are implemented "natively" and subsequently exported? Comparison provides information on the ordering relation for only two elements at a time. As a well-known corollary, any comparison-based sorting algorithm requires $\Omega(n \log n)$ applications of comparison to sort n elements [Knu98, Section 5.3.1]. The same holds true for comparators. In contrast, *distributive* sorting algorithms [Knu98, Section 5.2.5] such as radix-sort run in linear time. Interestingly, such distributive algorithms can be defined and extended *generically* to arbitrary first-order types while preserving their linear-time performance [Hen06]. In other words: It is possible to implement a time-efficient sort function for a new data type if we have access to time-efficient *sort functions* for the (ordered) types used in its implementation. If instead only comparison functions (or comparators for that purpose) are available we are back to the comparison-based sorting bottleneck.

Conclusion Starting with a seemingly innocuous question—What is a sort function?—we have arrived at a maybe surprising answer: Any consistently permutative function. In doing so we have flipped the preeminence of orders over sort functions on the head by showing that it is not necessary to be given a (presentation) of an order to define (what it means to be) a sort function. Comparisons, comparators and sort functions are parametrically polymorphically interdefinable. So conceptually and behaviorally they are interchangeable. From a performance point of view, there is a difference, however, depending on which one we implement "natively": Sort functions admit construction of generic distributive sorting algorithms that are not subject to the comparison-based sorting bottleneck.

References

[Hen06] Fritz Henglein. Generic discrimination: Partitioning and sorting complex data in linear time. TOPPS Report D-559, Department of Computer Science, University of Copenhagen (DIKU), December 2006. http://diku.dk/forskning/topps/bibliography.

[Hen07] Fritz Henglein. Intrinsic definition of sorting functions. TOPPS Report D-565, Department of Computer Science, University of Copenhagen (DIKU), April 2007. http://diku.dk/forskning/topps/bibliography.

[Knu98] Donald Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison Wesley, 2nd edition, 1998.

[4] We shall implicitly assume that the data type also has an effective equality test. This is not necessary if we adopt a different notion of sort function where keys can carry associated values ("payload").

Higher-Order Data Type Refinement

Joseph M. Morris Malcolm Tyrrell

School of Computing, Dublin City University, Dublin 9, Ireland

Lero - The Irish Software Engineering Research Centre

Our thesis is that data refinement and nondeterminacy are intimately related, and that a strong theory of data refinement can be constructed on a sufficiently rich theory of nondeterminacy. By *sufficiently rich* we mean that the theory supports demonic and angelic nondeterminacy, and that it presents an appropriate algebra to the programmer. By a *strong* theory of data refinement we mean primarily that it is higher-order, but also that it can account for refinement of various language constructs, with and without state. Data refinement is almost always explained in the context of a particular language construct, such as function, procedure, block, module, object, schema, state machine, or some other. The most basic setting for it, however, is that of data types, and in this context we call it *data type refinement*. That is our focus here, i.e. a theory of data higher-order data refinement for functional programming. The theory of nondeterminacy we employ is that described in [2, 3], but we assume no prior knowledge of it.

A *data type* provides a collection of related operations for manipulating data, while isolating users from the data's representation details. It has three parts: a *signature*, which gives the types of its operations, a *representation type*, which is the type used to represent the data, and a *body*, which gives the implementations of the operations. Operations are typically restricted to being first-order (as in, for example, all the theories in the survey work [1]), but no such restriction applies here. Table 1 shows an example of a signature and a data type for handling dictionaries. The operations include a constant and three second-order functions. In the signature $\mathsf{DictSig}$, Dict is a place-holder which abstracts the choice of representation type. In the data type DictDT, Dict is bound to an actual type, in this case the function type $\mathsf{Key} \to \mathsf{Value}$. The clause "($\mathsf{DictSig}$)" in the definition of DictDT declares that the data type can be accessed only through the signature $\mathsf{DictSig}$ — in particular, a client program which uses DictDT does not know how the type Dict is represented. The body of the data type is the tuple $(\mathsf{empty}, \mathsf{lookup}, \mathsf{add}, \mathsf{remove})$.

Informally, an $\mathcal{S}$-*client* is a program component which uses a data type with signature $\mathcal{S}$; the component only sees the signature. Formally, an $\mathcal{S}$-*client* is a term with a place-holder for a data type of signature $\mathcal{S}$. Binding a client to a data type is called *linking*. The program that results from linking $\mathcal{S}$-client κ to data type $\mathcal{A}$ is written $\kappa\,\mathbf{with}\,\mathcal{A}$; this is a term with the same standing as any regular term.

Data type refinement is a relation between two data types which describes when one may be safely replaced by the other. The theory of nondeterminacy provides a refinement relation $\sqsubseteq$ between terms, which we use to formulate data type refinement:

Definition: Let $\mathcal{A}$ and $\mathcal{C}$ be data types of signature $\mathcal{S}$. We say that $\mathcal{A}$

$$
\begin{array}{l}
\mathsf{DictSig} \triangleq \{ \\
\quad \textbf{type}\,\mathsf{Dict}; \\
\quad \begin{array}{lcl} \mathsf{empty} & : & \mathsf{Dict}; \\ \mathsf{lookup} & : & \mathsf{Dict} \to \mathsf{Key} \to \mathsf{Value}; \\ \mathsf{add} & : & \mathsf{Dict} \to (\mathsf{Key} \times \mathsf{Value}) \to \mathsf{Dict}; \\ \mathsf{remove} & : & \mathsf{Dict} \to \mathsf{Key} \to \mathsf{Dict} \end{array} \\
\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{DictDT} \triangleq (\mathsf{DictSig}) \{ \\
\quad \textbf{type}\,\mathsf{Dict} \triangleq \mathsf{Key} \to \mathsf{Value}; \\
\quad \begin{array}{llcl} \mathsf{empty} & & \triangleq & \lambda k' : \mathsf{Key} \cdot \mathsf{none}; \\ \mathsf{lookup} & d\,k & \triangleq & d\,k; \\ \mathsf{add} & d\,(k,v) & \triangleq & \lambda k' : \mathsf{Key} \cdot \textbf{if}\ k' = k\ \textbf{then}\ v\ \textbf{else}\ d\,k'; \\ \mathsf{remove} & d\,k & \triangleq & \lambda k' : \mathsf{Key} \cdot \textbf{if}\ k' = k\ \textbf{then}\ \mathsf{none}\ \textbf{else}\ d\,k' \end{array} \\
\}
\end{array}
$$

Table 1: A signature and a data type

> *is data refined by* $\mathcal{C}$, written $\mathcal{A} \sqsubseteq \mathcal{C}$, iff for all $\mathcal{S}$-client terms κ we have $\kappa\,\textbf{with}\,\mathcal{A} \sqsubseteq \kappa\,\textbf{with}\,\mathcal{C}$.

The definition of data refinement is not practically useful because it involves a quantification over client programs. A client-independent technique for establishing data refinement is provided by *simulation*. Informally, a simulation is a correspondence between the representation types of two compatible data types which is preserved by their operations. A technique for simulation is called *sound* if the existence of a simulation between data types $\mathcal{A}$ and $\mathcal{C}$ guarantees that $\mathcal{A}$ is data refined by $\mathcal{C}$.

Let $\mathcal{A}$ and $\mathcal{C}$ be data types with common signature $\mathcal{S}$, and let T and U be their representation types, respectively. Typically, simulation between $\mathcal{A}$ and $\mathcal{C}$ in its most general form involves the use of a pair of relations connecting T and U, i.e. an abstraction relation (from the more concrete type to the more abstract one), and a concretion relation (operating in the other direction). However, when functions may be nondeterministic as here, it turns out that simulation between $\mathcal{A}$ and $\mathcal{C}$ is fully captured by a single *function*. We opt to work with *abstraction functions* $g : U \to T$ (we could equally well have chosen concretion functions). It turns out that abstraction functions employ only angelic nondeterminacy.

Let b_A and b_C be the bodies of the data types $\mathcal{A}$ and $\mathcal{C}$, respectively. The types of b_A and b_C are the instantiations of the signature $\mathcal{S}$ at the representation types T and U respectively; we will write them $\mathcal{S}\,T$ and $\mathcal{S}\,U$. Typically $\mathcal{S}\,T$ and $\mathcal{S}\,U$ are product types, with each component of the product being a function type. For example,

$$\mathsf{DictDT}\,T \equiv T \times (T \to \mathsf{Key} \to \mathsf{Value}) \times (T \to (\mathsf{Key} \times \mathsf{Value}) \to T) \times (T \to \mathsf{Key} \to T)$$

The simulation technique works by constructing a function $(\mathcal{S}\,g) : \mathcal{S}\,U \to \mathcal{S}T$ and comparing $\mathcal{S}\,g\,b_C$ with b_A using term refinement. Our soundness result is the following:

> **Theorem: (Soundness)** Let $\mathcal{A}$ and $\mathcal{C}$ be data types of signature $\mathcal{S}$ with bodies b_A and b_C, respectively. If g is an abstraction function from the

representation type of $\mathcal{C}$ to the representation type of $\mathcal{A}$, then:

$$(b_A \sqsubseteq \mathcal{S}\, g\, b_C) \;\Rightarrow\; (\mathcal{A} \sqsubseteq \mathcal{C})$$

Defining the function $\mathcal{S}\, g$ requires us to be be able to *lift* functions through type constructors. For example, lifting functions $f_0 : T_0 \to U_0$ and $f_1 : T_1 \to U_1$ through a product type constructor just uses the well-known function product operator to give $f_0 \times f_1 : (T_0 \times T_1) \to (U_0 \times U_1)$. However, there is no general way to lift f_0 and f_1 through a function type constructor to give a function $f_0 \to f_1 : (T_0 \to T_1) \to (U_0 \to U_1)$. It can be done if the functions have so-called adjoints, but these rarely occur. This is where again nondeterminacy comes to the rescue: all "reasonable" functions have adjoints in the presence of nondeterminacy.

Function *adjoints* are, roughly speaking, approximations to inverses. Let $f : T \to U$ and $g : U \to T$. We call f a *left adjoint* of g and g a *right adjoint* of f iff $Id_T \sqsubseteq g \circ f$ and $f \circ g \sqsubseteq Id_U$. We have shown [4] that nondeterminacy allows us to define two operators on functions called the *L and R operators*. Given a function $f : T \to U$, f^L and f^R are functions of type $U \to T$ and, under appropriate conditions, f^L will be f's left adjoint and f^R will be f's right adjoint. The left-adjoint is sufficient to lift functions through the function type constructor, as needed for simulations. In fact, we define $f_0 \to f_1 \triangleq (\lambda g : T_0 \to T_1 \;\cdot\; f_1 \circ g \circ {f_0}^L)$.

Overall, nondeterminacy occurs in data refinement in three ways: in underspecification in the definition of data types, in the definition of abstraction functions, and in the definition of left adjoints. It turns out that a single kind of nondeterminacy is not sufficient to perform all three roles simultaneously. Rather we require dual nondeterminacy as described in [2, 3], i.e. one supporting both demonic and angelic nondeterminacy.

Although the theory above is described for a term language, it is also intended to serve as a foundation for data refinement in other programming language paradigms. For example, it is possible to add imperative features to the language and allow data types with procedures and state.

References

[1] W. de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. Cambridge University Press, New York, 1999.

[2] J. M. Morris and M. Tyrrell. Dually nondeterministic functions and functional refinement, 2006. Submitted.

[3] J. M. Morris and M. Tyrrell. Dual unbounded nondeterminacy, recursion, and fixpoints. *Acta Informatica*, 44(5), September 2007.

[4] Joseph M. Morris and Malcolm Tyrrell. An abundance of adjoints: Nondeterminacy as a source of Galois connections, 2007. Submitted.

Transfinite Semantics in the form of Greatest Fixpoint*

Härmel Nestra

Institute of Computer Science, University of Tartu
J. Liivi 2, 50409 Tartu, Estonia
harmel.nestra@ut.ee

It is sometimes useful to imagine program runs as they were able to overcome non-termination. According to this view, a computation that falls into an infinite loop or infinitely deep recursion continues after completing the infinite subcomputation.

One application of this is in the theory of program slicing. Program slicing is a program transformation technique where the aim is to omit some statements from a given program in such a way that executing the remaining program would handle all variables of our interest exactly the same way as the original program does. This transformation is used in several branches of software engineering; a good overview of program slicing and its applications is given by Binkley and Gallagher [1].

When a loop has no influence to the values of the interesting variables via data flow, slicing algorithms do not keep it. But if this loop does not terminate then the resulting program reaches farther in the code than the original program and thus may do assignments to the interesting variables that the original program never does. In order to treat this as a correct behaviour, we can say that the original program also makes these assignments but after an infinite number of steps.

Program semantics that follow this view are usually called transfinite. The idea of using transfinite semantics in program slicing theory was first proposed by Cousot [2]. This approach was followed later in the works of Giacobazzi and Mastroeni [4] and of us [7,6].

The word 'transfinite' actually fails to characterize the whole variety of computation processes that arises in this approach. In the case where only iterative loops can be non-terminating, the sequence of execution steps is really transfinite in the sense that the steps can be enumerated by ordinals so that their execution order corresponds to the natural order of ordinals. In the case of infinitely deep recursion, such an enumeration is impossible. In [6], we showed that the sequence of steps is more like a fractal structure.

In our current work, we express transfinite semantics in terms of greatest fixpoints of monotone operators on complete lattices of set-valued functions. This serves the following purposes.

1. In this framework, we give an exhaustive definition of semantics of infinitely deep recursion. We addressed infinitely deep recursion semantics also in [6]

* Partially supported by Estonian Science Foundation under grant no. 6713.

but there, the definition remained indeterminate in many cases. Other works on transfinite semantics do not handle recursion, they are limited with transfinite iteration.

2. In this framework, we describe a large family of semantics, including several variants of transfinite semantics as well as standard semantics, parametrically in a uniform way. The uniform parametric representation is somewhat similar to that in [6]. However, the semantics specification in [6] did not include any order relation, thus fixpoints were specified in ad-hoc manners while monotone operators together with their least and greatest fixpoints provide a standard framework for semantics.
3. Expressing semantics via greatest (or least) fixpoints of monotone operators enables to build Cousot's hierarchy of these semantics (see [2,3,4]). We have done some work also in this direction.

We prove that the monotone operators whose greatest fixpoints are used as our semantics are Scott-cocontinuous. This shows that the semantics can be achieved by iteration which is not transfinite, even if the semantics is transfinite.

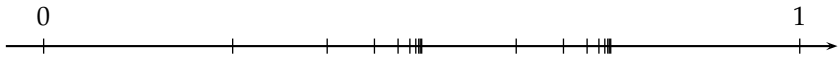
In order to achieve transfinite semantics in the form of greatest fixpoint, the usual execution traces are replaced with fractional traces where steps are enumerated by rational numbers from interval $[0;1]$. Fractional traces were introduced by us in [6].

For example, the execution trace of the swap program `z := x ; (x := y ; y := z)` at initial state $(\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 0)$ is

$$0 \mapsto (\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 0),$$
$$\tfrac{1}{2} \mapsto (\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 1),$$
$$\tfrac{3}{4} \mapsto (\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 1),$$
$$1 \mapsto (\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 1, \mathsf{z} \mapsto 1).$$

The assignments `z := x`, `x := y` and `y := z` are run within intervals $[0;\frac{1}{2}]$, $[\frac{1}{2};\frac{3}{4}]$ and $[\frac{3}{4};1]$, respectively. This is so independently of the initial state.

Even if the three assignments were replaced with arbitrary three statements S_1, S_2, S_3 and executed at any state, the same intervals would be reserved for the three statements. For example, if $S_1 = S_2 =$ `while true do skip` and $S_3 =$ `x := 0` then the execution trace of statement S_1 ; $(S_2$; $S_3)$ is depicted in the following figure; it suffices to indicate only the rationals occurring in the trace since the state does not change except probably at point 1 where the value of `x` changes to 0 whatever it was before:



Roughly, the reason why the usual traces are not satisfying for expressing transfinite semantics as a greatest fixpoint is that the greatest fixpoint would contain too many traces. Besides the desired traces, the semantics of an infinite loop would contain all traces having a desired trace as a prefix. Using fractional

traces solves this problem as the traces grow into depth rather than into length, the whole interval $[0;1]$ is always distributed between the statements of a program and no space is left for garbage.

Another possibility is to use trees instead of traces by accommodating the approach of Glesner [5] to the transfinite case. Actually, tree semantics and fractional trace semantics have very much in common. Fractional traces share both tree and trace properties: they reflect the deduction tree structure while keeping the execution order evident. Fractional traces form an intermediate category between trees and usual traces.

Our parametric framework includes both fractional trace semantics and tree semantics. Most of the results are obtained simultaneously for both.

In our approach, the semantics are not a priori deterministic. This is presumably not a big drawback since non-determinism is always introduced after non-termination.

References

1. Binkley, D. W., Gallagher, K. B.: Program Slicing. *Advances in Computers* **43** (1996) 1–50
2. Cousot, P.: Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Electronic Notes in Theoretical Computer Science* **6** (1997) 25 pp.
3. Cousot, P.: Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Theoretical Computer Science* **277** (2002) 47–103
4. Giacobazzi, R., Mastroeni, I.: Non-Standard Semantics for Program Slicing. *Higher-Order Symbolic Computation* **16** (2003) 297–339
5. Glesner, S.: A Proof Calculus for Natural Semantics Based on Greatest Fixed Point Semantics. *Electronic Notes in Theoretical Computer Science* **132** (2005) 75–93
6. Nestra, H.: Fractional Semantics. In Johnson, M., Vene, V. (eds.): *Proceedings of AMAST 2006. Lecture Notes in Computer Science* **4019** (2006) 278–292
7. Nestra, H.: Iteratively Defined Transfinite Trace Semantics and Program Slicing with respect to Them. PhD thesis, University of Tartu (2006) 119 pp.

Probabilistic Opacity *

Damas P. GRUSKA
Institute of Informatics, Comenius University,
Mlynska dolina, 842 48 Bratislava, Slovakia,
gruska@fmph.uniba.sk.

Abstract

We define a security concept 'probabilistic opacity' in a probabilistic timed process algebra framework. A process is opaque with respect to a given property over its traces (represented by predicate ϕ) if an observer (represented by an observational function) cannot decide whether the property has been satisfied. In general, opacity is undecidable even for finite state systems so we express both the security predicate and the observation function by means of finite state processes. The resulting security properties can be used for specifications and verifications of concurrent systems.

Keywords: probabilistic timed process algebras, timing attacks, opacity, information flow, security

Several formulations of a notion of system security can be found in the literature. Many of them are based on a concept of non-interference (see [9]) which assumes the absence of any information flow between private and public systems activities. More precisely, systems are considered to be secure if from observations of their public activities no information about private activities can be deduced. This approach has found many reformulations. To the most powerful ones belong a concept opacity. With its help many other security properties can be expressed (see [1]) and it can be formulated for different formalisms, computational models and nature or "quality" of observations.

Timing attacks have a particular position among attacks against systems security. They represent a powerful tool for "breaking" "unbreakable" systems, algorithms, protocols, etc. For example, by carefully measuring the amount of time required to perform private key operations, attackers may be able to find fixed Diffie-Hellman exponents, factor RSA keys, and break other cryptosystems (see [13]). This idea was developed in [3] where a timing attack against smart card implementation of RSA was conducted. In [11], a timing attack on the RC5 block encryption algorithm, in [14] the one against the popular SSH protocol and in [4] the one against web privacy are described.

*Work supported by the grant VEGA 1/3105/06 and APVV-20-P04805.

In the literature several papers on formalizations of timing attacks can be found. Papers [6, 7, 8] express attacks in a framework of (timed) process algebras. In all these papers system actions are divided into private and public ones and the security property expresses there is not an interference between them. More precisely, in [6, 7] it is required that on a level of system traces which do not contain internal actions one cannot distinguish between system which cannot perform private actions and system which can perform them but all of them are reduced to internal actions. In paper [8] a concept of public channels is elaborated. In the above mentioned papers also a slightly different approach to system security is presented - the system behaviour must be invariant with respect to composition with an attacker which can perform only private actions ([6], [7]) or with an attacker which can see only public communications ([8]).

The opacity property is based on a concept of observation functions and on a concept of predicates over system traces. It is similar to the concept of Non-information flow (NIF) (see [10]) but it allows more general observation functions (obs) in which an observation of an particular action might depend on context of its appearance. In the case of the static observation function each action is observed independently on its context. In case of the dynamic observation function an observation of an action may depend on the previous ones, in case of the orwellian and m-orwellian observation function an observation of an action depends on the all and m previous actions in the trace, respectively. The static observation function is the special case of m-orwellian one for $m = 1$. Opacity is defined for arbitrary predicate ϕ over traces ($Tr(P)$) of system actions. Roughly speaking, the observer cannot deduce validity of ϕ if there are two traces $w, w' \in Tr(P)$ such that $\phi(w), \neg\phi(w')$ and the traces cannot be distinguished by the observer i.e. $obs(w) = obs(w')$. Formally:

A predicate ϕ over $Tr(P)$ is opaque w.r.t. the observation function obs if for every trace w, $w \in Tr(P)$ such that $\phi(w)$ holds, there exists a trace $w', w' \in Tr(P)$ such that $\neg\phi(w')$ holds and $obs(w) = obs(w')$.

Clearly, opacity can capture more sofisticated security properties then just an execution of some private action as it is done in [6, 7, 8] . Moreover, since many of timing attacks described in the literature are based on observations of "internal" actions opacity can work also with this information what is not the case of the above mentioned papers. In this way we can consider timing attacks which could not be taken into account otherwise.

Generality of opacity brings some disadvantages. The opacity property is undecidable even for finite state systems. Hence we restrict the power of observation functions and predicates over traces by expressing them by (finite state) process algebra processes. Moreover since many attacks are based on statistical analyzes of system behaviour (see [13, 3, 11, 14]) instead of just "one single observation" or both observation function and predicate over traces might have probabilistic nature. Hence to obtain probabilistic opacity we will exploit probabilistic timed process algebra (in style of [12]) for process specifications as well as for specifications of observation functions and the predicates. With their help probabilistic version of opacity can be be formulated in terms of trace inclusion and so that it becomes decidable for finite state processes. Moreover some other

useful properties for it can be proved so that the resulting security properties can be used for specifications and verifications of concurrent systems.

References

[1] Bryans J., M. Koutny, L. Mazare and P. Ryan: Opacity Generalised to Transition Systems. In Proceedings of the Formal Aspects in Security and Trust, LNCS 3866, Springer, Berlin, 2006

[2] Busi N. and R. Gorrieri: Positive Non-interference in Elementary and Trace Nets. Proc. of Application and Theory of Petri Nets 2004, LNCS 3099, Springer, Berlin, 2004.

[3] Dhem J.-F., F. Koeune, P.-A. Leroux, P. Mestre, J.-J. Quisquater and J.-L. Willems: A practical implementation of the timing attack. Proc. of the Third Working Conference on Smart Card Research and Advanced Applications (CARDIS 1998), LNCS 1820, Springer, Berlin, 1998.

[4] Felten, E.W., and M.A. Schneider: Timing attacks on web privacy. Proc. 7^{th} ACM Conference on Computer and Communications Security, 2000.

[5] Focardi, R. and R. Gorrieri: Classification of security properties. Part I: Information Flow. Foundations of Security Analysis and Design, LNCS 2171, Springer, Berlin, 2001.

[6] Focardi, R., R. Gorrieri, and F. Martinelli: Information flow analysis in a discrete-time process algebra. Proc. 13^{th} Computer Security Foundation Workshop, IEEE Computer Society Press, 2000.

[7] Focardi, R., R. Gorrieri, and F. Martinelli: Real-Time information flow analysis. IEEE Journal on Selected Areas in Communications 21 (2003).

[8] Gorrieri R. and F. Martinelli: A simple framework for real-time cryptographic protocol analysis with compositional proof rules. Science of Computer Programing, Volume 50, Issue 1-3, 2004.

[9] Goguen J.A. and J. Meseguer: Security Policies and Security Models. Proc. of IEEE Symposium on Security and Privacy, 1982.

[10] Gruska D.P.: Observation Based System Security. To appear in Fundamenta Informaticae

[11] Handschuh H. and Howard M. Heys: A timing attack on RC5. Proc. Selected Areas in Cryptography, LNCS 1556, Springer, Berlin, 1999.

[12] Hansson, H. a B. Jonsson: A Calculus for Communicating Systems with Time and Probabilities. In Proceedings of 11th IEEE Real - Time Systems Symposium, Orlando, 1990.

[13] Kocher P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS and other systems. Proc. Advances in Cryptology - CRYPTO'96, LNCS 1109, Springer, Berlin, 1996.

[14] Song. D., D. Wagner, and X. Tian: *Timing analysis of Keystrokes and SSH timing attacks.* Pro.10th USENIX Security Symposium, 2001.

Symbolic Robustness Analysis of Probabilistic Timed Systems

Joost-Pieter Katoen, Mani Swaminathan, and Martin Fränzle
katoen@cs.rwth-aachen.de, {mani.swaminathan, fraenzle}@informatik.uni-oldenburg.de

Real-time systems, which have strict timing requirements, have emerged as an enabling technology for several important application domains such as air traffic control, telecommunications, and medicine, to name a few. Such systems are becoming increasingly pervasive, and hence rigorous methods and techniques to ensure their correct functioning are of utmost importance. Timed Automata (TA) [1] have been extensively studied as a formalism for modelling real-time systems. TA extend ω-automata by augmenting them with "clock" variables based on a dense-time model, which quantitatively captures the behaviour of the system with time. TA model-checkers such as UPPAAL [2, 3] and KRONOS [4] are now available and have been successfully used in industrial case studies such as [5].

The TA model provides for the behavioural description of real-time systems in terms of *non-determinism*, by having *transition-guards* and *location-invariants* that the clock values need to satisy. It however may be desirable to quantitatively express the relative *likelihood* of the system exhibiting certain behaviour, which is particularly relevant while considering properties such as fault-tolerance.

Probabilistic Timed Automata (PTA) [6, 7] are an extension of TA that model real-time systems in terms of discrete probability distributions annotating the transitions between locations, thereby expressing both *non-deterministic and probabilistic* behaviour. This model is then used to check system requirements expressed in PTCTL (Probabilistic Timed Computation Tree Logic), which is an extension of TCTL to deal with probability. The model-checking is based on a notion of *region equivalence* similar to that for usual TA. As with TA, the number of regions in this case is exponential in the number of clocks, rendering region-based model-checking of PTA impractical [6].

[6] therefore considers the narrower class of *probabilitic real-time reachability properties*, which can still express a number of useful system requirements, and presents an efficient symbolic algorithm for model-checking such properties against systems modelled as PTA. This algorithm is an adaptation of the standard *zone-based* algorithm used in TA model-checkers such as UPPAAL, and is fully-forward.[1] The probability of time-bounded reachability thus computed is however not exact, but rather an upper bound on the true probability of reachability. This technique is particularly useful in the verification of *invariance* or *safety* properties (where we are interested in the probability of reaching an "unsafe" target state) and has been applied to case-studies such as the bounded re-transmission protocol [6].

Given a PTA and a (bad) target state (location-zone pair) (l, B), the forward symbolic algorithm in [6] essentially computes the maximum probability (w.r.t all possible resolutions of the non-deterministic choices in the PTA's underlying *probabilistic timed structure*) with which the target state could be reached, or equivalently, the minimum probability p with which the target state could be avoided, i.e., $p = minprob(Reach \cap (l, B) = \emptyset)$. This analysis entails the computation of the set $Reach$ of (symbolic) states reachable with non-zero probability (performed by a fully forward state-space exploration analogous to that in TA model-checkers such as UPPAAL) in the form of a *zone-graph*, which is in fact a *Markov Decision Process*. This is then followed by the computation of the minimum probability p of target-avoidance by solving a system of linear in-equations obtained from the zone-graph, using standard techniques of linear-programming.

However, the existing analysis techniques for PTA consider an idealized behaviour of the clocks, in the sense that they are assumed to be *fully synchronous*, unlike in practice, where the clocks could actually *drift*. The effect of such imprecisions, modelled by a parameter $\varepsilon > 0$, has been studied for TA [8, 9], where it has been shown that a state that is un-reachable for perfect clocks, might actually be reachable for the slightest $\varepsilon > 0$ drift in the clocks. This leads to the definition of "robust" reachability for TA, where a state is considered to be *robustly reachable* iff it is reachable for *every* (i.e, even the slightest) $\varepsilon > 0$ drift in the clocks. Conversely, a (bad) state is considered not to be robustly reachable (and the TA is therefore "robustly safe") if it could be entirely avoided for a certain (small enough) $\varepsilon > 0$ drift.

In this paper, we consider PTA with drifting clocks that

[1] [7] presents a symbolic algorithm for PTA that entails a combination of forward and backward analyses, for the exact verification of the larger class of properties expressible in PTCTL, which is not considered in this paper.

could occur in practice. Such imprecisions play an adversarial role enhancing the non-deterministic behaviour of the system. We present a symbolic analysis technique that is "robust" w.r.t such imprecisions. In particular, we propose an extension to the zone-based fully forward algorithm for PTA [6] in order to deal with the effects of infinitesimal clock-drift. Our robustness notion is weaker than the classical definition for TA [8, 9], in the sense that the robustness margin (in case of a certification of robust safety) could potentially decrease with the iteration depth.

Given a PTA with clock-drift parameterized by $\varepsilon > 0$, with slopes in the range $[\frac{1}{1+\varepsilon}, 1+\varepsilon]$, we wish to compute $q = minprob(\forall i \in N\ \exists \varepsilon_i > 0 : Reach_i^{\varepsilon_i} \cap (l, B) = \emptyset)$. Here, q is the minimum probability of avoiding the target state (l, B) while considering drifting clocks, such that at any given iteration depth i, there exists a strictly positive value of the perturbation ε_i, for which the corresponding perturbed reach-set $Reach_i^{\varepsilon_i}$ at that iteration depth entirely avoids (l, B).

We conjecture that the computation of q above may be performed by simply replacing the time-successor of a zone by a form of zone-enlargement, called its "drift-neighbourhood", in the fully-forward zone-based algorithm of [6]. This results in an enlarged zone-graph consisting of states that are reachable with non-zero probability under the smallest clock-drift, and is obtained from the standard (i.e., non-robust) zone-graph by widening all *closed difference* constraints defining the zones by 1, to the next higher *open* constraints.[2] The computation of the minimum probability q of "robustly" avoiding the target state may then be performed by solving a system of linear in-equations as in the computation of p for the non-robust case, with $q \leq p$ always. Furthermore, the difference between the minimal probabilities p and q may be computed from the *strict-inequalities* of the zone-graph. This provides for a precise quantification of the change in the system behaviour owing to the additional non-determinism imposed by drifting clocks.

This paper thus attempts to address the interplay of *robustness* (w.r.t drifting clocks) and *randomness* (w.r.t transition probabilities) in the quantitative analysis of real-time systems. We are presently working on the correctness proof of our algorithm, by induction over the iteration depth. Following this, we intend to implement our algorithm in a practical PTA model-checker, with applications to case-studies such as the bounded re-transmission protocol and the IPv4 Zeroconf protocol [10].

Acknowledgements Joost-Pieter Katoen is supported by the Dutch Research Council NWO under grant 612.000.311. Mani Swaminathan and Martin Fränzle are supported by the Deutsche Forschungsgemeinschaft through the Graduiertenkolleg Trustsoft `http://trustsoft.uni-oldenburg.de` and the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, `www.avacs.org`).

[2] We assume the guards and invariants of the PTA to be *closed*, i.e., defined by non-strict inequalities.

References

[1] R. Alur and D. Dill, "A Theory of Timed Automata", *Theoretical Computer Science* **126**, Elsevier (1994), 183–235

[2] G. Behrmann, A. David, and K. Larsen, "A Tutorial on Uppaal", *Formal Methods for the Design of Real-Time Systems*, LNCS **3185**, Springer-Verlag (2004), 200–236

[3] J. Bengtsson and W. Yi, "Timed Automata: Semantics, Algorithms, and Tools", LNCS **3098**, Springer-Verlag (2004), 87–124

[4] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, S. Yovine, "KRONOS: A Model-Checking Tool for Real-Time Systems", *Proceedings of FTRTFT'98*, LNCS **1486**, Springer-Verlag (1998), 298–302

[5] M. Lindahl, P. Pettersson, and W. Yi, "Formal Design and Analysis of a Gear Controller", *International Journal on Software Tools for Technology Transfer* **3**, Springer-Verlag (2001), 353–368

[6] M. Kwiatkowska, G. Norman, R. Segala, and J. Sproston, "Automatic Verification of Real-time Systems with Discrete Probability Distributions", *Theoretical Computer Science* **282**, Elsevier (2002), 101-150

[7] M. Kwiatkowska, G. Norman, J. Sproston, and F. Wang, "Symbolic Model Checking for Probabilistic Timed Automata", *Information and Computation* **205**, Elsevier (2007), 1027-1077

[8] A. Puri, "Dynamical Properties of Timed Automata", *Discrete Event Dynamic Systems* **10**, Kluwer (2000), 87–113

[9] M. De Wulf, L. Doyen, N. Markey, and J.-F. Raskin, "Robustness and Implementability of Timed Automata", *Proceedings of FORMATS-FTRTFT'04*, LNCS **3253**, Springer-Verlag, 2004, 118-133

[10] M. Kwiatkowska, G. Norman, D. Parker, and J. Sproston, "Performance Analysis of Probabilistic Timed Automata using Digital Clocks.", *Formal Methods in System Design* **29**, Springer-Verlag (2006), 33–78

A probabilistic semantic paradigm for component-based security risk analysis

Gyrd Brændeland[1,2] and Ketil Stølen[1,2]

[1] Department of Informatics, University of Oslo, Norway
[2] SINTEF, Norway

Abstract. We propose a probabilistic semantic paradigm for component-based security risk analysis. By security risk, we mean behaviour that constitutes a risk with regard to ICT security aspects, such as confidentiality, integrity and availability. The purpose of this work is to investigate the nature of security risk in the setting of component-based system development. A better understanding of security risk at the level of components facilitates the prediction of risks related to introducing a new component into a system. The semantic paradigm provides a first step towards integrating security risk analysis into component-based system development.

1 Introduction

The flexibility of component-oriented software systems enabled by component technologies such as Sun's Enterprise Java Beans (EJB), Microsoft's .NET or the Open Source Gateway initiative (OSGi) gives rise to new types of security concerns. In particular the question of how a system owner can know whether to trust a new component to be deployed into a system. A solution to this problem requires integrating the process of security risk analysis in the early stages of component-based system development. The purpose of security risk analysis is to decide upon the necessary level of asset protection against security risks, such as a confidentiality or integrity breach. Unfortunately, the processes of system development and security risk analysis are often carried out independently with little mutual interaction. The result is expensive redesigns and unsatisfactory security solutions. To facilitate a tighter integration we need a better understanding of security risk at the level of components. But knowing the security risks of a single component is not enough, since two components can affect the risk level of each other. We therefore need a strategy for predicting system level risks that may be caused by introducing a new component. A better understanding of security risk at the level of components is a prerequisite for compositional security level estimation.

Our contribution is a novel semantic paradigm capturing: probabilistic behaviour; the composition of basic probabilistic components into composite components; and the notion of security risk as known from asset-oriented security risk analysis. In the following abstract we explain the intuition behind the extended semantic model. The full details including formal definitions can be found elsewhere [1].

2 Background

We build our semantic paradigm on top of the trace semantics of STAIRS [2, 3]. A trace is a sequence of events, representing a component run. There are two kinds of events: transmission and consumption of a message, where a message is a triple (s, re, tr) consisting of a signal s, a transmitter lifeline tr and a receiver lifeline re. Each event has a timestamp, which is a rational number.

3 Representing probabilistic components

A risk is the probability that an event will harm an asset with a given impact. In order to speak of component risks there need to be uncertainty in the component behaviour. Exposed to a threat the component suffers an unwanted incident, or resists is, with a certain probability. Hence, we represent components with risk behaviour as probabilistic components.

3.1 Closed components

A closed component has no external interfaces, i.e., it does not interact with the environment. Any closed component I can be represented by a probability space $(D_I, \mathcal{F}_I, f_I)$ where $\mathcal{F}_I$ is the cone-σ-field of D_I and f_I is the probability measure on $\mathcal{F}_I$ [5, 4]. The *cone-σ-field* is the smallest σ-field generated from the set of cones we obtain from any finite prefix of a trace in D_I. We let $\widetilde{D}_I$ denote the set of finite prefixes of the traces in D_I. The *cone* of a finite trace $t \in \widetilde{D}_I$ is the set of all traces in D_I with t as a prefix.

3.2 Open components

A component that is not closed is open. The probability of traces of open components depends on the probability of external behaviour. Such components cannot be represented directly as probability spaces. In order to represent open components in general, we adopt a reactive communication model using recording media to store incoming messages.

We represent an open component by a parameterized probability space, which is a triple of functions $(D_A(h), \mathcal{F}_A(h), f_A(h))$ that each takes a communication history as parameter and returns a set of traces $D_A(h)$, the cone-σ-field $\mathcal{F}_A(h)$ of $D_A(h)$ and the probability measure for $f_A(h)$. A given instance of a recording medium gives rise to *one* probability space with regard to an open component. It characterises the probabilistic behaviour of the component with respect to an environment whose behaviour is captured by the instance of the recording medium.

Let I be a closed composite component consisting of two distinct open basic components A and B that interact via recording media. Then we can compute the probability of the behaviour of I from the probabilities of the behaviours of A and B [1].

4 Extending the model to capture security risk

In a trace based semantics an *information security incident* is an event that reduces the value of one or more component assets. An *asset* is a special kind of component that has a value. A *threat scenario* is a sequence of events that ends with an information security incident. A *threat* is a component that may initiate a threat scenario through interacting with another component. *Risk* is the probability of an event that reduces the value of an asset. The *risk value* of a risk is a function of its probability, impact and affected asset. A *protection requirement* is a function that for all assets yields an upper bound for the acceptable risk value. We say that a component I satisfies protection requirements for all assets if all risks of I satisfy the acceptance level for all assets.

In the extended semantic model closed and open components are both five-tuples, where the first three elements are the same as for closed and open probabilistic components, respectively, and the last two are a set of component assets A_I and an impact function iv_I that yields the impact value for all events with regard to all assets $a_j \in A_I$.

For open components a given instance of a recording medium characterises the probabilistic behaviour of the component with respect to a threat. In order to obtain the risks of an open component A with regard to a certain threat T we look at the composition of A and T. In the case where we have a set of threats $T_1, \ldots, T_n$ we combine all threats into one composite component.

The full details of the extended semantic model can be found in [1].

5 Conclusion

We have outlined a semantic paradigm that captures probabilistic component behaviour. Using recording media to resolve mutual dependencies between open interacting components we have shown compositionality of probabilistic components [1]. Extending the semantic paradigm with security risk related aspects we can formally represent risk behaviour of components. The formal representation of security risk analysis results at the component-level facilitates integration of security risk analysis into all steps of the development process, from requirements specification to implementation.

References

[1] G. Brændeland and K. Stølen. A formal foundation for integrating security analysis into a component-based development process. Technical Report 363, University of Oslo, Department of Informatics, 2007.

[2] Ø. Haugen, K. E. Husa, R. K. Runde, and K. Stølen. Stairs towards formal design with sequence diagrams. *Software and System Modeling*, 4(4):355–357, 2005.

[3] A. Refsdal, K. E. Husa, and K. Stølen. Specification and refinement of soft real-time requirements using sequence diagrams. In *FORMATS*, volume 3829 of *Lecture Notes in Computer Science*, pages 32–48. Springer, 2005.

[4] A. Refsdal, R. K. Runde, and K. Stølen. Relating computer systems to sequence diagrams with underspecification, inherent nondeterminism and probabilistic choice. Part 1. Technical Report 346, University of Oslo, Department of Informatics, 2007.

[5] R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems.* PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1995.

A Hardware Independent Parallel Programming Model

Magne Haveraaen and Eva Suci
Institutt for Informatikk, University of Bergen, Norway
http://www.ii.uib.no/~magne/
http://www.ii.uib.no/~eva/

Abstract

Programming a highly parallel machine or chip can be formulated as finding an embedding of the computation's data dependency into the underlying hardware architecture. With the data dependency pattern of a computation extracted as a separate entity in a programming language, one has a powerful tool to code parallel programs.

Today's computational devices are rapidly evolving into massively parallel systems. Multi-core processors, e.g. the Cell processor and graphics processor units featuring 128 on-chip processors are all invented to explore parallelism at its most. The appearance of these new devices set new directions in computer science research, since the different hardware architectures come with different, new programming models, which makes the writing of portable, efficient parallel code difficult.

We want to develop an experimental unified programming model, based on the theory of Data Dependency Algebras and its embeddings, which aims to overcome these difficulties.

A general theory for describing the data dependency graph of a computation and its embedding into a hardware's space-time graph was introduced in [2]. In [1] this basic idea was taken a step further in the constructive recursive (CR) approach, where the computation is separated from its dependencies, allowing both to be programmed and manipulated independently of each other.

The dependency structure of a computation is defined as a separate data type – the Data Dependency Algebra (DDA), and the algorithm for solving a problem is given by a recursive function on this DDA.

The run-time parallel distribution and global communication pattern of a hardware layout, whether a parallel computer, a highly parallel graphics processor unit (GPU), a many-core CPU or a chip, is also defined by a separate data type, a space-time DDA. The embedding of a computation into the underlying hardware then becomes a task of finding an efficient mapping of the DDA of the computation into the space-time DDA of the hardware layout. This allows full control of the computation and explicit handling of the global interconnection network at a very high abstraction level.

A prototype compiler based on the CR aproach, called Saphire, was built to provide a simple way to generate parallel code from high level DDA descriptions for high performance computing architectures [4].

The flexibility offered by the CR framework seems to meet the need raised by current hardware programming issues. We would therefore like to extend this approach and show how it works for hardware.

