20th Nordic Workshop on Programming Theory

NWPT 2008

Tallinn, Estonia, 19-21 November 2008

Abstracts

TTÜ KÜBERNEETIKA INSTITUUT
Institute of Cybernetics at TUT

20th Nordic Workshop on Programming Theory

NWPT 2008

Tallinn, Estonia, 19–21 November 2008

Abstracts

Institute of Cybernetics at Tallinn University of Technology

Tallinn ○ 2008

20th Nordic Workshop on Programming Theory
NWPT 2008
Tallinn, Estonia, 19–21 November 2008
Abstracts

Edited by Tarmo Uustalu, Jüri Vain, and Juhan Ernits

Institute of Cybernetics at Tallinn University of Technology
Akadeemia tee 21, 12618 Tallinn, Estonia
http://www.ioc.ee/

Department of Computer Science, Tallinn University of Technology
Raja 15, 12618 Tallinn, Estonia
http://cs.ttu.ee/

ISBN 978-9949-430-24-6

© 2008 Institute of Cybernetics at TUT

Printed by Alfapress

Preface

This volume contains the abstracts of the talks to be presented at the 20th Nordic Workshop on Programming Theory, NWPT '08, to take place in Tallinn, Estonia, 19–21 November 2008.

The NWPT workshops are a forum bringing together programming theorists from the Nordic and Baltic countries (but also elsewhere). The previous workshops were held in Uppsala (1989, 1999, 2004), Aalborg (1990), Göteborg (1991 and 1995), Bergen (1992 and 2000), Åbo (Turku) (1993, 1998, 2003), Aarhus (1994), Oslo (1996, 2007), Tallinn (1997, 2002), Lyngby near Copenhagen (2001), Copenhagen (2005), Reykjavík (2006). This year 2008 it is Tallinn's turn again and the workshop series will be celebrating its 20th anniversary.

The scope of the meetings covers traditional as well as emerging disciplines within programming theory: semantics of programming languages, programming language design and programming methodology, programming logics, formal specification of programs, program verification, program construction, program transformation and refinement, real-time and hybrid systems, models of concurrent, distributed and mobile computing, tools for program verification and construction. In particular, they are targeted at early-career researchers as a friendly meeting where one can present work in progress but which at the same time produces a high-level post-proceedings compiled of the selected best contributions in the form of a special journal issue.

The workshop programme features four invited talks by distinguished researchers. We are proud to have on our programme talks by Dave Clarke (Katholieke Universiteit Leuven, Belgium), Vincent Danos (University of Edinburgh, UK), Martin Fränzle (Carl von Ossietzky Universität Oldenburg, Germany) and Margus Veanes (Microsoft Research, Redmond, WA, USA). The contributed part of the programme consists of 30 talks by authors from different European countries.

This edition of NWPT is sponsored by EXCS, the new Estonian Centre of Excellence in Computer Science, funded mainly by the European Regional Development Fund. We are grateful to the managements of the House of the Brotherhood of the Blackheads and the KUMU Art Museum for letting us their premises. For the third time, the Journal of Logic and Algebraic Programming have agreed to publish our post-proceedings and we are grateful in advance for their assistance.

Tarmo Uustalu, Jüri Vain

Tallinn, 12 November 2008

Organization

Programme Committee

Luca Aceto (Reykjavík University)
Michael R. Hansen (Danmarks Tekniske Universitet)
Anna Ingólfsdóttir (Reykjavík University)
Einar Broch Johnsen (Universitetet i Oslo)
Kim G. Larsen (Aalborg Universitet)
Bengt Nordström (Chalmers Tekniska Högskola, Göteborgs Universitet)
Olaf Owe (Universitetet i Oslo)
Gerardo Schneider (Universitetet i Oslo)
Tarmo Uustalu (Institute of Cybernetics at TUT, co-chair)
Jüri Vain (Tallinn University of Technology, co-chair)
Marina Waldén (Åbo Akademi)
Uwe Wolter (Universitet i Bergen)
Wang Yi (Uppsala Universitet)

Organizing committee

Tarmo Uustalu (Institute of Cybernetics at TUT)
Juhan Ernits (Inst. of Cybernetics / Dept. of Comp. Sci., TUT)
Monika Perkmann (Institute of Cybernetics at TUT)
Kristi Uustalu (Institute of Cybernetics at TUT)

Hosts

Institute of Cybernetics at Tallinn University of Technology
Department of Comptuer Science, Tallinn University of Technology

Sponsor

Estonian Centre of Excellence in Computer Science, EXCS
(funded mainly by the European Regional Development Fund)

Table of contents

Invited talks

Dave Clarke (Katholieke Universiteit Leuven)
Coordination via interaction constraints 7

Vincent Danos (University of Edinburgh)
A rule-based approach to the global dynamics of protein networks 8