Our model proposes to use DDA abstractions for describing parallel computations, hardware layouts or programming models, and embeddings. Once the dependency pattern of a computation is extracted and the recursive functions solving a problem are defined, they remain unchanged irrespectively of the available hardware resource. The underlying hardware architecture and the embedding of the compution into this can be also specified on a high-absraction level. Taking this into consideration, a suitably enhanced Saphire compiler will then be capable of producing parallel code for the different hardware architectures, without reinventing the problem solving code.

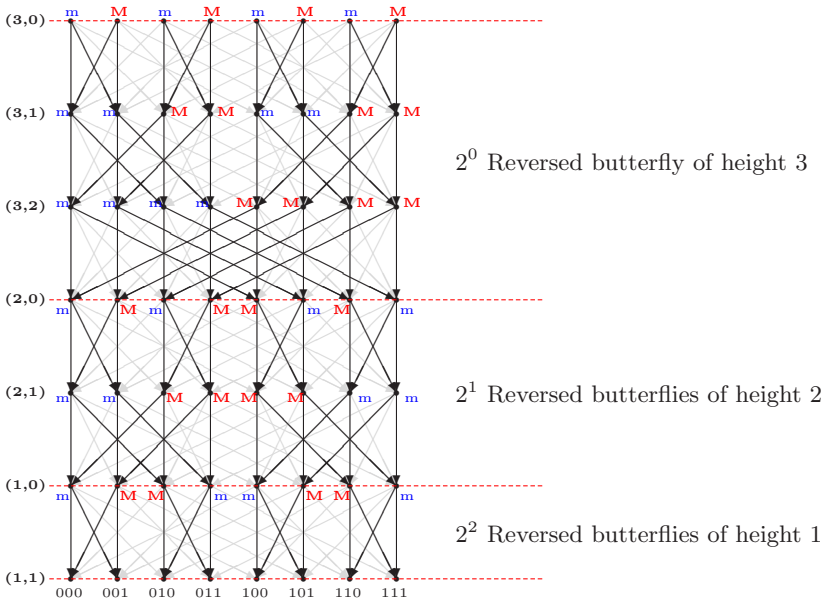


Figure 1: Bitonic sort DDA for 2^3 inputs and the underlying hardware communication layout: the hypercube space-time DDA of dimension 3 in 7 time-steps

The case-study of this presentation illustrates this idea. We extract the DDA of a non-trivial parallel computation, such as the bitonic sorting network, and define the computations as recursive functions on this DDA, expressed in Saphire terminology, and sketch its embeddings into the space-time layout descriptions of two different parallel architectures, the hypercube and a GPU.

The DDA of the bitonic sort can be seen as a combination of several reversed butterfly dependency graphs of different height, each subbutterfly corresponding to a bitonic merge, as seen in Fig. 1. Bitonic sorting is defined then as min/Max functions on the points of the graph. The figure also shows a straightforward embedding of this DDA into the underlying space-time communication layout (grey) of a hypercube.

An embedding consists of three functions, EP which defines how DDA points map into hardware space and time coordinates, ER which defines how a request branch at a DDA point is translated into an incoming communication channel, and ES which defines how a supply branch at a DDA point is translated into an outgoing communication channel. The embedding controls the utilisation of the available hardware resource.

When embedding the bitonic sort DDA into a GPU (Fig. 2), we consider NVIDIA's CUDA programming model [3] developed for general purpose computations. Here the GPU is considered as a coprocessor to the main CPU, capable of handling a huge number of threads, which are downloaded by the host onto the GPU in the form of a kernel. We can model CUDA as a special space-time architecture, where communication between threads in successive time-steps can be explicitely defined on a high abstraction level.

The present Saphire compiler is capable of producing parallel code from DDAs using the MPI message passing library. We are planning to enhance this prototype compiler to generate parallel code for other architectures as well, e.g., for CUDA. This will also allow us to easily test the efficiency of different embeddings, since they can be reformulated on a high abstraction level, and Saphire generates the parallel code.

We believe DDA abstractions can also be streched to tackle some aspects of FPGA programming, and it also seems promising to take a step further towards low-level hardware and investigate possible ways of combining DDAs with circuit designs.

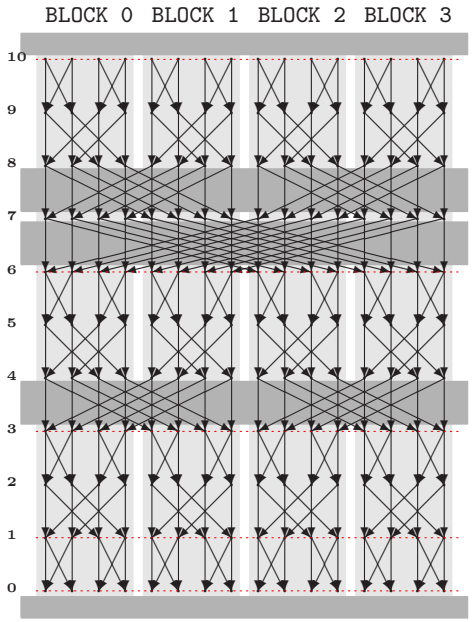


Figure 2: Bitonic sort DDA for 2^4 inputs, embedded into NVIDIA's CUDA programming model, executed by 4 kernel invocations, each consisting of 4 blocks of threads.

References

[1] Magne Haveraaen. Distributing programs on different parallel architectures. In David A. Padua, editor, *Proceedings of the 1990 International Conference on Parallel Processing (ICPP)*, volume II Software, pages 288–289. The Pennsylvania State University Press, University Park and London, 1990.

[2] W L. Miranker and A Winkler. Spacetime representations of computational structures. *Computing*, 32(2):93–114, 1984.

[3] NVIDIA. Compute unified device architecture programming guide (ver.0.8.2). http://www.nvidia.com.

[4] Steinar Søreide. *Compiling Saphire into Sequential and Parallel Code Using Assertions. Master Thesis.* Universitetet i Bergen, P.O. Box 7800, N-5020 Bergen, Norway, Spring 1998.

Analysis of Arithmetical Congruences on Low-Level Code

Stefan Bygde

Department of Computer Science and Electronics, Mälardalen University, Sweden
stefan.bygde@mdh.se

Abstract. Abstract Interpretation is a well known formal framework for abstracting programming language semantics. It provides a systematic way of building static analyses which can be used for optimisation and debugging purposes. Different semantic properties can be captured by so-called abstract domains which then easily can be combined in various ways to yield more precise analyses. The most known abstract domain is probably the one of intervals. An analysis using the interval domain yields bindings of each integer-valued program variable to an interval at each program point. The interval is the smallest interval that contains the set of integers possible for that particular variable to assume at that program point during execution. Abstract interpretation can be used in many contexts, such as in debugging, program transformation, correctness proving, Worst Case Execution Time analysis etc.
In 1989 Philippe Granger introduced a static analysis of arithmetical congruences. The analysis is formulated as an abstract interpretation computing the smallest (wrt. inclusion) congruence (residue) class that includes the set of possible values that that variable may assume during execution. The result of the analysis is a binding of each integer-valued variable at each program point to a congruence class (e.g. $\{x \in \mathbb{Z} \mid x \equiv 2 \; mod \; 3\}$). Applications for this analysis include automatic vectorisation, pointer analysis (for determining pointer strides) and loop-bound analysis (for detecting loops with non-unit strides). However, in the original presentation, the analysis is not well suited to use on realistic low-level code. By low-level code we mean either compiled and linked object code where high-level constructions has been replaced with target-specific assembly code, or code in a higher-level language written in a fashion close to the hardware. A good example of low-level code is code written for embedded systems which often is using advantages of the target hardware and/or using a lot of bit-level operations. Code for embedded systems is an increasingly important target for analysis, since it is often safety-critical. The reason that the congruence domain in its original presentation is not suitable for low-level code is mainly due to the three following properties of low-level code: A) Bit-level operations are commonly used in low-level code. Programs that contain bit-operations are not supported in the original presentation. For any computation of an expression which contain operations that has not been defined in the analysis, it has to assume that nothing is known about the result and assign the result to the largest congruence class (equal to $\mathbb{Z}$). This can potentially lead to very imprecise analysis results. B) The interpretation

of the values of integer-valued variables is not obvious (e.g. they can be signed or unsigned), the original presentation assumes that values has unambiguous representations. C) The value-domain is limited by its representation (integers are often represented by a fixed number of bits). In Grangers presentation integer-valued variables are assumed to take values in the infinite set of integers. Our contribution is to extend the theory of the analysis of arithmetical congruences to be able to handle low-level or assembly code, still in the framework of abstract interpretation.
This paper provides accurate definitions to the abstract bit-operations AND,NOT,XOR, left- and right shifting and truncation for the congruence domain in order to make the domain support these operations. We provide definitions for the operations together with proofs of their correctness. In these definitions care has been taken to the finite, fixed representation of integers as well as their sometimes ambiguous interpretations as signed or unsigned. With these definitions, congruence analysis can efficiently be performed on low-level code. The paper illustrates the usefulness of the new analysis by an example which shows that variables keep important parity information after executing a XOR-swap.

Hardware Modelling Language and Verification of Design Properties

Aske Brekling, Michael R. Hansen and Jan Madsen, IMM, DTU

As the complexity of chips grows, the methodology to build chips has to evolve. Today, chips are largely synthesized from high-level architectural descriptions that hide low-level details.

The majority of hardware designs are done using the most common hardware description languages, VHDL [5] or Verilog [10]. Both languages support high-level architectural descriptions, but allow hardware designers to incorporate low-level details in order to optimize for a particular hardware technology and directly synthesize using a restricted subset of the languages. However, chips may also be synthesized from software based models in much the same way as compilers produce executable code. Examples of such languages are Esterel, Lustre and Signal, see e.g. [3].

We have chosen to base our language for hardware models on Gezel [9]. It depends on reasonably few, simple and clean concepts, and it strikes a balance between software and hardware concerns that we believe suits the needs for a modern top-down approach to hardware design.

We give a semantics domain that can be used for hardware design languages like Gezel. With this semantics, we believe that a new Gezel-like language could be defined where the syntax reflects the semantics in a direct manner. We also show how the semantics can be used in connection with verification by relating the semantical domain to timed-automata [1]. We have experimented with verification of some examples, e.g. the Simplified Data Encryption Standard (SDES) Algorithm [8], using the UPPAAL system [2, 6].

An Introduction to Gezel

Gezel is a high-level hardware description language. It comes with an interpreter as well as a translator with VHDL as its target language. The interpreter provides means for simulation and debugging. The language does not have a formal semantics, and there is no tool for verification of Gezel specifications.

A Gezel specification specifies a number of *components* and their interconnections. A component in Gezel is made up of a *datapath* (`dp`) providing a number of *actions*, called *signal flow graphs* (`sfg`), and a *controller* expressed as a *finite state machine* (`fsm`) where one or more actions may be executed in each state transition. This model is called a *finite state machine with datapath* (FSMD). Figure 1 shows the structure of a FSMD, and Figure 2 shows the pattern for essential parts of a Gezel specification.

Overview and Build-up of the Semantics

The semantics provides definitions for modules. Modules are the building block for systems, and the concept is defined to facilitate hierarchical constructions. We provide three different terms for modules: basic module, composition of modules and top-level module.

A basic module is defined as a datapath and a controller. The controller is a finite state machine and the datapath consists of input- and output ports, signals, registers and a set of actions. An action is a single assignment program where registers, output ports and signals can be assigned a value. Composition of modules are basically modules composed in parallel (basic- or already composed modules) using single assignment programs to connect their ports. The top-level module basically describes the system as the module at the highest level in the hierarchical structure.

Verification of Specifications

Experiments with verification of specifications have been conducted using the MONA tool [7], UPPAAL [2, 6] and NuSMV [4]; so far the greatest results have been in using UPPAAL.

Each module in the semantics can be modelled in UPPAAL as a timed automata. Parallel composition of modules are simply parallel composition of timed automata in UPPAAL. The

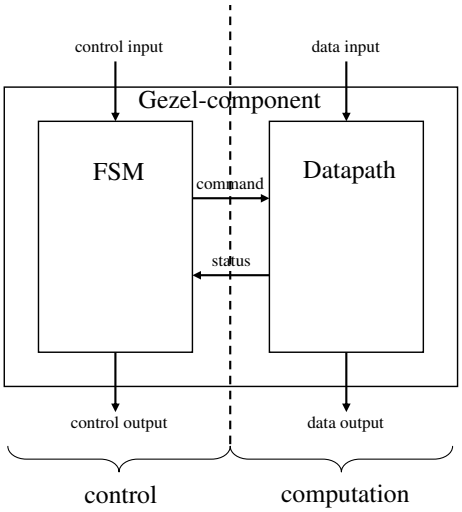


Figure 1: The FSMD model

(Datapath:

```
dp Name0(port list) {
  local register and signal  declarations
  sfg name1 { (non-branching)  actions }
  sfg name2 { (non-branching)  actions }
  ...
}
```

+

Control:

```
fsm controller_name(Name0) {
  initial state declaration
  auxiliary state declarations
  @state0 transition0
  @state1 transition1
  ...
}
```

) ... + Composition:

```
system id {
  Name0(n0,0,n0,1,...);
  Name1(n1,0,n1,1,...);
  ...
}
```

Composition in terms of *nets*
(implicitly introduced $n_{i,j}$)

Figure 2: Pattern for a Gezel program

greatest challenge in modelling the semantics in UPPAAL comes from assuring the correct sequence of execution of the single assignment programs. This is done by keeping track of which input ports and signals have been defined and assigning new values for output ports, signals and registers when all values required for the assignment have been defined.

Verifying properties of a system in UPPAAL is conducted using the UPPAAL requirement specification language. So far, functional properties, such as correct output, of systems as well as some structural properties, such as upper limit on clock cycles and register updates, have been expressed, and such verifications on examples have been explored.

Verification Results

We have conducted examples of verification on three high-level hardware designs. The three examples are specifications of two different greatest common divisor (gcd) algorithms (gcd1 and gcd2) and a specification of a SDES algorithm. Verification guarantees properties of the underlying algorithm, e.g. correct output for any given input, as well as other properties such as upper limits on the number of clock cycles for the algorithm to stabilize with a given input and upper limits on the number of register updates, to serve as an indicator of energy consumption.

The difference between the underlying gcd algorithms of the two specifications are that gcd1 uses simple subtraction to calculate the result, whereas gcd2 makes use of shift operations to make more efficient calculations.

For the gcd examples it was verified that both the specifications gave the correct 8-bit output given any two 8-bit inputs. This was done by testing the output against the result of an imperative implementation of a proven gcd algorithm implemented in a UPPAAL function for all possible combinations of inputs. Furthermore, it was verified that gcd1 calculated the greatest common divisor in maximally 511 clock cycles and 261 register changes, whereas gcd2 maximally used 26 clock cycles and 25 register changes in order to calculate the result. So as expected gcd2 was more efficient (i.e. used less clock cycles and register changes).

For the SDES example it was verified that for all combinations of 8-bit plaintexts and 10-bit keys, encryption followed by decryption gave the original plaintext. However, it was also verified that not all combinations of plaintexts and keys gave a different chiphertext than the original plaintext. This of course is a property of the underlying algorithm.

Each of the aforementioned Gezel specifications was verified in less than 25 minutes on a SUN Fire 3800 with 1.2GHz CPUs running Solaris 10. Until now, nothing in particular has been done to speed up the verification process.

Presented Topics

The focus of this talk will be on the ideas and concepts of a new hardware modelling language where the semantics is reflected in the syntax in a direct manner and specifying where Gezel fails to reflect the semantics. Also, making automated processes for steps from specification to verification of properties, using the semantics as a basis, will be given.

References

[1] R. Alur and D. L. D. Gerd. A theory of timed automata. *TCS*, 126(2):183–235, 2004.

[2] G. Behrmann, A. David, and K. G. Larsen. A tutorial on uppaal. *LNCS*, 3185:200–236, 2004.

[3] G. Berry. *The Foundations of Esterel*. MIT Press, 2000.

[4] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[5] Ieee. *Std 1076-2000: IEEE Standard VHDL Language Reference Manual*. IEEE, 2000.

[6] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[7] A. M. N. Klarlund. *Mona version 1.4: User manual*. BRICS, Department of Computer Science, University of Aarhus, Denmark, 2001.

[8] E. Schaefer. A simplified data encryption standard algotrihm. *Cryptologia 20(1)*, pages 77–84, 1996.

[9] P. Schaumont and I. Verbauwhede. Domain specific tools and methods for application in security processor design. *Design Automation for Embedded Systems 7*, pages 365–383, 2002.

[10] D. E. Thomas and P. R. Moorby. *The VERILOG Hardware Description Language*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

Test Driver Generation from Object-Oriented Interaction Traces

Frank S. de Boer[2] and Marcello M. Bonsangue[1] and
Andreas Grüner[3] and Martin Steffen[4]

[1] LIACS, Leiden, The Netherlands
[2] CWI Amsterdam, The Netherlands
[3] Christian-Albrechts-University Kiel, Germany
[4] University of Oslo, Norway

1 Introduction

Whereas object-orientation is established as a major paradigm for software development, testing methods specifically targeted towards object-oriented, class-based languages are less common. We propose a formal testing framework for object-oriented programs, based on the *observable trace semantics* of class components, i.e., for black-box testing. In particular, we propose a test specification language which allows to describe the behavior of the component under test in terms of the expected interaction *traces* between the component and the tester. The specification language is tailor-made for object-oriented thread-based programming languages like *Java* and *C#*, e.g., in that it reflects the nested call and return structure of thread-based interactions at the interface. From a given trace specification, a testing environment is *synthesized* such that component and environment represent an executable closed program.

The design of the specification language is a careful balance between two goals: using programming constructs in the style of the target language helps the programmer to specify the interaction without having to learn a completely new specification notation. On the other hand, *additional* expressions in the specification language which are usually not provided by the target language itself allow to specify the desired trace behavior in a concise, abstract way, hiding the intricacies of the required synchonization code at the lower-level programming language.

2 Test Driver Generation

Target Language We aim at a testing framework with a formal basis for testing software components written in object-oriented languages with synchronous message passing like *Java* or *C#*. A typed concurrent object calculus, based on impς introduced in [1] (see also [3]), and a corresponding open operational semantics for components[2] serve as a formal common ground for these languages. A component of the calculus consists of threads and classes. The semantics is given in two stages. The component-internal steps are defined without reference

to the environment. The external steps, in contrast, describe the component-environment interactions, i.e., interactions between the component and its environment. They are formalized by a labeled transition system, where the label of a transition represents the actual interaction.

Interface behavior A component-environment interaction is either a method call or a method return. A method call occurs at the interface if the callee is part of the component and the caller resides in the environment or vice versa. We call the first kind of method calls *incoming* and the latter *outgoing method calls*. An incoming method call results in an *outgoing return* (in case the method returns). The dual holds for outgoing calls.

The interface behavior of a component can be described by its set of interface traces. An interface, or interaction, trace of a component is the syntactical representation of a sequence of component-environment interactions that occur when executing the operational semantics. They are expressed by the corresponding sequence of transition labels.

Specification Language We propose a specification language for describing the desired interface behavior of a component. The language is designed under consideration of the following aspects:

- The behavior is formalized on the basis of interface traces. To this end, the language provides interactions statements for incoming and outgoing method calls and returns, which can be concatenated to describe an expected sequence of component-environment interactions.
- To allow specifications which correspond to possibly infinite sets of traces we add language constructs like variable declarations, conditionals, and tail recursion. To ease the use of the specification language for software developers, these language constructs are the same as in the target language.
- Certainly, there exist sequences of interactions that cannot be realized by any component. For instance, an incoming return cannot occur before the corresponding outgoing method call. The grammar and the type system of the specification language filters out most of these faulty specifications.

We give a semantics for the specification language which is similar to the semantics of the target language. However, the labeled transition system is different in that the possible incoming interactions are confined by the incoming interaction statements in the specification. Moreover, an important consequence of the above mentioned design decision is that we can specify a sequence of interaction statements of different threads such that the same order in the resulting trace is ensured.

Transformation We propose a transformation algorithm which generates target language code from a test specification for the methods of the tester classes. Basically, we had to tackle three main problems:

– To provide code which checks whether the component realizes an expected trace, we have to determine the possible orders of execution of interaction statements in the specification. We do this, by analyzing the possible control flow of the given specification.
– The tester should drive and observe the test, so that the component-tester interactions are realized in the specified order. However, our underlying theory shows that there always exist re-orderings which cannot be avoided but which are observable equal to the original order anyway[4]. We use a synchronization mechanism that ensures an order which is observable equal to the specified one.
– Unfortunately, we cannot identify all unrealizable specifications, statically. However, we can detect at runtime if in a certain situation the tester expects an impossible behavior of the component. In these cases, the tester stops the execution and reports a faulty specification.

3 Results

Specification language for a concurrent object calculus We formalize a specification language for object-oriented class-based concurrent components which specifies a component's behavior based on its interface trace and which, at the same time, has a similar look-and-feel as the target language. Moreover, we propose a transformation algorithm which synthesizes a tester program from a given specification, such that the tester and the component under test form an executable closed program.

Soundness of the transformation Our full research program also includes proofing soundness of the transformation in a later stage. This comprises

– *preservation of well-typedness* i.e., from a given well-typed specification, the transformation yields a well-typed tester program in the target language.
– *satisfaction* i.e., the traces accepted by the tester satisfy the specification.
– *detection of faulty specifications* i.e., irrealizable specifications are detected either statically or at run-time.

References

1. Abadi, M., Cardelli, L.: A Theory of Objects. Monographs in Computer Science. Springer-Verlag (1996)
2. Ábrahám, E., Grüner, A., Steffen, M.: Dynamic heap-abstraction for open, object-oriented systems with thread classes. Accepted to be published in SoSYM journal (2006).
3. Jeffrey, A., Rathke, J.: A fully abstract may testing semantics for concurrent objects. In: Proceedings of LICS '02. IEEE, Computer Society Press (2002)
4. Steffen, M.: Object-connectivity and observability for class-based, object-oriented languages. Habilitation thesis, Technische Faktultät der Christian-Albrechts-Universität zu Kiel (2006).

Test Purpose Directed Reactive Planning Tester for Nondeterministic Systems*

Jüri Vain[1], Kullo Raiend[2], Andres Kull[2], and Juhan P. Ernits[3]

[1] Dept. of Comp. Sci., Tallinn Univ. of Technology, Raja 15, 12618 Tallinn, Estonia
vain@ioc.ee
[2] Elvior, Mustamäe tee 44, 10621 Tallinn, Estonia
{kullo.raiend,andres.kull}@elvior.ee
[3] Institute of Cybernetics, Akadeemia 21, 12618 Tallinn, Estonia
juhan@cc.ioc.ee

Extended abstract

On-the-fly testing is widely considered to be the most appropriate technique for model-based testing of an implementation under test (IUT) modelled using nondeterministic models [7, 8]. We use the term *on-the-fly* to describe a test generation and execution algorithm that computes successive stimuli incrementally at runtime, directed by the test purpose and the observed outputs of the IUT.

The state-space explosion problem experienced by many offline test generation methods is reduced by the on-the-fly techniques because only a limited part of the state-space needs to be kept track of at any point in time. On the other hand, exhaustive planning is difficult on-the-fly due to the limitations of the available computational resources at the time of test execution.

The simplest approach to the selection of test stimuli is to apply the so called random walk strategy where no test sequence has an advantage over the others. It is inefficient because it is based on the random exploration of the state space and leads to test cases that are unreasonably long and nevertheless may leave the test purpose unachieved. To overcome this deficiency additional heuristics are applied for guiding the exploration of the state space [4, 9]. The other extreme of guiding is exhaustive planning by solving constraint systems at each step. For instance, the witness trace generated by model checking provides possibly optimal selection of the next test stimulus. The critical issue in the case of explicit state model checking algorithms is the size and complexity of the model leading to the explosion of the state space specially in cases such as "combination lock" or deep loops in the model [2].

In this paper we propose a balance between the tradeoffs of using a simple heuristic and the exhaustive planning methods for on-the-fly testing. We apply the principles of reactive planning to the problem of test planning under uncertainty. Reactive planning operates in a timely fashion and hence can cope with highly dynamic and unpredictable environments [10]. Just one subsequent input

* Full version of the paper has been accepted to the Conference of Automated Software Engineering, November 5-9, 2007, Atlanta, Georgia, USA.

is computed at every step, based on the current context. Instead of producing a complete test plan with branches (test tree), a set of decision rules is produced. We construct these rules by offline analysis based on the given IUT model and the test purpose.

The key assumption is that the IUT model is presented as an output observable nondeterministic state machine [5, 6] either in the form of an FSM or an EFSM in which all transition paths are feasible [1, 3]. From the IUT model we synthesise a reactive planning tester that is able to generate test inputs on-the-fly depending on the observed reactions of the IUT and the test purpose without having a preset test tree generated in advance. The proposed approach leads to a tester that directs the IUT efficiently towards the user-defined test purpose during test execution.

A test purpose is a specific objective or a property of the IUT that the tester is set out to test. We focus on test purposes that can be defined as a set of traps associated with the transitions of the IUT model [2]. The goal of the tester is to generate a test sequence so that all traps are visited at least once during the test.

We synthesise the tester as an EFSM where the rules for online planning derived during the tester synthesis are encoded into the transition guards of the EFSM. At each step only the rules associated with the outgoing transitions of the current state of the EFSM are evaluated to select the next transition with the highest gain. Thus, the number of rules that need to be evaluated at each step is relatively small.

The decision rules are constructed taking into account the reachability of all trap-equipped transitions from a given state and the length of the paths to them. Also, the current value (visited or not) of each trap is taken into account. The decision rules are derived by performing reachability analysis from the current state to all trap-equipped transitions by constructing the shortest path trees. The gain functions that are the terms of the decision rules are derived from the shortest path trees by simple rewrite rules.

The resulting tester drives the IUT from one state to the next by generating inputs and by observing the outputs of the IUT. When generating the next input the tester takes into account which traps have been visited in the model before. The execution of the decision rules at the time of the test execution is significantly faster than finding the efficient test path by state space exploration algorithms but nevertheless leads to the test sequence that is lengthwise close to optimal.

In the context of our examples, the reactive planning tester is more efficient at runtime than random choice and anti-ant algorithms. The planning feature of the reactive planner results in significantly shorter average test sequence lengths. The reactive planner outperforms the anti-ant algorithms in cases where more directed search is presumed, i.e. the test purpose covers the model partially.

The next step in evaluating the applicability of the reactive planning tester on further case studies requires an implementation of tool support that is part of the future work. In this work we made the assumption that all transition paths

in the EFSM are feasible but future work involves removing the feasible paths assumption.

Acknowledgements This work was partially supported by the Estonian Science Foundation under grant No 5775, by ELIKO Competence Centre project "Integration Platform for Development Tools of Embedded Systems", by the Estonian Doctoral School in Information and Communication Technology, and Tiigriülikool+ programme of the Estonian Information Technology Foundation.

References

1. A. Y. Duale and M. U. Uyar. A method enabling feasible conformance test sequence generation for EFSM models. *IEEE Trans. Comput.*, 53(5):614–627, 2004.
2. G. Hamon, L. de Moura, and J. Rushby. Generating efficient test sets with a model checker. In *2nd International Conference on Software Engineering and Formal Methods*, pages 261–270, Beijing, China, Sep 2004. IEEE Computer Society.
3. R. M. Hierons, T.-H. Kim, and H. Ural. On the testability of SDL specifications. *Comput. Networks*, 44(5):681–700, 2004.
4. H. Li and C. P. Lam. Using anti-ant-like agents to generate test threads from the UML diagrams. In *TestCom*, pages 69–80, 2005.
5. G. Luo, G. von Bochmann, and A. Petrenko. Test selection based on communicating nondeterministic finite-state machines using a generalized Wp-method. *IEEE Trans. Softw. Eng.*, 20(2):149–162, 1994.
6. P. H. Starke. *Abstract Automata*. North-Holland, Amsterdam: Elsevier, 1972.
7. M. Veanes, C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, and N. Tillmann. Model-based testing of object-oriented reactive systems with Spec Explorer, 2005. Tech. Rep. MSR-TR-2005-59, Microsoft Research.
8. M. Veanes, C. Campbell, and W. Schulte. Composition of model programs. In J. Derrick and J. Vain, editors, *FORTE*, volume 4574 of *Lecture Notes in Computer Science*, pages 128–142. Springer, 2007.
9. M. Veanes, P. Roy, and C. Campbell. Online testing with reinforcement learning. In K. Havelund, M. Núñez, G. Rosu, and B. Wolff, editors, *FATES/RV*, volume 4262 of *Lecture Notes in Computer Science*, pages 240–253. Springer, 2006.
10. B. C. Williams and P. P. Nayak. A reactive planner for a model-based executive. In *Proc. of 15th International Joint Conference on Artificial Intelligence, IJCAI*, pages 1178–1185, 1997.

Finite-State Call-Chain Abstractions for Deadlock Detection in Multithreaded Object-Oriented Languages

Frank S. de Boer[1], and Immo Grabe[2]

[1] CWI, Amsterdam, The Netherlands
[2] Christian-Albrechts-University Kiel, Germany

1 Introduction

Concurrent object-oriented programming languages are hard to analyse and verify, as the interleaving of activities leads to numerous different executions of the same program. We address a typical source of errors in such programs namely deadlocks. In a well known paper Coffman et al.[2] gave four necessary criteria for the occurrence of a deadlock (here adapted to an object-oriented setting):

mutual exclusion An object provides exclusive access to its internal state.

hold and wait While waiting to get the lock of one object a thread can hold the locks of other objects.

no preemption Threads cannot be forced to release exclusive access to an object.

circular waiting A circular chain of threads each waiting to get a lock the next thread in the chain holds.

To avoid deadlocks we have to identify program code which can lead to such a situation. In the presence of concurrency a mechanism for mutual exclusion is indispensable. The mechanisms for waiting and for preemption are language specific. Circular waiting, however, is program specific and thereby amenable to validation. We use program analysis techniques, in particular interprocedural analysis, to address the problem of deadlocks. For an introduction to program analysis see [3].

This paper deals with Java programs. The Java language provides a mechanism for mutual exclusion by locks of objects [1]. Code is declared to be accessed exclusively — using the lock mechanism — by the keyword *synchronized*. A thread that has gained access to synchronized code keeps the lock until the code is completed, and cannot be forced from the outside to release the lock. This explains the need to analyse whether a program is vulnerable to circular waiting, i.e., deadlocks.

2 Deadlock Detection Algorithm

We consider a Java program consisting of a finite number of classes, objects, and threads. We give several levels of abstractions to get a finite-state representation of the call-chains of the threads. We can exploit this representation by model checking to detect possible deadlocks. Our model is an overapproximation of the program which can lead to false negatives, i.e., deadlocks reachable only in the model and not in the program. We give a notion of refinement to reduce the likelihood of those false negatives.

We introduce for the sake of this paper so called *method automata* and *thread automata*. As a first step we abstract from the data and the object instances by construction of method automata representing the calling behaviour of the methods. Then we take those method automata to construct automata describing the behaviour of a single thread. By model checking deadlock freedom of these thread automata, we get the desired property or a counterexample representing an execution leading to a deadlock.

Method Automaton By giving a finite automaton for each method describing the calling behaviour of that method, we abstract from the concrete object instances and the internal steps of a method. An execution of a thread can be modelled as a push-down automaton moving through the method-automata. We assume the nodes of the method automata to be disjoint. By this we can relate every step of the push-down automaton to a particular method automaton in a particular class.

Call-Chain Automaton We characterize balanced call-chains by a context-free-language. We construct the call-chain automata from the method automata by stepwise extension. Initially the automaton consist only of the starting node gradually all reachable nodes are included. The nodes of the call-chain automaton represent either a call or a return and are labelled by tokens of our language. By this we can generalize the reachability problem to a context-free-language reachability problem following a formalism presented by Reps in [4]. We do not count the number of recursive calls in a run but allow an arbitrary number. By this abstraction we get rid of the obligation to maintain a history, e.g., a stack. On the other hand this introduces a source of imprecision.

Deadlock Detection To detect a deadlock we annotate the call-chain automata states with the set of locks the thread *may* own in this state. Without counting, there are situations where it is only possible to say that a lock may be held by a thread, not that the thread actually holds it. These sets can be derived from the call-chain automaton. Using the labelling of the call-chain automaton we derive the involved class from the method automata. A deadlock occurs if the circular waiting situation described above can evolve. We verify this by model checking deadlock freedom of the call-chain automata.

Refinement A counterexample to the deadlock freedom property is a path in the model leading to a deadlock. In general such a deadlock need not be a deadlock of the program. If we have found such a false negative we have to refine our model. We can do this by refining to more precise method automata by taking into account from which method they are called and reconstruction of the thread automata. These more refined thread automata can be constructed in the same way as in the initial case.

3 Results

We use an automata representation of the program to derive the automata representation of its behaviour. The chosen abstractions are sound.

Lemma 1 (Soundness of abstraction). *The abstraction is sound, i.e.:*

$$[\![t]\!]_{trace} \subseteq [\![A_{thread}]\!]_{trace}$$

where t is a thread of the program and A_{thread} is the call-chain automaton of t.

Our representation of the program behaviour is an overapproximation of the program behaviour. Proving a safety property of our model implies that the property also holds for the program.

Theorem 1 (Soundness of the algorithm). *The algorithm is sound, i.e.:*

$$DLF(A_{thread}) = true \Rightarrow \nexists\ e \in [\![P]\!]_{exec}.deadlock(e),$$

where DLF is a predicate stating deadlock freedom and e is an execution of program P.

We have given a finite program abstraction that is feasible for model checking. A notion of refinement is developed that allows us to scale the precision of our abstraction to the level needed for the verification. Using these techniques, we can prove the absence of deadlocks for a Java program.

References

1. Erika Ábrahám, Andreas Grüner, and Martin Steffen. Abstract interface behavior of object-oriented languages with monitors. In Roberto Gorrieri and Heike Wehrheim, editors, *FMOODS*, volume 4037 of *Lecture Notes in Computer Science*, pages 218–232. Springer, 2006.
2. Edward G. Coffman Jr., M. J. Elphick, and Arie Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, 1971.
3. Flemming Nielson, Hanne-Riis Nielson, and Chris L. Hankin. *Principles of Program Analysis*. 1999.
4. Thomas W. Reps. Program analysis via graph reachability. *Information & Software Technology*, 40(11-12):701–726, 1998.

Reasoning on Responsiveness – Extending CSP-Prover by the model R

D. Gift Samuel[1], Yoshinao Isobe[2], Markus Roggenbach[1*]

[1] University Swansea, United Kingdom
[2] AIST, Tsukuba, Japan

The process algebra Csp [4, 11, 15, 1] provides a well-established, theoretically thoroughly studied, and in industry often applied formalism for the modelling and verification of concurrent systems. Csp has been successfully applied in areas as varied as distributed databases, parallel algorithms, train control systems [3], fault-tolerant systems [2], and security protocols [15].

Fixing one syntax, Csp offers different semantical models, each of which is dedicated to special verification tasks. The traces model $\mathcal{T}$, e.g., covers *safety properties*. *Liveness properties* can be studied in more elaborate models. *Deadlock analysis*, e.g., is best carried out in the stable-failures model $\mathcal{F}$, the failures-divergences model $\mathcal{N}$ allows for *livelock analysis*. The analysis of *fairness properties* requires models based on infinite traces, see [11, 1] for further details.

Recently, the Csp realm of models has been extended by the newly designed *stable-revivals model* $\mathcal{R}$ [13]. On the practical side, the model $\mathcal{R}$ is appropriate to study *responsiveness* [12], which is a significant property in the context of component-based system design. From a theoretical point of view, the model $\mathcal{R}$ turns out to be fully abstract with respect to detecting when some system of processes can fail to make progress despite one or more of them having unfinished business with other(s). However, this comes at the price that certain algebraic properties fail to hold in the new model, among them the law $\sqcap$–$\Box$-distributivity $(P \,\Box\, Q) \sqcap R = (P \sqcap R) \,\Box\, (Q \sqcap R)$ is the most prominent example.

Csp-Prover [5, 8, 6, 7] is an interactive theorem prover for Csp based on Isabelle/HOL [10]. With its theorem proving approach Csp-Prover complements the established model checker FDR [9] as a proof tool for Csp. Csp-Prover is generic in the various Csp models, currently, it fully supports the traces model $\mathcal{T}$ and the stable-failures model $\mathcal{F}$.

Csp-Prover provides a deep encoding of Csp, and, consequently, also allows for meta theorems on the semantics implemented. Mistakes found, e.g., in the typing of the semantical functions of the predecessor of the model $\mathcal{N}$ [17, 16], or in the algebraic laws for the model $\mathcal{F}$ [6] demonstrates that, soon, presentations of similar models and axiom schemes will only be 'complete' once they have been accompanied by mechanised theorem proving [14].

In this paper, we report on an ongoing project of encoding the model $\mathcal{R}$ in Csp-Prover, see Figure 1. Csp-Prover provides a large re-usable part, e.g., theories on CMS and CPO, which provide techniques for dealing with recursive process definitions, or the Csp syntax. On top of this re-usable part, each Csp

* This work has been supported by the EPSRC Project EP/D037212/1.

model needs to be defined individually with its domain, its semantical functions, and its proof infrastructure, where – thanks to the close relation between the Csp models – sharing and re-use is possible. Encoding a Csp model includes two major parts, namely *specifying the model* and *proving properties* about this specification. Due to Isabelle's concept of conservative extensions, these two parts are often intertwined.

On the specification side, we first need to encode the domain of the model $\mathcal{R}$. Mathematically, this domain is a powerset reduced to 'healthy' elements. The notion of healthiness arises in a natural way, when the domain is interpreted as the collection of possible process observations. In Isabelle/HOL, we mirror this construction with a type definition. Already in this context, Isabelle/HOL requires the proof of properties, namely, that the newly defined types are non-empty.

The next step is to specify the refinement order, under which $\mathcal{R}$ forms a complete lattice. Here, we declare our domain to be an instance of the Isabelle/HOL type class 'order'. To this end we have to discharge the proof obligation that the inverse refinement order of $\mathcal{R}$ is a partial order. We also prove that the inverse refinement order forms a pointed cpo: this result allows us to re-use Csp-Prover's theory on CPO. As the model $\mathcal{R}$ is currently only defined in the CPO approach, we refrain from linking our domain to Csp-Prover's theory on CMS.

Next comes the specification of the semantical functions which map the Csp syntax to the domain. In Isabelle/HOL, these definitions come with proof obligations. Given, for example, healthy denotations of two processes P and Q, we have to prove that the denotation of the external choice $(P \,\Box\, Q)$ between P and Q is healthy as well. While Roscoe regards it in [13] as 'mechanical calculation' that all operators preserve $\mathcal{R}$'s healthiness conditions, these proofs turn out to be a real challenge in Isabelle/HOL.

The final step in order to instantiate Csp-Prover with the model $\mathcal{R}$ is to prove that all Csp operators are continuous w.r.t. the inverse refinement order. Given this continuity result, the semantics of recursive process definitions in Csp is defined via Tarski's fixed point theorem, which also offers proof support in the form of fixed point induction.

The proof infrastructure for the model $\mathcal{R}$ is based on the encoding described above. It consists of a collection of algebraic laws, which have been proven to be correct w.r.t. the encoding, as well as of a collection of tactics combining these laws.

Our instantiation of Csp-Prover with the model $\mathcal{R}$ as laid out in this abstract is nearly complete: there are open continuity proofs, and more proof infrastructure needs to be developed.

References

1. A. E. Abdallah, C. B. Jones, and J. W. Sanders, editors. *Communicating Sequential Processes, the first 25 years.* LNCS 3525. Springer, 2005.
2. B. Buth, J. Peleska, and H. Shi. Combining methods for the livelock analysis of a fault-tolerant system. In *AMAST'98*, LNCS 1548. Springer, 1998.

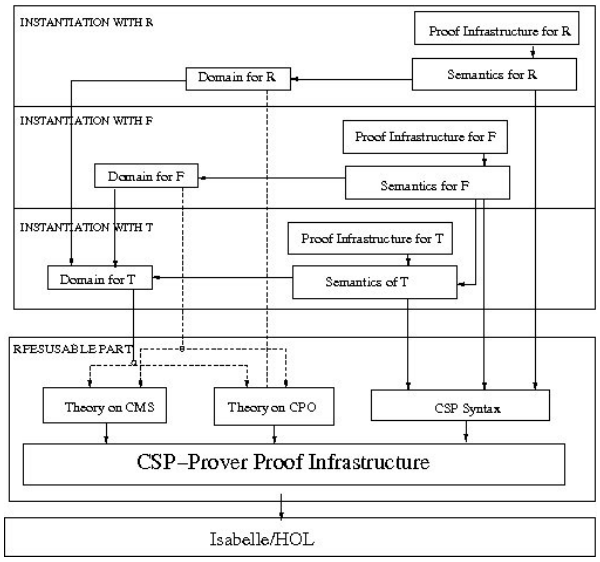


Fig. 1. Model R in Csp-Prover

3. B. Buth and M. Schrönen. Model-checking the architectural design of a fail-safe communication system for railway interlocking systems. In *FM'99*, LNCS 1709. Springer, 1999.
4. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
5. Y. Isobe and M. Roggenbach. A generic theorem prover of CSP refinement. In *TACAS 2005*, LNCS 3440. Springer, 2005.
6. Y. Isobe and M. Roggenbach. A complete axiomatic semantics for the csp stable failures model. In *CONCUR 2006*, LNCS 4137. Springer, 2006.
7. Y. Isobe and M. Roggenbach. Proof principles of CSP – CSP-Prover in practice. In *LDIC 2007*. Springer, 2007. To appear.
8. Y. Isobe, M. Roggenbach, and S. Gruner. Extending CSP-Prover by deadlock-analysis: Towards the verification of systolic arrays. In *FOSE 2005*, Japanese Lecture Notes Series 31. Kindai-kagaku-sha, 2005.
9. F. S. E. Limited. Failures-divergence refinement: FDR2. `http://www.fsel.com/`.
10. T. Nipkow, L. C. Paulon, and M. Wenzel. *Isabelle/HOL*. LNCS 2283. Springer, 2002.
11. A. Roscoe. *The theory and practice of concurrency*. Prentice Hall, 1998.
12. A. W. Roscoe, J. N. Reed, and J. E. Sinclai. Machine-verifiable responsiveness. AVOCS 2005, 2005.
13. B. Roscoe. Revivals, stuckness and responsiveness, 2005. Unpublished Draft. Available at `web.comlab.ox.ac.uk/oucl/work/bill.roscoe/publications/105.pdf`.
14. B. Roscoe, 2006. Private conversation.
15. P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. *The Modelling and Analysis of Security Protocols: the CSP Approach*. Addison-Wesley, 2001.
16. H. Tej. *HOL-CSP: Mechanised Formal Development of Concurrent Processes*. BISS Monograph Vol. 19. Logos Verlag Berlin, 2003.
17. H. Tej and B. Wolff. A corrected failure-divergence model for CSP in Isabelle/HOL. In *FME'97*, LNCS 1313. Springer, 1997.

Dependently Typed Array Programs Don't Go Wrong

Abstract

Kai Trojahner Clemens Grelck
University of Lübeck
Institute of Software Technology and Programming Languages
{trojahner,grelck}@isp.uni-luebeck.de

1. Introduction

The array programming paradigm adopts arrays as the fundamental data structures of computation. Such arrays may be vectors, matrices, tensors, or objects of even higher dimension. In particular, arrays may also be scalar values such as the integers which form the important special case of arrays without any axes. Array operations work on entire arrays instead of just single elements. This makes array programs highly expressive and introduces data-parallelism in a natural way. Hence, array programs lend themselves well for parallel execution on parallel computers such as recent multi-core processors (Sutter 2005; Grelck 2005). Prominent examples of array languages are APL (Iverson 1962), J (Iverson 1995), MATLAB (Moler et al. 1987), and, more recently, SAC (Grelck and Scholz 2006).

A powerful concept found in array programming languages is shape-generic programming: algorithms may be specified for arrays of arbitrary size and even an arbitrary number of axes. For example, array addition works for scalars as well as for vectors and matrices. However, this flexibility introduces some non-trivial constraints between function arguments: *Uniform array operations* such as element-wise addition require both their arguments to have exactly the same number of axes and the same size. The situation is even more complicated for operations like array selection: The length of the selection vector (a vector of indices) must equal the number of array axes and the *values* of said indices must range within the array bounds.

Interpreted array languages like APL, J, and MATLAB come with a large number of built-in operations. Each operation performs dynamic consistency checks on the structural properties of its arguments. In contrast, SAC is a compiled language aimed at providing utmost computing speed. Clearly, in this setting dynamic checks are undesirable since their repeated execution is costly in terms of run-time overhead. Moreover, uncertainty about whether or not program constraints are met hampers program optimization, leading to poor performance characteristics. Finally, run-time checks do not provide any confidence in program correctness since a program may run correctly for a long time before showing any erroneous behavior. For all these reasons, it is desirable to statically verify the various program constraints by means of formal methods.

Type systems are light-weight formal methods for program verification. A program that has been shown to be well-typed at compile-time is guaranteed not to exhibit certain undesired behavior at run-time. The amount of program errors ruled out by a type system depends on both the programming language to be checked and the expressive power of its types. A language is called type-safe if its operational semantics ensures that well-typedness is preserved by the reduction rules and that the evaluation of well-typed programs never gets into an undefined state in which no further reduction rule applies (Pierce 2002).

In this paper, we present a special type system for static verification of array programs. The system uses families of array types that do not only reflect the type of an array's elements but also contain an expression describing its shape. For specific arrays such as shape vectors, singleton types are employed to capture a vector's value in its type. The type system is based on a novel variant of dependent types: Vectors of integers are used to index into the family of array types. These vectors are themselves indexed from a sort family using an integer. Like in other approaches using indexed types such as DML (Xi and Pfenning 1999), type checking then proceeds by checking constraints on the index expressions, which is decidable. However, dealing with index vectors requires machinery beyond classical theorem provers for presburger arithmetic. To illustrate the approach, we introduce the nucleus of a programming language that allows for the convenient specification of higher-order shape-generic functional array programs. We aim at providing full type-safety, i.e. that all well-typed array programs will yield a value. In short: Dependently typed array programs don't go wrong!

2. A Foundation for Type-Safe Array Programming

The appropriate abstraction for treating the various kinds of arrays uniformly are true multidimensional arrays. As shown in Fig. 1, each array is characterized by its *rank* denoting the number of axes, and its *shape vector* describing the extent of each axis. Since the shape vector contains a natural number for each axis, the rank is implicitly encoded. The elements contained in an array are represented as a potentially empty sequence of *quarks* called the *data vector*. Quarks are the standard values of functional programming languages: constants, function values, and tuples (omitted in this abstract), but here they cannot exist without arrays. Together, the shape vector and the data vector form a unique representation of arrays. As shown in Fig. 2, array terms and quarks are defined mutually recursive.

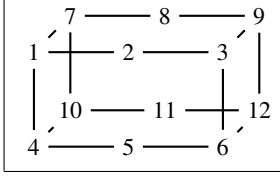
Array	Rank	Shape vector
1	0	[]
1 2 3	1	[3]
1 2 3 4 5 6	2	[2 3]
	3	[2 2 3]

Figure 1. Multidimensional arrays are characterized by their rank and shape vector.

For example, a 2×2 matrix of integers may be represented as $\langle (2,2), (1,2,3,4) : \texttt{int} \rangle$. Further examples are shown in Fig. 3. The type annotation describes the type of the quarks in the data vector. This is necessary to determine the type of the entire array if it does not contain any quarks. Resembling the relationship between quarks and array terms, there are quark types and array types. The array types of a given quark type form a type family in which we select specific types using a type index vector representing the array shape. The type of our above example would thus be $[\texttt{int}|(2,2)]$. Purely for reasons of readability, we may freely abbreviate the types of scalar arrays by writing Q instead of $[Q|()]$.

Using the abstraction quark $\lambda x.\, e$, we can define arrays of functions such as the identity function on scalar integers. Corresponding to the abstraction quark, there is also a quark type $T \to T$ representing function types. Since all array elements must have the same type, the type of the variable in the abstraction is unnecessary:

$$\langle (), (\lambda x.\, x) : \texttt{int} \to \texttt{int} \rangle : [\texttt{int} \to \texttt{int}|()].$$

Only scalar arrays of functions like the example above may be applied to an argument. We define β-reduction on arrays by syntactical substitution whose definition omitted in this abstract. Note that the following reduction does not take the type of the function argument into account:

$$\langle (), (\lambda x.\, t_1) : A \to B \rangle\ t_2 \to [x \mapsto t_2]t_1.$$

Finally, the index abstraction quark $\lambda' x.\, t$ serves to abstract a type index out of a term. Its type is the dependent function quark type $\Pi x : I.\, T$. The index variables may then be used to form index terms. However, like in other approaches using indexed types like DML (Xi and Pfenning 1999), the grammar of these index expressions is limited to retain decidability. The application of dependent functions $t\ 'i$ substitutes the index term i into both the types and terms of t:

$$\langle (), (\lambda' x.\, t) : \Pi x : I.\, T \rangle\ 'i \to [x \mapsto i]t.$$

I	::=	$\texttt{nat} \mid \texttt{natvec}(i) \mid \{I\ \texttt{in}\ ..i\}$	Index sorts
i	::=	$x \mid c \mid i \texttt{+} i \mid c^i \mid (c^*) \mid i \vec{+} i$	Index terms
	$\mid$	$i \texttt{++} i \mid \texttt{take}(i,i) \mid \texttt{drop}(i,i)$	
T	::=	$[Q\|i] \mid \texttt{num}(i) \mid \texttt{numvec}(i)$	Array types
Q	::=	$\texttt{int} \mid T \to T \mid \Pi x : I.\, T$	Quark types
t	::=	$\langle (c^*), (q^*) : Q \rangle \mid t\ t \mid t\ 'i$	Array terms
	$\mid$	$\texttt{gen}\ x\ \texttt{<}\ t\ \texttt{with}\ t \mid t\texttt{[}t\texttt{]}$	
q	::=	$c \mid \lambda x.\, e \mid \lambda' x.\, e$	Quarks
v	::=	$\langle (c^*), (q^*) : Q \rangle$	Array values

Figure 2. A simplified syntax for typed array programs

Array	Array expression
1	$\langle (), (1) : \texttt{int} \rangle$
1 2 3	$\langle (3), (1,2,3) : \texttt{int} \rangle$
1 2 3 4 5 6	$\langle (2,3), (1,2,3,4,5,6) : \texttt{int} \rangle$
	$\langle (2,2,3), (1,..,12) : \texttt{int} \rangle$

Figure 3. Arrays represented as array expressions.

Using dependent function types, we may, for the first time in the history of array programming, give an accurate type to the shape-generic addition operation for integer arrays of arbitrary but equal rank and shape:

$$\texttt{add} : \Pi r{:}\texttt{nat}.\, \Pi s{:}\texttt{natvec}(r).\, [\texttt{int}|s] \to [\texttt{int}|s] \to [\texttt{int}|s].$$

The two layers of index abstraction are typical for shape-generic array programs: we first abstract out the rank r to select the sort of the shape vector s from the sort family of index vectors. Then, the second Π type abstracts out the shape s which is used to index the array types.

The definition of the shape-generic array addition extends an addition operation $+$ for integer scalars to arbitrary arrays using a `gen` expression. To simplify the notation, we use a supercombinator notation that is equivalent to a nesting of scalar function definitions.

$$\texttt{add}\ 'd\ 's\ a\ b = \texttt{gen}\ x\ \texttt{<}\ \texttt{shape}\ a\ \texttt{with}\ a\texttt{[}x\texttt{]} + b\texttt{[}x\texttt{]}.$$

The expression $\texttt{gen}\ x\ \texttt{<}\ s\ \texttt{with}\ e$ is a simplified variant of the WITH-loop, the versatile array comprehension used in SAC. It takes a new identifier x, a non-negative vector of integers s, and a scalar-valued expression e that may contain x as a free variable. The expression evaluates to an array of shape s in which the element at index iv is computed by substituting x in e with iv. In the example, we use the built-in function `shape` to determine the shape of the result array based on the shape of the arguments. We may not simply use the index identifier s here because indices will not be evaluated at run-time. The selection operation $a\texttt{[}iv\texttt{]}$ takes an array a and a vector iv whose length equals the rank of a and whose value denotes a valid position within a.

Even for this simple example, verification requires many constraints between array ranks, shapes, and values to be checked:

1. `gen` requires `shape` a to have rank one,
2. `gen` demands the value of `shape` a to be non-negative,
3. $a\texttt{[}x\texttt{]}$ requires x to have rank one,
4. $a\texttt{[}x\texttt{]}$ requires the length of x to match the rank of a,
5. $a\texttt{[}x\texttt{]}$ requires a value of x between $\vec{0}$ and `shape` a,
6. $b\texttt{[}x\texttt{]}$ similar,
7. $+$ demands rank zero arguments,
8. `gen` expects $a\texttt{[}x\texttt{]} + b\texttt{[}x\texttt{]}$ to have rank zero,
9. to meet the specification, the result must have the same shape as a.

The array type $[Q|i]$ contains information about an array's shape, but not about its value. To capture the value of integer arrays, we use two families of singleton types. The members of $\texttt{num}(i)$ and $\texttt{numvec}(i)$ are integer scalars and integer vectors, re-

spectively. For example, $\texttt{numvec}((2,3))$ is the type of the value $\langle(2),(2,3) \texttt{ : int}\rangle$. Every $\texttt{num}(x)$ is also a $[\texttt{int}|()]$ and every $\texttt{numvec}(i)$ is also a $[\texttt{int}|(l)]$ where l denotes the sort index of the index vector i. In this context, `shape` maps array types to singleton types, e.g. in the example `add`, `shape` a gives a $\texttt{numvec}(s)$ because a is of type $[\texttt{int}|s]$.

Singleton types are employed whenever a value must be asserted to lie within a specific range. For example, the selection vector in the selection must be of type `numvec` to assert the selection is in-bounds. Furthermore, in an expression `gen` x `<` s `with` e, s must be a $\texttt{numvec}(s')$ such that the compiler can verify the new array is not meant to have negative extent. During type checking of e, x is then regarded as a $\texttt{numvec}(x')$ where x' is an index vector ranging between $\vec{0}$ and s'.

By merely classifying `shape` a and x as `numvec`, we have already found a proof for the constraints 1 and 3 in the above list. However, in general the constraints must be verified using a theorem prover for formulas in *presburger vector arithmetic*, an extension of presburger arithmetic to vectors of integers having arbitrary length. For example, constraint 2 requires to verify the following formula:

$$\forall d.\, d \geq 0 \Longrightarrow \forall s^d.\, s \geq_\wedge 0^d \Rightarrow s \geq_\wedge 0^d.$$

In this formula, s^d means that s is an arbitrary vector of length d. 0^d creates a vector of zeros of length d. Finally, $\geq_\wedge$ denotes the conjunctive extension of the relation symbol $\geq$, meaning $a \geq_\wedge b$ if $\forall i. a_i \geq b_i$. The other constraints encountered in `add` are resolved similarly.

A more complex example illustrating further type checking challenges is the generalized selection `gsel`. It serves to select not just a particular array element, but an entire subarray. For example, an individual row may be selected from a matrix by accessing the matrix with a selection vector of length one whose value must index within the rows of the matrix. For selection vectors of length zero, `gsel` acts as the identity function, whereas for full-length selection vectors it behaves like the regular selection.

$$\begin{array}{l}\texttt{gsel} : \Pi d{:}\texttt{nat}.\,\Pi s{:}\texttt{natvec}(d).\\ \qquad \Pi l{:}\{\texttt{nat in }..d{+}1\}.\,\Pi i{:}\{\texttt{natvec}(l)\texttt{ in }..\texttt{take}(l,s)\}.\\ \qquad [\texttt{int}|s] \to \texttt{numvec}(i) \to [\texttt{int}|\texttt{drop}(l,s)]\\ \texttt{gsel}\ {}'d\ {}'s\ {}'l\ {}'i\ a\ v = \texttt{gen}\ x < \texttt{drop}(\texttt{length}\ v, \texttt{shape}\ a)\\ \qquad\qquad \texttt{with}\ a\texttt{[}v\texttt{++}x\texttt{]}\end{array}$$

Using four index abstractions, the type of `gsel` concisely describes the constraints between the shape of the array and the length and value of the selection vector. The subset notation $\{I \texttt{ in } ..i\}$ allows to specify a strict upper bound on the value of type indices. The implementation exemplifies the three essential structural operations on vectors: the concatenation a`++`b appends vector b to vector a. $\texttt{take}(i,v)$ yields a new vector by selecting the first i elements from v, whereas $\texttt{drop}(i,v)$ discards those elements.

Due to their significance for shape-generic array programming, `take`, `drop` and `++` exist as built-in functions and as index terms. This means that the theorem prover must be able to verify constraints containing these three structural operations. As an example, we present the formula that must be verified in order to show that the selection $a\texttt{[}v\texttt{++}x\texttt{]}$ is valid:

$$\begin{array}{l}\forall d.\,\forall l.\, d \geq 0 \wedge l \geq 0 \wedge l < d+1 \Longrightarrow\\ \forall s^d.\,\forall i^l.\,\forall x^{d-l}.\, s \geq_\wedge 0^d \wedge i \geq_\wedge 0^l \wedge i <_\wedge \texttt{take}(l,s) \wedge\\ x \geq_\wedge 0^{d-l} \wedge x <_\wedge \texttt{drop}(l,s) \Rightarrow i\texttt{++}x \geq_\wedge 0^d \wedge i\texttt{++}x <_\wedge s.\end{array}$$

3. Conclusion

In this abstract, we have illustrated key ideas of a new system for type-safe array programming. By introducing quarks as a representation of array content, we achieved that every value in the language is an array. To gain type-safety, the system employs a new restricted variant of dependent types to express constraints between array ranks, shapes, and values. Type checking is then carried out by verifying properties in presburger vector arithmetic, an extension of conventional presburger arithmetic to integer vectors of arbitrary length.

Due to the space limitations of this abstract, we omitted some features of the language as well as its operational semantics, the typing rules and a proof of type-safety. In particular, the system presented contains no tuples or dependent pairs. These are essential for representing arrays of arrays and will also be discussed in a full article.

In the future we would like to extend the system towards polymorphic array programming to allow for more programming convenience. Similar to the work done by Xi (Xi and Pfenning 1999), we intend to allow type arguments to be automatically reconstructed if omitted.

References

C. Grelck. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming*, 15(3):353–401, 2005.

Clemens Grelck and Sven-Bodo Scholz. SAC: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.

K.E. Iverson. *A Programming Language*. John Wiley, New York City, New York, USA, 1962.

K.E. Iverson. *J Introduction and Dictionary*. Iverson Software Inc., Toronto, Canada, 1995.

C. Moler, J. Little, and S. Bangert. *Pro-Matlab User's Guide*. The MathWorks, Cochituate Place, 24 Prime Park Way, Natick, MA, USA, 1987.

B.C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0262162091.

H. Sutter. A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 2005.

H. Xi and F. Pfenning. Dependent Types in Practical Programming. In A. Aiken, editor, *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99), San Antonio, Texas, USA*, pages 214–227. ACM Press, 1999.

Note on a simple type system for non-interference

Steffen van Bakel and Maria Grazia Vigliotti

Department of Computing, Imperial College London,
180 Queen's Gate, London SW7 2BZ, UK

In recent years *secure information flow* has attracted a great deal of interest, spurred on by the spreading of mobile devices and nomadic computation, and has been studied in some depth for both programming languages and process calculi. In this paper we shall speak of the "language-based approach" when referring to programming languages and of the "process-algebraic approach" when referring to process calculi.

The language-based approach is concerned with the avoidance of secret information leakage or corruption through the execution of programs, i.e. with the security properties of *confidentiality* and *integrity*. The property of confidentiality, which appears to be the most studied, is usually formalised via the notion of *non-interference*, meaning that secret inputs should not have an effect on public outputs, since this could allow -in principle- a public user to reconstruct sensitive information. Non-interference may be achieved in various ways: via program analysis, type systems, semantics equivalencies, etc. In most cases the languages are equipped with a type system or some other tool to enforce the compliance of programs to the desired security property.

In the process-algebraic approach the focus is on the notion of *external observer*, who ideally has nothing to do with the specification and implementation of a given system, and should not be able to infer any secret by interacting with it. The process-algebraic approach is concerned with secret events not being revealed while processes communicate, i.e. actions that involve sensitive or confidential data should have no effect on public actions.

In process algebra, many non-interference properties are formalised in a way similar to programming languages, i.e. program analysis, type systems, semantics equivalencies. In the last few years a variety of properties have been proposed for process calculi, mostly based on trace equivalence or bisimulation, ranging from the simple property of *Non-deducibility on Composition* to more complicated ones (see [3] for a review). Methods for static detection of insecure processes have not been largely studied for process calculi. In [6,7] type systems which characterise a non-inference property have been proposed

[1] Email: svb@doc.ic.ac.uk, mgv98@doc.ic.ac.uk

for the π-calculus. More sophisticated type systems have been extensively studied in [8,9] for variants of the π-calculus, which combine the control of security with other correctness concerns. More recently, Crafa and Rossi proposed in [2] a simple security type system for the π-calculus, which consists essentially of a simplification of that used by Hennessy [7], ensuring the absence of explicit information flows. All those type systems include specific analysis on the values passed on a channel.

Pottier [11] proposed a very simple view on non-interference via a type system for the π-calculus which does not involve any extra typing information on the values passed over channels. The great appeal of this type system is its simplicity in characterising non-interference only; in fact, Pottier calls this system 'simple', and we will use his terminology in this work. The limitation of Pottier's work, with respect to the 'simple type system' is the lack of a robust semantic notion of non-interference. In this work we will address this issue specifically. The notion of *Persistent Non-deducibility on Composition* developed for CCS [5,4] has shown to be quite natural, also because it preserves the notion of non-interference of the language-based approach and in the process-algebraic approach [5]. Our work aims to show that the 'simple type system' can be adapted to standard CCS and that it characterises the semantic notion of Persistent Non-deducibility on Composition. This means that any typeable process is persistently deducible on composition. However, there exist processes that are considered secure according the notion of persistence, yet they are not typeable. Therefore, the set of typeable processes according to Pottier's type system is strictly smaller than the class of processes included in Persistent Non-deducibility on Composition relation.

In process algebraic approach, differently from language based security, no distinction is made between input events and output events, neither at the level of semantics definitions of security not at the level of type systems. In this work we aim to address this issue in the framework of a simple type system: that is to define a notion of non-interference which matches closely the one in the language-based approach. The basic idea is to view channels as information carriers, so that emitting a secret on an output channel can be considered safe, while inputting a secret may lead to some kind of leakage. Thus, we modify the 'simple type system' so that the notion of non-interference matches closely that of programming languages. That is to view channels as information carriers rather than as "events", so that the process $a_h(x).\overline{b_l}\langle e\rangle$, which emits on a low channel a value received on a high channel, is considered insecure, while $\overline{a_h}\langle v\rangle.\overline{b_l}\langle v\rangle$, which emits successively a value v on a high channel and on a low channel, is considered secure. The second example would not be be typeable in the 'simple type system' nor would it be considered secure with the standard semantic notions of non-interference.

We consider CCS here instead of the π-calculus because we wish to focus on the specific issues of non-interference in the simplest model possible. It is clear that our work could be easily extended to the π-calculus, with little

extra effort.

References

[1] G. Boudol and I. Castellani. Non-interference for Concurrent Programs and Thread Systems, *Theoretical Computer Science* 281(1): 109-130, 2002.

[2] S. Crafa and S. Rossi. A Theory of Non-interference for the π-calculus. In *Proceedings of Symposium on Trustworthy Global Computing '05*, volume 3705 of *LNCS*, Springer-Verlag, 2005.

[3] R. Focardi and R. Gorrieri. Classification of Security Properties (Part I: Information Flow). In *Foundations of Security Analysis and Design - Tutorial Lectures* (R. Focardi and R. Gorrieri, Eds.), volume 2171 of *LNCS*, Springer, 2001.

[4] R. Focardi and S. Rossi Information Flow Security in Dynamic Contexts In *Proceedings of of the IEEE Computer Security Foundations Workshop*, pages 307–319, EEE Computer Society Press, 2002.

[5] R. Focardi and S. Rossi and A. Sabelfeld. Bridging Language-Based and Process Calculi Security. In *Proceedings of FoSSaCs'05*, volume 3441 of *LNCS*, Springer-Verlag, 2005.

[6] M. Hennessy and J. Riely. Information flow vs resource access in the asynchronous π-calculus. *ACM TOPLAS* 24(5): 566-591, 2002.

[7] M. Hennessy. The security π-calculus and non-interference. *Journal of Logic and Algebraic Programming* 63(1): 3-34, 2004.

[8] K. Honda and V. Vasconcelos and N. Yoshida. Secure information flow as typed process behavior. In *Proceedings of ESOP'00*, volume 1782 of *LNCS*, pages 180-199. Springer-Verlag, 2000.

[9] K. Honda and N. Yoshida. A uniform type structure for secure information flow. In *Proceedings of P : (OP : (L'02*, pages 81-92. January, 2002.

[10] R. Milner. *Communication and Concurrency.* Prentice-Hall International, 1989.

[11] F. Pottier, A Simple View of Type-Secure Information Flow in the π-Calculus. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, pages 320–330, 2002.

[12] A. Sabelfeld and D. Sands. Probabilistic Non-interference for Multi-threaded Programs. In *Proceedings of 13th Computer Security Foundations Workshop*, IEEE, 2000.

[13] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes.* Cambridge University Press, 2001.

[14] D. Volpano and G. Smith and C. Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security* 4(3):167–187, 1996.

A Method for Parameterizing Type Systems over Relational Information

Tobias Gedell Daniel Hedin
{gedell, utter}@cs.chalmers.se
Chalmers University of Technology

September 3, 2007

The precision of many program analyses improves if certain properties about the analyzed program are known. Examples of such properties are aliasing information and nil pointer freeness. For instance, recently proposed information flow analyses use an integrated alias analysis to improve their precision. In these cases, the alias analysis is built in, and cannot easily be replaced, something which makes the resulting analyses hard to modify or extend.

In this work we present a method for parameterizing program analyses with relational information about the program execution.

The method is based on a definition of a general format for carrying the information — the *abstract environment mappings* and the accompanying semantic demand the *solution* — and probing the information — the *expression views*.

The definition of solutions and expression views is programming language dependent; once they are fixed for a given language they provide a sound base of relation information about the program execution.

To exemplify the use of the method, we show how it can be applied to a standard type system for the language to strengthen it to satisfy a modified progress property without changing the underlying semantics. Furthermore, we show how the same parameterization could be used to form a transformation of this language into a total variant of it, in order to establish progress for the original program.

By using the same parameterization to achieve the same thing using two different base analyses — the type system and the transformation – we are able to compare the two approaches. Once a clean interface to the additional information has been defined, the direct approach of parameterizing the type system seems to be beneficial over the transformation approach, given that the correctness of the transformation has to be established.

The result of parameterizing an analysis is an analysis that takes a number of decision procedures for the parameterized views and becomes a family of analyses indexed over the corresponding abstract environment mappings. The benefit of this method over, e.g. fusing the analysis into the type system, is that we can easily instantiate it with the result of different analyses; all that needs to be done is forming decision procedures for the parameterized views.

The method presents an easy and elegant way of increasing the precision of many program analyses — type systems being our main example. Furthermore, the modularity of the approach extends to the proof of correctness. The analysis is proved correct w.r.t. properties assumed of the generic format. Each instantiation is proved to satisfy those properties; in this direction, we show how the results of a class of abstract interpretations can be used to instantiate the parameterized analyses.

All proofs of this work with the exception of the proofs of the transformation have been formalized in the theorem prover Coq; in particular, we have formalized all details specific to the important technical development of the paper.

Relational Soundness and Optimality Proofs for Simple Partial Redundancy Elimination

Ando Saabas and Tarmo Uustalu
Institute of Cybernetics at Tallinn University of Technology
Akadeemia tee 21, EE-12618 Tallinn, Estonia; {ando|tarmo}@cs.ioc.ee

Introduction In a previous work of ours, we have argued that type systems are a compact and useful way of describing dataflow analyses [2, 6]. There are several benefits to them: type systems can explain analyses and optimizations well, they can be used for transformation of functional correctness proofs as soundness of the optimization is straightforward by structural induction on type derivations.

In this work in progress, we investigate the type-systematic approach further, and look at partial redundancy elimination, an optimization which is interesting in several aspects. Firstly, it is a highly non-trivial optimization, which performs code motion, i.e. changes the structure of the code. It is also important to show the optimality of the algorithm since it is not immediately obvious that the optimization does not reduce the runtime performance of a program due to code motion.

Partial redundancy elimination (PRE) is a widely used compiler optimization that eliminates computations that are redundant on some but not necessarily all paths in a program. As a consequence, it performs both code motion and global common subexpression elimination.

This optimization is notoriously tricky to perform and has been extensively studied since it was invented by Morel and Renvoise [4]. There is no single canonical algorithm for performing the optimization. Instead, there is a plethora of subtly different ones. The most straightforward algorithms by Xue and Knoop [3] and Paleri et al. [5] consist of four dataflow analyses.

In this paper, we look at a simplified version of PRE, which is more conservative in the sense that it does not eliminate all computations possible, but is more easily presentable, consisting of two dataflow analysis. All techniques we use here are also applicable for the full version of PRE.

The language we consider is WHILE. To simplify presentation, we consider expressions to be either variables, constants or expressions with a single operator. PRE seeks to avoid re-evaluations of operator expressions.

Type system for Simple PRE The algorithm comprises of two dataflow analyses, a backward anticipability analysis and a forward partial availability analysis which uses the results of the anticipability analysis. The anticipability analysis computes for each program point which expressions will be evaluated on all paths before any of their operands are modified. The partial availability analysis tells which expressions have already been evaluated on some paths at a program point where that expression was anticipable, and later not modified. This means that the partial availability analysis depends on the anticipability analysis. There are two possible optimizations for assignments. If we know that an expression is anticipable after an assignment, it means that the expression will definitely be evaluated later on in the program, so a new variable can be introduced, to carry the result of the evaluation. If we know that an expression is partially available at an assignment, we can assume it has already been computed and replace the expression with the new variable holding its value. If neither case holds, we leave the assignment unchanged. To make a partially available expression fully available, we must perform code

$$\frac{a \notin e \quad a \notin d \vee x \in FV(a)}{x := a : d\backslash\{a' \mid x \in FV(a')\} \cup \{a\}, e \longrightarrow d, e \cup \{a\}\backslash\{a' \mid x \in FV(a')\} \cap d \;\hookrightarrow x := a} :=_{1\text{pre}}$$

$$\frac{a \notin e \quad a \in d \quad x \notin FV(a)}{x := a : d\backslash\{a' \mid x \in FV(a')\} \cup \{a\}, e \longrightarrow d, e \cup \{a\}\backslash\{a' \mid x \in FV(a')\} \cap d \;\hookrightarrow nv(a) := a; x := nv(a)} :=_{2\text{pre}}$$

$$\frac{a \in e}{x := a : d\backslash\{a' \mid x \in FV(a')\} \cup \{a\}, e \longrightarrow d, e \cup \{a\}\backslash\{a' \mid x \in FV(a')\} \cap d \;\hookrightarrow x := nv(a)} :=_{3\text{pre}}$$

$$\frac{}{\mathsf{skip} : d, e \longrightarrow d, e \;\hookrightarrow \mathsf{skip}}\ \text{skip}_{\text{pre}} \qquad \frac{s_0 : d, e \longrightarrow d'', e'' \;\hookrightarrow s_0' \quad s_1 : d'', e'' \longrightarrow d', e' \;\hookrightarrow s_1'}{s_0; s_1 : d, e \longrightarrow d', e' \;\hookrightarrow s_0'; s_1'}\ \text{comp}_{\text{pre}}$$

$$\frac{s_t : d', e \longrightarrow d, e' \;\hookrightarrow s_t' \quad s_f : d', e \longrightarrow d, e' \;\hookrightarrow s_f'}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : d', e \longrightarrow d, e' \;\hookrightarrow \mathsf{if}\ b\ \mathsf{then}\ s_t'\ \mathsf{else}\ s_f'}\ \text{if}_{\text{pre}} \qquad \frac{s_t : d, e \longrightarrow d, e \;\hookrightarrow s_t'}{\mathsf{while}\ b\ \mathsf{do}\ s_t : d, e \longrightarrow d, e \;\hookrightarrow \mathsf{while}\ b\ \mathsf{do}\ s_t'}\ \text{while}_{\text{pre}}$$

$$\frac{d, e \leq d_0, e_0 \quad s : d_0, e_0 \longrightarrow d_0', e_0' \;\hookrightarrow s' \quad d_0', e_0' \leq d', e'}{s : d, e \longrightarrow d', e' \;\hookrightarrow \forall a \in e_0\backslash e.nv(a) = a; s'; \forall a \in e'\backslash e_0'.nv(a) = a}\ \text{conseq}_{\text{pre}}$$

Figure 1: Type system for latest analysis

motion, i.e. move evaluations of expressions to program points at which they are not partially available, but are partially available at the successor points.

We now present the two analyses as type systems. A type is a pair $(d, e) \in (\wp(\mathbf{AExp}) \times \wp(\mathbf{AExp}))$ satisfying the constraint $e \subseteq d$, where d is an anticipability type, and e is an partial availability type. Subtyping $\leq$ is set inclusion, i.e. $\subseteq$. Typing judgements are of the form $s : d', e \longrightarrow d, e'$, stating that if expressions in d are anticipable at the end of the program s, then the expressions in d' are anticipable in the beginning of the program and moreover, if the expressions in e are partially available in the beginning of the program, then the expressions in e' are partially available in the end of the program.

The type system is given in Figure 1 (ignore for now the gray boxes, which represent the optimization component of the type system). Anticipability is a backwards analysis, while partial availability a forward one. This is reflected in the type system, where a modified d holds in the pretype, while a modified e holds in the posttype for an assignment. The rest of the rules are standard.

The optimization component of the type system is colored gray in Figure 1. The introduction of new variables happens in two places, before assignments (if the necessary conditions are met) and at subtyping. An already computed value is used if an expression is partially available (rule $:=_{3\text{pre}}$). If it is not available, but is anticipable, and the assignment does not change the value of the expression, then the result is recorded in a new variable (rule $:=_{2\text{pre}}$). Code motion is performed by the the subtyping rule, which introduces new variable definitions when there is weakening or strengthening of types (this typically happens at the beginning of loops and at the end of conditional branches). The auxiliary function nv is used for generating a fresh variable from an expression.

Soundness (in the sense of preservation of semantics) of the optimized program can be shown using the relational method [1], by showing that the original and optimize program simulate each other wrt. to the equality relation $\sim$ on states, which is parameterized by the typing judgement that was the basis of the optimization.

Let $\sigma \sim_e \sigma'$ denote that two states σ and σ' agree on $e \subseteq \mathbf{AExp}$ in the following sense: $(\forall x \in \mathbf{Var}.\sigma(x) = \sigma'(x)) \wedge (\forall a \in e.[\![a]\!]\sigma = \sigma'(nv(a)))$. We have the following correctness theorem.

Theorem 1 (Correctness of simple PRE) *If $s : d', e \longrightarrow d, e' \hookrightarrow s_*$ and $\sigma \sim_e \sigma_*$, then*
— $\sigma \succ s \rightarrow \sigma'$ implies the existence of σ_*' such that $\sigma' \sim_{e'} \sigma_*'$ and $\sigma_* \succ s_* \rightarrow \sigma_*'$,
— $\sigma_* \succ s_* \rightarrow \sigma_*'$ implies the existence of σ' such that $\sigma' \sim_{e'} \sigma_*'$ and $\sigma \succ s \rightarrow \sigma'$.

The proof is by induction on the structure of the typing derivation.

It is possible to show more than just preservation of semantics using the relational method. One can also show that the optimization is actually an improvement in the sense that the number of evaluations of an expression on any given program path cannot increase. This means that no new computations can be introduced which are not used later on in the program. This is not obvious, since code motion might introduce unneeded evaluations.

To show this property, there must be a way to count the expression uses. This can be done via a simple instrumented semantics, which counts the number of evaluations of every expression. The semantic judgement is of the form $(\sigma, r) \succ s \rightarrow (\sigma', r')$, where σ and σ' are usual states, and r and r' are mappings from expressions to values, which show how many times a particular expression has been evaluated. The corresponding equivalence relation between the states is the following. Let $(\sigma, r) \sim_e (\sigma', r')$ denote that two states (σ, r) and (σ', r') agree on $e \subseteq \mathbf{AExp}$ in the following sense: $(\forall x \in \mathbf{Var}.\sigma(x) = \sigma'(x)) \wedge (\forall a \in e.[\![a]\!]\sigma = \sigma'(nv(a)))$. Furthermore $\forall a \notin e.r'(a) \leq r(a)$ and $\forall a \in e.r'(a) \leq r'(a) + 1$.

Here, partial availability types serve as an "amortization" mechanism. The intuitive meaning of an expression being in the type of a program point is that there will be a use of this expression somewhere in the future, where this expression will be replaced with a variable already holding its value. Thus it is possible that a computation path of an optimized program has one more evaluation of the expression before this point than the corresponding computation path of the original program due to an application of subsumption. This does not break the improvement argument, since the type increase at the subsumption point contains a promise that this evaluation will be taken advantage of ("amortized") in the future.

Theorem 2 (Improvement property of simple PRE) *If* $s : d', e \longrightarrow d, e' \hookrightarrow s_*$ *and* $\sigma \sim_e \sigma_*$, *then*
— $(\sigma, r) \succ s \rightarrow (\sigma', r')$ *implies the existence of* (σ'_*, r'_*) *such that* $(\sigma', r') \sim_{e'} (\sigma'_*, r'_*)$ *and* $(\sigma_*, r_*) \succ s_* \rightarrow (\sigma'_*, r'_*)$,
— $(\sigma_*, r_*) \succ s_* \rightarrow (\sigma'_*, r'_*)$ *implies the existence of* (σ', r') *such that* $(\sigma', r') \sim_{e'} (\sigma'_*, r'_*)$ *and* $(\sigma, r) \succ s \rightarrow (\sigma', r')$.

Again, proof is by induction on the structure of the type derivation.

To prove that an optimization is really optimal in the sense of achieving the best possible improvement (which simple PRE really is not), we would have to fix what kind of modifications of a given program we consider as possible transformation candidates (they should keep the control structure and only hoist and remove expression evaluations, they should not take advantage of the real semantics of expressions etc.). The argument would have to compare the optimization to other semantics-preserving transformation candidates.

Conclusion We have shown that the type-systematic approach to dataflow analyses scales up to complicated analyses such as partial redundancy elimination. Furthermore, using this approach, it is possible to prove properties beyond soundness, like optimality results. Soundness of the optimization makes it possible to transform a program's proof along the program guided by its analysis type derivation. While we have presented it on a simpler form of PRE, the same technique is applicable for full PRE.

Acknowledgement This work was supported by the Estonian Science Foundation grant no. 6940, the Estonian Doctoral School in ICT, the EU FP6 IST integrated project MOBIUS.

References

1. N. Benton. Simple relational correctness proofs for static analyses and program transformations, in *Proc. of 31st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL 2004*, pp. 14–25, ACM Press, 2004.

2. M. J. Frade, A. Saabas, T. Uustalu. Foundational certification of data-flow analyses. In *Proc. of 1st IEEE and IFIP Int. Symp on Theor. Aspects of Software Engineering, TASE 2007*, pp. 107–116. IEEE CS Press, 2007

3. J. Xue, J. Knoop. A fresh look at PRE as a maximum flow problem. In A. Mycroft, A. Zeller, eds., *Proc. of 15th Int. Conf. on Compiler Construction, CC 2006*, *Lect. Notes in Comput. Sci.*, v. 3923, pp. 139–154. Springer, 2006.

4. E. Morel, C. Renvoise. Global optimization by suppression of partial redundancies. *Commun. of the ACM*, v. 22, n. 2, pp. 96–103, 1979.

5. V. K. Paleri, Y. N. Srikant, P. Shankar. Partial redundancy elimination: a simple, pragmatic, and provably correct algorithm. *Sci. of Comput. Program.*, v. 48, n. 1, pp. 1–20, 2003.

6. A. Saabas, T. Uustalu. Program and proof optimizations with type systems. Submitted to *J. of Logic and Algebraic Program.*

Gradual Release: Unifying Declassification, Encryption and Key Release Policies (Extended Abstract)

Aslan Askarov Andrei Sabelfeld
Department of Computer Science and Engineering
Chalmers University of Technology
412 96 Göteborg, Sweden

Full version of this paper has appeared in *Proceedings of the 2007 IEEE Symposium on Security and Privacy, Berkely/Oakland, California, May 2007.*

Information security [4] has a challenge to address: enabling information flow controls with expressive information release (or declassification) policies [4, 7, 5]. In a scenario of systems that operate on data with different sensitivity levels, the goal is to provide security assurance via restricting the information flow within the system. However, allowing no flow whatsoever from secret (*high*) inputs to public (*low*) outputs (as prescribed by *noninterference* [3]) is too restrictive because many systems deliberately declassify information from high to low.

Characterizing and enforcing declassification policies is the focus of an active area of research [5]. However, existing approaches tend to address selected aspects of information release, exposing the other aspects for possible attacks. It is striking that these approaches fall into two mostly separate categories: revelation-based (as in information purchase, aggregate computation, moves in a game, etc.) and encryption-based declassification (as in sending encrypted secrets over an untrusted network, storing passwords, etc.). It is essential that declassification policies support a combination of these categories: for example, a possibility to release the result of encryption should not be abused to release cleartext through the same declassification mechanism.

This paper introduces *gradual release*, a policy that unifies declassification, encryption, and key release policies. As we explain below, the latter is not only a useful feature, but also a vital component for connecting revelation-based and encryption-based declassification. We model an attacker's knowledge by the sets of possible secret inputs as functions of publicly observable outputs. The essence of gradual release is that this knowledge must remain constant between releases. Gradual release turns out to be a powerful foundation for release policies, which we demonstrate by formally connecting revelation-based and encryption-based declassification.

When it comes to handling encryption, there is a demand for expressing rich policies beyond declassification at the point of encryption. To this end, a desirable ingredient in declassification policies is reasoning about released keys. In bit commitment, premature revelation of the bit should be prevented by not releasing the secret key until necessary. In a media distribution scenario—when large media is distributed in encrypted form, and the key is supplied on the date of media release—early key release should be prevented. In addition, key release policies are important for *mental poker* [6, 2, 1] (for playing poker without a trusted third party), where the participants reveal their keys for each other at the end of the game, in order to prove that they were not cheating during the game. In this protocol too, it should not be possible to release secret keys prematurely or encrypt with a key that has already been released.

Gradual release allows for reasoning about newly generated and released keys. In fact, this combination turns out to be crucial for connecting revelation-based and encryption-based declassification. We show that gradual release for revelation-based declassification can be represented by a rewardingly simple encryption-based declassification: declassifying an expression corresponds to encrypting the expression with a fresh key and immediately releasing the key.

As a result, gradual release is, to the best of our knowledge, the first framework to unify revelation-based and encryption-based declassification policies. Furthermore, we show that gradual release can be provably enforced by security types and effects.

References

[1] A. Askarov and A. Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In *Proc. European Symp. on Research in Computer Security*,

volume 3679 of *LNCS*, pages 197–221. Springer-Verlag, Sept. 2005.