Martin Fränzle (Carl von Ossietzky Universität Oldenburg)
Engineering constraint solvers for the analysis of hybrid systems 9

Margus Veanes (Microsoft Research, Redmond)
Ten years of model-program theory at Microsoft: from research to product impact 10

Contributed talks

Lennart Beringer
Relational bytecode correlations 11

Sukriti Bhattacharya and Agostino Cortesi
Property-driven program slicing 15

Maksym Bortin, Christoph Lüth and Dennis Walter
A certifiable formal semantics of C 19

Morten Dahl and Hans Hüttel
Type inference for a correspondence certifying type system 22

Markus Degen, Peter Thiemann and Stefan Wehr
Contract monitoring and call-by-name evaluation 25

Romain Demangeon
Type systems for the termination of mobile processes 28

Johan Dovland, Einar Broch Johnsen, Olaf Owe and Martin Steffen
Lazy behavioral subtyping 31

Stephen Fenech, Gordon Pace and Gerardo Schneider
Detection of conflicts in electronic contracts 34

Pavel Grigorenko
Higher-order attribute semantics of flat declarative languages 37

Damas Gruska
Quantification of information flow for value-passing process algebra 40

Nan Guan, Wang Yi, Zonghua Gu, and Ge Yu
Improving scalability of model-checking for minimizing buffer requirements of synchronous dataflow graphs 43

Abubakar Hassan, Ian Mackie and Shinya Sato
Translating interaction nets to C 47

Dag Hovland
A type system for usage of software components 51

Hans Hüttel and Willard Thór Rafnsson
Secrecy in mobile ad-hoc networks 54

Peter A. Jonsson and Johan Nordlander
On building a supercompiler for GHC 57

István Knoll, Anders P. Ravn and Arne Skou
A semantics for a real-time actor language 60

Marcel Kyas, Andries Stam, Martin Steffen and Arild Torjusen
A specification-driven interpreter for testing asynchronous Creol component 63

Timo Nummenmaa
A method for modelling probabilistic object behaviour for simulations of formal specifications 66

Cristian Prisacariu
Extending Kleene algebra with synchrony: completeness and decidability 69

Stefan Ratschan and Jan-Georg Smaus
Finding errors of hybrid systems by optimising an abstraction-based quality estimate 72

Rivo Roo and Juhan Ernits
Model-based testing of a web-based positioning application 75

Adrian Rutle, Alessandro Rossini, Yngve Lamo and Uwe Wolter
A category-theoretical approach to the formalisation of version control in MDE 78

Adrian Rutle, Alessandro Rossini, Yngve Lamo and Uwe Wolter
Automatic definition of model transformations at the instance level 81

Felix Schernhammer and Bernhard Gramlich
On operational termination of deterministic conditional rewrite systems 84

Neva Slani and Francisco Martins
Secure open networks 87

Florian Stenger and Janis Voigtländer
Parametricity for Haskell with imprecise error semantics 90

Peter Sørensen and Jan Madsen
Consistency check for component-based design of embedded systems using SAT-solving 93

Claus Thrane, Uli Fahrenberg and Kim G. Larsen
Quantitative simulations of weighted transition systems 97

Claus Thrane, Uffe Sørensen and Kim G. Larsen
Slicing for Uppaal 100

Peter Ölveczky and Daniela Lepri
Towards model checking bounded response in real-time Maude 103

Coordination via interaction constraints

Dave Clarke
Dept. of Computer Science, Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
dave.clarke@cs.kuleuven.be

Wegner described coordination as constrained interaction. This talk will explore a literal interpretation of this idea by using constraint satisfaction as the basis of a coordination model. Our model derives from the behavioural constraints underlying Arbab's Reo coordination model, though extends it considerably. Coordination in Reo emerges from the composition of the local behavioural constraints of primitives such as channels in a component connector. Expressing behavioral possibilities as constraints provides a new computational model for component connectors, which also offers a clear intensional description of behaviour. By exploring variations on the constraint paradigm, such as open constraints and partial solutions, more refined interaction patterns can be expressed, including interaction with an unknown world and on-the-fly-constraint generation.

A rule-based approach to the global dynamics of protein networks

Vincent Danos
Laboratory for Foundations of Computer Science, School of Informatics,
University of Edinburgh
Informatics Forum, 10 Crichton Street, Edinburgh EH8 9AB, United Kingdom
vdanos@inf.ed.ac.uk

Over the past decade, networks have emerged as a biological paradigm. High-throughput experimental techniques, bioinformatics and improved data analysis have made a global systemic view possible. However statistical studies of network structure (degree distributions, motifs, etc.) are limited by the fact that network descriptions (usually plain graphs) are static. One would like to investigate the network dynamics. Bridging this gap between network structure and function poses serious challenges. One needs high-quality network data. One also needs new modelling techniques since the traditional ones cannot cope with the inherent combinatorial complexity. We present an other approach grounded in the computer science of distributed systems, that allows us to consider truly global cellular systems, and investigate their dynamics.

Engineering constraint solvers for the analysis of hybrid systems

Martin Fränzle
Abteilung Hybride Systeme, Department für Informatik,
Carl von Ossietzky Universität Oldenburg
Escherweg 2, D-26121 Oldenburg, Germany
fraenzle@informatik.uni-oldenburg.de

Safety-critical embedded systems often operate within or even comprise tightly coupled networks of both discrete-state and continuous-state components. The behavior of such hybrid discrete-continuous systems cannot be fully understood without explicitly modeling and analyzing the tight interaction of their discrete switching behavior and their continuous dynamics, as mutual feedback confines fully separate analysis to limited cases. Tools for building such integrated models and for simulating their approximate dynamics are commercially available. Simulation is, however, inherently incomplete and has to be complemented by verification, which amounts to showing that the coupled dynamics of the embedded system and its environment is well-behaved, regardless of the actual disturbance and the influences of the application context, as entering through the open inputs of the system under investigation. Basic notions of being well-behaved demand that the system under investigation may never reach an undesirable state (safety), that it will converge to a certain set of states (stabilization), or that it can be guaranteed to eventually reach a desirable state (progress).

Within this talk, we concentrate on automatic verification and analysis of hybrid systems, with a focus on fully symbolic methods manipulating both the discrete and the continuous state components symbolically by means of predicative encodings and dedicated constraint solvers. We provide a brief introduction to hybrid discrete-continuous systems, demonstrate the use of predicative encodings for compactly encoding operational high-level models, and continue to a set of constraint-based methods for automatically analyzing different classes of hybrid discrete-continuous dynamics. Covering the range from non-linear discrete-time hybrid systems to probabilistic hybrid systems, advanced arithmetic constraint solvers will be used as a workhorse for manipulating large and complex-structured Boolean combinations of arithmetic constraints arising in their analysis tasks.

Ten years of model-program theory at Microsoft: from research to product impact

Margus Veanes
Foundations of Software Engineering Group, Microsoft Research
One Microsoft Way, Redmond, WA 98052, USA
margus@microsoft.com

Model programs, written in C# or AsmL, were recently adapted in the Windows organization at Microsoft as the standard for doing behavioral modeling of application-level network protocols. The main focus currently is on model-based testing, which is a cornerstone of the wider protocol quality assurance effort that is a part of Microsoft's commitment to comply with the requirements of the Department of Justice and the European Union. We look back at how this technology evolved in Research and ended up in the Product land. Looking ahead, model-based development, supported by various formal methods, will be an integrated part of protocol development. We look at some recent advances and challenges in model-program analysis.

Relational bytecode correlations
(Extended abstract)

Lennart Beringer

Institut für Informatik, Ludwig-Maximilians-Universität München
Oettingenstrasse 67, 80538 München, Germany
beringer@tcs.ifi.lmu.de

Abstract. We present a calculus for tracking indistinguishability relationships between values through pairs of bytecode programs. The calculus may serve as a certification mechanism for a variant of non-interference as well as for a restricted form of transformation validation. Contrary to previous type systems for non-interference, no restrictions are imposed on the control flow structure of programs. Objects, static and virtual methods are included, and heap-local reasoning is supported by frame rules. In combination with polyvariance, the latter enable the modular verification of programs over heap-allocated data structures, which we illustrate by verifying and comparing different implementations of list copying. The material is based on a complete formalisation in Isabelle/HOL.

1 Introduction

Non-interference is a well-known program property in the area of language-based security [6]. In its most basic form for a simple imperative language, it is formulated by separating the program variables into public (low security) L and private (high security) variables H. The property then requires that the program preserves the relation $=_L$ between states, which is to say that the final states of two executions agree on all variables in L whenever the initial states did.

A similar property specifies the semantic validity of program transformations: two executions (now of *different* programs) commencing in *identical* states should yield *identical* states, or should at least yield return states which agree on all variables that are relevant for the ensuing program continuation.

In this paper, we outline a new proof system that may be used for certifying properties of both kinds, for a fragment of sequential Java bytecode. The calculus generalises the two disciplines by considering two program phrases, a relation constraining pairs of initial states, and a relation constraining return values and final states. Its interpretation requires that any two terminating executions of the phrases commencing in states related by the former relation yield configurations that are related by the latter relation. In addition to the basic instructions for transferring values between the operand stack and local variables, the fragment of bytecode considered includes object creation and manipulation, static and virtual methods, and conditional and unconditional jumps.