[2] J. Castellà-Roca, J. Domingo-Ferrer, A. Riera, and J. Borrell. Practical mental poker without a TTP based on homomorphic encryption. In *Progress in Cryptology-Indocrypt*, volume 2904 of *LNCS*, pages 280–294. Springer-Verlag, Dec. 2003.

[3] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, Apr. 1982.

[4] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[5] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Computer Security*, 2007. To appear.

[6] A. Shamir, R. Rivest, and L. Adleman. Mental poker. *Mathematical Gardner*, pages 37–43, 1981.

[7] S. Zdancewic. Challenges for information-flow security. In *Proc. Programming Language Interference and Dependence (PLID)*, Aug. 2004.

A Framework for Formal Reasoning about Distributed Webs of Trust

(Work in progress)

Fredrik Degerlund[1,2], Mats Neovius[1,2] and Kaisa Sere[2]

[1] Turku Centre for Computer Science (TUCS)
[2] Åbo Akademi University
Joukahainengatan 3-5
FIN-20520 Åbo, Finland
{Fredrik.Degerlund, Mats.Neovius, Kaisa.Sere}@abo.fi

1 Background concepts

Formal methods have emerged as a means of reasoning about computer programs in a mathematical-logical manner. One of the major branches is stepwise refinement, in which an abstract specification is gradually turned into a concrete program while using specific rules to prove each transformation step. The *Action Systems formalism* [1] is a well-studied example of such a correct-by-construction approach. Having its roots in Dijkstra's guarded command language [5] and the refinement calculus [2], it is based on a firm theoretical foundation. The formalism is suitable for development of parallel programs, and it has gradually been adapted to further usage scenarios such as context-aware systems [7, 8].

Even though traditional formal methods can be successfully used to prove that a piece of software does behave correctly from a mathematical point of view, there is no guarantee that individual human beings behave as expected by their peers. To solve this problem of partially social nature, the concept of *trust* has emerged as a cornerstone in *social computing* scenarios. Trust is a central concept in activities involving human input and action. Such services are, however, often developed without the use of mathematically rigorous formal methods. In a general purpose specification language such as the Action Systems formalism, trust scenarios can in theory be described and dealt with. However, this is in practice cumbersome, since trust relationships can be complex, and the developer would have to take a stand on how to represent trust, how it is updated, how trust-related criteria are handled etc.

To overcome these issues, Degerlund and Sere [3, 4] have proposed a *domain-specific* extension the Action Systems formalism, providing developers with mechanisms to express and reason about trust. These mechanisms involve a special *coordination language* allowing the developer to separate trust-related functionality from the rest of the system, and, thus, deal with the two issues separately. Mechanisms for providing trust as well as requiring a certain amount of trust are supported by the language. Trust values are expressed as triples consisting of belief, disbelief and ignorance values as suggested by Jøsang, and trust computations are typically, but not necessarily, performed by using Jøsang's subjective logic [6]. The language can be translated into traditional action systems, i.e., no new expressibility is provided. Instead, the focus lies entirely on providing convenient constructs facilitating the work of the system developer.

2 Towards distributed webs of trust

In our current research, we revise and extend the trust coordination language proposed by Degerlund and Sere. In their original work, *entities* (or *agents*) were considered omniscient with respect to the participants in the system and their trust values in other entities. This approach may constitute a bottleneck as to the scalability of the model. In practice, it is also not always reasonable to expect each entity to know everybody else's trust values for all other entities. In our revised framework, we enable reasoning about *distributed webs of trust* by mimicking the mechanisms of friendship and trust in traditional, non-computer implemented social scenarios. Distinct entities may have their private, or local, relationships to other entities, without everybody in the network knowing about the existence thereof, nor the specific trust values involved. We also no longer assume that all entities are known from the beginning and remain in the network during operation. Instead, new entities may dynamically appear, as well as leave unexpectedly.

From a technical point of view, we move away from a model where each node performing a trust computation requires access to the data of a trust relation graph covering the whole system. Instead, we allow for fully distributed computation of trust values, in which each node only needs access to trust data regarding its portion of the computation. Sharing of partial results between nodes is possible and necessary, in contrast to the original model in which each trust computation is performed by a single node in one step. To achieve dynamic handling of entities, each entity maintains its own set of known peers, as well as a set of trust values related to those entities. We also no longer assume that trust is represented as trust triples, but allow for application specific decisions regarding the matter. This enables the use of alternative means of expressing trust, as well as for the use of trust fusion algorithms that operate on other structures than trust triples. The separation of trust from the rest of the functionality is preserved in our model, as well as developed further, allowing for componentization.

3 Conclusions

Information technology is rapidly developing towards pervasive and ubiquitous applications assisting us in making our everyday decisions. As such, they need to be reliable. Technical reliability can already be achieved by traditional formal methods, but modelling human interaction demands new perspectives. We are developing a coordination language for use within the Action Systems formalism in order to mimic human perception of trust. This language allows for componentization by separating trust, as well as for reasoning about trust in a fully distributed manner. We believe that this will dramatically extend the applicable areas for mathematically verified applications, and that it will make us comfortable relying on microprocessors making critical decisions in our favour.

References

[1] R.J.R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In *PODC '83: Proc. of the second annual ACM symposium on Principles of distributed computing*, pages 131–142, New York, NY, USA, 1983. ACM Press.

[2] R.J.R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag New York, 1998.

[3] F. Degerlund and K. Sere. A framework for incorporating trust into the action systems formalism (work in progress). In L. Aceto and Anna Ingólfsdóttir, editors, *Proc. for the 18th Nordic Workshop on Programming Theory (NWPT'06)*, Reykjavík, Iceland, October 2006. Reykjavík University. Abstract.

[4] F. Degerlund and K. Sere. A framework for incorporating trust into formal systems development. In *Proc. of the 4th International Colloquium on Theoretical Aspects of Computing (ICTAC 2007)*. Springer, 2007. To appear.

[5] E. Dijkstra. *A Discipline of Programming*. Prentice Hall International, 1976.

[6] A. Jøsang. Artificial reasoning with subjective logic. In *Proc. of the 2nd Australian Workshop on Commonsense Reasoning*, 1997.

[7] M. Neovius, K. Sere, L. Yan, and M. Satpathy. A formal model of context-awareness and context-dependency. In *SEFN '06: Proc. of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 177–185, Washington, DC, USA, 2006. IEEE Computer Society.

[8] L. Yan and K. Sere. A formalism for context-aware mobile computing. In *ISPDC '04: Proc. of the Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, pages 14–21, Washington, DC, USA, 2004. IEEE Computer Society.

Controlling timing channels in multithreaded programs

Alejandro Russo
Department of Computer Science and Engineering
Chalmers University of Technology
412 96 Göteborg, Sweden

Information-flow analysis studies whether an attacker can obtain confidential information by observing how the input of a system affect its output. Information can be disclosed by different mechanisms or channels. This presentation follows the line of *language-based information-flow security* [SM03]. Information-flow analysis is typically performed by static program analysis. As a consequence, it is possible to guarantee *end-to-end* securities, as confidentiality, by just analyzing the whole code of a given system.

Confidentiality policies could be precisely characterized by using program semantics. Moreover, they can be provably enforced by traditional mechanisms as type systems. *Noninterference* is a well known end-to-end property of programs that expresses the freeness of flows from more secret security levels to less secret ones. In other words, a variation in the confidential input of a program does not produce any variation of its public outputs. The attacker model defines what the attacker can observe about the execution of programs. For the noninterference property, the attacker can only inspect the public input and output states.

Language-based information-flow techniques deal with mechanisms used by programming languages to convey information. These mechanisms include assignments and branching instructions. Confidentiality of data can be preserved if programs are free of illegal *explicit and implict* flows [DD77]. On one hand, explicit flows can leak information by assigning confidential values to public variables. For instance, the program $l := h$ leaks the secret value of h by assigning it directly to the public variable l. Implicit flows, on the other hand, can use control constructs in the language to leak information. As an example, the program $\texttt{if}\ h > 0\ \texttt{then}\ l := 1\ \texttt{else}\ l := 2$ leaks if $h > 0$ or not by using the construct $\texttt{if} - \texttt{then} - \texttt{else}$. Even though there is no direct assignment of secret values to public variables, the final value of l depends on the secret value h.

Besides explicit and implicit flows, programming languages can present other mechanisms to leak information that were not originally designed for that purpose. These kind of mechanism are referred as *covert channels*[Lam73]. For example, the execution time of a program, memory consumption, and concurrency features can be used to leak confidential information. This presentation describes techniques to deal with *covert channels* introduced by some concurrent features. More precisely, it proposes remedies for leaks produced by exploiting scheduler properties through the timing behavior of threads in order to modify how the public variables are updated. This covert channel is called *internal timing covert channel* [VS99] and is the main focus of the talk. The presentation is based on the work done in [Rus07] and consists on four parts that are briefly described as follows.

Securing Interaction between Threads and the Scheduler

Existing approaches to specifying and enforcing information-flow security often present non-standard semantics, lack of compositionality, inability to handle dynamic threads, scheduler dependence, and efficiency overhead for code that results from security-enforcing transformations. Particularly, Volpano and Smith propose a special primitive called `protect` in order to remove internal timing leaks. By definition, $\texttt{protect}(c)$ takes one atomic step in the semantics with the effect of executing c until termination. Internal timing leaks are removed if every computation that branches on secrets is wrapped by $\texttt{protect}()$ commands. However, implementing `protect` imposes a major challenge. We suggest a remedy for some of the described shortcomings and a framework that allows the implementation of a generalized version of `protect`. More precisely, it introduces a novel treatment of the interaction between threads and the scheduler. A permissive noninterference-like security specification and a security type system that provably enforces this specification are obtained as a result of such

interaction. The type system guarantees security for a wide class of schedulers and provides a flexible treatment of dynamic thread creation. The proposed techniques relies on the modification of the scheduler in the run-time environment. *This part is based on a paper accepted to the 19th IEEE Computer Security Foundations Workshop* [RS06a].

Security for Multithreaded Programs under Cooperative Scheduling

In some scenarios, the modification of the run-time environment might not be an acceptable requirement. In this light, we present a transformation that eliminates the need for `protect` under cooperative scheduling. In fact, no additional interactions, besides yielding control to a thread, are needed in order to avoid internal timing leaks. Variations in the transformation can enforce both termination-insensitive and termination-sensitive security specifications in a language with dynamic thread creation. *This part is based on a paper accepted to the Andrei Ershov International Conference on Perspectives of System Informatics* [RS06b].

Closing Internal Timing Channels by Transformation

For those scenarios where the scheduler is preemptive and behaves as round robin, we present a transformation that closes the internal timing channel for multithreaded programs. The transformation is based on spawning dedicated threads, whenever computations may affect secrets, and carefully synchronizing them. Moreover, the transformation only rejects programs that have symptoms of illegal flows inherent from sequential settings. *This part is based on a paper accepted to the 11th Annual Asian Computing Science Conference* [RHNS06].

A Library for Secure Multi-threaded Information Flow in Haskell

Recently, Li and Zdancewic have proposed an approach to provide information-flow security via a library rather than producing a new language from the scratch. They show how to implement such a library in Haskell. We propose an extension of Li and Zdancewic's library that provides information-flow security for multithreaded programs. The extension provides reference manipulation, a run-time mechanism to close internal timing leaks, and a flexible treatment of dynamic thread creation. In order to provide such features, the library combines some ideas presented in the previous parts together with some other ones taken from literature: type system with effects, singleton types, projection functions, cooperative round-robin schedulers, and type classes in Haskell. Moreover, an online-shopping case study has been implemented in order to evaluate the proposed techniques. The case study reveals that exploiting concurrency to leak secrets is feasible and dangerous in practice and shows how the library can help to avoid internal timing leaks. Up to the publication date, this is the first implemented tool to guarantee information-flow security in concurrent programs and the first implementation of a case study that involves concurrency and information-flow policies. *This part is based on a paper accepted to the 20th IEEE Computer Security Foundations Symposium* [TRH07].

References

[DD77] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[Lam73] B. W. Lampson. A note on the confinement problem. *Comm. of the ACM*, 16(10):613–615, October 1973.

[RHNS06] A. Russo, J. Hughes, D. Naumann, and A. Sabelfeld. Closing internal timing channels by transformation. In *Proc. Annual Asian Computing Science Conference*, LNCS, December 2006.

[RS06a] A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *Proc. IEEE Computer Security Foundations Workshop*, pages 177–189, July 2006.

[RS06b] A. Russo and A. Sabelfeld. Security for multithreaded programs under cooperative scheduling. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer-Verlag, June 2006.

[Rus07] A. Russo. Controlling timing channels in multithreaded programs. Licentiate Thesis, Chalmers University of Technology, Gothenburg, May 2007.

[SM03] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.

[TRH07] T. Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in haskell. In *Proc. IEEE Computer Security Foundations Symposium*, July 2007.

[VS99] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *J. Computer Security*, 7(2–3):231–253, November 1999.

UML state machines: Fairness Conditions specify the Event Pool *

Harald Fecher[†], Heiko Schmidt[‡] and Jens Schönborn[§]

Abstract

The communication between different instances of UML state machines is handled by using underlying event pools but the UML standard leaves the behavior completely unspecified. Thus, in general, liveness properties cannot be verified. We give semantics of Streett fairness constraints for the event pool and present an algorithm that turns a set of fairness constraints into an abstract event pool. Since common fairness suffers from the drawback that the time until something good happens may be unbounded the presented algorithm allows the modeler to specify such a bound. The resulting abstract event pool can be further refined, justified by an example where a priority scheme on events is introduced.

1 Introduction

The communication between different instances (objects) of UML state machines is handled by using underlying event pools: An object (caller) can call a method of another object (callee), thereby sending an event to the callee. This event is first received by an implicit, usually not modeled event pool of the callee. At a later point in time, the event is provided by the callee's event pool to the callee's state machine and then, triggers transitions or is discarded otherwise. The UML standard [4] leaves the behavior of the event pool completely unspecified. This flexible treatment has the disadvantage that, in general, liveness properties cannot be verified. Therefore, the modeler should be provided with a facility to restrict the event pool's behavior. We propose adding Streett fairness constraints [3] to the state machine. This allows the modeler to specify fairness constraints without modeling the event pool of the state machine, thereby following the philosophy of the UML standard that the event pool should be left as unspecified as possible. Furthermore, as stated in [1, 2], liveness is robust (independent of the granularity of transitions) and simple (abstraction of complicated time bounds), but it suffers from the drawback that the time until something good happens may be unbounded. Thus the modeler should be enabled to specify a bound for the maximum of time until something good must happen.

Example 1 *A model of a pedestrian crossing on a road controlled by a traffic light is depicted on the left of Figure 1. This model ensures mutual exclusion for the use of the road, but not liveness, i.e., that a request for green light will be serviced eventually. Fairness constraints are needed to avoid possible starvation, and for practical application, this fairness should be bounded.*

Contribution. In order to enable reasoning with liveness properties, we extend UML state machines by fairness constraints for the underlying event pools. We give semantics of unbounded (common) and bounded Streett fairness constraints for the event pool, which is not straightforward, since a non-empty event pool must always provide an event. We present an algorithm that turns a set of fairness constraints into an abstract event pool satisfying the fairness constraints by its structure. As a consequence, traces rejected by the fairness constraints no longer occur in the composition of event pool and state machine. The abstract event pool is modeled using a simplified variant of state machines with a straightforward semantics in terms of labeled transition systems. We argue, by means of an example where a priority scheme on the events is added, that this is feasible, because thereby the event pool can be refined later in the modeling process.

*This work is in part financially supported by the DFG project *FE 942/1-1*.

[†]hfecher@doc.ic.ac.uk

[‡]hsc@informatik.uni-kiel.de

[§]jes@informatik.uni-kiel.de

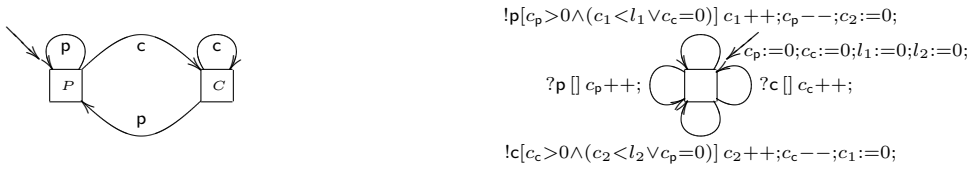


Figure 1: Left: A state machine model of a pedestrian crossing on a road controlled by a traffic light. Pedestrians request green light using a button triggering an event p, an inductor generates events c for cars. The states P and C represent green light for pedestrians resp. cars. These events are sent from the environment via the event pool to the state machine. Right: An abstract model of an event pool guaranteeing fairness between the events p, c in the sense that they cannot be neglected more than l_1-, resp. l_2-, times.

2 State machine and event pool with fairness

Fairness constraints on the state machines specify constraints on the event pool. For the definition of the event pool, we use a variant of state machines in order to enhance readability for software engineers. The semantics of this state machine variant is straightforward. Events sent to the state machine are first immediately received (input enabledness) by the event pool (?e) and will later be sent (provided) to the state machine (!e) via handshake communication. The sender of the event e needs to synchronize with ?e of the event pool and the directive !e of the event pool needs to synchronize with a transition having event e as label, or if no such transition exists, the event is discarded.

2.1 Bounded and unbounded fairness.

We generalize the common notion of fairness, expressed by Streett fairness constraints, such that also finite fairness constraints can be expressed, similar to [1, 2]. A set of fairness constraints of the form (E, F, l), where E and F are sets of events and $l \in \mathbb{N} \cup \{\infty\}$, specifies that at most l events from F may be sent to the state machine, before sending an event from E is required, provided such an action exists in the event pool. A trace t of the event pool, being a sequence of symbols ?e (receive) and !e (send), is accepted by a set of fairness constraints if for each fairness constraint (E, F, l) and for each index k in the trace there are no events from E currently stored in the event pool (i.e., the prefix of t until index k contains equally many occurrences of ?e and !e) or there is no immediately preceding subtrace with more than l occurrences of F send actions (!f with f $\in F$) without any occurrence of E send actions (!e with e $\in E$).

Note that bounded fairness specifications can introduce deadlock: Consider the fairness specifications $\{(\{e_1\}, \{e_2, e_3\}, 1), (\{e_3\}, \{e_1, e_2\}, 1)\}$ and the situation, where $!e_2$ has been dispatched and there are occurrences of all events in the event pool. The first tuple requires $!e_1$ to appear before a possible $!e_3$, whereas the second tuple requires $!e_3$ to appear before a possible $!e_1$.

The bound of a fairness constraint can be decreased via refinement. When unbounded fairness constraints are used, i.e. the bounds equal ∞, the resulting event pool is not yet concrete, and therefore the bound has to be decreased via refinement. A concrete event pool will provide, if not empty, exactly one event.

2.2 Transformation

We present the algorithm for generating the event pool from a given set of fairness constraints in Figure 2. The algorithm is divided into two parts: in the first loop on the set of events the basic I/O behavior is modeled and in the second loop on the set of fairness constraints the behavior is further restricted.

Example 2 *In Example 1, pedestrians' requests should not starve cars' requests and vice versa. This is specified by the fairness constraints* $\{(\{\mathsf{c}\}, \{\mathsf{p}\}, l_1), (\{\mathsf{p}\}, \{\mathsf{c}\}, l_2)\}$ *with reasonable values of* l_1, l_2 *which yields an event pool as depicted on the right of Figure 1.*

Define one state s, source and target of all transitions

1. For every event e of the state machine
   a) Define a counter c_e for the number of occurrences of e in the event pool
   b) Define a transition for receiving e labeled $?e[]c_e++;$
   c) Define a transition for sending e labeled $!e[c_e > 0]c_e--;$
2. For every fairness constraint $A_i = (E_i, F_i, l_i)$
   a) Define limit counter c_i
   b) For every $e \in F_i$, add to the transition for sending e
      i. Guard ... $\wedge (c_i < l_i \vee \bigwedge_{e' \in E_i} c_{e'} = 0)$
      ii. Action ...; $c_i++;$
   c) For every $e \in E_i$ add action $c_i := 0;$ to the transition for sending e

Figure 2: The algorithm for generating an abstract event pool from a given set of fairness constraints.

3 Adding priority via refinement

In this section we present an example which shows the feasibility of our approach since it admits further refinement, such as defining priority on events.

We have shown how (bounded) fairness can be realized in the event pool. However, it is also often useful to model priority on events. This can contradict existing fairness constraints, and to solve this, we give fairness constraints higher priority than priority specifications, i.e., a non-prioritized event is handled, in case this is required by a fairness constraint.

Example 3 *We extend the state machine from Example 1 as follows: An emergency state is added, which is requested by an approaching ambulance and turns all lights turn red. This emergency request (*e*) has higher priority than pedestrians' and cars' requests, being handled as soon as it appears. The state machine and its resulting event pool are presented in Figure 3. Here, priority is translated into a hierarchical nesting of state machine states, utilizing the fact that in UML transitions with sources of inner states have priority over transitions with sources of outer states.*

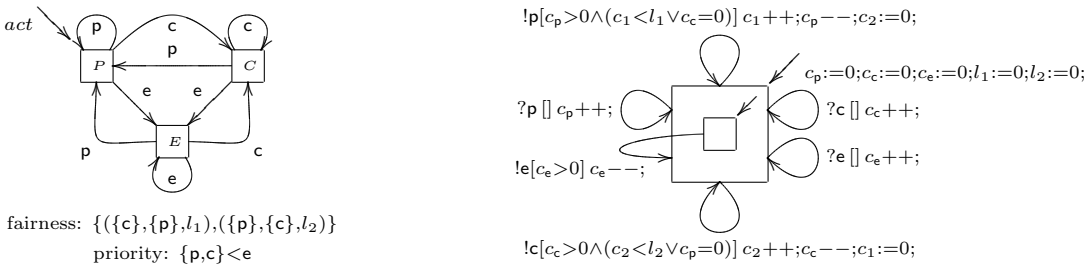


Figure 3: Left: Prioritized pedestrian crossing. A high-priority event e switches into an emergency mode that turns all lights red, issued, e.g., by an ambulance. Right: The resulting abstract event pool.

References

[1] K. Chatterjee and T. A. Henzinger. Finitary winning in omega-regular games. In H. Hermanns and J. Palsberg, editors, *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 257–271. Springer, 2006.

[2] N. Dershowitz, D. N. Jayasimha, and S. Park. Bounded fairness. In N. Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 304–317. Springer, 2003.

[3] E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research*, volume 2500 of *Lecture Notes in Computer Science*. Springer, 2002.

[4] Object Management Group. *UML Superstructure Specification, v2.1.1 formal/2007-02-03*, 2007.

Visualising program transformations in a stepwise manner

Marta Pląska[1], Marina Waldén[1] and Colin Snook[2]

[1] Åbo Akademi University/TUCS, Joukahaisenkatu 3-5A, 20520 Turku, Finland
[2] University of Southampton, Southampton, SO17 1BJ, UK

1. Introduction

For designing and developing complex, correct and reliable systems formal methods are the most beneficial approach. However, a formal methodology could be difficult for industry practitioners due to its mathematical notation. Hence it needs to be supported by more approachable platform, which would give guidance both for industry and research world representatives. The Unified Modelling Language (UML) with its semi-formal notation gives the intuitive image of the system and is commonly used within the industry. In the stepwise development of our system we combine UML with the formal methods approach and refinement patterns.

For a formal top-down approach we use the Event B formalism [11] and associated proof tool to develop the system and prove its correctness. Event-B is based on Action Systems [5] as well as the B Method [1]. It is related to B Action Systems [15] for developing distributed systems in B. With the Event-B formalism we have tool support for proving the correctness of the development. In order to translate UML models into Event B, the UML-B tool [13] is used. UML-B is a specialisation of UML that defines a formal modelling notation combining UML and B.

The first phase of the design approach is to state the goals that the system needs to satisfy. Afterwards the functional requirements of the system are specified using natural language and illustrated by various UML diagrams, such as statechart diagrams that depict the behaviour of the system. The system is built up gradually in small steps using superposition refinement [4, 10]. We strongly rely on patterns in the refinement process, since these are the cornerstones for creating *reusable* and *robust* software [3, 8]. UML diagrams and corresponding Event B code are developed for each step simultaneously. To get a better overview of the design process, we benefit from the *progress diagrams* [12] that illustrate only the refinement-affected parts of the system. These diagrams are based on well-known and widely-used UML statechart diagrams. In our previous research we introduced progress diagrams, which were illustrated by a case study. Here we focus on exploring refinement patterns in view of progress diagrams, as their combination supports constructing large software systems in an incremental and layered fashion. Moreover, this combination facilitates to master the complexity of the project and to reason about the properties of the system.

2. UML, Event-B and refinement patterns.

We use the Unified Modelling Language™ (UML) [6], as a way of modelling not only the architecture of a system, but also its behaviour and data structure. UML provides a graphical interface and documentation for every stage of the (formal) development process. We focus on the statechart diagram, as it shows the possible states of the object and the transitions between them.

In order to be able to reason formally about the abstract specification, we translate it to the formal language Event-B [11]. An Event-B specification consists of a model and its context that depict the dynamic and the static part of the specification, respectively. They are both identified by unique names. The context contains the sets and constants of the model with their properties and is accessed by the model through the SEES relationship [1]. The dynamic model, on the other hand, defines the state variables, as well as the operations on these. In order to be able to ensure the correctness of the system, the abstract model should be consistent and feasible [11]. Each transition of a statechart diagram is translated to an event in Event-B. The feasibility and the consistency of the specification are then proved using the Event-B prover tool [2].

It is convenient not to handle all the implementation issues at the same time, but to introduce details of the system to the specification in a stepwise manner. Stepwise refinement of a specification is supported by the Event-B formalism. In the refinement process an abstract specification *A* is transformed into a

more concrete and deterministic system *C* that preserves the functionality of *A*. We use the superposition refinement technique [4, 10, 15], where we add new functionality, i.e., new variables and substitutions on these, to a specification in a way that preserves the old behaviour.

In order to guide the refinement process and make it more controllable, *refinement patterns* [15] are used. In this paper we are focusing on data refinement and event refinement. The example of the former is shown in the statechart diagram in Fig 1b (splitting states into substates), while examples of the latter are given in Fig 1c (adding new transitions to use refined states) and Fig 1d (separating existing transitions), when the abstract specification is as in Fig. 1a. Furthermore, we can also consider a pattern for adding the same behaviour to several states (orthogonal regions [14]) as a type of data and event refinement. The pattern types are illustrated in more detail with *progress diagrams* [12] to give an abstraction and graphically-descriptive view documenting the applied patterns in each step.

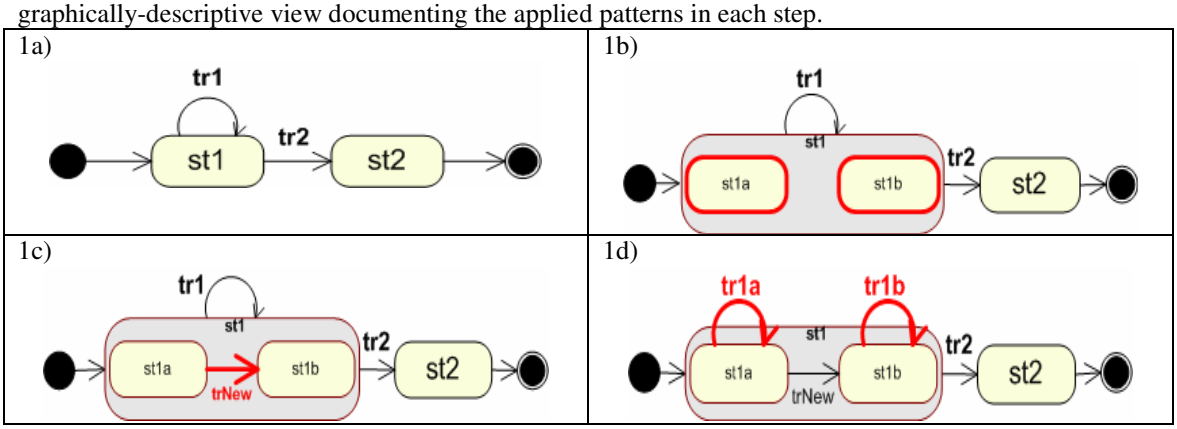


Fig. 1. Refinement patterns

3. Modelling refinement steps – progress diagrams

In most cases it is difficult to get an overview of the refinement process, since the size of the system grows during the development. Although refinement patterns help the designers and developers to refine the system, we combine them with the idea of progress diagrams [12]. The progress diagram is a table that is divided into a description part and a diagram part. The tabular part briefly describes the features and design patterns in the system of the current development step. It also depicts how states and transitions (initiated, refined or anticipated) are refined, as well as new variables that are added with respect to these features. The diagram part gives a supplementary view of the current refinement step and is in fact a fragment of the statechart diagram. Hence, with the progress diagrams we give a more detailed documentation of the refinement patterns and visualise the refinement step, showing the most important features and changes. Our approach results in a clear and legible graphical view of the system.

In order to illustrate the idea of *progress diagrams* in combination with *refinement patterns*, we use the abstract system (shown in Fig.1a) consisting of two states (*st1* and *st2*) and two transitions (*tr1* and *tr2*). We refine it in one step to the concrete system shown in Fig.1c, where the state (*st1*) is partitioned into substates (*st1a* and *st1b*) and the anticipating transition *trNew* is added between the new substates. The progress diagram of this sample *refinement step* is depicted in Fig.2. Here we also assume that a new variable *yy* is added to the refined system, as illustrated in the rightmost column of the progress diagram.

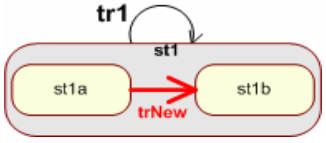
Description	States	Ref. States	Transitions	Ref. Transitions	New Var.
1st refinement step: • creating hierarchical substates (in state *st1*) – data ref. • adding new transition concerning the substates (*trNew*) – event ref.	*st1*	*st1a* *st1b*	-	*trNew*	yy
					

Fig. 2. Example of a progress diagram

4. Conclusion

This paper presents a compact approach to documentation of the stepwise system development focusing on the design patterns. Formal methods and verification techniques are used to ensure that a development is correct. Our approach uses the B Method as a formal framework and allows us to address modelling at different levels of abstraction. The use of progress diagrams during the incremental construction of large software systems provides legible and accessible documentation, whereas the refinement patterns facilitate the (complex) development process. We benefit from the progress diagrams, as we concentrate only on the refined part of the system. The combination of descriptive and visual approaches enables us to focus on the details we are most interested in. Furthermore, it helps to better understand the refinement steps and patterns used in the development, giving a clear overview of the development.

Design patterns in UML and B have been studied previously. Chan et al. [7] work on identifying patterns at the specification level, while we are interested in refinement patterns. The refinement approach on design patterns was presented by Ilič et al. [9]. They focused on using design patterns for integrating requirements into the system models via model transformation. This was done with strong support of the Model Driven Architecture methodology, which we do not consider in this paper. Instead we provide an overview of the development from the patterns.

Our approach is not only helpful for the developers, but also for those that later will try to reuse the exploited features of the system. A clear and compact form of progress diagrams in combination with refinement patterns is appropriate both for industry developers and researchers. Since progress diagrams do not involve any mathematical notation, they are useful for presenting the development steps to non-formal methods colleagues.

As future work tool support will be designed to generate a full refinement chain automatically from the initial specification and a series of progress diagrams in a stepwise manner. Although progress diagrams already appear to be a viable graphical view of the system development, further experimentation on complex case studies is envisaged leading to possible enhancements of the progress diagrams.

References

[1] J.R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] J.R. Abrial, S. Hallerstede, F. Metha, C. Metayer and L. Voisin. Specification of Basic Tools and Platform, RODIN Deliverable 3.3 (D10), http://rodin.cs.ncl.ac.uk/deliverables/rodinD10.pdf, 2005

[3] J. Arlow and I. Neustadt. *Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML.* Addison-Wesley, 2004.

[4] R.J.R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In: *Proc. of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 131-142, 1983.

[5] R.J.R. Back and K. Sere. From modular systems to action systems. *Software - Concepts and Tools,* 13, pp. 26-39, 1996.

[6] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language - a Reference Manual*. Addison-Wesley, 1998.

[7] E. Chan, K. Robinson and B. Welch. Patterns for B: Bridging Formal and Informal Development. In *Proc. of 7th International Conference of B Users (B2007): Formal Specification and Development in B*, LNCS 4355, pp. 125-139, 2007. Springer.

[8] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1995.

[9] D. Ilič and E. Troubitsyna. A Formal Model Driven Approach to Requirements Engineering. TUCS Technical Report No 667, Åbo Akademi University, Finland, February 2005.

[10] S.M. Katz. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337-356, April 1993.

[11] C. Metayer, J.R. Abrial and L. Voisin. *Event-B Language*, RODIN Deliverable 3.2 (D7), http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf (May 2005)

[12] M. Pląska, M. Waldén, C. Snook. Documenting the Progress of the System Development, In *Proc. of Workshop on Methods, Models and Tools for Fault Tolerance,* Oxford, UK, July 2007.

[13] C. Snook and M. Butler. U2B - a tool for translating UML-B models into B. In *UML-B Specification for Proven Embedded Systems Design*, chapter 5. Springer, 2004.

[14] C. Snook and M. Waldén. Refinement of Statemachines using Event B semantics. In *Proc. of 7th International Conference of B Users (B2007): Formal Specification and Development in B*, Besançon, France, LNCS 4355, January 2007, pp. 171-185. Springer.

[15] M. Waldén and K. Sere. Reasoning About Action Systems Using the B-Method. *Formal Methods in Systems Design 13(5-35),* 1998. Kluwer Academic Publishers.

Refinement, Compliance and Adherence of Policies in the Setting of UML Interactions

Bjørnar Solhaug[1,2] and Ketil Stølen[2,3]
[1]Dep. of Information Science and Media Studies, University of Bergen
[2]SINTEF ICT [3]Dep. of Informatics, University of Oslo
Email: {*bjors,kst*}*@sintef.no*

Abstract. The UML is the *de facto* standard for information system specification, but offers little support for policy specification. We introduce extensions to the sequence diagram notation suitable for expressing policy rules. A formal semantics is defined which gives a precise meaning to the specifications and allows the formalization of the notions of refinement and compliance of policies. The relation between a policy and systems for which the policy applies is formalized, defining what it means for a system to adhere to the policy.

1 Problem Characterization

Policy frameworks are increasingly being used for the management of information systems, and there exists a number of languages that have been developed for the purpose of policy specification and analysis. UML [4] is the *de facto* standard for the modeling and specification of information systems, but offers little support for the description of policies. To what extent can policies be expressed in UML, and in what way should policies be related to ordinary UML system documentation?

We define a policy as a set of rules governing the choices in the behavior of a system [5], where the rules are classified into *permissions*, *obligations* and *prohibitions*. This classification is based on deontic logic [3], and is found in several approaches to policy specification as well as in standards issued by ISO/IEC and ITU-T [2].

A policy refers to system behavior, so UML diagrams for specifying dynamic aspects of systems are natural candidates for policy specification. UML 2.0 sequence diagrams describe system behavior by showing how messages are sent between entities. We have chosen to focus on these since most of the dynamic UML diagrams can be expressed in terms of such interactions. This means that our results for sequence diagrams generalize to several other dynamic UML diagrams. The sequence diagram notation is, however, not a policy specification language [6]. A foundational task is therefore to introduce constructs to the syntax customized for expressing deontic constraints. A formal semantics should be provided to ensure a precise interpretation, and to pave the way for formal methods and analyses.

The process of developing a policy specification should be supported by a formal notion of *refinement*. Refinement means to add information to a specification,

bringing it closer to an implementation. The relation between a policy specification, which can be at any level of abstraction, and a policy implementation should be formally captured by a *compliance relation*. Compliance with a specification means that the implementation is correct, and with a formal compliance relation, this can be formally verified.

A core issue with respect to policy specification and analysis is to understand the relation between a policy specification and a system for which the policy applies. We specify systems with ordinary UML sequence diagrams where the semantics is the denotational trace semantics of STAIRS [1]. A formal *adherence relation* precisely defines what policy adherence of system to policy means.

2 Approach

We extend the sequence diagram notation with constructs for deontic rules. The extension is modest and conservative so that people that use the UML can easily understand and use the extended language. We define a formal semantics for the policy notation that builds on the STAIRS semantics for sequence diagrams. The interpretation of the deontic constraints is captured by defining a relation between the Kripke semantics of deontic logic [3] and the STAIRS trace semantics.

Given our language and formal semantics for policy specification, we define the notions of policy refinement and policy compliance. For policy specifications P_A and P_C, $P_A \rightarrow_r P_C$ denotes that the more concrete specification P_C is a refinement of the more abstract specification P_A. This is illustrated at the left hand side of Figure 1. The refinement relation is transitive, supporting a stepwise, incremental specification process by allowing any number of refinement steps between P_A and P_C. The refinement relation also supports compositional refinement, allowing a modular specification process.

A policy specification documents the requirements to the policy that eventually will be implemented. We also formalize a compliance relation '$\rightarrow_c$', where $P \rightarrow_c P_I$ denotes that the implementation P_I complies with the specification P. This is illustrated at the bottom left in Figure 1. The compliance relation is defined such that if $P_A \rightarrow_r P_C$ and $P_C \rightarrow_c P_I$, then $P_A \rightarrow_c P_I$.

An important part in our development of a policy specification language is to formalize the relation between a policy specification and a UML sequence diagram specification of the system for which the policy applies. What does it mean that a system specification adheres to a policy, and how is this verified?

The relation between the policy specification and the system specification is the adherence relation '$\rightarrow_a$' depicted horizontally in Figure 1 . By formally defining this relation, adherence to a policy is precisely defined and can be formally verified. Moreover, the refinement and compliance relations for the policy and system specifications can be utilized in both development and analysis by ensuring that adherence results propagate downwards through refinement and compliance.

The system specifications are depicted to the right in Figure 1 with the STAIRS

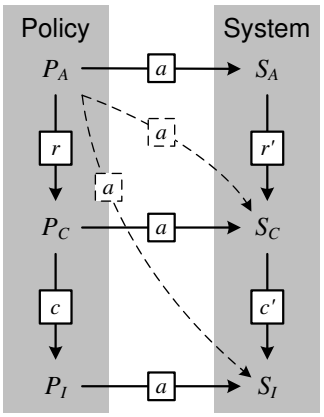


Figure 1: Formal relations

relations of refinement '$\rightarrow_{r'}$' and compliance '$\rightarrow_{c'}$'. Invariance of adherence through refinement and compliance is expressed by the following definition:

$$P_A \rightarrow_a S_A \stackrel{\text{def}}{\Longleftrightarrow} (\forall P_I : \forall S_I : P_A \rightarrow_c P_I \wedge S_A \rightarrow_{c'} S_I \Rightarrow P_I \rightarrow_a S_I)$$

This means that if policy adherence is established at an abstract level, refinement of policy specification and system specification independently will ensure adherence at the concrete level. The dashed, diagonal relations of Figure 1, for example, illustrates persistency of policy adherence through system refinement and compliance.

Policy specification with UML has not been much investigated despite the broad use of UML and the increased interest in policies. By some quite modest extensions of the syntax, accompanied by a formal semantics, useful methods can be developed for the support of specification and analysis. Policy specification using UML is highly interesting also because it facilitates the formalization and analysis of the relation between a policy and a system when the latter is documented in UML.

References

[1] Ø. Haugen, K. E. Husa, R. K. Runde, and K. Stølen. STAIRS towards formal design with sequence diagrams. *Journal of Software and Systems Modeling*, 4:355–367, 2005.

[2] ISO/IEC. *ISO/IEC FCD 15414, Information Technology - Open Distributed Processing - Reference Model - Enterprise Viewpoint*, 2000.

[3] P. McNamara. Deontic Logic. Stanford Encyclopedia of Philosophy, 2006. http://plato.stanford.edu/entries/logic-deontic/.

[4] Object Management Group. *Unified Modeling Language: Superstructure, Version 2.0*, 2005. www.omg.org.

[5] M. Sloman. Policy Driven Management for Distributed Systems. *Journal of Network and Systems Management*, 2:333–360, 1994.

[6] B. Solhaug, D. Elgesem, and K. Stølen. Specifying Policies Using UML Sequence Diagrams – An Evaluation Based on a Case Study. In *Proceedings of POLICY 2007*, pages 19–28. IEEE Computer Society, 2007.

Generalized Sketches and Model-driven Architecture

Adrian Rutle[1], Yngve Lamo[1] and Uwe Wolter[2]
[1]Bergen University College, NORWAY
[2]Dept. of Informatics, University of Bergen, NORWAY

ABSTRACT

Software developers increasingly recognize the need for portable, scalable and distributed applications which are reliable, secure and run at high performance. These applications are also required to be produced both faster and at a lower cost. An approach to achieve this is the Object Management Group's Model Driven Architecture which involves automatic development processes and model management in addition to executable models. This makes the importance of formal modeling languages more recognizable since the automatization of processes requires formal models. However, current modeling languages are either semi-formal, ambiguous, or both, therefore, a generic framework for formal, diagrammatic software specification is inevitable. In this presentation, Generalized Sketches will be introduced as a generic framework for the specification of modeling languages and transformations between them. Then an example will be presented to show how Generalized Sketches may be used in Model Driven Architecture.

Introduction and Motivation

Evidently, formalization of modeling languages and transformations between them is an important step in Model Driven Architecture (MDA) [7]. In contrast to the traditional software development processes where models are used only for documentation purposes, in MDA, models are considered first-class citizens. In MDA, building applications is started by the construction of abstract, platform-independent models (PIM) of system properties and behavior. These models are automatically transformed into one or more platform-specific models (PSM) which are used by code-generators to generate application code [6]. The transformation processes are specified by transformation definition languages and are executed by transformation tools. The involvement of these tools in the development processes requires that the models and the transformations between them should be defined formally. This implies the necessity of techniques which can be used to specify formal models and formal transformation definitions.

In addition, using formal modeling techniques provides mechanisms for model de-composition and integration; mechanisms for verification of correctness, consistency and validity of models and transformation definitions; as well as mechanisms for reasoning about models and transformations. Thus the basis for executable models will be a step nearer accomplishment.

Related Work

Currently, the Unified Modeling Language (UML) is the most used diagrammatic modeling language in software engineering. A huge effort is done by the Object Management Group (OMG) to formalize UML. This effort has resulted in special languages such as the Meta Object Facility (MOF) [9], which can be used to specify all other OMG languages; and the Queries, Views and Transformations (QVT) language which is used to specify transformations between MOF-compliant languages [8]. The QVT's approach of transformation requires that modeling languages are MOF-compliant in order to enable model transformation and integration. This approach might solve some of the problems concerning model definition and transformation, however, it is not the optimal solution for the problem because of the continuous development of modeling languages that are not necessarily MOF-compliant. In fact, MOF only provides the syntax for modeling languages, hence the semantics is still needed to be formalized.

Another language which is used for definition of transformations is the ATLAS Transformation Language (ATL) [5]. ATL is currently available as an open source project under the Eclipse Modeling subproject. The ATL framework consists of a transformation language (ATL), a virtual machine, and an IDE for writing transformation definitions. ATL is a hybrid language –both declarative and imperative. The declarative style is used to specify rules for matching source and target patterns. During application of the rules, a target pattern will be created in the target model whenever a source pattern is found. ATL presumes that the source and target languages are well-formed without providing mechanisms to check the well-formedness of the languages. ATL defines the semantics of transformation rules, however, this semantics is not graph-based.

Goals and Approaches

The purpose of our research is to investigate and apply the potentials of Category Theory (CT) and the ideas which propose Generalized Sketches (GS) as a mathematical formalism for formalization of modeling languages [4, 3, 2] and their transformations [10]. GS is a graph-based specification format which borrows its main ideas from both first order logic and categorical logic [2]. GS provides mechanisms for specification and manipulation of models in an abstract, generic and formal way.

In GS, a modeling language L is represented by a meta-sketch S_L which in turn is based on a diagrammatic signature Σ_L. A Σ_L-sketch can be seen as an instance of S_L,

$L-$models correspond to visualizations of Σ_L-sketches, and model transformations correspond to sketch operations. Σ_L is a collection of diagrammatic *predicate symbols*, and, each predicate symbol corresponds to a constraint or construct which can be set by L. Σ_L-sketches are categorical structures which consist of nodes, arrows and diagrams which are marked with predicate labels from Σ_L. By having a graph-based logic in which the arrow-thinking style is provided, the relationship between the syntax and the semantics of diagrams in Σ_L-sketches is made formal and more compact.

Heterogenous model transformations can be formalized in the following way. For the Language $L^{'}$, we can define the signature $\Sigma_{L'}$ and the meta-sketch $S_{L'}$. The transformation between constructs of L and $L^{'}$ can now be specified formally as a sketch operation which is given by a sketch morphism $\phi : S_L \rightarrow S_{L'}$ (Figure 1). Then a Σ_L-sketch, I_L, which is an instance of the meta-sketch S_L can be transformed to a $\Sigma_{L'}-$sketch, $I_{L'}$, by applying the sketch operation as shown in the figure and explained in the example below. The commutativity requirement of the diagram in the figure is to assure that the transformation is correct. For homogenous model transformation, we use the same procedure for $\Sigma_L = \Sigma_{L'}$.

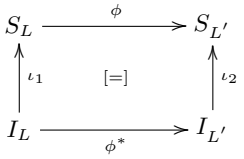


Figure 1: Generic Model Transformation

Example

A simplified meta-sketch, S_{PIM}, of a subset of a PIM language is shown in Figure 2. The figure explains the relations between classes and attributes. In this language, attributes must have exactly one kind of visibility: public, private or protected. This constraint is given by the total mapping, $attrVisibility$, and the curly braces including the possible visibility types, $\{pub, prv, prt\}$. Associations are omitted for briefness, and operations are not part of this PIM language.

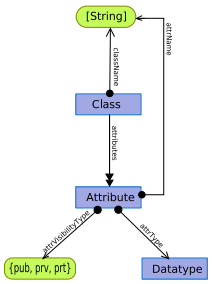


Figure 2: S_{PIM}, the meta-sketch of a PIM language

In Table 1, some constraints (or predicates) from Σ_{PIM} with their semantics in terms of sets and mappings are shown. Predicates can be combined to set the necessary constraints,

name	arity	visualization	semantic
"node"	•	A	set
[cover]	• → •	A –f→ B	$\forall b \in B : \exists a \in A \mid f(a) = b$
[total]	• → •	A •–f→ B	$\forall a \in A : \exists b \in B \mid f(a) = b$
[partial]	• → •	A ○–f→ B	$\exists a \in A \mid \nexists b \in B \mid f(a) = b$
[multivalued]	• → •	A –f↠ B	$\forall a \in A \mid a \in Dom(f) : f(a) \subseteq \mathrm{P}(B)$

Table 1: Signatures Σ_{PIM} and Σ_{PSM}

e.g. the mapping between *Class* and *Attribute* in S_{PIM} is marked with two predicates: *[cover]*, meaning that all attributes must exist as a field in some class, and *[multivalued]*, meaning that each class may have *0...** attributes.

Figure 3 shows a simplified meta-sketch, S_{PSM}, of a subset of a PSM language. Operations are part of this PSM language. There is a semantic overlap between the signatures of the PIM and the PSM languages, therefore, there is no need for a signature mapping. In most cases, especially when very different languages are concerned, a signature mapping is necessary to make the alignment of the two languages possible. A signature mapping is given by a signature morphism which is a mapping between predicate labels such that the shape graphs of the predicate labels are preserved.

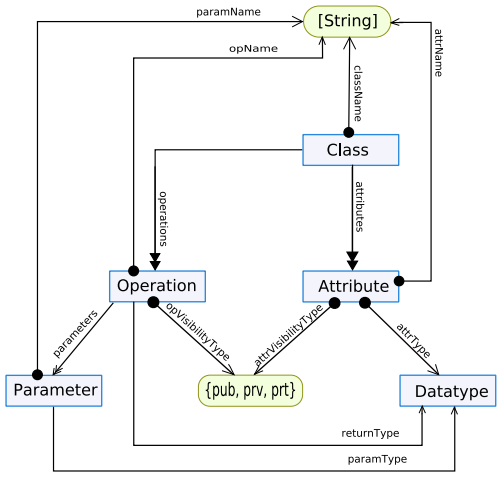


Figure 3: S_{PSM}, the meta-sketch of a PSM language

In Figure 4, we show a very simple transformation definition as it is defined by the GS formalism. This transformation is often used in the transformation of PIMs to PSMs, where a low level model –including operations– is generated from the high level model. For each public attribute *attr* from the PIM, we will generate a private attribute and two public operations –a getter and a setter– to access *attr*. In the transformation, we declare that for any match of the diagram SRC in the source model, the diagram TRG will be generated in the target model. This transformation can be specified diagrammatically and, we can put any kind of formal constraints on the transformation and its source and target models.

Examples of constraints which must be defined for this transformation are:

- $getterOp; returnType = attr; attrType$, stating that the return type of the getter of each attribute must be

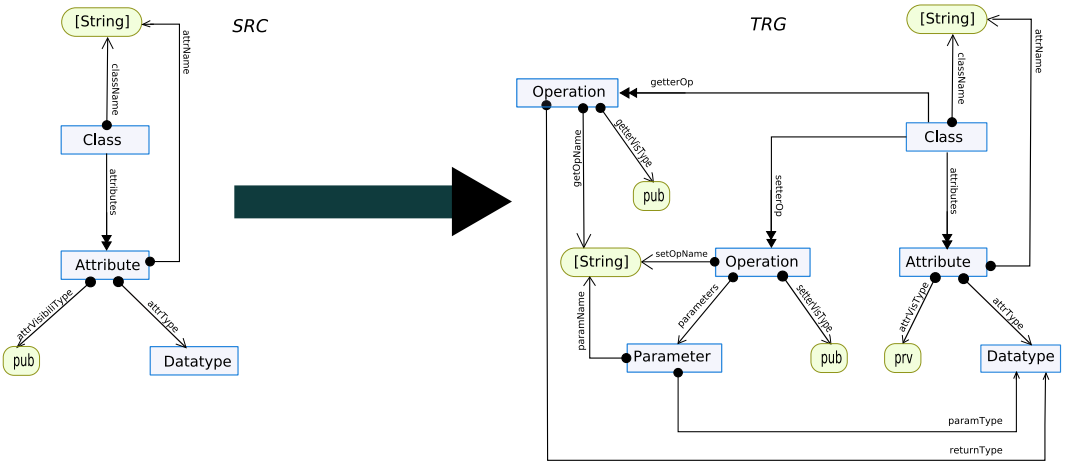


Figure 4: Transformation definition.

the same as the type of the attribute.

- $setterOp; param; paramType = attr; attrType$, stating that the type of the parameter of the setter of each attribute must be the same as the type of the attribute.
- $getterOp; op1Name = "get" + attr; attrName$, stating that the name of the getter of each attribute must be the same as the name of the attribute prefixed by "get".
- $setterOp; op2Name = "set" + attr; attrName$, stating that the name of the setter of each attribute must be the same as the name of the attribute prefixed by "set".

Concluding Remarks

The example shows how a transformation between two modeling languages (which are specified in GS) is defined. That kind transformation may be achieved also by using ATL or QVT. However, in GS, this process is both diagrammatic, allowing us to define the transformation visually; formal, allowing us to compose transformations and verify easily that the target model is an instance of the target metamodel; as well as language-independent, the transformation can be applied to the metamodel of any source language which has an occurrence of SRC, and any target language in which TRG can be expressed.

Currently, the focus of our research is on accomplishing the theories of GS and the design of tools for the application of these theories. Our tools will be implemented as plugins to Eclipse and will be proposed as a subproject of the Eclipse Modeling project [1]. The tools will be intended for three groups of users.

- Those who are interested in formalization of languages by designing diagrammatic signatures for those languages.
- Those who are interested in comparison and alignment of languages by definition of transformations between those languages.
- Those who are interested in using the signatures and transformations to define domain-specific models and then transform them to models in other modeling or programming languages, i.e. automatic code-generation.

1. REFERENCES

[1] *Eclipse Modeling subproject's Web site.* http://www.eclipse.org/modeling/.

[2] Z. Diskin and U. Wolter. Generalized Sketches: A Universal Logic for Diagrammatic Modeling in Software Engineering. *ENTCS*, 2007. Submitted.

[3] Zinovy Diskin. *Mathematics of UML: Making the Odysseys of UML less dramatic. Practical foundations of business system specifications*, chapter 8, pages 145–178. Kluwer Academic Publishers, 2003.

[4] Zinovy Diskin and Boris Kadish. Generalized sketches as an algebraic graph-based framework for semantic modeling and database design. Research Report M-97, Faculty of Physics and Mathematics, University of Latvia, August 1997.

[5] Frédéric Jouault and Ivan Kurtev. On the architectural alignment of atl and qvt. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1188–1195, New York, NY, USA, 2006. ACM Press.

[6] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: practice and promise.* Addison-Wesley, 1 edition, April 2003.

[7] Object Management Group (OMG). Moden Driven Architecture Guide. (1.0.1), june 2003. http://www.omg.org/cgi-bin/doc?omg/03-06-01.

[8] Object Management Group (OMG). MOF Query, Views and Transformations. Number 2.0, November 2005. http://www.omg.org/cgi-bin/doc?ptc/2005-11-01.

[9] Object Management Group (OMG). Meta Object Facility (MOF) Core Specificaiton. (2.0), january 2006. http://www.omg.org/cgi-bin/doc?formal/2006-01-01.

[10] Laura Rivero, Jorge Doorn, and Viviana Ferraggine. *Encyclopedia of Database Technologies and Applications*, chapter Mathematics of Generic Specifications for Meodel Management I and II, pages 351–366. Information Science Reference, 2005.

Algebraic and Coinductive Characterizations of Semantics Provide General Results for Free

David de Frutos Escrig, Carlos Gregorio Rodríguez
Department of Sistemas Informáticos y Computación
Universidad Complutense de Madrid
defrutos,cgr@sip.ucm.es

Abstract

There have been quite a number of proposals for behavioural equivalences for concurrent processes, and many of them are presented in van Glabbeek's linear time-branching time spectrum, where in particular an axiomatization for each one of these semantics is provided. Recently we have been able of presenting also a coinductive characterization of these semantics by means of *bisimulations up-to*, which give us the possibility to apply coinductive technique to their study. The most important property of these two approaches is their generality: once we have captured the essence of many different semantics in the same way we are able to get quite general results, which can be proved once for all, covering not only the semantics that have just been considered, but any other that could be defined in those frameworks having some quite general properties that are the only ones used when proving our results. We illustrate these asserts by means of a new and quite interesting result by Aceto, Fokkink and Ingolfsdottir, where they prove how to get for free an axiomatization of the equivalence induced by a behaviour preorder coarser than the ready simulation from that of the given preorder. We have got much simpler and shorter proofs of the main results in their paper that besides are valid not only for the semantics in the spectrum that they need to consider one by one, but for any other semantics fulfilling the quite general and simple properties that we need in our general proofs.

1 Short Extended Abstract

Most of the most popular semantics for concurrent processes appear in van Glabbeek´s ltbt. In its famous paper [1] all of them are characterized by means of adequate testing scenarios, that generate not only an equivalence relation between processes, but even more importantly, a natural preorder relation that has this equivalence relation as kernel. Bisimulation semantics is the strongest of the semantics in the spectrum, and it is proved to be much finer than simulation equivalence, since bisimulation having a pure coinductive definition is much stronger than mutual simulation. Ready simulation semantics appears as a nice compromise between them, and it has been proved that it is the strongest semantics having a large

collection of desirable properties. In particular, all the semantics in the spectrum that are weaker than it can be finitely axiomatizable, and this is true both for the equivalences and the preorders that define them. Besides, we have recently proved that also both admit coinductive characterizations based on either bisimulations up-to [2] or I-simulations up-to [3]. These two kind of characterizations have as main property that of providing uniform definitions instead of the quite different notions that are needed to define the extensional models of each of the semantics, take for instance traces, refusals, ready sets or the quite complex models for all the simulation semantics. It is true that in some cases these explicit models allow to foresee some interesting properties, that are either satisfied by a class of semantics, or even by any reasonable semantics, but if one needs to consider one by one all these models to get the corresponding formal proof of these results then things can become quite boring, and even a bit frustrating since one does not finally know which are the fundamental properties that justify the results.

Quite recently Aceto, Fokkink and Ingolfsdottir have given to us in [4] the perfect example to support our claims. They have showed how to get an axiomatization for the semantic equivalence defined by a given preorder from that of this one, thus showing a natural connection between both. But even if the construction seem to be quite general their proof is based in several results they only know how to prove considering the corresponding extensional semantics one by one. More in detail, they use the concept of *cover equation* in order to get a kind of basis for the sound equations under an axiomatization. The restriction to cover equations allows a (relatively) general proof for the central Theorem 1 in their paper, by means of proof induction. In order to prove that cover equations are indeed sufficient to prove the desired completeness of the defined axiomatization their main needed technical result is their Lemma 3, that says that whenever an inequation $t + x \leq u + x$ is sound, then the inequation $t \leq x$ is too. The proof of this lemma for the ten different semantics in the spectrum weaker than ready simulation takes up to twenty five pages of lengthy reasonings that are based on the extensional definitions of the semantics. Instead, our coinductive characterization of behaviour preorders coarser than the ready simulation preorder by means of simulations up-to the equivalences defined by them, allows us to get a general proof which extends for lees than half a page. Moreover by means of the study of these coinductive characterizations we have set into the light some very simple properties of reasonable semantics such as initials preserving and action factorised, which are the only ones we need in our proofs as hypothesis, and of course are satisfied by all the semantics in the spectrum.

But the algebraic characterizations of the semantics have also proved to be very useful when we studied the details of the rest of the proof of their Theorem 1. There the application of an inequational axiom in the derivation of any sound inequation can be made under an arbitrary context C, and this generates an structural induction on the form of the context. This is a natural descendent induction where the involved context is simplified by removing its root, and then the induction hypothesis is applied. The problem is that under the root of the context there are several sons, but its hole is only contained in one of them, and the rest of the sons have to be adequately taken into account. This is why in the original proof the cover equations were considered, because thanks to their simplicity the desired

result could be proved surrounding those technical difficulties. We tried to use the algebraic characterization of the semantics to get a new simpler proof of the theorem, and we found that the key idea was to reverse that structural induction, what means to enlarge the hole of the context to reduce the depth at which it appears. This is made by including into the hole the node over it and all its descendants. This produces a more elegant proof which avoids the necessity to restrict ourselves to cover equations, and whose main fundamental fact is a closure property of the simple contextual axiom $t \leq u \Longrightarrow b(t + x) \leq b(u + x)$ that generates the axiom $b(t + x) + b(u + x) \approx b(u + x)$, that Aceto et al. cleverly discovered as the key to construct the axiomatization of the equivalence induced by a behaviour preorder starting from the axiomatization of this preorder.

Let us conclude that the combination of the different characterizations of the semantics has proved to be very useful to get general proofs of their properties. Our new coinductive characterization of behaviour preorders and equivalences by means of (bi)simulations up-to has proved to be much more powerful than its simplicity made to suspect. At the same time their well known algebraic characterizations have found new applications thanks to a global study of the subject. This is why we advocate for the continuation of the study of all these topics that will serve us to get a much better knowledge of the complex structure relating all the proposed semantics for processes and their general properties.

References

[1] R. J. v. Glabbeek, Handbook of Process Algebra, Elsevier, 2001, Ch. The Linear Time – Branching Time Spectrum I: The Semantics of Concrete, Sequential Processes, pp. 3–99

[2] David de Frutos-Escrig and Carlos Gregorio-Rodríguez. Bisimulations up-to for the linear time-branching time spectrum. In *CONCUR 2005 - Concurrency Theory, 16th International Conference*, volume 3653 of *Lecture Notes in Computer Science*, pages 278–292. Springer, 2005.

[3] D. de Frutos-Escrig and C. Gregorio-Rodríguez, Simulations up-to and canonical preorders (extended abstract), in: Structural Operational Semantics SOS 2007, volume to appear of Electronic Notes in Theoretical Computer Science, Elsevier, 2007.

[4] L. Aceto, W. Fokkink, A. Ingólfsdóttir, Ready to preorder: get your BCCSP axiomatization for free!, in: Proc. 2nd Conference on Algebra and Coalgebra in Computer Science - CALCO'07, Vol. to appear of Lecture Notes in Computer Science, Springer, 2007.

Guarded and Mendler-Style Structured (Co)Recursion in Circular Proofs (Extended Abstract)

Tarmo Uustalu
Institute of Cybernetics at Tallinn University of Technology,
Akadeemia tee 21, EE-12618 Tallinn, Estonia; tarmo@cs.ioc.ee

Varmo Vene
Dept. of Computer Science, University of Tartu,
J. Liivi 2, EE-50409 Tartu, Estonia; varmo@cs.ut.ee

Designing a usable language for total functional programming based on structured (co)recursion is remarkably difficult. It amounts to designing a term calculus for a type language with inductive (least fixpoint) and coinductive (greatest fixedpoint) types that is satisfactory both metatheoretically and practically.

The straightforward option is to support (co)iteration or primitive (co)recursion in their basic forms. Term calculi with the corresponding constructs have an immediate semantic basis in the initial algebra (final coalgebra) semantics of (co)inductive types. Unfortunately, this option is impractical.

Alternatively, one might support general-recursor like combinators subject to syntactic conditions controlling the marshalling of recursive call arguments (the collection of corecursive call results), so-called guardedness conditions [6]. This approach is considerably more convenient for the programmer, but leads to a cumbersome and error-prone metatheory.

An approach combining the benefits of the basic structured recursion combinators and general recursor like combinators governed by syntactic conditions is so-called type-based termination, based on structured (co)recursors à la N. P. Mendler [8, 5, 10, 13, 7, 2, 1]. The general-recursor like behavior of such combinators is controlled by restrictive types employing universal quantification. The semantics combines initial algebras (final coalgebras) with (variants of) the Yoneda lemma and recursion schemes from comonads.

Common to both the approach of syntactic conditions and the Mendler-style approach is the issue of choosing how far to go: which structured (co)recursion scheme should be supported as primitive?

We argue that one possible canonical choice is given by a calculus of "circular proofs". Inspired by circular sequent calculi for classical predicate and modal logics with inductive and coinductive definitions [12, 11, 3], we look at an intuitionistic propositional sequent calculus where a proof is a rational derivation tree with every path satisfying a natural progress condition, cf. [4, 9]. An equivalent formulation is a higher-order sequent calculus where proof rules are higher-order but proofs are wellfounded. The progress condition can be stated either as a syntactic condition or à la Mendler. The (co)recursion scheme of circular proofs is an interesting form of lazily simultaneous (co)recursion: primitive (co)recursion with subsidiary simultaneous primitive (co)recursions on structurally smaller arguments. This scheme is of greater direct expressivity than the standard guarded by destructors (constructors) (co)recursion and Mendler-style course-of-value (co)recursion usually employed in the guarded and Mendler-style approaches.

Acknowledgements This work was supported by the Estonian Science Foundation grant no. 6940 and the EU FP6 IST coordination action TYPES.

References

1. A. Abel. Termination checking with types. *Theor. Inform. and Appl.*, 38(4):277–319, 2004
2. G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Math. Structures in Comput. Sci.*, 14(1):97–141, 2004.
3. J. Brotherston. Cyclic proofs for first-order logic with inductive definitions. In B. Beckert, ed., *Proc. of 14th Int. Conf. on Automated Reasoning with Analytic Tableaux and Related Methods, TABLEAUX 2005*, v. 3702 of *Lect. Notes in Artif. Intell.*, pp. 78–92. Springer-Verlag, 2005.
4. R. Cockett. Deforestation, program transformation, and cut-elimination. In A. Corradini, M. Lenisa, and U. Montanari, eds., *Proc. of 4th Wksh. on Coalgebraic Methods in Computer Science, CMCS '01*, v. 44(1) of *Electron. Notes in Theoret. Comput. Sci.*. Elsevier, 2001.
5. P. J. de Bruin. Inductive types in constructive languages. PhD thesis. Univ. of Groningen, 1990.
6. E. Giménez. Codifying guarded definitions with recursion schemes. In P. Dybjer and B. Nordström, eds., *Selected Papers from 2nd Int. Wksh. on Types for Proofs and Programs, TYPES'94*, v. 996 of *Lect. Notes in Comput. Sci.*, pp. 39–59. Springer-Verlag, 1995.
7. E. Giménez. Structural recursive definitions in type theory. In K. G. Larsen, S. Skyum, and G. Winskel, eds., *Proc. of 25th Int. Coll. on Automata, Languages and Programming, ICALP'98*, v. 1443 of *Lect. Notes in Comput. Sci.*, pp. 397–408. Springer-Verlag, 1998.
8. N. P. Mendler. Recursive types and type constraints in second-order lambda-calculus. In *Proc. of 2nd Ann. IEEE Symp. on Logic in Computer Science, LICS '87*, pp. 30–36. IEEE CS Press, 1987.
9. L. Santocanale. A calculus of circular proofs and its categorical semantics. In M. Nielsen and U. Engberg, eds., *Proc. of 5th Int. Conf. on Foundations of Software Science and Computation Structures, FoSSaCS 2002*, v. 2303 of *Lect. Notes in Comput. Sci.*, pp. 357–371. Springer-Verlag, 2002.
10. Z. Spławski. Proof-theoretic approach to inductive definitions in ML-like programming languages versus second-order lambda calculus. PhD thesis. Wrocław Univ., 1993.
11. C. Sprenger and M. Dam. On the structure of inductive reasoning: circular and tree-shaped proofs in the μ-calculus. In A. D. Gordon, ed., *Proc. of 6th Int. Conf. on Foundations of Software Science and Computation Structures, FoSSaCS 2003*, v. 2620 of *Lect. Notes in Comput. Sci.*, pp. 425–440. Springer-Verlag, 2003.
12. C. Stirling and D. Walker. Local model checking in the modal mu-calculus. *Theor. Comput. Sci.*, 89(1):161–177, 1991.
13. T. Uustalu and V. Vene. A cube of proof systems for the intuitionistic predicate μ, ν-logic. In M. Haveraaen and O. Owe, eds., *Selected Papers from 8th Nordic Wksh. on Programming Theory, NPWT '96*, pp. 237–246. Univ. of Oslo, 1997.

FLACOS 2007

The 1st Workshop on Formal Languages and
Analysis of Contract-Oriented Software

Compositionality and Compatibility of Service Contracts

Joseph C. Okika and Anders P. Ravn
Aalborg University, Denmark
{ojc, apr}@cs.aau.dk

1 Introduction

Service Oriented Architecture (SOA) is a way of reorganizing series of previously operational software applications and support infrastructure into an interconnected set of services, each accessible through standard interfaces and messaging protocols. Once all the elements of an enterprise architecture are in place, existing and future applications can access these services as necessary. This architectural approach is particularly applicable when multiple applications running on varied technologies and platforms need to communicate with each other. It promotes services that are distributed, heterogeneous, autonomous and open in nature. In this way, enterprises can mix and match services to perform business transactions with less programming effort.

In recent times, Service Oriented Architecture is being employed in developing service based applications. While this is a welcome idea by a versed majority of developers and vendors in the software industry, a lot of issues are yet to be resolved. For instance, collaboration presumes a minimum level of mutual trust and where ever trust is not considered sufficient, contracts become the alternative mechanism to reduce risks. Therefore, the possibility has arisen to have a detailed contract specification and of course verification for these service contracts in order to ensure a more reliable and dependable application. Two different web services from two different platforms or from two different organizations should be able to inter-operate based on the agreed contract.

Techniques and tools to handle service contracts or that support analysis, verification and validation is important. There should be a possibility to represent service contracts whether for intra/inter-organizational applications that allows the use of existing analysis and verification verification techniques. Our goal is to formulate a formal representation for the functional aspects of service contracts. Further, we investigate how to analyze and verify this service contracts with emphasis on compositionality and compatibility.

2 Service Contracts

A contract is a specification of the way a consumer of a service will interact with the service provider. A service contract specify a set of preconditions, post-conditions, quality of service levels for non-functional aspects. Contract is also a runtime dependency between the provider and the consumer. Contracts for web services including its specification and verification is of paramount importance in today's enterprise applications because of the high risks involved in using untrusted services from unknown providers. Since the provider of the service and the client that requests the service do not know each other in advance, the issue of trust cannot be guaranteed. Therefore, forming contracts becomes a way to manage the risks that comes with the interaction among these inter-organizational services. Service contracts may specify different aspects of a service. There are two major groups of these aspects.

- **Non-functional** Service contracts include descriptions of non-functional QoS (Quality of Service) requirements such as availability, accessibility, integrity, performance, reliability, security, among others. Most of these non-functional aspects are being investigated or covered by some standards [2, 1, 5]. Further, several works on formal approach employ Deontic logic as found in [8, 3, 4].
- **Functional** Service contracts should also include a description of functional requirements. An earlier treatment of contracts in an object-oriented paradigm is Design by Contract [6]. Here, the functional specification is achieved through assertions; which consists of preconditions, post-conditions and invariants.

We note that these functional specifications are important in order to express what a client should do to make a service request and what the provider will do in return. These functional aspects serve as a starting point for analysis and verification.

2.1 Representation of Functional Service Contracts (FSC)

In order to discuss about the compositionality and compatibility of service contracts, a formal representation of a functional service contract is needed.

$$FSC \quad \text{is a pair } SI : SP \tag{1}$$

SI is the set of service operations (SO) while SP are a set of sequences of operation names. Note that names occurring in sequences of SP are included in SI

Service Operations A service operation is a tuple:

$$SO \quad = \quad \langle pre_{SO}, post_{SO} \rangle \tag{2}$$

pre_{SO} : a precondition of the service is a logical expression which must be satisfied before the service can be called.

$post_{SO}$: postcondition is logical expression describing the effect/result of the invocation of the service. Thus, the service guarantees that it will satisfy postcondition if the precondition is true when called.

We define a function *sops* that retrieves all the service operations of a given service interface as:

$$sops \quad : \quad SI \rightarrow SO \tag{3}$$

Service Protocols We use a notation similar to regular expressions to describe the sequence of service operation calls in FSC. For instance, in a trading system, (details not given due to space constraints) startSale() should be called first, followed by some number of alternating calls to enterItem() and scanItem() and finishSale should be called last.

salsProt1: $(startSale()(\{enterItem()|scanItem()\})^{+}); finishSale()$

3 Compatibility and Compositionality

We consider compatibility and compositionality as steps in the analysis, and verification of service contracts. A service contract is compatible (operationally) if it is able to work with a given service. It focuses on the alignment of internal activities of participants with their contract obligations. Service Contract $SC1$ and $SC2$ each having all its SO satisfied are compatible when $SC1$ is at least as capable as $SC2$ and when $SC1$ can substitute $SC2$. We consider

this definition as a variant of refinement. Thus service contracts are compatible if there is a refinement relation between them. Note that $SC1 \cup SC2 \subseteq sops(SI)$.

The notion of refinement here is based on an ordering relation $\sqsupseteq$ on service contracts; `SC1` $\sqsupseteq$ `SC2`. `SC1` is a refinement of `SC2` if `SC1` can replace `SC2` in any service. That is `SC1` satisfies at least the same set of service operations as `SC2`. The refinement relation is reflexive and transitive. We employ the use of theorem proving to support the reasoning about service contracts in this context. One proof obligation is that refinement relation holds. We use the notion of parallel composition which allows new service contracts to be made from two (or more) service contracts to reason about the compositionality.

Semantics Given two functional service contracts $FSC1$ and $FSC2$, their parallel composition, $FSC1 \parallel FSC2$ is given structural operational semantics (or small step semantics from [7]) with the restriction that the sequences protocols are not altered. When composing service contracts, if each service satisfies (is compatible with) its service contracts, then the entire composite service contracts is satisfied. For instance, the resulting composite service contracts involves the execution of both $FSC1$ and $FSC2$ but the execution can be interleaved.

The functional aspects of service contracts (preconditions, post conditions and service protocols) are an integral part of the overall service contracts. Its formal specification enables a rigorous analysis and verification. Existing techniques for reasoning such as theorem proving can be employed to reason about compatibility of service contracts.

References

[1] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. *BPEL4WS, Business Process Execution Language for Web Services Version 1.1.* IBM, 2003.

[2] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. *Web Services Description Language (WSDL) 1.1.* W3C, 1.1 edition, March 2001. URL: http://www.w3c.org/TR/wsdl.

[3] Guido Governatori. Representing business contracts in *ruleml*. *International Journal of Cooperative Information Systems*, 14(2-3):181–216, 2005.

[4] Guido Governatori and Zoran Milosevic. A formal analysis of a business contract language. *International Journal of Cooperative Information Systems*, 15(4):659–685, 2006.

[5] Nickolas Kavantzas, David Burdett, Gregory Ritzinger, Tony Fletcher, and Yves Lafon. Web services choreography description language version 1.0. W3C working draft, W3C, December 2004. http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/.

[6] Bertrand Meyer. *Object-oriented software construction (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[7] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications: a formal introduction.* John Wiley & Sons, Inc., New York, NY, USA, 1992.

[8] Cristian Prisacariu and Gerardo Schneider. A formal language for electronic contracts. In Marcello Bonsangue and Einar Broch Johnsen, editors, *9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'07)*, volume 4468 of *Lecture Notes in Computer Science*, pages 174–189. Springer, June 2007.

Towards Model-Checking Contracts

Cristian Prisacariu Gerardo Schneider

Department of Informatics – University of Oslo,
P.O. Box 1080 Blindern, N-0316 Oslo, Norway.

{cristi, gerardo}@ifi.uio.no

Abstract

We understand by a *contract* a document written in natural language which engages several parties into a transaction, and which stipulates commitments (obligations, rights, and prohibitions) of the parties. Moreover the contract specifies also reparations in case of contract violation (i.e. some obligations or prohibitions are not respected). Because the human language is ambiguous by nature, contracts (written in plain English, for example) are inherent ambiguous. This ambiguity can, and many times is exploited by the parties involved in the contract. The purpose of our research is to eliminate this ambiguity as much as possible and to automate the process of designing, negotiation and monitoring of contracts. For this purpose contracts should be amenable to formal analysis (including model-checking) and thus should be written in a formal language.

There are currently several different approaches aiming at defining a formal language for contracts. Some works concentrate on the definition of contract taxonomies [1], while others look for formalizations based on logics (e.g. classical [4], modal [3], deontic [6] or defeasible logic [5]). Other formalizations are based on models of computation (e.g. FSMs [7], Petri Nets [2], or process calculi [12]). None of the above has reached enough maturity as to be considered *the* solution to the problems of formal definition of contracts. In our opinion, the most promising approach to formalizing contracts is the one based on logics of actions [13, 14] (i.e. actions found in contracts). It has been argued for the need to base deontic logic on a theory of actions which would solve many of the paradoxes deontic logic faces.

The method of *model-checking* is an old and established field of computer science. Model-checking has been applied in several fields of computer science from hardware circuits to concurrent programs. The idea of *model-checking electronic contracts* is extremely new and of great interest. From our knowledge there has been no attempt of using the classical model-checking techniques (or an established model-checking tool) on a real electronic contract example. Model-checking tools usually describe the system as an automata-like structure and the property to be checked in a temporal logic (like CTL, LTL, or μ-calculus). With the contract written in a formal language with semantics based on a Kripke-like structure we wold have the automaton input for the model-checking tools.

Our aim is to define a general framework for describing in a uniform way both the contract and the properties.

In [11] we have provided a formal language for writing contracts, which allows to write (conditional) obligations, permissions and prohibitions over (names of) human *actions* as well as reparations in case of violations. The language is specially tailored for representing statements found in contracts and is proven to avoid major deontic paradoxes (for motivations and design decisions we refer the reader to [11]). Here we briefly sketch and discuss the ideas behind the syntax of the contract language $\mathcal{CL}$.

$$\mathcal{C}ontract := \mathcal{D} \,;\, \mathcal{C}$$
$$\mathcal{C} := \phi \mid \mathcal{C}_O \mid \mathcal{C}_P \mid \mathcal{C}_F \mid \mathcal{C} \wedge \mathcal{C} \mid [\alpha]\mathcal{C} \mid \langle\alpha\rangle\mathcal{C} \mid \mathcal{C}\,\mathcal{U}\,\mathcal{C} \mid \bigcirc\mathcal{C} \mid \Box\mathcal{C}$$
$$\mathcal{C}_O := O(\alpha) \mid \mathcal{C}_O \oplus \mathcal{C}_O \quad \mathcal{C}_P := P(\alpha) \mid \mathcal{C}_P \oplus \mathcal{C}_P \quad \mathcal{C}_F := F(\alpha) \mid \mathcal{C}_F \vee [\alpha]\mathcal{C}_F$$

A contract consists of two parts: *definitions* ($\mathcal{D}$) and *clauses* ($\mathcal{C}$). $\mathcal{D}$ specifies the *assertions* ϕ (which ranges over Boolean expressions including arithmetic comparisons, like *the budget is more than 200$*), and the atomic actions present in the clauses. Actions α are formalized by the $\mathcal{CAT}$ action algebra which we see later. $\mathcal{C}$ is the general *contract clause*. $\mathcal{C}_O$, $\mathcal{C}_P$, and $\mathcal{C}_F$ denote respectively *obligation*, *permission*, and *prohibition* clauses. $\wedge$, $\oplus$, and $\vee$ may be thought as the classical conjunction, exclusive disjunction, and disjunction.

We borrow from Propositional Dynamic Logic (PDL) the syntax $[\alpha]\phi$ to represent that after performing α (if it is possible to do so), ϕ must hold. The $[\cdot]$ notation allows having a *test*, where $[\phi?]\mathcal{C}$ must be understood as $\phi \Rightarrow \mathcal{C}$. The syntax $\langle\alpha\rangle\phi$ captures the idea that there must exist the possibility of executing action α, in which case ϕ must hold afterwards. Following temporal logic (TL) [9] notation we have $\mathcal{U}$ (*until*), $\bigcirc$ (*next*), and $\Box$ (*always*) with intuitive semantics as in TL. Thus $\mathcal{C}_1 \mathcal{U} \mathcal{C}_2$ states that $\mathcal{C}_1$ should hold until $\mathcal{C}_2$ holds. $\bigcirc\mathcal{C}$ intuitively states that the $\mathcal{C}$ should hold in the next moment.

In [10] we have defined a new algebraic structure to provide a well-founded formal basis for $\mathcal{CL}$ and to help give a direct semantics to the language. The main contributions of the algebra are: (1) A formalization of actions found in contracts; (2) The introduction of a different kind of negation over actions; (3) A restricted notion of resource-awareness; and (4) A standard interpretation of the algebra over specially defined guarded rooted trees. The algebra is proven to be complete w.r.t. the interpretation as trees.

We called our algebra $\mathcal{CAT}$ to stand for *algebra of concurrent actions and tests*. $\mathcal{CAT} = (\mathcal{CA}, \mathcal{B})$ is formed of an algebraic structure $\mathcal{CA} = (\mathcal{A}, +, \cdot, \&, \mathbf{0}, \mathbf{1})$ which defines the concurrent actions, and a Boolean algebra $\mathcal{B} = (\mathcal{A}_\mathbf{1}, \vee, \wedge, \neg, \bot, \top)$ which defines the tests. Special care has to be taken when combining actions and tests under the different algebraic operators.

Actions of $\mathcal{A}$ are constructed from atomic actions $\mathcal{A}_B$ by applying: *choice* ($+$), *sequence* ($\cdot$), or *concurrent* composition ($\&$). Tests of $\mathcal{A}_\mathbf{1}$ are constructed with the classical boolean operators ($\vee$, $\wedge$, and $\neg$) and the constants ($\bot$ and $\top$). Moreover, $(\mathcal{A}, +, \cdot, \mathbf{0}, \mathbf{1})$ is an idempotent semiring and $(\mathcal{A}, +, \&, \mathbf{0}, \mathbf{1})$ is a commutative and idempotent semiring. We consider a *conflict relation* over the set of atomic actions $\mathcal{A}_B$ (denoted by $\#_\mathcal{C}$) defined as: $a \#_\mathcal{C} b \overset{def}{\iff} a\&b = \mathbf{0}$. The intuition of the conflict relation is that if two actions are in conflict then the actions cannot be executed concurrently.

With the actions and their interpretation as guarded rooted trees it is simple now to give a direct semantics to the language $\mathcal{CL}$. In [11] the semantics of $\mathcal{CL}$ was given through a translation into a variant of μ-calculus. We have used $\mathcal{CL}$ and the μ-calculus semantics in [8] to do model-checking of an example from a real contract.

We have used the state-of-the-art tool NuSMV and followed the steps: (1) first translate the conventional contract into the formal language $\mathcal{CL}$; (2) translate syntactically the $\mathcal{CL}$ specification into the extended μ-calculus $\mathcal{C}\mu$ to obtain a Kripke-like model (a labeled transition system, LTS) of the $\mathcal{C}\mu$ formulas (representing the semantics of the $\mathcal{CL}$ specification); (3) translate the LTS into the input language of NuSMV and perform model checking using NuSMV; (4) in case of a counter-example given by NuSMV, interpret it as a $\mathcal{CL}$ clause and repeat the model checking process until the property is satisfied; (5) finally, repair the original contract by adding a corresponding clause, if applicable.

Besides classical properties like liveness, safety, or response which can be specified using LTL or CTL we are also interested in properties more specific to electronic contracts. The first kind of properties is *search properties*. Examples of search properties include: listing of the obligations, or prohibitions of one of the parties in the contract; listing of the rights that follow after the fulfilling of an obligation; or what are the penalties for whenever violating an obligation or prohibition, and so on. A second kind of reasoning about a contract (formalized in our contract language) is quantitative reasoning with respect to the amount of obligations or of rights a contract imposes. Examples of such properties include: the client would like to know if the contract can be changed so that the client has a smaller number of obligations, or a greater number of rights; if quantitative measures are inserted into the language (like time or amounts) one could ask to minimize time between certain events, or try to change the contract so that a minimal(maximal) amount is reached.

References

[1] A. Beugnard, J.-M. Jézéquel, and N. Plouzeau. Making components contract aware. *IEEE Computer*, 32(7):38–45, 1999.

[2] A. Daskalopulu. Model Checking Contractual Protocols. In L. Breuker and Winkels, editors, *Legal Knowledge and Information Systems, JURIX 2000*, pages 35–47. IOS Press, 2000.

[3] A. Daskalopulu and T. S. E. Maibaum. Towards Electronic Contract Performance. In *12th International Conference and Workshop on Database and Expert Systems Applications*, pages 771–777. IEEE C.S. Press, 2001.

[4] H. Davulcu, M. Kifer, and I. V. Ramakrishnan. CTR-S: A Logic for Specifying Contracts in Semantic Web Services. In *Proceedings of WWW2004*, pages 144–153, May 2004.

[5] G. Governatori. Representing business contracts in RuleML. *International Journal of Cooperative Information Systems*, 14:181–216, 2005.

[6] G. Governatori and A. Rotolo. Logic of violations: A gentzen system for reasoning with contrary-to-duty obligations. *Australatian Journal of Logic*, 4:193–215, 2006.

[7] C. Molina-Jiménez, S. K. Shrivastava, E. Solaiman, and J. P. Warne. Run-time Monitoring and Enforcement of Electronic Contracts. *Electronic Commerce Research and Applications*, 3(2), 2004.

[8] G. Pace, C. Prisacariu, and G. Schneider. Model checking contracts - a case study. In K. Namjoshi and T. Yoneda, editors, *ATVA'07*, volume 4762 of *LNCS*, pages 82–97. Springer, 2007.

[9] A. Pnueli. Temporal logic of programs, the. In *FOCS'77*, pages 46–57. IEEE Computer Society Press, 1977.

[10] C. Prisacariu and G. Schneider. An algebraic structure for the Action-Based Contract language CL - Theoretical Results. Technical Report 361, Department of Informatics, University of Oslo, Oslo, Norway, July 2007.

[11] C. Prisacariu and G. Schneider. A formal language for electronic contracts. In M. Bonsangue and E. B. Johnsen, editors, *FMOODS'07*, volume 4468 of *LNCS*, pages 174–189. Springer, 2007.

[12] G. Salaün, L. Bordeaux, and M. Schaerf. Describing and reasoning on web services using process algebra. In *ICWS '04*, page 43. IEEE Computer Society, 2004.

[13] G. H. V. Wright. Deontic logic. *Mind*, 60:1–15, 1951.

[14] G. H. V. Wright. Deontic logic: A personal view. *Ratio Juris*, 12(1):26–38, 1999.

ConSpec: A Formal Language for Policy Specification

Irem Aktug
KTH
irem@nada.kth.se

Katsiaryna Naliuka
UNITN
naliuka@dit.unitn.it

1 Introduction

As mobile devices become increasingly popular, the problem of secure mobile application development gains importance. Mobile devices contain personal information, which users wish to protect. They also provide access to costly functionality, such as GSM services and GPRS connections. It is necessary to provide *controlled* access to the sensitive resources through fine-grained, at times application specific, constraints on execution.

A *security policy* specifies the set of acceptable executions and can thus be used to define how and under what conditions a sensitive resource can be accessed. For instance, a user policy may limit the number of SMSs that are sent by an application per hour in order to prevent spamming. A program adheres to a policy if all its executions are acceptable by the policy. Several techniques exist to ensure that an application complies to a policy. *Static verification* techniques analyze the program code in order to construct a mathematical proof that no execution of the program can violate the policy. Though such an analysis provides full assurance, due to the complexity of the problem, static verification is most often unfeasible in the resource-critical environment of the mobile device. *Runtime monitoring*, on the other hand, observes the behavior of a target program at runtime and terminates it if it does not respect the policy. Monitoring can effectively enforce many practically useful security policies [8]. However, it creates performance overhead since each *security relevant action* of the program should be detected and checked against the policy.

Our framework combines static with dynamic techniques in order to enforce security policies on mobile devices in the most effective way. It is similar to the *model-carrying code* approach [9], in that security specifications can be enforced at the three stages of the application lifecycle: the *development*, *installation* and *runtime* phases.

Development Phase: We associate with the application a *contract* [2], a piece of data that describes its *security-relevant behavior*. In the development phase, the contract is a specification of the intended security-relevant behavior of the application by the producer. The compliance of the application to the contract can be checked using static verification. The analysis can be performed by the producer or a trusted third party, which then signs the application and the contract by its private key. This analysis is performed by powerful machines rather than the mobile devices, and can make use of knowledge available to the developer (e.g. program specifications, annotations derived from the source code). Instead of signing with a private key, *proof-carrying code* techniques [7] can be used to convey assurance in program-contract compliance. If contract compliance can not be statically verified, then an execution monitor can be *inlined* in the program at this stage so that compliance is ensured at runtime [4, 3].

Installation Phase: Before the program is installed on the device, a formal check is needed to show that the security-relevant behavior of the application given by the contract is acceptable by the user policy; we call this process *contract-policy matching*. In the case where the contract does not match to the policy, adherence to policy can be enforced by inserting a monitor to the environment of the program in the installation phase.

Runtime: At runtime, the behavior of an application may be checked against a policy by monitoring.

The main contribution of this paper is the language *ConSpec* (Contract Specification Language), which can be used for specifying both user policies and application contracts in a framework such as the one described above. ConSpec aims for a balance of language expressiveness and tractability of the various tasks identified in the framework. For example, the problem of matching a contract against a policy reduces to the language containment problem for such automata, if policies and contracts are captured with automata on infinite strings. The complexity of this task (see e.g. [5, 6]) severely restricts the expressive power of the language. We provide a semantics for ConSpec and we briefly explain how the design of ConSpec renders possible the formal treatment of several of the activities in the framework mentioned above.

2 ConSpec Language

ConSpec is strongly inspired by the policy specification language PSLang, which was developed by Erlingsson and Schneider [3] for runtime monitoring. PSLang policies consist of a set of variable declarations representing the security state, followed by a list of security relevant events, where each event is accompanied by a piece of Java-like code that specifies how the security state variables should be updated in case the event is encountered in the current state. PSLang trades the formalisation of the monitor to the simplicity of inlining it. While a PSLang policy text is intended to encode a security automaton, a formal semantics for PSLang is not provided. Such a task is not trivial due to the power of the programming language constructs that can be used in the updates. Furthermore, contract-policy matching would be undecidable when such an expressive language is used. ConSpec is a more restricted language than PSLang: the domains of the security state variables are finite and we have used a guarded-command language for the updates where the guards are side-effect free and commands do not contain loops. The simplicity of the language then allows for a comparatively simple and elegant semantics. ConSpec has the additional scope construct for expressing security requirements on different levels. Case studies show that this feature is necessary for expressing many interesting real-life policies [10]. ConSpec is expressive enough to write policies on multiple executions of the same application, and on executions of all applications of a system, in addition to policies on a single execution of the application and on lifetimes of objects of a certain class.

Assume that the method `Open` of the class `File` is used to create files (when the argument `mode` has the value "CreateNew") or open files (when the argument `mode` has the value "Open") for reading or writing. Similarly, the method `Open` of the class `Connection` opens a connection and the method `AskConnect` asks the user for permission to open a connection. The policy "*Application must, after accessing an existing file, get approval from the user before opening a connection*" is expressed in ConSpec as follows:

```
SCOPE Session
SECURITY STATE
      bool accessed = false;
      bool permission = false;

BEFORE File.Open(string path, string mode, string access)
PERFORM
      mode.equals("CreateNew") -> { skip; }
      mode.equals("Open") -> { accessed = true; }


BEFORE Connection.Open(string type, string address)
PERFORM
      !accessed -> { permission = false; }
      accessed && permission -> { permission = false; }

AFTER string answer = GUI.AskConnect()
PERFORM
      answer.equals("Yes") -> { permission = true; }
```

We begin by specifying that the policy applies to each single execution of an application. Scope declaration is followed by the *security state declaration*: the security state of the example policy is represented by the boolean variables `accessed` and `permission`, which are both false initially to mark, respectively, that no file has been accessed and that no permissions are granted when the program begins executing. The example policy contains three *event clauses* that state the conditions for and effect of the security relevant actions: call to the method `File.Open`, call to the method `Connection.Open` and return from the method `GUI.AskConnect`. The types of the method arguments are specified along with representative names, which have the event clause as their scope. The *modifiers* BEFORE and AFTER mark whether the call of or the normal return from the method specified in the event clause is security relevant (exceptional returns can be specified by the modifier EXCEPTIONAL). Whatever the execution history, whenever the application calls the method `File.Open`, it should be creating a new file (the first guard) or it should be opening an existing file (the second guard). In order to decide if the application is allowed to open a connection, history of existing file accesses and user permissions should be consulted. If the current history contains an access to an existing file (that is if `accessed` is true), then an attempt to open the connection is allowed only if the security state variable `permission` is true. The only way this variable has the value true at a certain point in the execution is, if the last execution of the method `GUI.AskConnect` has returned "Yes". Notice that the policy does not allow the same permit to be used for several connections since `permission` is set to false before each connection. If neither of these conditions hold for the current call, that is if an existing file has been accessed but no permission for connection was granted, then it is a violating call as none of the guards are satisfied.

For such a language, an elegant semantics can be given in terms of security automata [8].

3 ConSpec in Use

Current work focuses on the formal treatment of the following tasks related with policy enforcement, based on ConSpec semantics:

Matching One way to match a ConSpec contract against a ConSpec policy is to check that the language of the contract automaton is included in the language of the policy automaton. Since the domains of the security state variables are bounded, the extracted automata have finitely many states and standard methods for checking language inclusion for automata can be facilitated for contract-policy matching (see for instance [1]).

Monitoring Given a program and a ConSpec policy with scope `Session`, the concept of monitoring can be formalized by defining the co-execution of the corresponding ConSpec automaton with the program. Such co-executions are a subset of the set of interleavings of the individual executions of the program and the automaton. Co-executions satisfy the following condition: when the execution of the program component is projected to its security relevant action executions, each *before* action is immediately preceded by a transition of the automaton for the same action; dually, each *after* action is immediately followed by a corresponding automaton transition. It is simple, then, to show that the program component of the co-execution adheres to the given policy, as the co-execution includes an accepting trace of the automaton for the program execution.

Monitor Inlining Inlining a ConSpec policy with scope `Session` can be performed similar to inlining a PSLang policy (see [3] for details). The correctness of such a monitor inlining scheme can be proven by setting up a bisimulation relation between the states of the inlined program and the states of the co-execution of the original program with the ConSpec automaton (induced by the policy). If the party responsible for the inlining is not trusted, the proof-carrying code approach can be used. An annotation scheme can be devised that produces annotations for a given policy, so that if the annotations are valid for a program, then the program adheres to the policy. The inlining party can use certain information about the inlining (e.g. the variables used to represent the security state) to prove the verification conditions resulting from annotating the program with this scheme. A proof can then be shipped to the consumer which enables the correctness of the inlining to be verified on the mobile device, based on the same scheme.

4 Conclusion

In this paper, we present the language ConSpec, which can be used for specifying both policies and contracts in various security enforcement related tasks of the application lifecycle. We provide a formal semantics for the language which enables formal proofs to be constructed for these taks. Currently, we are formalizing enforcement techniques using ConSpec for sequential programs as summarized in Section 3. As future work, we aim to extend our approach to applications where multiple threads can perform security-relevant actions.

Acknowledgements The authors thank Dilian Gurov and Fabio Massacci for valuable comments and discussions.

References

[1] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Form. Methods Syst. Des.*, 1(2-3):275–288, 1992.

[2] N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Security-by-contract: Toward a semantics for digital signatures on mobile code. In *European PKI Workshop: Theory and Practice (to appear)*, 2007.

[3] Ú. Erlingsson. *The inlined reference monitor approach to security policy enforcement.* PhD thesis, Department of Computer Science, Cornell University, 2004.

[4] Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: a retrospective. In *Proc. of the Workshop on New Security Paradigms (NSPW '99)*, pages 87–95, New York, NY, USA, 2000. ACM Press.

[5] J. E. Hopcroft. On the equivalence and containment problems for context-free languages. *Theory of Computing Systems*, 3(2):119–124, 1969.

[6] H. B. Hunt and D. J. Rosenkrantz. On equivalence and containment problems for formal languages. *J. ACM*, 24(3):387–396, 1977.

[7] G. C. Necula. Proof-carrying code. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997.

[8] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[9] R. Sekar, V.N. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. *ACM SIGOPS Operating Systems Review*, 2003.

[10] A. Zobel, C. Simoni, D. Piazza, X. Nuez, and D. Rodriguez. Business case and security requirements. Public Deliverable of EU Research Project D5.1.1, S3MS- Security of Software and Services for Mobile Systems, http://s3ms.org, October 2006.

An Integrated Platform for Contract-oriented Development

Hakim Belhaouari
Laboratoire d'Informatique de Paris 6
104 avenue du President Kennedy
75016 Paris, France
Hakim.Belhaouari@lip6.fr

Frederic Peschanski
Laboratoire d'Informatique de Paris 6
104 avenue du President Kennedy
75016 Paris, France
Frederic.Peschanski@lip6.fr

Introduction

Contract-orientation prescribes the definition of precise checkable interfaces - or services - for software components. The *agreement* of contracts between clients and service providers is traditionally enforced using runtime monitoring techniques. In the recent years, static verification techniques were developed to assist contract-oriented development. Static verification allows to reduce the overhead caused by the monitoring logic at runtime. Moreover, and most importantly, inconsistencies in contract specifications can be found earlier in the development cycle.

The goal of the Tamago platform we overview in this paper is to provide a comprehensive set of tools to assist in contract-oriented component software development. It provides a specification language similar to an interface description language extended with observable properties, first-order logic assertions (preconditions, postconditions and invariants) and descriptions of *interface behaviors* [3, 8, 10] based on finite-state automata with conditional transitions. A set of static analysis tools are proposed to detect contract inconsistencies at the specification level, and also provides a support for automated testing of implementations.

A contract compiler is proposed that currently targets the Java programming language. The generated monitoring logic adopts a flexible contained-based architecture that can be attached dynamically to components implementing the business logic. It also supports various *percolation patterns*.

Specification language for contracts

To illustrate the Tamago specification language, we give a possible definition of a contract for the traditional bank account example as follow:

```
service BankAccount {
    property readonly double balance init 0;

    property constant int id;
    invariant balance ≥ 0 fail "negative balance";

    method void credit(double amount) {
        pre amount > 0 fail "amount must be strictly positive";
        post balance = balance@pre + amount
    }

    method void withdraw(double amount) {
        pre amount ≤ balance fail "not enough money";
        post balance = balance@pre - amount
    }

    behavior {
        init navail;
        states{
              state navail { allow credit; }
              state avail { allow credit,withdraw; }
        }
        transitions {
              transition from empty to notempty with credit;
              transition from notempty to empty with withdraw
                        when balance = 0;
        }
    } // end behavior
} // end service
```

The first part of the specification is self-explaining and resembles traditional software contracts. The notion of *observable properties* is important for contracts defined on interfaces rather than implementations. The second part of the contract describes a simple automaton with two states: navail and avail. Any account starts with a balance of 0 and is thus not available for withdrawal. As soon as a credit is performed, both operations becomes available. The converse transition is through method withdraw only when the balance is zero. Beyond such service-level contracts, the language also supports component-level contracts (with require/provide clauses), assembly contracts (architecture of interconnected components) as well as contracts for composite components (components with an inner architecture of sub-components). It also supports *behavioral subtyping* that leverage reuse of specifications as well as their underlying implementations through object-oriented inheritance.

Contract Analysis

Our main design objective is to offer a good compromise between *expressiveness* and *tractability*. To illustrate the second point, we develop a set of tools to analyze contract specifications at design time.

Structural analysis. The first tool performs a structural analysis of the contracts. Beyond type checking, the tool use finite-state automata techniques to detect inconsistencies such as unreachable states in interface behaviors. It supports service subtyping and composition. The algorithm we employ is a simplified variant of the synchronization of parallel automata [11], only without the calculation of synchronized transitions and corresponding optimizations (see Algorithm 1(a)). It starts with the merging of all the initials states of the provided services in a unique start state (*initState*). For each such merged states, the *traversal* of the product automaton works as follows. First, if the state to traverse is marked, then the corresponding branch stops here. Otherwise, the state is marked and for all the transitions starting from one of the members of the current state we compute the next merged state (*nstate*). We then add the resulting abstract transition to the product automaton (*add*). Finally, we reiterate the traversal on the next state until all the abstract states and transitions have been taken into account. Since we go through all the abstract transitions, the algorithm also detects the set of unreachable states (unmarked).

Abstract interpretation. The second static analysis we perform tries to uncover inconsistencies in the dynamics of the service and component specifications. The purpose of the algorithm is to simulate their concrete behaviors. It takes the product automaton generated by the structural analysis tool and tries to "find" its concrete states and transitions. The algorithm (see Algorithm 1(b)) starts with the domains of values of the observables properties in the initial state of the product automaton, forming the initial context of the interpretation (*initContext*). These domains may be of course unknown at first (or infinite, e.g. arbitrary strings which we also interpret as unknown). The algorithm then use depth-first search to animate the context (*animate*). For each animated state, we compare the current context with its previously known context (*loadContext*). If no change is observed then we found a fixed point and stop the process for this branch. If the context is new, we associate it to the current sate (*saveContext*). Then, for each (abstract) transition, we try to evaluate its firing condition as well as the precondition (and invariant) for the corresponding functionality (*eval*). If we are able to falsify these conditions, then the transition may not be fired, which we report as an inconsistency. The (concrete) states from this transition are also reported unreachable. If the conditions are true, we advance to the next state silently. The last case is if the evaluation is only partial, then we assume the transition enabled and fire it but we report the indecision as a warning. The most important step is with the postcondition (and invariant) that we first try to falsify/satisfy (and abort if it is false), but also analyze to find conditions of the form prop = expr. We take such equality as a definition and record it as an *effect* if we are able to evaluate expr to a finite domain. The process extends to finite domains comparisons through the use of a simple CSP solver (*solveContext*[1]). We reiterate the process on the further transitions with the new set of constraints on properties. Because the dynamics of a service may be infinite, the actual implementation ultimately relies on a maximal depth parameter to enforce a global fixed point.

```
function mergeBehaviors(Bs):
    result ⇐ emptyBehavior()
    initState ⇐ {q0 ∈ B | B ∈ Bs}
    traversal(result,initState,Bs)
    return result

function traversal(result,state,Bs):
    if isMark(state) then
        return // fixed point
    else
        mark(state)
    end if
    for all { s1 --f/c--> s2 | s1 ∈ state} ∈ Bs
        nstate ⇐ (state \ {s1}) ∪ {s2}
        add(result, state --f/c--> nstate)
        traversal(result,nstate,Bs)
    end for
```

(a) Generation of the product automaton

```
function interpret(B ≝ {q0, Q, T}):
    ctx ⇐ initContext(B)
    animate(ctx,q0)

function animate(ctx,state):
    old ⇐ loadContext(state)
    if old=ctx then
        return // fixed point
    else
        saveContext(state,ctx)
    end if
    for all state --{p}f{q}/c--> nstate ∈ T
        if ¬eval(ctx,p∧c∧Inv) then
            error // Unreachable
        end if
        nctx ⇐ solveContext(ctx,q∧Inv) // Inv invariant
        animate(nctx,nstate)
    end for
```

(b) Abstract Interpreter

Figure 1: Used Algorithms

Automated testing. The abstract interpretation of the service and component specifications corresponds to a form of systematic and automated testing of the specifications. We also use the output - a partial evaluation of the concrete automaton - to generate a testing infrastructure. The difference with the abstract interpretation is that here the transitions are tested not only from the point of view of the contract itself but also from the point of view of the environment, i.e. we can test for external arguments given by users. Moreover, it is not the contract that we test but the actual implementation(s). There is in fact almost nothing new from an algorithmic point of view here: a simple depth-first search is performed again. We give more details about the actual implementation of the algorithm. One difference with abstract interpretation is that for each (partial) concrete transition, the precondition is used as a unit test for the corresponding functionality, and the postcondition serves as an oracle, following [4]. An interesting fact is that the precondition (plus invariant and firing condition for a transition) allows to reflect on the (inbounds) testing coverage for the considered functionality/transition. The negation also gives the

[1]The algorithm we use is similar to a CSP solver *without* forward checking because we are not looking for a specific solution for the constraints.

outbounds test. The boundary tests should also be inferred when possible, but this is left for a future work.

The problem with the CSP technique we employ is that it does not apply directly on complex user-defined types. For this we use a notion of *builder* that wraps arbitrary types to comparable and finite domain types (e.g. integer intervals). Also, the builders are used to generate data sets for testing purposes. We illustrate the use of builders with the following example. Suppose a constraint (e.g. a precondition) stating str.size() > 3 where str is a platform-dependent string object. The first step is to generate a constraint corresponding to this statement, which will be of kind $X > C$ where X is a CSP variable and C an integer constant. For the variable str, the algorithm asks to the builder for arbitrary strings (i.e. StringBuilder) to generate a CSP variable X of integer type corresponding to the method size(). Of course, a builder needs to know the semantics of the objects it wraps. Given the constraints, the domain of X is updated with 3 as a lower bound. Most interestingly, the current implementation of the builder for strings is able to interpret the automaton of all strings of minimum size 3 and thus generate a word for the variable str that satisfies the initial constraint.

Contract compilation

The main objective of the contract compiler is to generate the monitoring logic that enforces the behavioral contracts at runtime, only for the remaining constraints after the static analyzes. A non-trivial aspect of code generators for contract assertion checking in presence of behavioral subtyping is that of *percolation patterns* [9, 12]. Many approaches follow the Eiffel percolation pattern, which builds assertions by OR-ing preconditions and AND-ing postconditions [5, 7, 9]. In general, what must be effectively allowed "at most" is the weakening of preconditions and/or strengthening of postconditions [6]. The Eiffel percolation remains consistent with these, but denotes rather weak constraints as we may "safely" contradict the parents' preconditions at the specification level. In fact, there is not a single best solution for the percolation problem, as explained in [9] and various algorithms have been proposed in the literature: *exact pre/post*, *exact pre*, *exact post*, *plug-in* (Eiffel approach [7]), *weak plug-in* or *guarded plug-in*, *relaxed plug-in*, etc. Indeed, the choice of a specific percolation algorithm may depends on the context of use. For example, in a secure environment a restricting percolator may be preferable whereas a general plug-in system may benefit from more relaxed assertions. Another important role of the verification code is to safely decide the correct responsibilities in case of *contract violation*. In fact, the selected percolation pattern interferes with this detection issues, since the responsibility for violation may vary depending on the chosen policy. To allow for the maximum flexibility in percolation patterns, Tamago adopts a container-based architecture to support the choice of the specific percolator to use at deployment time. The advantage is that the service implementors are allowed to offer a panel of available percolators, which are then selected by either the providers or the clients at deployment time depending on their specific context of use. The architecture also allows the dynamic and hierarchical composition of containers so that other extra-functional aspects can be integrated in the process (e.g. containers for security, QoS monitoring, etc.).

Ongoing work

The Tamago platform is now a functional prototype implemented in Java that has been experimented in a graduate course on software components and design by contract. Beyond the pedagogical interest of the framework, we started a project to adapt the model to existing and largely deployed OSGi component model [1]. A particular interest is to extend the OSGi service discovery protocol with contract-based queries, which would enable semantic match-making.

The analysis algorithms and testing tools correspond to a recent addition in the framework. The tools already allow thorough investigations of the service semantics. However, we would like to extend the algorithms, most notably by enabling the verification of the logical property, using an automatic theorem prover. The LeanTap algorithm [2] is currently being adapted to the platform. An important step that we initiated recently was to go deeper into the formalization of the contract model. Our objective is to establish correctness proofs for the verification algorithms. Note however that these tools are more about giving hints about possible inconsistencies/incompletenesses in the specifications that about proving them (in)correct.

1. REFERENCES

[1] Osgi alliance. `http://www.osgi.org/`, 2007.

[2] B. Beckert and J. Posegga. leantap: Lean tableau-based deduction. *J. Autom. Reasoning*, 15(3):339–358, 1995.

[3] A. Brown, S. Johnston, and K. Kelly. Using service-oriented architecture and component-based development to build web service application. *Rational Software white paper*, 2002.

[4] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The jml and junit way. In *ECOOP*, volume 2374 of *LNCS*, pages 231–255. Springer, 2002.

[5] R. B. Findler, M. Latendresse, and M. Felleisen. Behavioral contracts and behavioral subtyping. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference*, pages 229–236. ACM Press, 2001.

[6] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. volume 16, pages 1811–1841. ACM Press, 1994.

[7] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997. second edition.

[8] R. Reussner, I. Poernomo, and H. W. Schmidt. Reasoning about software architectures with contractually specified components. In *Component-Based Software Quality*, volume 2693 of *LNCS*, pages 287–325. Springer, 2003.

[9] H. Toth. On theory and practice of assertion based software development. *Journal of Object Technology*, 4(2):109–129, March 2005.

[10] S. Viswanadha and D. Kapur. Ibdl: A language for interface behavior specification and testing. In *COOTS*, pages 235–248. USENIX, 1998.

[11] D. Zampunieris and B. L. Charlier. An efficient algorithm to compute the synchronized product. volume 00, page 77, Los Alamitos, CA, USA, 1995. IEEE Computer Society.

[12] A. M. Zaremski and J. M. Wing. Specification matching of software components. volume 6, pages 333–369, New York, NY, USA, 1997. ACM Press.

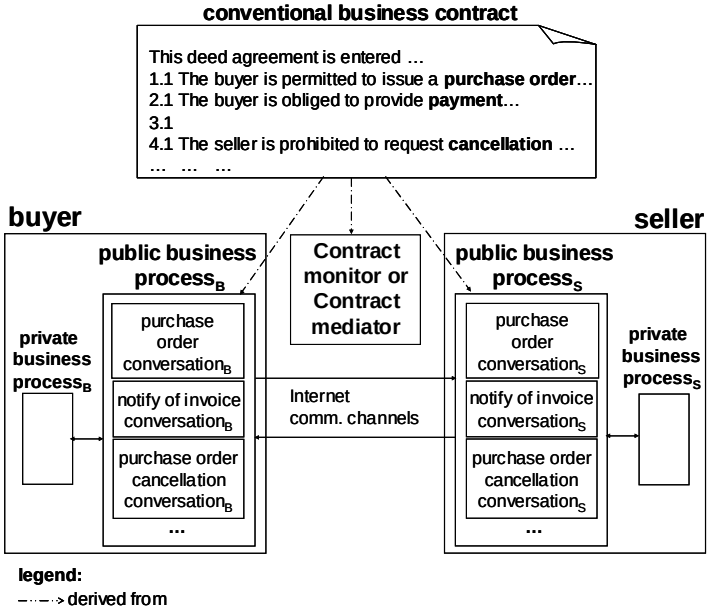
On Contract Compliant Business Process Coordination

Carlos Molina-Jimenez and Santosh Shrivastava
School of Computing Science
Newcastle University
Newcastle upon Tyne, NE1, 7RU

{Carlos.Molina, Santosh.Shrivastava}@ncl.ac.uk

1. Introduction

We assume that business-to-business (B2B) interactions between trading partners are being regulated by electronic contracts. The business scenario that motivated our research is depicted in the figure below. The buyer and the seller represent two autonomous organizations (trading partners) who have chosen to conduct business by means of exchanging messages over Internet communication channels. To interact, the buyer and the seller need to expose to each other, part of their business interfaces, in the figure, these interfaces are represented by the public business process$_B$ and public business process$_S$, respectively. To preserve their autonomy, the buyer and the seller conceal behind their public business processes those aspects of their business that they do not wish to disclose; in the figure, this is represented by the private business process$_B$ and private business process$_S$.



The B2B interactions between the buyer and the seller can be regarded as the execution of a global (cross-organizational) business process composed out of public business process$_B$ and public business process$_S$. Typically, a cross organizational business process can be constructed out of a set of small conversations (joint activities or dialogs) such as *issue a purchase order, send invoice, cancel purchaser order*, etc. We highlight some technical issues involved in ensuring that an execution of a cross-organizational business process is compliant with the corresponding electronic contract.

2. Contract Representation

The electronic representation of terms and conditions of the contract should be such that it can be utilized at design time by partners for validating their public processes and at run time for monitoring/mediating business interactions between trading partners, ensuring that these indeed correspond to meeting the rights and obligations that each interacting entity has promised to honour.

It is not well understood yet what formal language or languages should be used for contract description. To start with, such a language should be expressive enough to capture the permissions, obligations and prohibitions together with their time and order of execution constrains stipulated in the contract. Secondly, the language should produce a model that is simple enough to reason (perhaps by means of existing model-checking tools) about the logical consistency of the interaction; this requirement suggests that the formal notation should abstract away irrelevant implementation details. Finally and in direct conflict with the previous requirement, the language should produce a notation that can be mapped by the programmer into existing middleware technology and e-business standards such as RosettaNet, ebXML, CORBA, J2EE, Web services etc. From this observation it seems very unlikely that a single contract description language is enough; several equivalent descriptions of the contract might be needed with different level of abstractions. Considering our personal research experience we are planning to focus our efforts on the lowest two, perhaps three, levels [1]. The interest in implementation-independent contract notations is that they become declarative as their level of abstraction increases; this is a feature that we would like to explore as it is relevant in adaptable contractual business interactions. We believe that in practice, long term business interactions are likely to experience changes to adapt to new market conditions. On this basis, we speculate that the permissions, obligations and prohibitions stipulated in the original conventional contract should be split into two sets: those that stipulate the core business operations and are not expected to change and those that stipulate the complimentary business operations and are very likely to experience changes (within or outside the scope of the contract) during the contract life time. We suspect that the static set should be described in an implementation oriented notation and hard coded in an imperative language; on the other hand, the changeable set should be described in a declarative notation, so that it can be read, understood and changed by non-technical people such as business managers, as needed at run-time. In the literature, this approach is known as policy-driven or rule-driven and is a promising research avenue that needs exploring.

3. Process coordination

Naturally, business process executions at each partner must be coordinated at run-time to ensure that partners are performing mutually consistent actions. Distributed coordination mechanisms are attractive in B2B settings where all the partners are autonomous entities. A primitive B2B conversation typically involves exchange of one or more electronic business documents (e.g., a purchase order, shipping notification, invoice notification, etc.) and has various QoS constraints (timing, security, message validation, etc.). RosettaNet Partner Interface Processes (PIPs) are a good example of such conversations. The QoS constraints are expected to be met despite encountering software and hardware related problems (e.g., clock skews, unpredictable transmission delays, message loss, corrupted messages, node crashes, etc.). A failure to deliver a valid message within its time constraint could cause mutually conflicting views of an interaction (one party regarding it as timely whilst the other party regarding it as untimely). A conflict can also arise if a sent message is delivered but not taken up for processing due to some message validity condition not being met at the receiver (the sender assumes that the message is being processed whereas the receiver has rejected it).

In a loosely coupled system, it could take a long time before such inconsistencies are detected. Subsequent recovery actions - frequently requiring compensation - may turn out to be quite costly. It is best to deal with this problem at the source rather than at business process level by ensuring that interacting parties get a mutually consistent view on how a given interaction completed. The sender needs a timely assurance that the sent document will be processed by the receiver, and the receiver needs the assurance that if it accepts the document for processing, the sender is informed of the acceptance in a timely manner; in all other cases, the interaction returns failure exceptions to both the parties. Whereas existing middleware solutions have concentrated on providing generic client-centric communication primitives (such as RPCs), in the world of loosely coupled peer-to peer entities, we speculate that we need messaging abstractions with bi-lateral (multi-lateral) consistency guarantees. This is a challenging problem to solve, since we need to provide consistency without unduly compromising loose coupling. Encapsulating a business conversation within a single atomic transaction would not be considered practical as it would introduce tight coupling. Instead we need some form of a state synchronisation protocol that can be executed to reach an agreed outcome. Some ideas on how this could be achieved are presented in [2,3] where distributed as well as centralised synchronisation approaches are discussed.

We believe that the issue here is all about the execution of business processes with state notification to guarantee the observance of safety properties. The problem has practical relevance and can be reduced to the task of finding a set of synchronisation constraints that guarantee that a given list of safety properties in never violated. A more challenging problem is to present the business partners with a minimal set (which is not necessarily unique) of synchronisation constraints that satisfy the requirements. The problem can be also presented as follows: given the specification of a business process whose execution results in the violation of one or more safety properties, find the minimal synchronisation constraints to be introduced into the specification to remedy the problem.

Acknowledgements

This work is funded in part by UK Engineering and Physical Sciences Research Council (EPSRC), Platform Grant No. EP/D037743/1, “Networked Computing in Inter-organisation Settings”

References

[1] Carlos Molina-Jimenez, Santosh Shrivastava and John Warne, “A Method for Specifying Contract Mediated Interactions”, In Proc. of the IEEE International Enterprise Distributed Object Computing Conference (EDOC 2005), Sep. 2005, Enschede, The Netherlands, pp. 106-115.
[2] Carlos Molina-Jimenez, Santosh Shrivastava, “Maintaining Consistency Between Loosely Coupled Services in the Presence of Timing Constraints and Validation Errors”, Proc. the 4th IEEE European Conf. on Web Services (ECOWS’06), Dec. 2006, Zurich, pp. 148-157.
[3] Carlos Molina-Jimenez, Santosh Shrivastava and Nick Cook, “Implementing Business Conversations with Consistency Guarantees Using Message-oriented Middleware”, To appear in Proc. of the IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007), Oct. 2007, Annapolis, Maryland U.S.

Transforming Web Services Choreographies with Priorities and Time Constraints Into Prioritized-Time Petri Nets

Valentín Valero, María Emilia Cambronero, Gregorio Díaz, and Juan José Pardo *

Departamento de Sistemas Informáticos, Escuela Politécnica Superior de Albacete
Universidad de Castilla-La Mancha, Campus Universitario s/n. 02071. Albacete, SPAIN
{Valentin.Valero, MEmilia.Cambronero, Gregorio.Diaz, Juan.Pardo}@uclm.es

Abstract. A Web Service is a self-describing, self-contained modular application that can be published, located, and invoked over a network, e.g. the Internet. Web Services composition provides a way to obtain value-added services by combining several Web Services. Then, the composition of Web Services is suitable to support enterprise application integration. WS-CDL (Web Services Choreography Description Language) is a W3C candidate recommendation for the description of peer-to-peer collaborations for the participants in a Web Services composition. In this paper we focus our attention on the timed aspects of WS-CDL, as well as in the collaborations that require some kind of priorization. Then, our goal is to provide a WS-CDL definition of prioritized collaborations and a semantics for them by means of timed Petri nets.

1 Introduction

Web Services Choreography and Orchestration specifications are aimed at the composition of interoperable collaborations between any type of party regardless of the supporting platform or programming model used by the implementation of the hosting environment.

In this paper we focus our attention on the Choreography layer, and specifically our intention is the description of timed and prioritized interactions. Then, we use a prioritized-timed model of Petri nets to capture the main elements of WS-CDL, thus providing a formal framework to precisely describe the behaviour of the parties involved in a choreography. The motivation is therefore twofold, on the one hand we obtain a graphical representation of this behaviour in terms of prioritized-timed Petri nets, which can be very helpful for the software designer in order to have a complete view of the Composed Web Service and the interactions that take place among the different participants. But Petri nets are also a formal tool, they allow to describe not only a static vision of a system, but its dynamic behaviour too. Then, we can use the Petri net representation to validate and verify the Composed Web Service.

In the literature we can find some related works, Yang Hongli et. al. [9] have also made a translation of WS-CDL into a formal model, in that case a small language (CDL), for which they provide a formal semantics. R. Hamadi and B. Benatallah [2] have proposed a Petri net-based algebra to model Web Services Control flows, thus constructions like sequence, choice, iteration, parallelism, discriminator, selection and refinement are considered in that paper, but they do not consider timed or prioritized interactions. Another Petri net representation of Web Services composition can be found in [3], in this case the starting point are descriptions written in BPEL4WS [1] and they are translated into a particular class of Petri nets called *workflow modules*. There is also another work that translates BPEL4WS into Petri nets [6]. We have found a timed Petri net representation of Web Services Flows in [5], in this case just the flow of messages and methods are considered, and the starting point is WSDL (Web Service Description Language) [7].

2 WS-CDL with priorities

WS-CDL has a *choice* construct, which allows us to choose among some different activities. Actually, when some of them are possible, it is assumed that the selection criteria are non-observable. Then,

* Supported by the spanish government (cofinanced by FEDER founds) with the project TIN2006-15578-C02-02, and the JCCLM regional project PAC06-0008-6995.

what we are proposing is that in some cases these criteria could be observable or known by the parties, i.e., they can be aware of the level of priority of interactions. Accordingly, we propose an extension of WS-CDL with priorities. Priorities are associated with interactions, as natural numbers, with the usual interpretation, the greater the number, the greater priority for the corresponding activity in the system. Therefore, we introduce the *priority* attribute in interaction activities, in order to indicate the priority level of the corresponding interaction. The interpretation of this attribute is the natural one, in case of conflict only the most prioritary interactions are allowed.

3 Prioritized-Time Petri Nets

The specific model of timed-prioritized Petri net that we consider for the translation is a prioritized extension of Merlin's nets [4, 8]. Due to the lack of space we omit a complete description of Merlin's nets, we just provide a brief description of the specific model that we consider.

In this model transitions are assigned both a time interval and a static priority. The time interval restricts the instants at which a transition is allowed to be fired[1], whereas the priority is used to resolve conflicts. Then, priorities are only used in case of conflict, when at a given marking two or more transitions are simultaneously enabled, then only the most prioritary one is allowed to be fired at that moment.

When a transition becomes enabled, a local clock associated with it is set, then the transition can fire when its clock has a value in the time interval associated with the transition. Furthermore, no time may elapse when a local clock has reached its latest firing time. The firing of a transition takes no time to complete, so they are immediate. It can also be the case for the fired transition to become enabled again at the new marking, in that case its local clock is reset.

We restrict our attention to a particular class of PTPNs, for which no transition will be enabled more that once at a time, i.e., it will never be the case that two or more instances of the same transition are enabled at a certain instant. With this restriction we avoid the semantic problems that appear in Merlin's nets when multiple enablings of transitions is allowed (see [8]).

4 PTPN Semantics for WS-CDL with Priorities

In this section we provide a PTPN semantics for a subset of WS-CDL with priorities. In the PTPN representation we will label each transition with the roletypes that are involved in its execution, but notice that it can also be the case that no specific RoleType is involved in the execution of a transition, in that case we will omit this information in the graphical representation of the PTPN. The obtained PTPNs will be 1-safe, which means that for every reachable marking we will have at most one token on every place. Furthermore, all of the generated PTPNs will have one initial place[2], which activates the PTPN when it is marked, and two exit places, which do not have any postconditions and cannot be marked simultaneously. These exit places correspond to the correct or erroneous termination of the system represented by the PTPN.

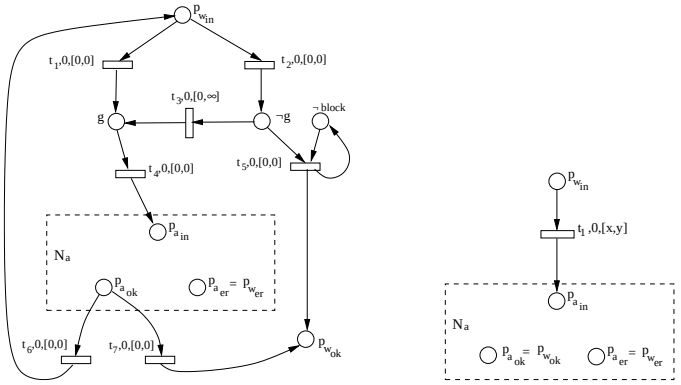
The translation is defined compositionally, so for each WS-CDL element a corresponding PTPN is provided. However, in some cases no translation is needed, because some elements are not relevant or their information is in fact integrated or used in other WS-CDL elements. Thus, elements like *RoleTypes, RelationShipTypes, Participants* and *Channels* are not translated. The same occurs with *InformationTypes* and *Variables* in general, although we allow time variables that are used to delay the execution of a workunit or to fix its execution at a certain instant. These variables can be used under some restrictions, with the goal to delay the execution.

For some WS-CDL elements the translation is simple, so we omit it. This is the case of choreographies, some basic activities (*Assign, Silent* and *Noaction*) and the *Sequence* and *Parallel* ordering structures. The more interesting cases are interaction activities, workunits and the *Choice* ordering

[1] Earliest and Latest Firing Time, with respect to the enabling time.
[2] This does not mean that this is the only initially marked place.

structure. Interaction activities may have associated both a *time-out* attribute and a priority attribute. The PTPN representation in this case is simple, the *time-out* is used to establish the time interval of the transition representing the interaction, and its priority is taken from the priority attribute.

Workunits are translated differently depending on their type. We may have workunits with time guards (using time variables) and general workunits. Fig. 1.a illustrates the translation for the general case (repetitive and guarded workunit), whereas Fig. 1.b illustrates the translation for workunits that are used to delay the execution.



(a) General Case (b) Delayed Workunit

Fig. 1. Workunit Translation

Finally, the translation for the *Choice* ordering structure requires again a distinction of cases. The point is that we may have (general) repetitive-guarded workunits as alternatives, and we must discard them when their guards are evaluated to false. Then, the translation is defined depending on the arguments of the choice. The most interesting case is that of a general guarded and repetitive workunit as alternative, for which observe that the choice is not resolved by evaluating the guard, but executing the first action in the activities inside the workunit. Unfortunately, due to the lack of space we cannot include the figures illustrating the *choice* translation.

References

1. T. Andrews et. al. BPEL4WS – Business Process Execution Language for Web Services. Version 1.1. May 2003., *http://www.ibm.com/developerworks/library/specification/ws-bpel/.*
2. Rachid Hamadi and Boualem Benatallah. A Petri Net-based Model for Web Service Composition. In *ADC '03: Proceedings of the 14th Australasian database conference*, 2003.
3. A. Martens. Analyzing web service based business processes. In *Proc. of International Conference on Fundamental Approaches to Software Engineering (FASE'05), Edinburgh, Scotland*, volume 3442 of *LNCS*, pages 19–33. Springer-Verlag, 2005.
4. P. Merlin. *A Study of the Recoverability of Communication Protocols*. PhD. Thesis, Univ. of California. 1974.
5. Johnson P. Thomas, Mathews Thomas, and George Ghinea. Modeling of Web Services Flow. In *IEEE International Conference on E-Commerce, Newport Beach, California, USA*, pages 391–398, June 2003.
6. H.M.W. Verbeek and W.M.P. van der Aalst. Analyzing BPEL Processes using Petri Nets. In *Proceedings of the Second International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management*, pp. 59–78, 2005.
7. W3C. Web Services Description Language (WSDL). Version 1.1. *http://www.w3.org/TR/wsdl/.*
8. J. Wang. *Timed Petri Nets, Theory and Application*. Kluwer Academic Publishers, 1998.
9. Hongli Yang, Xiangpeng Zhao, Zongyan Qiu, Geguang Pu, and Shuling Wang. A Formal Model for Web Service Choreography Description Language (WS-CDL). *International Conference on Web Services (ICWS'06), pp. 893-894, IEEE Computer Society Press. 2006.*

WST: A Tool for Verifying Web Services systems

María Emilia Cambronero, Valentín Valero, and Gregorio Díaz *

Departamento de Sistemas Informáticos
Escuela Politécnica Superior de Albacete
Universidad de Castilla-La Mancha
Campus Universitario s/n. 02071. Albacete, SPAIN
{MEmilia.Cambronero, Valentin.Valero, Gregorio.Diaz}@uclm.es

Abstract. In this paper we introduce a tool called Web Service Translation tool, WST for short, which we are developing to support a model-driven methodology, intended to the analysis, design, implementation, validation and verification of Web Services with time restrictions, called Correct-WS methodology. This methodology works by making several translations, from phase to phase in the life cycle. An important feature of this methodology is that at each phase the system is represented by XML models. Then, we use XSL Transformations (XSLT), which is a language for transforming XML documents.
The purpose of these translations is to obtain Web Services description models with Timed Automata for validating and verifying Web Services with time restrictions. For that, we use UPPAAL tool, which is used to simulate and analyse the behavior of Real-Time Systems described by Timed Automata.

1 Introduction

In some cases time restrictions are considered in the description of Web Services. Thus, not only the correct functioning of the system, but also the times required to perform some actions or to react to some possible events are of importance. Therefore, we deal, in this work, with real-time systems, for which some actions must be made in a bounded period of time. But notice that the order of these times can be greater than what is usually seen in classical real-time applications. Traditionally, real-time is associated with time restrictions in the order of seconds, milliseconds or even smaller (in some cases nanoseconds). But in Web Services the time restrictions are usually linked with transactions, for which the times considered can be in the order of minutes, hours or even days. These are still real-time systems, as we have a timeliness restriction for a transaction to be completed or canceled, or for a system to provide an answer to a query. For instance, we can think of a failure for a bank to receive a large electronic funds transfer on time, which may result in huge financial losses or we could consider a Seat Reservation System, in which the reservations are maintained for a limited period of time (several days).

The motivation of this paper is focused on the following points:

- It is difficult for non-XML expert to implement Web Services systems using WS languages, such as WS-CDL [1] and WS-BPEL [2], which are based in XML.
- Although there are several middleware platforms that support Web Services development, there is a lack of a solid methodological base for this.

Thus, the main goal of this work is the development of a tool, which allows us to fulfill these different motivation points.

2 Web Services Translation tool

In this section we describe the tool that we are developing to support the Correct-WS methodology [3]. This tool is called Web Service Translation tool, WST, and by means of it, we will be able to

* Supported by the spanish government (cofinanced by FEDER founds) with the project TIN2006-15578-C02-02, and the JCCLM regional project PAC06-0008-6995.

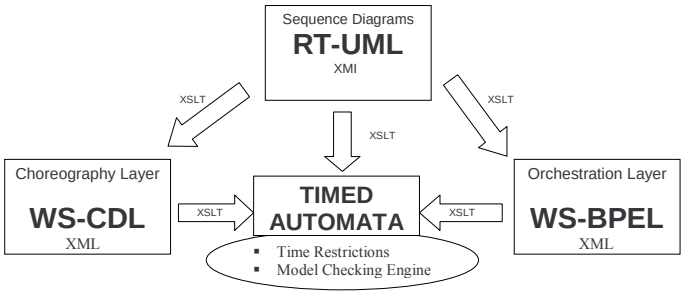


Fig. 1. Translation process in the current WST

perform the translations indicated in Fig. 1. This tool uses XSLT [4], XML Stylesheets Language for Transformation.

Web Services Translation tool (WST) is an integrated environment for translating RT-UML documents into WS-CDL specifications, RT-UML documents into Timed Automata [5], WS-CDL specifications into Timed Automata and it will also cover these relationships with WS-BPEL instead of WS-CDL. We can see in Fig. 2 the parts that are currently under development and the parts in which we are currently working. It is still in a Beta-state of development, in the sense that much effort must be still dedicated to cover all the indicated goals. Currently, the WST tool covers the phases corresponding to WS-CDL, which are shown in Fig. 2.

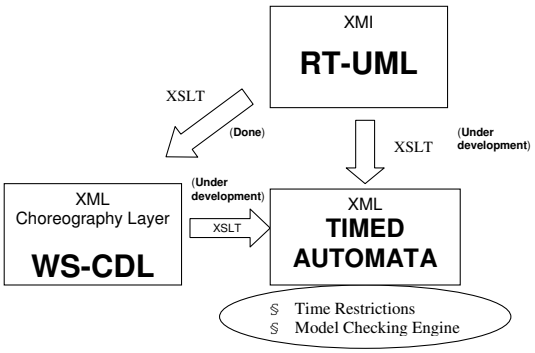


Fig. 2. Translation process for WS-CDL documents

WST applies XSL Stylesheets to an initial XML document to obtain another XML document, for instance we create XSL Stylesheets in order to translate an initial WS-CDL document into another XML document representing the Timed Automata system.

The first step in the translation process consists in designing a RT-UML sequence diagram to model the system. Currently, the design part of WST tool is under development, for this reason we are using a UML modeling tool for that purpose, specifically the Enterprise Architect [6], which allows us to design the RT-UML diagram and then to export it in a XMI document. Once we have designed the RT-UML diagram by using Enterprise Architect tool, we can open it in WST, and also the corresponding XMI in order to run the process translation on it.

Figure 3 shows the interface for the Web Service Translation tool once we have opened a RT-UML diagram. As you can see, WST has a menu, in the upper part, which currently consists of the following elements:

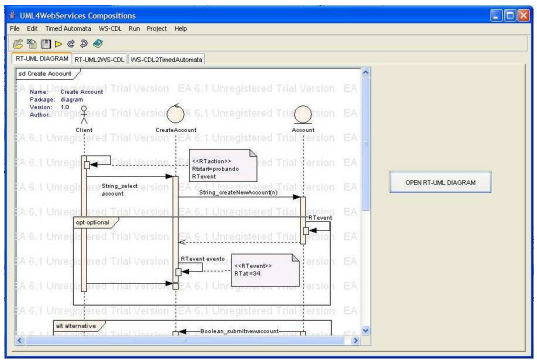


Fig. 3. A RT-UML Diagram in WST tool

- **File menu**: It consists of the submenus, which open and save WS-CDL or XMI documents, and allow to close WST tool.
- **Timed Automata menu**: It saves the generated timed automata file.
- **WS-CDL menu**: It allows to save the generated WS-CDL document. Once we have obtained the WS-CDL document from the corresponding RT-UML diagram, we can save it in a file, in order to use it later.
- **Help menu**: At present it shows general information about WST and a help guide.

The translations are easily obtained, for instance, once you have selected the translation to Timed Automata you just need to do the following:

1. Firstly, opening the WS-CDL document. Once the file is opened, it can be visualized in the left hand-side window in the main interface of WST.
2. After that, clicking on the Transform button.
3. Once the translation is made, the result is shown in the right window of the WST tool.
4. Finally, the resulting XML UPPAAL [7] document must be saved, in order to use it with the UPPAAL tool.

As explained above, the obtained document as a result of the translation can be directly used with the UPPAAL tool in order to validate the obtained system and verify the properties of interest.

References

1. Nickolas Kavantzas et al. Web Service Choreography Description Language (WSCDL) 1.0. http://www.w3.org/TR/ws-cdl-10/.
2. Assaf Arkin, Sid Askary, and et. al. Ben Bloch. Web Services Business Process Execution Language Version 2.0, December 2004. http://www.oasis-open.org/committees/download.php/10347/wsbpel-specification-draft-120204.htm.
3. M Emilia Cambronero Piqueras. *Description and Verification of Multimedia Systems and Web Services with Time Constraints*. PhD. Thesis, University of Castilla-La Mancha, 2007.
4. James Clark. XSL Transformations (XSLT) Version 1.0. Technical Report REC-xml-19980210, W3C, 1998. http://www.w3.org/TR/xslt.
5. Rajeev Alur and David L. Dill. Automata For Modeling Real-Time Systems. In *ICALP*, pages 322–335, 1990.
6. Enterprise Architect 6.5, 2007. http://www.sparxsystems.com.au/ea_downloads.htm.
7. Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL: Status and Developments. Number 1254, pages 456–459. LNCS, Springer–Verlag, June 1997.

A Theory of Contracts for Web Services*

Giuseppe Castagna
PPS (CNRS) Université Paris 7
Paris, France

Nils Gesbert
LRI (CNRS) Université Paris-Sud
Orsay, France

Luca Padovani
ISTI, Università degli Studi di Urbino,
Urbino, Italy

The Web Service Description Language (WSDL) [7] provides a standardised technology for describing the interface exposed by a service. Such a description includes the service location, the format (or *schema*) of the exchanged messages, the transfer mechanism to be used (i.e. SOAP-RPC, or others), and the *contract*. In WSDL, contracts are basically limited to one-way (asynchronous) and request/response (synchronous) interactions. The Web Service Conversation Language (WSCL) [2] extends WSDL contracts by allowing the description of arbitrary, possibly cyclic sequences of exchanged messages between communicating parties. Other languages, such as the Abstract Web Service Business Execution Language (WS-BPEL) [1], provide even more detailed descriptions by defining the subprocess structure, fault handlers, etc. While the latter descriptions are much too concrete to be used as contracts, they can be approximated and compared in terms of contracts that capture the external, observable behaviour of a service.

Documents describing contracts can be published in repositories so that Web services can be *searched* and *queried*. These two basic operations assume the existence of some notion of contract equivalence to perform service discovery in the same way as, say, type isomorphisms are used to perform library searches. The lack of a formal characterisation of contracts only permits excessively demanding notions of equivalence such as syntactical equality. In fact, it makes perfect sense to further relax the equivalence into a *subcontract preorder* (denoted by $\preceq$ in this paper), so that Web services exposing "larger" contracts can be *safely* returned as results of queries for Web services with "smaller" contracts.

In this work we develop a formal theory that precisely defines what "larger" and "smaller" mean, and which safety properties we wish to be preserved. Along the lines of [5] we describe contracts by a simple CCS-like syntax consisting of just three constructors: prefixing, denoted by a dot, and two infix choice operators + representing the *external choice* (the interacting part decides which one of alternative conversations to carry on); $\oplus$ representing the *internal choice* (the choice is not left to the interacting part). Thus $\alpha.\sigma$ is the contract of services that perform an action α and then implement the contract σ, $\sigma \oplus \tau$ is the contract of services that may decide to implement either σ or τ, while $\sigma + \tau$ is the contract of services that according to their client's choice, will implement either σ or τ.

Following CCS notation, actions are either write or read actions, the former being topped by a bar, and one being the *co-action* of the other. Actions can either represent *operations* or *message types*. As a matter of facts, contracts are behavioural types of processes that do not manifest internal moves and the parallel structure.

*A very preliminary version of this work was presented on January 20, 2007, Nice, France, at PLAN-X 2007, the 5th ACM SIGPLAN Workshop on Programming Language Technologies for XML.

Contracts are then to be used to ensure that interactions between clients and services will always succeed. Intuitively, this happens if whenever a service offers some set of actions, the client either synchronises with one of them (that is, it performs the corresponding co-action) or it terminates. The service contract will then allow us to determine the set of clients that *comply* with it, that is that will successfully terminate any session of interaction with the service.

Of course the client will probably be satisfied to interact with services that offer more than what the searched contract specifies. Intuitively we want to define an order relation on contracts $\sigma \preceq \tau$ such that every client complying with services implementing σ will also comply with services of contract τ. In particular, we would like the $\preceq$ preorder to enjoy some basic properties. The first one is that it should be safe to replace (the service exposing) a contract with a "more deterministic" one. For instance, we expect $\overline{a} \oplus \overline{b}.c \preceq \overline{a}$, since every client that terminates with a service that may offer either $\overline{a}$ or $\overline{b}.c$ will also terminate with a service that systematically offers $\overline{a}$. The second desirable property is that it should be safe to replace (the service exposing) a contract with another one that offers more capabilities. For instance, we expect $\overline{a} \preceq \overline{a} + \overline{b}.d$ since a client that terminates with services that implement $\overline{a}$ will also terminate with services that leave the client the choice between $\overline{a}$ and $\overline{b}.d$. If taken together, these two examples show the main problem of this intuition: it is easy to see that a client that complies with $\overline{a} \oplus \overline{b}.c$ does not necessarily comply with $\overline{a} + \overline{b}.d$: if client and service synchronise on b, then the client will try to write on c while the service expects to read from d. Therefore, under this interpretation, $\preceq$ looks as not being transitive:

$$\overline{a} \oplus \overline{b}.c \preceq \overline{a} \quad \wedge \quad \overline{a} \preceq \overline{a} + \overline{b}.d \quad \Longrightarrow \quad \overline{a} \oplus \overline{b}.c \preceq \overline{a} + \overline{b}.d\,.$$

The problem can be solved by resorting to the theory of *explicit coercions* [3, 6]. The flawed assumption of the approach described so far, which is the one proposed in [5], is that services are used carelessly "as they are". Note indeed that what we are doing here is to use a service of "type" $\overline{a} + \overline{b}.d$ where a service of type $\overline{a} \oplus \overline{b}.c$ is expected. The knowledgeable reader will have recognised that we are using $\preceq$ as an *inverse* subtyping relation for services. If we denote by $:>$ the subtyping relation for services, then $\overline{a} \oplus \overline{b}.c \;\; :> \;\; \overline{a} + \overline{b}.d$ and so what we implicitly did is to apply subsumption [4] and consider that a service that has type $\overline{a} + \overline{b}.d$ has also type $\overline{a} \oplus \overline{b}.c$. The problem is not that $\preceq$ (or, equivalently, $:>$) is not transitive. It rather resides in the use of subsumption, since this corresponds to the use of *implicit* coercions. Coercions have many distinct characterisations in the literature, but they all share the same underlying intuition that coercions are functions that embed objects of a smaller type into a larger type "without adding new computation" [6]. For instance it is well known that for record types one has $\{a{:}s\} :> \{a{:}s; b{:}t\}$. This is so because the coercion function $c = \lambda x^{\{a:s;b:t\}}.\{a = x.a\}$ embeds values of the smaller type into the larger one. In order to use a term of type $\{a{:}s; b{:}t\}$ where one of type $\{a{:}s\}$ is expected we first have to embed it in the right type by the coercion function c above, which erases (masks/shields) the b field so that it cannot interfere with the computation. Most programming languages do not require the programmer to write coercions, either because they do not have any actual effect (as in the case of the function c since the type system already ensures that the b field will never be used) or because they are inserted by the compiler (as when converting an integer into the corresponding float). In this case we speak of *implicit* coercions. However some programming languages (e.g. OCaml) resort to *explicit* coercions because they have a visible effect and, for instance, they cannot be inferred by the compiler.

Coercions for contracts have an observable effect, therefore we develop their metatheory in term of explicit coercions. However, coercions can be inferred so they can be kept implicit in the language and automatically computed at static time. Coming back

to our example, the embedding of a service of type $\overline{a}$ into $\overline{a} \oplus \overline{b}.c$ is the identity, since we do not have to mask/shield any action of a service of the former type in order to use it in a context where a service of the latter type is expected. On the contrary, to embed a service of type $\overline{a} + \overline{b}.d$ into $\overline{a}$ we have to mask (at least) the $\overline{b}$ action of the service. So in order to use it in a context that expects a $\overline{a}$ service we apply to it a *filter* that will block all $\overline{b}$ messages. Transitivity being a logical cut, the coercion from $\overline{a} + \overline{b}.d$ to $\overline{a} \oplus \overline{b}.c$ is the composition of the two coercions, that is the filter that blocks $\overline{b}$ messages. So if we have a client that complies with $\overline{a} \oplus \overline{b}.c$, then it can be used with a service that implements $\overline{a} + \overline{b}.d$ by applying to this service the filter that blocks its $\overline{b}$ messages. This filter will make the previous problematic synchronisation on b impossible, so the client can do nothing but terminate.

Filters thus reconcile two requirements that were hitherto incompatible: On the one hand we wish to replace an old service by a new service that offers more choices (that is *width subtyping*, e.g. $\sigma \mathrel{:>} \sigma + \tau$) and/or longer interaction patterns (that is *depth subtyping*, e.g. $a \mathrel{:>} a.\sigma$) and/or is more deterministic (e.g. $\sigma \oplus \tau \mathrel{:>} \sigma$). On the other hand we want clients of the old service to seamlessly work with the new one.

Two observations to conclude this brief overview. First, the fact that we apply filters to services rather than to clients is just a presentational convenience: the same effect as applying to a service a filter that blocks some actions can be obtained by applying to the client the filter that blocks the corresponding co-actions. Second, filters must be more fine grained in blocking actions than restriction operators as defined for CCS or the π-calculus. These are "permanent" blocks, while filters are required to be able to modulate blocks along the computation. For instance the filter that embeds $(a.(a + b)) + b.c$ into $a.b$ must block b only at the first step of the interaction and a only at the second step of the interaction.

References

[1] A. Alves, A. Arkin, S. Askary, C. Barreto, et al. *Web Services Business Process Execution Language Version 2.0*, January 2007. `http://docs.oasis-open.org/wsbpel/2.0/CS01/wsbpel-v2.0-CS01.html`.

[2] A. Banerji, C. Bartolini, D. Beringer, V. Chopella, et al. *Web Services Conversation Language (*WSCL*) 1.0*, March 2002. `http://www.w3.org/TR/2002/NOTE-wscl10-20020314`.

[3] K. Bruce and G. Longo. A modest model of records, inheritance and bounded quantification. *Information and Computation*, 87(1/2):196–240, 1990.

[4] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. A previous version can be found in Semantics of Data Types, LNCS 173, 51-67, Springer, 1984.

[5] S. Carpineti, G. Castagna, C. Laneve, and L. Padovani. A formal account of contracts for Web Services. In *WS-FM, 3rd Int. Workshop on Web Services and Formal Methods*, number 4184 in LNCS, pages 148–162. Springer, 2006.

[6] G. Chen. Soundness of coercion in the calculus of constructions. *Journal of Logic and Computation*, 14(3):405–427, 2004.

[7] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana. *Web Services Description Language (*WSDL*) Version 2.0 Part 1: Core Language*, 2006. `http://www.w3.org/TR/2006/CR-wsdl20-20060327`.

Towards A Delegation Framework for Aerial Robotic Mission Scenarios

P. Doherty[1] and J.-J. Ch. Meyer[2]

[1]Dept of Computer and Information Science,
Linköping University,
Sweden
patdo@ida.liu.se

[2]Dept of Information and Computing Sciences,
Utrecht University,
The Netherlands
jj@cs.uu.nl

Abstract. The concept of delegation is central to an understanding of the interactions between agents in cooperative agent problem-solving contexts. In fact, the concept of delegation offers a means for studying the formal connections between mixed-initiative problem-solving, adjustable autonomy and cooperative agent goal achievement. In this paper, we present an exploratory study of the delegation concept grounded in the context of a relatively complex multi-platform Unmanned Aerial Vehicle (UAV) catastrophe assistance scenario, where UAVs must cooperatively scan a geographic region for injured persons. We first present the scenario as a case study, showing how it is instantiated with actual UAV platforms and what a real mission implies in terms of pragmatics. We then take a step back and present a formal theory of delegation based on the use of 2APL and KARO. We then return to the scenario and use the new theory of delegation to formally specify many of the communicative interactions related to delegation used in achieving the goal of cooperative UAV scanning. The development of theory and its empirical evaluation is integrated from the start in order to ensure that the gap between this evolving theory of delegation and its actual use remains closely synchronized as the research progresses. The results presented here may be considered a first iteration of the theory and ideas.

A Security-by-Contracts Architecture for Pervasive Services

Fabio Massacci, Nicola Dragoni, and Ida S.R. Siahaan

DIT, Universitá di Trento, via Sommarive 14, 38050 Povo, Trento, Italy

1 Security-by-Contract (S×C)[3]

The paradigm of pervasive services [1] envisions a nomadic user traversing a variety of environments and seamlessly and constantly receiving services from other portables, handhelds, embedded or wearable computers. When traversing environments the nomadic user does not only invoke services according a web-service-like fashion (either in push or pull mode) but also download *new* applications that are able to exploit its computational power in order to make a better use of the unexpected services available in the environment. These *pervasive client downloads* will appear because service providers will try to exploit the computational power of the nomadic devices to make a better use of the services available in the environment. To address the challenges of this paradigm we propose the notion of *security-by-contract* (S×C), as in programming-by-contract, based on the notion of a mobile contract that a pervasive download carries with itself. It describes the relevant security features of the application and the relevant security interactions with its nomadic host.

S×C **Framework.** The framework of S×C is shaped by four stake-holders: mobile operator, service provider or developer, mobile user and third party security service providers. Application developers are responsible to provide a *contract*, i.e. a formal, complete and correct specification of the behavior of an application for what concerns relevant security actions (Virtual Machine (VM) API Calls, Operating System Calls). Each "application" consists of four components: *executable code*, *run-time level contract*, *proof of compliance*, and *application credentials*. By signing the code the developer binds the code with the stated claims on its security-relevant behavior thus providing a semantics to digital signatures. An example of a contract is "After Personal Information Management (PIM) was opened no connections are allowed".

Users and mobile phone operators are interested in that any software deployed on their platform is secure by declaring security policy. A *policy* is a formal complete specification of the acceptable behavior of applications to be executed on the platform for what concerns relevant security actions (Virtual Machine API Calls, Operating System Calls). An example of policy is "After PIM was accessed only secure connections can be opened i.e. url starts with "https://" ".

A contract should be negotiated and enforced during development, at time of delivery and loading, and during execution of the application code by the mobile platform. Fig. 1 shows the phases of the S×C life-cycle. S×C security architecture (Fig. 2) has two goals: supporting the application and service life cycle by guaranteeing the security of the channel between parties as well as authenticity of the parties and non-repudiation

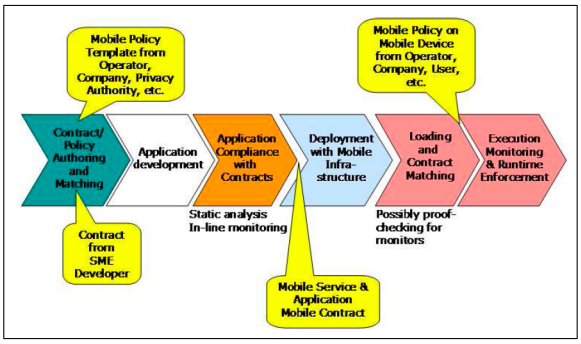


Fig. 1. Application/Service Life-Cycle

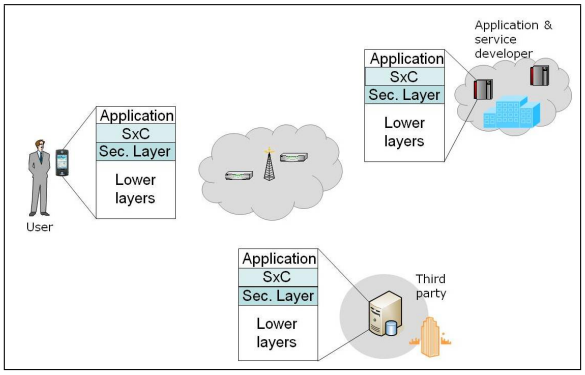


Fig. 2. S×C Architecture

of communication actions for charging and billing, and enabling trust relationships between stakeholders, i.e. authenticity and integrity of exchanged data elements.

2 Automata Modulo Theory ($\mathcal{AMT}$) [4]

We solved the problem of matching the security claims of the code with the security desires of the platform of S×C in [2] with only a meta-level algorithm showing how we can combine policies at different levels of details.The actual mathematical structure and algorithm to do the matching is specified in [4]. The key idea is based on the introduction of the concept of *Automata Modulo Theory* ($\mathcal{AMT}$). $\mathcal{AMT}$ enables us to define very expressive and customizable policies as a model for *security-by-contract* as in [2] and model-carrying code [6] by capturing the infinite transitions into finite transitions labeled as expressions in defined theories.

To represent a security behavior, provided by the contract and desired by the policy, a system can be represented as an automaton where transitions corresponds to the in-

voked methods as in the works on model-carrying code [6]. In this case, the operation of contract matching is a *language inclusion* problem.

$\mathcal{AMT}$ **Theory.** The theory of $\mathcal{AMT}$ is a combination of the theory of Büchi Automata (BA) with the Satisfiability Modulo Theories (SMT) problem. SMT problem pushes the envelope of formal verification based on effective SAT solvers. In contrast to classical security automata we prefer to use BA because besides safety properties, there are also some liveness properties which have to be verified. An example of liveness is "The application uses all the permissions it requests".

Definition 1 (Automaton Modulo Theory ($\mathcal{AMT}$)). *A tuple $A_{\mathcal{T}} = \langle E, S, q_0, \Delta_{\mathcal{T}}, F\rangle$ where E is a set of formulae in the language of the theory $\mathcal{T}$, S is a finite set of states, $q_0 \in S$ is the initial state, $\Delta_{\mathcal{T}} : S \times E \rightarrow 2^S$ is labeled transition function, and $F \subseteq S$ is a set of accepting states.*

$\mathcal{AMT}$ operations for intersection and complementation require that the theory is closed under intersection and complementation (union is similar to the standard one). We consider only the *complementation of deterministic* $\mathcal{AMT}$, because in our application domain all security policies are naturally deterministic (as the platform owner should have a clear idea on what to allow or disallow) (further details in [4]).

On-the-Fly State Model Checking with Decision Procedure. We are interested in finding counterexamples faster and we combine algorithm based on Nested DFS [5] with decision procedure for SMT. The algorithm takes as input the midlet's claim and the mobile platform's policy as $\mathcal{AMT}$ and then starts a DFS procedure over the initial state. When a suspect state which is an accepting state in $\mathcal{AMT}$ is reached we have two cases. First, when a suspect state contains an error state of complemented policy then we report a security policy violation without further ado. Second, when a suspect state which is an accepting state in $\mathcal{AMT}$ does not contain an error state of complemented policy we start a new DFS from the suspect state to determine whether it is in a cycle, in other words it is reachable from itself. If it is, then we report availability violation.

Theorem 1. *Let the theory $\mathcal{T}$ be decidable with an oracle for the SMT problem in the complexity class $\mathcal{C}$ then:*

1. *The non-emptiness problem for $\mathcal{AMTT}$ is decidable in $LIN - TIME^{\mathcal{C}}$.*
2. *The non-emptiness problem for $\mathcal{AMTT}$ is $NLOG - SPACE^{\mathcal{C}}$.*

References

1. J. Bacon. Toward pervasive computing. *IEEE Perv.*, 1(2):84, 2002.
2. N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Security-by-Contract: Toward a Semantics for Digital Signatures on Mobile Code. In *Proc. of EuroPKI*, 2007.
3. N. Dragoni, F. Massacci, C. Schaefer, T. Walter, and E. Vetillard. A security-by-contracts architecture for pervasive services. In *Proc. of SecPerU*, 2007.
4. F. Massacci and I. Siahaan. Matching Midlet's Security Claims with a Platform Security Policy using Automata Modulo Theory. *NordSec*, To Appear.
5. S. Schwoon and J. Esparza. A note on on-the-fly verification algorithms. Tech Rep 2004/06, Univ. Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, 2004.
6. R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. In *Proc. of SOSP*, 2003.

Applications of a Formal Contract Description Language to the Investment Banking Domain

Jean-Marc Eber
LexiFi Technologies
jeanmarc.eber@lexifi.com

Abstract

In our presentation, we firstly introduce the audience to the particular domain we are addressing, namely investment banking financial problems. After a quick presentation of the typical problems encountered by this industry when manipulating complex financial contracts, we show the practical benefits of a clearly defined operational semantics. We then give some intuition for understanding the problem of pricing complex contracts, a fundamental problem in this industry. We show that even if many different pricing models appear in the finance theory literature, a unified approach is indeed possible, and can be interpreted as a denotational semantics of our contract description language.

Our company, LexiFi, has developed an industrial solution around these ideas, mostly implemented as an application around a core library implemented in OCaml. We will present some typical usage examples, showing the combined use of LexiFi's pricing compiler and contract management engine. We emphasize the importance of some implementation details (for instance run-time code generation and partial evaluation) to achieve good run-time efficiency.

We present some user experience and discuss typical pitfalls to avoid. We also want to discuss with the audience future research and development directions for manipulating contracts.

Compositional contract specification for REA (Abstract)

Fritz Henglein Ken Friis Larsen Jakob Grue Simonsen
Christian Stefansen
Department of Computer Science, University of Copenhagen (DIKU)
{henglein,kflarsen,simonsen,cstef}@diku.dk

September 19th, 2007

Contracts When entrepreneurs enter contractual relationships with a large number of other parties, each with possible variations on standard contracts, they are confronted with the interconnected problems of *specifying* contracts, *monitoring* their execution for performance[1], *analyzing* their ramifications for planning, pricing and other purposes prior to and during execution, and *integrating* this information with accounting, workflow management, supply chain management, production planning, tax reporting, decision support *etc.*

Andersen, Elsborg, Henglein, Simonsen and Stefansen [AEH+06] define a typed language for compositional contract specification:[2]

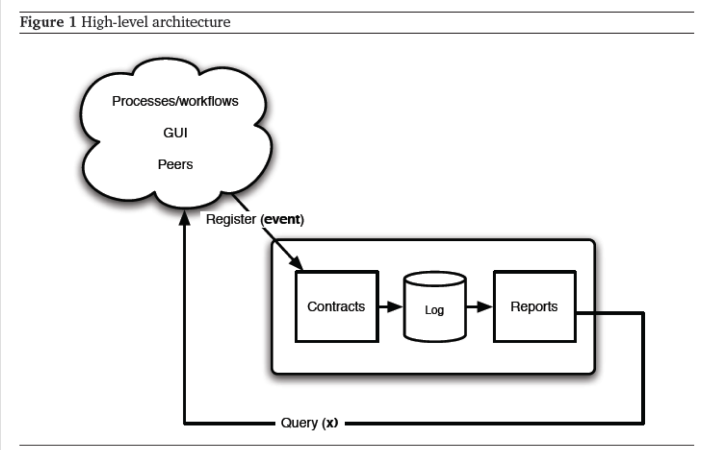
$$\begin{array}{rcl} c & ::= & \text{Success} \mid \text{Failure} \mid f(\vec{a}) \mid \text{transmit}(A_1, A_2, R, T \mid P).\, c \\ & \mid & c_1 + c_2 \mid c_1 \parallel c_2 \mid c_1; c_2 \end{array}$$

Success denotes the *trivial* or *(successfully) completed* contract: it carries no obligations on anybody. Failure denotes the *inconsistent* or *failed* contract; it signifies breach of contract or a contract that is impossible to fulfill. For a Boolean predicate P the contract expression $\text{transmit}(A_1, A_2, R, T \mid P).\, c$ represents a contract where the *commitment* $\text{transmit}(A_1, A_2, R, T \mid P)$ must be satisfied first. The commitment must be *matched* by a *transmit event* $e = \text{transmit}(v_1, v_2, r, t)$ of resource r from agent v_1 to agent v_2 at time t such that $P(v_1, v_2, r, t)$ holds. After matching, the residual contract is c in which A_1, A_2, R, T are bound to v_1, v_2, r, t, respectively. Note that A_1, A_2, R, T are binding variable occurrences whose scope is P and c. In this fashion the subsequent contractual obligations expressed by c may depend on the actual values in event e; such as a payment being due 8 days after delivery of the goods. The *contract combinators* $\cdot + \cdot, \cdot \parallel \cdot$ and $\cdot;\cdot$ compose subcontracts according to contract composition patterns: by alternation, concurrently, and sequentially, respectively. A (contract) context is a finite set of named contract template declarations of the form $f(\vec{X}) = c$. By using the *contract instantiation* (or *contract application*) construct $f(\vec{a})$ contract templates may be (mutually) recursive, which, in particular, captures repetition of subcontracts. Contract template definitions occur only at top level.

The language operates at two levels: the base level of (primitive commitments requiring the occurrence of) economic events such as transfer of resources between economic agents, reflecting the basic ontological concepts of the REA accounting

[1] *Performance* in contract lingo refers to *compliance* with the *promises* (contractual commitments) stipulated in a contract; nonperformance is also termed *breach of contract.*

[2] The types are elided here.



model [McC82]; and the compositional level of contract combinators. In commercial contracts only transfers of resources (goods, service, money) are included. Production events could be included, too, however. Indeed, with the compositional level being parametric in the base language, any kind of event types could be included in the base language, also outside the realm of economic events.

A contract-based event-driven architecture An event-driven architecture (EDA) [EDA06], is, loosely speaking, a software architecture that is organized around (data representing) *events* that *drive* system/component state transitions which in turn may generate events and other observable outputs.

We are presently working on developing a *contract-based* EDA for enterprise resource planning (ERP) systems [Wik]. Its high-level architecture is depicted in Figure 1. In this architecture, contracts such as standard or customized sales agreements, leases, *etc.* can be installed (entered) dynamically. Installed contracts are then matched against incoming events; after matching an event a contract is converted into an explicit representation—again as a contract—of the residual obligations.

Being *data* (residual) contracts have additional uses beyond monitoring their execution. They can be inspected, audited, analyzed and changed in response to failures to perform. Standard or customized report functions can be installed that at any point in time can be applied to the log of registered events alone (*ex-post* reports such as payments received) or, more interestingly, to *both* the log *and* the current contract states. An example of such an *ex-ante* analysis could be inventory restocking required to fulfil future demand based on both currently open orders and previously expedited orders. A basic ex-ante analysis for extracting deadline-ordered task lists has been described for the commercial contracts of Andersen *et al* [AEH$^+$06]. Peyton-Jones and Eber [JE03] have demonstrated sophisticated compositional pricing analysis for financial contracts. The key point here is that such analyses are defined once and for all for *all* definable contracts in an expressive language, not just a fixed finite set of *given* contract templates. Consequently, a custom contract not used before is automatically covered and does not require development of specialized analysis software.

Contracts can be thought of as declarative formal (behavioral, temporal) interface specifications in the spirit of software design by contract [Mey97], without any requirement for modeling real-world contracts. As such a contract functions as a behaviorial type for the process that generates the (expected) events. In ERP systems most basic processes that generate events such as order entry and financial bookkeeping are not automated, but performed by humans. Since a contract is an explicit representation of what events are expected (allowed) to happen next, contracts specifications can be used to automatically derive a user interface that prompts and guides the user through contract execution, guaranteeing that all and only relevant user interface options are provided at any given point during execution.

For an automated (executable) process that generates events contract residuation provides run-time verification. Conceivably, with both process code and contract specification in hand it should principally be possible to prove *statically* that a process always complies with its contract. This is bound to require a rather drastic limitation of expressive power of the base language to achieve practical analyzability while retaining sufficient expressiveness and generality for the intended domain-specific applications, however. Where such a "soft spot" is—and whether it exists at all—remains to be seen for now.

Acknowledgements The above reflects ongoing work within the 3d generation Enterprise Resource Planning Systems Project (3gERP.org), a collaboration between Copenhagen Business School, University of Copenhagen and Microsoft Development Center Copenhagen made possible by a grant by the Danish National Advanced Technology Foundation.

The section on contracts is excerpted from Andersen, Elsborg, Henglein, Jakobsen, Stefansen [AEH+06]; Figure is from Larsen, Simonsen, Stefansen [LSS07]; both with permission by the authors.

References

[AEH+06] Jesper Andersen, Ebbe Elsborg, Fritz Henglein, Jakob Grue Simonsen, and Christian Stefansen. Compositional specification of commercial contracts. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(6):485–516, November 2006.

[EDA06] *Workshop on Event Driven Architecture*, October 2006. http://www.haifa.il.ibm.com/Workshops/oopsla2006/present.html.

[JE03] Simon Peyton Jones and Jean-Marc Eber. *How to Write a Financial Contract*. Palgrave Macmillan, 2003. In: The Fun of Programming.

[LSS07] Ken Friis Larsen, Jakob Grue Simonsen, and Christian Stefansen. Towards a new high-level architecture for ERP systems (position paper). October 2007. http://www.3gERP.org/workshop.

[McC82] William E. McCarthy. The REA accounting model: A generalized framework for accounting systems in a shared data environment. *The Accounting Review*, LVII(3):554–578, July 1982.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997. ISBN 0-13-629155-4.

[Wik] Wikipedia. Enterprise resource planning. http://en.wikipedia.org.

On the formal representation of contracts: veriification and execution monitoring

(Abstract for FLACOS'07)

Robert Craven Alessio Lomuscio Hongyang Qu
Marek Sergot Monika Solanki

Department of Computing, Imperial College London
180 Queen's Gate, London SW7 2BZ, UK
{rac101,alessio,hongyang,mjs,monika}@doc.ic.ac.uk

We describe the main elements of a project we have recently started on the formal representation of contracts and the business processes within which they are enacted, outlining the motivation, technical objectives, and preliminary work completed so far. Our aim is to integrate and then build upon three separate strands of previous work.

(1) the use of the event calculus for tracking the evolving state of a contract as it is enacted [5]. The contract state consists of the obligations, permissions, and powers ('capacities', 'competences') of the contracting parties and the values of various other state variables. The system monitors a stream of recorded event occurrences and maintains the current state of the contract as well as giving access to all past states if required, for instance for auditing purposes. We use a Java implementation of the event calculus with an XML encoding of contract terms and event occurrences. The system has been applied to a range of examples, mostly concerning Service Level Agreements in the context of Utility Computing.

(2) executable specifications of 'open agent societies' [1, 2]. Although not directly targeted at the representation of contracts and business processes, this work shares many of the essential features. The specification of an 'open agent society' consists of four components: first, the possible behaviours, causal relations, and physical capabilities of the member agents; second, constitutive norms defining institutional concepts and relations (such as, in a contract setting, 'designated carrier', 'mode of delivery', 'supervising engineer', and so on) and specifying the powers ('capacities', 'competences') of agents to create new instances of institutional relations or to effect changes; third, permissions, prohibitions and obligations of the agents; and fourth, sanctions, enforcement policies, and recovery procedures that deal with the performance of forbidden actions and non-compliance with obligations. We have used both the event calculus and the action language $\mathcal{C}+$ [6] to execute such specifications. An advantage of the event calculus is that it is easily and efficiently implemented for certain computational tasks, specifically computing states from recorded event narratives, and

enables the full power of logic programming to be used for the specification of time-independent auxiliary concepts. An advantage of the $\mathcal{C}+$ language, which intuitively resembles the event calculus in many respects, is that it can be given an explicit semantics in terms of labelled transition systems. It thereby provides a bridge to a wide range of existing tools and techniques, such as model checkers in particular. We also have an extended form of $\mathcal{C}+$ specifically designed for representing institutional and normative concepts [10, 3]. This language extends labelled transition systems with features to distinguish between compliant (ideal, acceptable, permitted, legal) and non-compliant (sub-ideal, unacceptable, forbidden, illegal) states and behaviours. Normative system properties expressed in temporal logics such as CTL can then be verified by means of standard model checking techniques (specifically the model checker NuSMV).

(3) symbolic model checking techniques for the verification of normative and temporal epistemic properties of multi-agent systems [8]. This strand of work is based on an extended form of the 'interpreted systems' framework [4] used for analysing epistemic properties in multi-agent systems and distributed computer systems. 'Deontic interpreted systems' [9] add a means of analysing epistemic system properties when components of the system fail to function according to their specification. We are exploring the application of these techniques to the formal modelling of web services. The model checking of web service behaviour has remained limited to verifying simple termination, safety, and liveness properties. When viewed as a multi-agent system, however, the system composition can be analysed by considering additional properties which capture the knowledge acquired by services during their interactions. We believe that the specification and verification of these epistemic properties will facilitate greatly the analysis of a number of properties of the system, including Service Level Agreements and contracts which define the allowed (acceptable, legal) behaviours of the parties in the composition. A small example is presented in [7]. We use a specialised system description language (ISPL) paired with a symbolic model checker (MCMAS) optimised for the verification of temporal epistemic properties [8]. This formalism can be seen as adding to the formalisms described in (2) above the concepts of an agent's local state and its local protocol/behaviour.

Our aim is to integrate these three strands of work. We want to support both (1) run-time monitoring of contract execution and implementation of the enabling infrastructure (e.g., web services), and (2) off-line (design-time) verification of contract and system properties.

For illustration we will present (in simplified form) one of the examples we are using to drive the development. It concerns a contract for the production and phased delivery of a complex artefact comprising several components (in the actual example, a large software product commissioned and specified by the client). The contract specifies an agreed schedule of monitoring and progress report points, delivery milestones for the various phases of the project, and mechanisms allowing the client to request changes to the specification. These changes can be minor modifications, or major revisions such as cancellation of a complete sub-component.

The emphasis in the example (as in many other examples of contracts) is not so much on obligations and sanctions, which are relevant but comparatively peripheral, but rather on detailing the terms under which the delivered product is to be regarded as meeting its specification, and the powers and procedures

by means of which changes to the specification are requested, approved, and effected. The example can be further elaborated by adding details of how requests are passed to an appropriately empowered agent for approval (resembling examples often used in the literature on web service composition), or by considering the case where the supplier sub-contracts part of the construction to one or more other parties, with a similar structure of reporting, delivery, and change request points in the sub-contract.

We will present a stylised version of the example and sketch some of the options for its formal representation.

References

[1] A. Artikis, J. Pitt, and M. Sergot. Animated specifications of computational societies. In C. Castelfranchi and L. Johnson, editors, *Proceedings of Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 1053–1062. ACM Press, 2002.

[2] Alexander Artikis, Marek Sergot, and Jeremy Pitt. Specifying Norm-governed Computational Societies. *ACM Transactions on Computational Logic*, 2007. To appear.

[3] Robert Craven and Marek Sergot. Agent strands in the action language nC+. *Journal of Applied Logic*, 2007. To appear.

[4] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge*. MIT Press, Cambridge, 1995.

[5] Andrew D. H. Farrell, Marek Sergot, Mathias Sallé, and Claudio Bartolini. Using the event calculus for tracking the normative state of contracts. *International Journal of Cooperative Information Systems*, 14(2–3):99–129, June & September 2005.

[6] E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain, and H. Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 153(1–2):49–104, 2004.

[7] A. Lomuscio, H. Qu, M. Sergot, and M. Solanki. Verifying temporal epistemic properties of web service compositions. In *Proc. 5th International Conference on Service-Oriented Computing (ICSOC'07), Vienna, September 2007*, LNCS. Springer Verlag, 2007. To appear.

[8] A. Lomuscio and F. Raimondi. MCMAS: a tool for verifying multi-agent systems. In *Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2006)*. Springer-Verlag, 2006.

[9] A. Lomuscio and M. J. Sergot. Deontic interpreted systems. *Studia Logica*, 75(1):63–92, October 2003.

[10] Marek Sergot and Robert Craven. The deontic component of action language nC+. In L. Goble and J-J. Ch. Meyer, editors, *Deontic Logic and Artificial Normative Systems. Proc. 8th International Workshop on Deontic Logic in Computer Science (DEON'06), Utrecht, July 2006*, LNAI 4048, pages 222–237. Springer Verlag, 2006.

Monitor-based Runtime Reflection

– Extended Abstract –

Martin Leucker and Christian Schallhart

Institut für Informatik · Technische Universität München · Germany

1 Introduction

In this extended abstract, we sketch the concepts of runtime verification, monitor-oriented programming, and monitor-based runtime reflection and discuss their similarities and differences. Runtime verification is mainly concerned with monitoring a (correctness) property at runtime, i.e. when executing a system. Monitor-oriented programming aims at a programming methodology that allows for the execution of code whenever monitors observe a violation of a given correctness property. Runtime reflection is an architecture pattern that is applicable for systems in which monitors are enriched with a diagnosis and reconfiguration phase. We briefly discuss their possible use in contract enforcement.

2 Runtime verification

Runtime verification [5] deals with checking whether an execution of a system under scrutiny satisfies or violates a given correctness property. It aims to be a *lightweight* verification technique complementing techniques such as *model checking* [4] and common testing techniques [2].

As in some model checking approaches, in runtime verification, a correctness property φ is usually formulated in some linear temporal logic, such as LTL [6], and automatically translated into a *monitor*. Such a monitor is then used to check the *current* execution of a system or a (finite set of) *recorded* execution(s) for satisfaction of the property φ. In the previous case, we speak of *online monitoring* while in the latter we speak of *offline monitoring*.

As runtime verification does not consider each possible execution of a system, but just a given subset, it shares similarities with *testing*, which is also usually not complete. While runtime verification shares also many similarities with *model checking*, there are important differences:

- While in model checking, *all executions* of a given system are examined to answer whether they satisfy a given correctness property φ, which corresponds to the language inclusion problem, runtime verification answers whether a *single execution* satisfies φ, which is called the word problem. For most logical frameworks, the word problem is far lower complexity than the inclusion problem.
- While model checking, especially when considering LTL, considers *infinite* traces, runtime verification deals with *finite* executions—as executions have necessarily to be finite.
- While in model checking a complete model is given allowing to consider arbitrary positions of a trace, runtime verification, especially when dealing with online monitoring, considers finite executions of increasing size. For this, a monitor should be designed to consider executions in an *incremental fashion*.

These differences make it necessary to adapt the concepts developed in model checking to be applicable in runtime verification. For example, the second item asks for coming up with a semantics for LTL on finite traces [1].

Furthermore, from an application point of view, it is important to know that, as only observed executions are checked, runtime verification is applicable for black box systems for which no system model is at hand, in contrast to model checking.

In contrast to *Monitor-oriented programming* and *Runtime Reflection*, runtime verification typically does *not* interfere with the system under scrutiny. Thus, when a violation has been observed, it typically does not influence the program's execution.

3 Monitor-oriented programming

Monitoring-Oriented Programming (MOP) [3], proposed by Feng and Rosu, is a software development methodology, in which the developer specifies desired properties using a variety of (freely definable) specification formalisms,

along with code to execute when properties are violated or validated. The MOP framework automatically generates monitors from the specified properties and then integrates them together with the user-defined code into the original system. Thus, it extends ideas from runtime verification by means for *reacting* on detected violations (or validations) of properties to check. This allows the development of *reflective* software systems: A software system can monitor its own execution and the subsequent execution is influenced by the code a monitor might execute–again influencing the behavior of the monitor etc.

4 Runtime Reflection

Monitor-based runtime reflection or short *runtime reflection* (RR) is an architecture pattern for the development of reliable systems. The main idea is that a *monitoring layer* is enriched with a *diagnosis layer* and a subsequent *mitigation layer*. We first show the pattern with respect to information flow in a conceptual manner, before presenting a realization for a distributed system.

The layered view The architecture consists of four layers as shown in Figure 1, whose role will be sketched in the subsequent paragraphs.

Logging—Recording of system events. The role of the *logging layer* is to observe system events and to provide them in a suitable format for the monitoring layer. Typically, the logging layer is realized by adding code annotations within the system to build. However, separated standalone loggers, logging for example network traffic, also can realize this layer as well. While the goal of a logger is to provide information on the current run to a monitor, it must not assume (much) on the properties to be monitored.

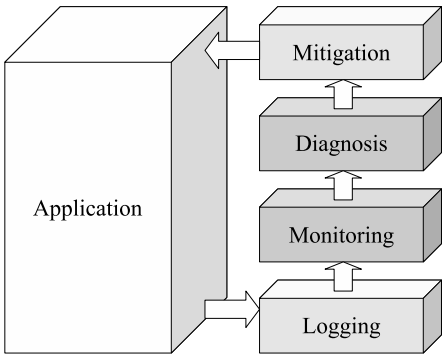


Fig. 1. An application and the layers of the runtime reflection framework.

Monitoring—Failure detection. The *monitoring layer* consists of a number of monitors (complying to the logger interface of the logging layer) which observe the stream of system events provided by the logging layer. Its task is to detect the presence of failures in the system without actually affecting its behavior. It is typically implemented via *automatically generated monitors* which—each locally with respect to a certain subsystem or system's component—monitor often *safety properties*. A typical example is the exclusion of certain critical system states, e. g., one always wants to ensure that $\neg(critical_1 \wedge critical_2)$ holds. If a violation of a safety property is detected in some part of the system, the generated monitors will respond with an alarm signal for subsequent diagnosis.

Diagnosis—Failure identification. We deliberately separate the identification of symptoms from the detection of failures in terms of a dedicated diagnosis system. The *diagnosis layer* collects the verdicts of the distributed monitors and deduces an explanation for the current system state.

For this purpose, the diagnosis layer infers a (minimal) set of system components, which must be assumed faulty in order to explain the currently observed system state. The procedure is solely based upon the results of the monitors and general information on the system. Thus, the diagnostic layer is not directly communicating with the application.

Mitigation—Failure isolation. The results of the system's diagnosis are then used in order to *isolate* the failure, if possible. However, depending on the diagnosis and the occurred failure, it may not always be possible to re-establish a determined system behavior. Hence, in some situations, e. g., occurrence of fatal errors, a recovery system may merely be able to store detailed diagnosis information for off-line treatment.

The distributed-system view So far, we merely discussed the tier-structure of our architecture, while we did mostly ignore the distributed nature of it. However, the distribution is oriented towards the layering of the framework: the logging layer and the monitoring layer consist both of a number of different software components, which are distributed throughout the system under scrutiny. Each local monitor computes a verdict on the locally observed event stream and provides this verdict for further, subsequent diagnosis regarding the system's general status. The diagnosis and mitigation layers, in contrast to logging and monitoring, are realized in terms of centralized components, which collect the information of the monitors in order to compute and react upon a global system view.

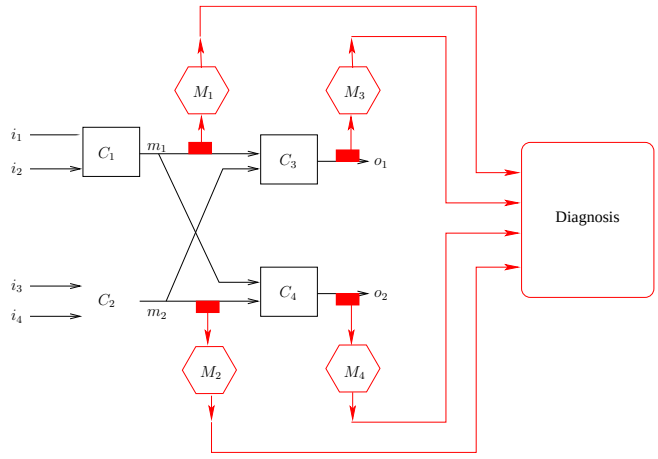


Fig. 2. Distributed monitoring & diagnosis.

For instance, consider Figure 2, where we show an example application consisting of four distributed components, $C_1, \ldots, C_4$. To monitor the overall system behavior of this application, we employ four dedicated monitoring applications, $M_1, \ldots, M_4$, to the system. Each monitor M_i is then locally observing the output of a single component, C_i, and computes its verdict on the correctness of the observed output stream so far. These distributed verdicts are then transmitted back to the central diagnosis component for further treatment via the application's communication infrastructure which, depending on the nature of the system, may be a physical bus system or merely remote procedure calls, for instance.

4.1 MOP versus RR

RR differs from monitor-oriented programming in two dimensions. First, MOP aims at a programming methodology, while RR should be understood as an architecture pattern. This implies that MOP support has to be tight to a programming language, for example Java resulting in jMOP, while in RR, a program's structure should highlight that it follows the RR pattern. The second difference of RR in comparison to MOP is that RR introduces a diagnosis layer not found in MOP.[1]

5 Runtime Reflection and Contract Enforcement

According to [7], a *contract* is a document which engages several parties in a transaction and stipulates their obligations, rights, and prohibitions, as well as penalties in case of contract violation. *Contract enforcement* terms the problem of monitoring contract fulfillment as well as enforcing the penalty, when a contract violation has been observed. Thus, contract enforcement apparently matches runtime reflection: While monitoring contract fulfillment provides suitable properties to verify at runtime, the enforcement of penalties asks for mitigation. However, a detailed study in this direction has to be done.

References

1. Andreas Bauer, Martin Leucker, and Christian Schallhart. Monitoring of realtime properties. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, 2006. 260–272.
2. Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005.
3. Feng Chen and Grigore Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In *Object-Oriented Programming, Systems, Languages and Applications(OOPSLA'07)*, 2007. to appear.
4. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
5. Séverine Colin and Leonardo Mariani. Run-time verification. In *Model-based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005.
6. Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, Providence, Rhode Island, October 31–November 2 1977. IEEE Computer Society Press.
7. Cristian Prisacariu and Gerardo Schneider. A formal language for electronic contracts. In *9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'07)*, volume 4468 of *LNCS*, pages 174–189, Paphos, Cyprus, June 2007. Springer.

[1] Clearly, in the MOP framework, a diagnosis can be carried out in the code triggered by a monitor. This yields a program using the MOP methodology and following the RR pattern.

Contracts as a support to static analysis of open systems

Extended Abstract

Nadia Bel Hadj Aissa, Dorina Ghindici, Gilles Grimaud, and Isabelle Simplot-Ryl

IRCICA/LIFL CNRS UMR 8022, Univ. Lille 1, INRIA Futurs
Parc Scientifique de la Haute Borne
50, avenue Halley - BP 70478
59658 Villeneuve d'Ascq Cedex, France

Static analysis is a powerful tool to establish various properties of programs. The analysis is often directed by the call graph of the programs (*e.g* [3]) and thus is not well suited to open object-oriented systems, or sometimes consider that when a method is called, all its parameter escape to any control (*e.g.* [1]). In this work in progress, we introduce the notion of contract as a support to openness and extensibility: we propose an analysis computing the contract of a method relying on contracts of methods used by it, and respecting the requirements of the methods using it. Thus the analysis is compositional and the properties of composed code can be deduced from contract composition.

We give two examples of applications in the context of open object-oriented systems and dynamic loading of applications: one for a distributed WCET computation method for small embedded systems and one for computation of information flows in Java programs. These examples allow us to present two types of contracts: the contracts that are introduced by the designer of the system, and the contracts that are automatically computed by the analysis with regard to the first ones.

1 General idea

Static analysis of programs has been used for a long time to deduce and prove properties of systems. In this work, we are interested in static analysis of open object oriented systems supporting dynamic class loading, typically Java programs. The openness of the systems combined with subclassing and overriding leads to classical problems when coming to analysis: it is not possible to rely on any call graph or on any calling context when analyzing a piece of code. We here consider two opposed worlds: static analysis and dynamically evolving systems. Thus, we aim at computing properties of a system, based on a static description, which does not always correspond to the system that will be executed. This can be a disadvantage when the analysis is used for example for optimization purposes and this is not acceptable when it is used for example for security properties.

Thus, we propose an analysis, which uses the notion of contract to describe the minimal behaviour that a piece of code guaranties relying on the contracts of the pieces it uses. As we focus on object-oriented languages, the natural grain of the analysis is the method. Thus, we propose to add to each method signature a contract that describes it behaviour regarding to the property we are analyzing and to analyze each method with respect to the contracts of the called methods. This contract is i) produced by the static analysis, ii) used by static analysis.

The analysis of a single method can be done in a very classical way, but instead of unfolding and analyzing the called methods, we use the contract of the called methods. Contracts of other methods can be obtained in several ways depending on various factors:

- Other methods have already been analyzed and their contract has been computed by the analysis, their are available in some repository,
- Some methods have not been analyzed. Then:
  - The user can provide a contract, either because the method cannot been analyzed, *e.g.* for native methods in Java, or because the method is not available, *e.g.* when considering mobile code,
  - In some cases, the analysis itself can compute the contract that the called method must respect.

The last case is the most automatic one, since the analysis itself can specify the required contract, but it mainly depends on the nature of contracts: this is possible in the example of Section 2, since contracts have mainly the form of equations on integers. Contracts of Section 3 are more complex and still need user interactions when the code is not available.

2 Application to WCET computation

In real-time systems, the prediction of the worst case execution time (WCET) of a program comes twofold. First, find and express the timing behavior in the worst case of a code unit (*i.e.* intra-method analysis). Second, combine these properties to find out the end-to-end timing behavior of a composition of code units (*i.e.* inter-method analysis). We aim to establish that an open system can continue to respect strict deadlines if a new method overriding one of the existing methods is loaded.

When a method `A` contains an open call site `B` (*i.e.* Open call sites are the one whose sets of target methods resolved statically may not contain all methods invoked at runtime), the idea is that the static analysis generates a contract that describes the constraints the target methods must respect.

```
void A(){
   if (exp)
      statement;
   else
      B();
}
```

Fig. 1. Example 1

Consider the example of Figure 1. Computing the WCET of the method `A`, which contains a simple alternative construct leads to Equation 1, where W denotes the WCET. Then, the WCET of method `B` is the maximum of all the WCET of the methods overriding `B` (`B'` overrides `B` is denoted $\texttt{B'} \sqsubseteq \texttt{B}$), this leads to Equation 2. The contract the method `A` must respect says that its WCET is less than the deadline defined by the specifications of the system (see Equation 3). This imposes constraints on the WCET of any new method overriding `B` and willing to be executed on the system as described by Equation 4.

$$W(\texttt{A}) = W(\texttt{if}) + W(\texttt{exp}) + Max(W(\texttt{statement}), W(\texttt{B})) \quad (1)$$

$$= W(\texttt{if}) + W(\texttt{exp}) + Max(W(\texttt{statement}), \underset{\texttt{B'}\sqsubseteq\texttt{B}}{Max}(W(\texttt{B'}))) \quad (2)$$

$$deadline \geq W(A) \quad (3)$$

$$\geq W(\texttt{if}) + W(\texttt{exp}) + Max(W(\texttt{statement}), \underset{\texttt{B'}\sqsubseteq\texttt{B}}{Max}(W(\texttt{B'})) \quad (4)$$

Then the contract $Y \geq W(\texttt{B})$ where $Y = deadline - W(\texttt{if}) - W(\texttt{exp})$ is added to the contract repository.

Each time a method that is `B` or that overrides `B` is loaded, its WCET must satisfy the contracts otherwise it is rejected. When a method depends on several open calls, its contract depends on several variables. When one of the methods is loaded and accepted by

the system, contracts are updated progressively while respecting the end-to-end deadline, thus becoming more and more restrictive. In this way, methods that are loaded on the system must respect the contracts of the repository and can add contracts for forthcoming methods.

3 Application to information flow

In the constantly evolving computer systems, security becomes a critical factor. One of the main concerns of current security issues is the flow of private data in Java programs; the private data must satisfy a security policy (e.g. must be accessible only to authorized users throughout the control flow of the program). A number of works of information flow use static analysis. They are mostly dedicated to close world: they analyze a system and track illicit flows. These concerns are also (and maybe more) important in modern ubiquitous systems where systems support multiple applications and mobile code, provided by various issuers.

```
String passwd;
boolean login(String s){
   if(s.equals(passwd))
      return true;
   return false;
}
```

Fig. 2. Example 2

Then, the desired security policy can be defined as a contract. For some methods, contracts can be defined by the user or deducted by the analysis starting from the predefine contracts. In an open world, the untrusted code loaded in an already existing system must meet the desired security requirements of the code already loaded. This must be the case for newly loaded methods but also for methods overriding already loaded methods: overriding cannot be a way to bypass the security controls. Thus, we need to define an order on contracts and to accept new applications/code if their contract is stronger than the requirements of the existing system.

In [2], we have defined a framework, which ensures that security policies are meet for an open system even when new applications are loaded. Let us consider the example in Figure 2. The security policy of the application states that the field `passwd` is private and some user who can observe the behaviour of the program must not deduct it. The contract of the method `login` says that there exists a flow from `passwd` to `s` and to the `return` value. This information can be confronted with the requirements of users of `login`. Moreover, as `passwd` is used with the open site `equals`, we must know its contract. For the moment, the analysis takes the contract of `equals` in the repository and uses it. In the future, we aim at being able to compute the contract required for the missing code as in the case of WCET. The general framework is the same but the expression of security contracts and flows between data is more complex than the expression of deadlines.

References

1. Matthew Q. Beers, Christian Stork, and Michael Franz. Efficiently verifiable escape analysis. In *Proc. of ECOOP 2004 - Object-Oriented Programming, 18th European Conference*, volume 3086 of *LNCS*, pp. 75–95, Oslo, Norway, 2004.
2. Dorina Ghindici, Gilles Grimaud, and Isabelle Simplot-Ryl. Embedding verifiable information flow analysis. In *Proc. 4th Annual Conference on Privacy, Security and Trust*, pp. 343–352, Toronto, Canada, 2006. McGraw-Hill.
3. Andrew C. Myers. Jflow: practical mostly-static information flow control. In *Proc. 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'99)*, pp. 228–241. ACM Press, 1999.

Describing Software Components with Parametric Contracts

Ralf H. Reussner
Institute for Program Structures and Data Organisation
Universität Karlsruhe (TH), Germany

1. Challenges in Component Specification

The underlying motivation for parametric contracts for software components stems from a rather general question: How to specify a component, given the fact that most properties of a component depend on the component's deployment environment.

Before discussing this question, we will give a short definition of the entities we have in mind when using the term "software component".

> A *software component* is a contractually specified building block for software which can be readily used. *Readily used* firstly means the component can be used without that the software architect or system deployer (as users of the component) having to understand the component's internals. Secondly, the use of a component is restricted to three different actions: (a) component composition with other components to form composite components, (b) component deployment on resources such as virtual machines, application servers or containers and (c) component use during run-time, i.e., calling component services.

By this, it becomes clear that classes are not components, as use by inheritance needs a thorough understanding of the internals of the superclass when overwriting inherited methods. When using models we do not have this problem of inheritance, but the lack of the explicit specification of the external dependencies of a module also requires the consideration of their internals before use. As components are a new entity (compared to other software encapsulation mechanisms, such as classes or modules), we are interested in meta-modelling them. Here we have to face the challenge that component properties are not constant, but depend heavily on four influencing factors:

The implementation: For obvious reasons the implementation of a component influences its functional and extra-functional properties. Of course, the implementation can be described in a more abstract specification.

The external services called: Extra-functional properties of a component depend on such properties of external components. But also the provision of functional properties (e.g., whether an implemented service can actually be offered) depends on the presence of external services. (If one of several external services that are required is missing, the component cannot offer all its implemented services, but possibly a non-empty subset of them.)

The execution environment: In particular the extra-functional properties of the component execution system influence directly the extra-functional properties of the component services (as the speed and any failure of the execution system directly influences the component). In modern enterprise systems, the execution environment is a stack of rather complex systems, including hardware, the operating systems (with virtualisation mechanisms on various levels), possibly a virtual machine and a component container in an application server.

The usage profile: The frequency and parameters of service calls to a component also influence their extra-functional properties. As an example a download of a kilobyte is faster than the download of a megabyte.

Any meaningful component meta model which is concerned with the description of extra-functional properties has to take all these factors as explicit parameters of the component properties into account. For the specification of the functional properties, the usage profile can be neglected.

In any case, for a reasoning mechanism on the properties of component-based systems, it is important to have component meta models that are parameterised over the above mentioned influence factors. This allows to compose the actual component models isomorphically to the component composition (i.e., the architecture of a component-based system). By the specification of the system architecture, the parameters are filled and the actual properties of the single components and the composed components (namely the system) can be computed. In the area of extra-functional properties, this computation is called "prediction".

Due to this role of components, we claim that the componentisation of software also brings advances when systematically predicting extra-functional properties (such as reliability or performance) of software systems in contrast to the current view, where primarily software components are seen as a means to software reuse. As a consequence, one of the major motivations of software architectures, the aim to reason explicitly with extra-functional properties during software-design, may benefit a lot from focusing on *component based* software architectures.

2. Contractual Use of Components

Much of the confusion about the term "contractual use" of a component comes from the double meaning of the term "use" of a component. The "use" of a component refers (as discussed above) refers to different usage-times:

1. the usage of a component during run-time. This is, calling services of the component.
2. the usage of a component during composition time or deployment. This is, placing a component in a new reuse-context, as it happens when architecting, installing or reconfiguring systems.

Depending on the above case, contracts play a different role. Before actually defining contracts for components, we briefly review the design-by-contract principle from an abstract point of view. According to [3, p. 342] a contract between the client and the supplier consists of two obligations:

- The client has to satisfy the precondition of the supplier.
- The supplier has to fulfil its postcondition, if the precondition was met be the client.

Each of the above obligations can be seen as a benefit for the other party. (The client can count on the postcondition if the precondition was fulfiled, while the supplier can count on the precondition). Putting it in one sentence:

> If the client fulfils the precondition of the supplier, the supplier will fulfil its postcondition.

How can we translate this principle to software components? We can consider the user of a component as the client, and the component as the supplier of services. It is important to note that each service of the component may have a contractual specification (i.e., its pre- and postcondition), but these service-contracts (or a collection of these contracts) are *not* the contract of the component, as we will discuss in the following. To formulate contracts for components, we also have to identify the pre- and postconditions and the user of a component. But what is a precondition, postcondition and user, depends on the case of use as listed above (run-time or composition-time). Let us first consider the component's use at run-time. The use of a component at run-time is calling its services. Hence, the user of a component C are all components connected to C's provides interface(s).

The precondition for that kind of use is the precondition of the service, likewise the postcondition is the postcondition of the service. Actually, this shows that this kind of use of a component is in no way different as using a method. Therefore, the authors do consider this case as the use of a *component service*, but *not* as the use of a *component*. Likewise, the contract to be fulfilled here from client and supplier is a *method contract* as described by Meyer already 1992. There is nothing component specific in this kind of contracts!

The other case of component usage (usage at composition time) is the actual important case when talking about the contractual use of components. This is the case when architecting systems out of components or deploying components within existing systems for reconfiguration. Again, in this case a component C is acting as a supplier, and the environment as a client. The component C offers services to the environment (i.e., the components connected to C's provides interface(s)). According to the above discussion of contracts, these offered services are the postcondition of the component, because it specifies what the client can expect from a working component. Also according to the Meyers above mentioned description of contracts, the precondition specifies the component C expects from its environment in order to enable C to offer its services (as stated in its postcondition). Hence, the precondition of a component is stated in its requires-interfaces.

Analogously to the above single sentence formulation of a contract, we can state:

> If the user of a component fulfils the components' required interface (offers the right environment) the component will offer its services as described in the provided interface.

Note that checking the satisfaction of a requires interface includes checking whether the contracts of required services (the service contracts specified in the requires-interface(s)) are sub-contracts of the service contracts stated in the provides interfaces of the required components. The notion of a subcontract is described in [3, p. 573] like contravariant typing for methods: A contract c' is a subcontract of contract c, if (a) the precondition of c' is weaker than or equal to the precondition of c and (b) the postcondition of c' is stronger than or equal to the postcondition of c.

> The interfaces involved by contract checking belong to separate components and are connected by bindings. Checking contractual use of components are interoperability checks and therefore have a boolean result.

Hence, when architecting systems (i.e., introducing new components), we have to check the bindings of their requires-interfaces to the used environmental provides-interfaces. When replacing a component with a newer one, we not only have to check their contract (i.e., the bindings of their requires-interfaces to the used components, as mentioned above), but also the contracts of the using environmental components (i.e., the bindings from the provides-interfaces), because one has to ensure that by a replacement non of the existing local contracts have been broken.

There is a range of formalisms used for specifying pre- and postconditions, defining a range of interface models for components (see for extensive discussions and various models, e.g., [2, 6]).

Another degree of freedom in the abstract principle of design-by-contract is the time of their deployment. Component contracts as discussed here describe the deployment of components at composition-time. This stresses the importance of contracts which are statically checkable. When a system is architected or reconfigured, one is aware of the possibility of introducing errors. Therefore, the direct feedback about the success of introducing (or replacing) a component into a system is very helpful in practice because it

can assure the absence of composition errors. Opposed to that run-time checks can only show the presence of composition errors when detecting a contract violation. This is particularly bad, because the person using the system and triggering the error is most commonly not the person reconfiguring or architecting the system who may find it hard to trace back the reconfiguration step which introduced the error.

3. Parametric Contracts

We have seen that the enforcement of contractual use of software components is equivalent to interoperability checks between interfaces. An interoperability check has a boolean outcome: Either there are interoperability errors detected, or not. In practice this is not sufficient for two reasons: Firstly, in particular in bottom-up reuse scenarios (where components are created before and independently from system architectures) a component rarely fits directly in a new reuse context, because for a component developer it is hard to foresee all possible reuse contexts of a component in advance (i.e., during design-time).

Secondly, there is a broad consensus on that extra-functional properties (such as performance or reliability metrics) of components need to be specified for components. However, as with the different unforeseeable reuse scenarios above, the component developer cannot foresee all the possible extra-functional properties of the environment. But to make a specification of the extra-functional properties of the component, the component developer has to make strong assumptions on the properties of the execution environment. Coming back to our discussion about component contracts, this means that in practice one single pre- and postcondition of a component will not be sufficient:

1. the precondition of a component is not satisfied by a specific environment while the component itself would be able to provide a meaningful subset of its functionality.
2. a weaker postcondition of a component is sufficient in a specific reuse context (i.e., not the full functionality of a component will be used). Due to that, the component itself will require less functionality at its requires-interface(s), i.e., will be satisfied by a weaker precondition.

Hence, what we need are not static pre- and postconditions, but *parametric contracts* [4]. In case 1 a parametric contract computes the postcondition in dependency on the strongest precondition guaranteed by a specific reuse context (hence the postcondition is parametric with the precondition). In case 2 the parametric contract computes the precondition in dependency on the postcondition (which acts as a parameter of the precondition). For components this means that provides- and requires-interfaces are not fixed, but a provides-interface is computed in dependency on the actual functionality a component receives at its requires-interface and a requires-interface is computed in dependency on the functionality actually requested from a component in a specific reuse context. Hence, opposed to classical contracts, one can say:

> Parametric contracts link the provides- and requires-interface(s) of the same component. They have a range of possible results (i.e., new interfaces).

Interoperability is a special case now: if a component is interoperable with its environment, its provides-interface will not change. If the interoperability check fails, a new provides-interface will be computed.

4. Conclusion

This paper discussed contractual usage of software components and argued for requires-interfaces as precondition of components and provides-interfaces as postcondition. Parametric contracts describe the relation between pre- and post-conditions. By this, the external influences given by properties of the external services (as the parameter "precondition") are taken into account in the computation of the post-condition (i.e., the provides interfaces). The Palladio Component Model (PCM)[1] is a component meta model which is based on parametric contracts and, as the first of its kind, models all above mentioned influence factors on components as explicit parameters. It is well documented and freely available under

`http://sdqweb.ipd.uni-karlsruhe.de/wiki/Palladio_Component_Model`

Please note that sections 2 and 3 are revised passages from [5].

References

[1] S. Becker, H. Koziolek, and R. H. Reussner. Model-based Performance Prediction with the Palladio Component Model. In *Workshop on Software and Performance (WOSP2007)*. ACM Sigsoft, February5–8 2007.

[2] B. Krämer. Synchronization constraints in object interfaces. In B. Krämer, M. P. Papazoglou, and H. W. Schnmidt, editors, *Information Systems Interoperability*, pages 111–141. Research Studies Press, Taunton, England, 1998.

[3] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ, USA, 2 edition, 1997.

[4] R. H. Reussner. The use of parameterised contracts for architecting systems with software components. In W. Weck, J. Bosch, and C. Szyperski, editors, *Proceedings of the Sixth International Workshop on Component-Oriented Programming (WCOP'01)*, June 2001.

[5] R. H. Reussner and H. W. Schmidt. Using Parameterised Contracts to Predict Properties of Component Based Software Architectures. In I. Crnkovic, S. Larsson, and J. Stafford, editors, *Workshop On Component-Based Software Engineering (in association with 9th IEEE Conference and Workshops on Engineering of Computer-Based Systems), Lund, Sweden, 2002*, Apr. 2002.

[6] A. Vallecillo, J. Hernández, and J. Troya. Object interoperability. In A. Moreira and S. Demeyer, editors, *Object Oriented Technology – ECOOP '99 Workshop Reader*, number 1743 in LNCS, pages 1–21. Springer-Verlag, Berlin, Germany, 1999.

Task Scheduling in Rebeca

Mohammad Mahdi Jaghoori · Frank S. de Boer · Marjan Sirjani

1 Introduction

Rebeca [3] (reactive objects language) is an actor [2] based language with formal semantics, which can be used at a high level of abstraction for modeling concurrent and distributed reactive systems. Reactive objects (called rebecs) run in parallel and can communicate by asynchronous message passing. Rebecs have no explicit receive statement; instead, incoming messages are queued. A rebec has a message server for each message it can handle. A message sever (also called a method) is defined as a piece of sequential code, which, among others, may include sending messages.

All rebecs must implement a message server '`initial`'. At creation, a rebec has the '`initial`' message in its queue. At each step, each rebec executes (the message server corresponding to) the message at the head of the queue and then removes it from queue (i.e., there is no intra-object concurrency). In this paper, we allow rebecs to define their own scheduling policies (which has been traditionally FIFO in Rebeca). The scheduling policy of each rebec, upon receiving a message, determines where in the queue the message should sit; however, it cannot preempt the currently running method.

Task automata [1] is a new approach for modeling real time systems with non-uniformly recurring computation tasks; where tasks are generated (or triggered) by timed events. Tasks, in this model, are represented by a triple (b, w, d), where b and w are, respectively, the best-case and worst-case execution times, and d is the deadline. A task automaton is said to be schedulable if there exists a scheduling strategy such that all possible sequences of events generated by the automaton are schedulable in the sense that all associated tasks can be computed within their deadlines. It is shown in [1] that, among other cases, with a non-preemptive scheduling strategy, the problem of checking schedulability for task automata is decidable.

In this paper, we add real time constraints to Rebeca and present a compositional approach based on task automata for schedulability analysis of timed Rebeca models. In this approach, instead of just best-case and worst-case execution times, the behavior of each task is given (in terms of timed automata) and used in the schedulability analysis. These timed automata may in turn generate new tasks. Task automata, as introduced in [1], cannot model tasks generated *during* the execution of another task.

2 The Timed Rebeca Model

For each rebec, the message servers are modeled as timed automata, in which actions may include sending messages, either to the same rebec, called *self calls*, or to other rebecs. Since message servers always terminate, every execution of the corresponding automata also stops at a state with no outgoing transition. The modeler also gives an abstract behavior of the environment for each rebec in terms of a timed automaton (called the *driver* automaton). The driver automaton models the (expected) timings for arrival of messages to the rebec, together with their deadlines. The driver automaton is similar to task automata in the sense that receiving a message corresponds to generating a new task.

M. M. Jaghoori · F.S. de Boer
CWI, Amsterdam, The Netherlands; E-mail: {jaghouri, f.s.de.boer}@cwi.nl

M. Sirjani
University of Tehran and IPM, Tehran, Iran; E-mail: msirjani@ut.ac.ir

$$((s,s'),[B_1,\ldots,B_l]) \xrightarrow[c\ ,\ X]{?m(d)}_H ((u,s'),sched([B_1,\ldots,B_l],m(d))) \quad [recieve]$$
$$if\ s \xrightarrow[c\ ,\ X]{?m(d)}_T u$$

$$((s,s'),[B_1,\ldots,B_l]) \xrightarrow[c\ ,\ X]{m(d)}_H ((s,u),sched([B_1,\ldots,B_l],m(d))) \quad [self-call]$$
$$if\ (s' \xrightarrow[c\ ,\ X]{!m(d)}_{B_1} u)\ and\ (m \in M)$$

$$((s,s'),[B_1,\ldots,B_l]) \xrightarrow[c\ ,\ X]{!m(d)}_H ((s,u),[B_1,\ldots,B_l]) \quad [send]$$
$$if\ (s' \xrightarrow[c\ ,\ X]{!m(d)}_{B_1} u)\ and\ (m \notin M)$$

$$((s,s'),[B_1,\ldots,B_l]) \rightarrow_H error \quad [overflow]$$
$$if\ (l=q)\ and\ ((s \xrightarrow[c\ ,\ X]{?m(d)}_T u)\ or\ (s' \xrightarrow[c\ ,\ X]{!m(d)}_{B_1} u\ and\ m \in M))$$

$$((s,s'),[B_1,B_2,\ldots,B_l]) \rightarrow_H ((s,start(B_2)),[B_2,\ldots,B_l]) \quad [context-switch]$$
$$if\ s' \in final(B_1)$$

Fig. 1 Calculating the edges of the behavior automata

A timed automaton is identified by a finite set of locations N (including an initial location n_0); a set of actions Σ; a set of clocks C; location invariants $I : N \rightarrow \mathcal{B}(C)$; and, the set of edges $\rightarrow \subseteq N \times \mathcal{B}(C) \times \Sigma \times 2^C \times N$, where $\mathcal{B}(C)$ is the set of all clock constraints. An edge written as $s \xrightarrow[c\ ,\ X]{a} s'$ means that action a may change state s to s' by resetting the clocks in X, if clock constraints in c hold. In the sequel, assume that the sets of messages handled by different rebecs are disjoint and their union is $\mathcal{M}$.

Definition 1 A rebec R is formally defined as $[(m_1 : A_1, \ldots, m_n : A_n), T, \mathcal{C}]$, where

- $M = \{m_1, \ldots, m_n\} \subseteq \mathcal{M}$ is the set of messages handled by R;
- $A_i = (N_i, \rightarrow_{A_i}, \Sigma, C_i, I_i, n_{0_i})$ is a timed automaton representing the message server handling m_i.
- $T = (N_T, \rightarrow_T, \Sigma_T, C_T, I_T, n_T)$ is a timed automaton modeling the rebec's environment (the driver).
- $\mathcal{C}$ is a set of clocks shared by all A_i and T (called the global clocks).

The action set of A_i is defined to be $\Sigma = \{!m | m \in M\} \cup \{!m(d) | m \in \mathcal{M} \wedge d \in I\!N\}$; and, the action set of the driver automaton is $\Sigma_T = \{?m(d) | m \in M \wedge d \in I\!N\}$. Intuitively, the driver automaton is similar to task automata in the sense that executing an action in the driver (i.e., receiving a message from another rebec) creates a new task. However, a rebec may send messages to itself (self calls), which also result in new (internal) tasks being generated. According to the definition of Σ, internal tasks are not necessarily assigned deadlines. Internal tasks without an explicit deadline (called *delegation*) inherit the (remaining) deadline of the task that generates them (parent task).

Delegation implies that the internal task (say t') is in fact the continuation of the parent task (say t). Notice that unconstrained loops in delegations result in nonschedulability, because deadline becomes smaller every time. To bound delegation loops, one can use the global clocks $\mathcal{C}$. A common scenario for delegation happens when a task t creates an instance of t' to continue the computation, after another task (say y) is executed. In such cases, if t' is scheduled before y is executed, it would need to create another instance of itself (t'). This results in a loop in calling t'.

As mentioned above, the driver automaton has the same syntax as a task automata, but it models only the messages sent by other rebecs (does not include internal tasks). Therefore, analyzing the driver alone is not enough for determining schedulability of the rebec. Instead, schedulability analysis should be performed on the automaton obtained by executing the abstract behavior of the message servers as controlled by the driver automaton.

Definition 2 (Behavior Automaton) The behavior automaton for a rebec R (cf. Definition 1) is a timed automaton $H = (S_H, \rightarrow_H, \Sigma_H, C_H, I_H, s_H)$ where

- $S_H = error \cup \big(N_T \times (\bigcup_{i \in [1..n]} N_i) \times (M \cup \{emp\})^q\big)$, where N_T and N_i are the sets of locations of T and A_i, respectively, and q is a statically computable bound on the length of schedulable queues.

- $\Sigma_H = \{!m(d) | m \notin M\} \cup \{?m(d) | m \in M\} \cup \{m(d) | m \in M\}$, where $d \in \mathbb{N}$ denotes the deadline.
- $C_H = C_T \cup \big(\bigcup_{i \in [1..n]} C_i\big) \cup \mathcal{C}$, where C_i and C_T are the sets of clocks for A_i and T, respectively.
- For each state $u = (s_T, s, Q)$, if $s \in N_i$ then $I_H(u) = I_T(s_T) \wedge I_i(s)$.
- The initial state s_H is $\big((n_T, start(A_1)), [A_1]\big)$, where n_T is the initial location of T; and, A_1 is the automaton corresponding to the '`initial`' message server.
- The edges $\rightarrow_H$ are defined with the rules in Figure 1.

In Figure 1, functions $start(A)$ and $final(A)$, respectively, give the initial location of A, and the set of locations in A with no outgoing transitions. Function *sched* puts the given message in the queue based on the scheduling policy of rebec R. Each state of the behavior automaton is written as $((s, s'), [B_1, \ldots, B_l])$, where $B_1, \ldots, B_l$ show the automata corresponding to the messages in the queue (empty queue elements are not written); s shows the current state in the driver; and, s' shows the current state in B_1. Notice that self calls are modeled as *internal* actions, while send and receive operations to/from other rebecs are *visible* actions. As discussed in the next section, sends and receives of different rebecs must match.

Assume that b_{min} is the smallest best-case execution time of the automata A_i representing the message servers in R; and, d_{max} is the longest deadline for the tasks that may be triggered on R. In Definition 2, one can statically compute $q = d_{max}/b_{min}$, as the bound on the length of schedulable queues. It means that the behavior automaton for each rebec is finite state and computable.

The schedulability analysis can be performed in a way similar to task automata. Schedulability can be verified by resetting a fresh clock (say x_i) whenever a new task (with deadline d_i) in scheduled into the queue. From every state, if $x_i \geq d_i$ for some task in the queue, the behavior automata should move to the *error* state. Consequently, the schedulability problem reduces to the reachability of the *error* state.

As a timed automaton, the semantics of the behavior automaton can be defined in terms of a timed transition system. The states of this transition system are pairs (S_H, u) where S_H is a location of the behavior automata and u is a clock assignment. Considering the delay transitions, the semantics of the behavior automaton is related to the semantics of the automata in the definition of a rebec:

$$((s_1, s_2), [B_1, \ldots, B_l], u) \xrightarrow{\delta} ((s_1', s_2'), [B_1, \ldots, B_l], u + \delta) \text{ iff } \begin{cases} (s_1, u_T) \xrightarrow{\delta} (s_1', u_T + \delta); \text{ and,} \\ (s_2, u_B) \xrightarrow{\delta} (s_2', u_B + \delta) \end{cases}$$

where, $((s_1, s_2), [B_1, \ldots, B_m])$ is a state of the behavior automaton; u_T and u_B represent the projection of u on the clocks of T and B_1, respectively; and, $\delta \in \mathbb{R}_+$ is a positive real valued number.

3 Compatibility checking

After performing schedulability analysis for each rebec separately, one should check if the driver automaton for each rebec correctly models the messages sent to that rebec. Notice that due to the schedulability of all rebecs, an action $?m(d)$ implies that m can be finished within d time units. Therefore, an action $!m(d')$ (requiring that m should finish within d' time units) can match $?m(d)$ only if $d \leq d'$.

To check the compatibility of the driver automata with the definition of the rebecs in the model, one can compute the synchronous product of the behavior automata of all rebecs. When computing the synchronous product of these automata, $?m(d)$ and $!m(d')$ can synchronize and become an internal action only if $d \leq d'$ (besides matching the timing constraints). The behavior automata of all rebecs are compatible if every send action can be matched by a corresponding receive.

Before computing the synchronous product, the information in the states of the behavior automata (the contents of the queue, etc.) can be abstracted away. Different internal actions (of the general form $m(d)$) can also be treated as one internal action τ.

References

1. Elena Fersman, Pavel Krcal, Paul Pettersson, and Wang Yi. Task automata: Schedulability, decidability and undecidability. 2007. Accepted for publication in Information and Computation (to appear).
2. Carl Hewitt. Procedural embedding of knowledge in planner. In *Proc. the 2nd International Joint Conference on Artificial Intelligence*, pages 167–184, 1971.
3. Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S. de Boer. Modeling and verification of reactive systems using Rebeca. *Fundamamenta Informaticae*, 63(4):385–410, 2004.