The usage of an explicitly relational judgement form differentiates our approach from most static analyses presented for non-interference in the literature. These employ

non-relational type systems or abstract interpretations, but are often restrictive in at least some of the following respects. First, occurrences of low events (such as assignments to public variables) are often forbidden to occur in code regions whose execution is predicated on a private variable, even if these occurrences do not affect the final states. Secondly, semantic equivalences of e.g. the two branches of a high conditional are not taken into account, even if the equivalence may be detected by simple program transformations or program analyses such as copy propagation. Thirdly, the effect of method invocations on heap objects is usually constrained along the axis of visibility (i.e. the order of the security lattice separating low from high), irrespectively of the reachability of objects from the method parameters. Finally, type systems for non-interference are often restricted to bytecode of essentially structured control flow in the sense that the analysis (or its correctness) depends on the availability of annotations which partition the code into regions corresponding to high-level code structures. The calculus proposed in the present paper addresses all these issues. First, we allow in principle arbitrary events to occur in high branches, as long as such events are statically identifiable as being compatible with the desired security condition on final states. Second, our formulation is easily seen to be compatible with semantics preserving code transformations and incorporates a form of copy propagation. Thirdly, we improve on the modularity of verification by including frame rules which allow one to embed judgements at application sites where additional heap objects are present, and do not impose an explicit effect-limitation along the axis of visibility. Finally, we do not require annotations that essentially enforce structured control flow but instead treat arbitrary unstructured code.

Our calculus is based on a notion of abstract states that approximates the equality between values held in the storage locations (local variables, operand stack positions, object fields) of a concrete JVM state. Abstract states comprise similar components as concrete states, but contain abstract anonymous values ("colours") instead of concrete runtime values. Occurrences of the same abstract value in different abstract storage locations represents equality of the concrete values in the corresponding locations of a compatible concrete state. On integer values, different abstract values may well abstract one and the same concrete value. Address values, in contrast, are required to be abstracted to different colours, resulting in the separation discipline necessary for admitting the above-mentioned frame rules [5]. Due to the necessarily approximative nature of the described value abstractions our calculus represents a conservative static analysis and may be seen as an explicitly relational abstract interpretation or type system. Indeed, while we expect our calculus to be formally embeddable in relational general-purpose program logics such as Benton's Relational Hoare Logics [3], we do not include any form of logical reasoning in the calculus.

Symbolic assertions in our calculus consist of pairs of abstract states, together with an administrative component that associates basic type information as well as a security level to each colour in use. These structures are called relational shape descriptions (RSD's for short) and are interpreted over pairs of concrete states. In addition to the constraints arising from the interpretation of the two abstract states contained in an RSD, the concrete states have to satisfy the following property. Whenever a colour occurs in both abstract states, the concrete values interpreting the colour in the two concrete states must be indistinguishable according to the security level associated with this colour in

the administrative component. In particular, any colour of integer type that is associated with the public security level must be interpreted identically in the two concrete states. Thus, RSD's generalise the information contained in abstract states of non-relational type systems for non-interference in that the colours introduce an additional layer of abstraction: the security levels are not directly associated with variables, and occurrences of a colour in the two abstract states may well be in different (abstract) storage locations.

The main judgement form of our calculus associates a pair of phrases with a pre-RSD and a post-RSD, and is interpreted in a (relational) partial-correctness style. Indeed, a special case of the interpretation asserts that the final states of two terminating executions commencing in states related by the pre-RSD are related by the post-RSD. However, the formal interpretation of judgements is more general as it extends this pre-post-relationship to all separated state extensions of the abstract states explicitly mentioned in the judgement. This interpretation not only makes the frame rule easy to justify but also yields a stronger guarantee than previous type systems as non-interference is directly guaranteed also for states containing additional objects.[1]

The proof system includes rules for pairs of correlated instructions (for example, allocations of objects of the same class in both phrases) as well as rules that affect only one of the two phrases. Proof rules of the latter kind are used for code segments where the two phrases proceed in a non-correlated way (roughly speaking, such segments correspond to high code regions, but in the absence of a formal separation into high and low code regions this notion is only phenomenologically observed rather than actively enforced) and also for simple instructions transferring abstract values between the components of abstract states. Due to the occurrence of uncorrelated method invocations, the unary rules are isolated as a separate auxiliary proof system which is embedded into the main relational proof system using dedicated rules. Frame rules are given for both judgement forms, as are axiom rules that extract assumptions from appropriate (polyvariant) unary and relational proof contexts, and further structural rules.

In order to maintain the copy propagation information embodied in abstract states across method invocations, judgements are required to preserve colours and abstract addresses. This means that the administrative component of the post-RSD must contain the administrative component of the pre-RSD, and that each abstract object in the abstract heap of an initial abstract state must be present (albeit with potentially different abstract values in the abstract fields) in the corresponding abstract heap of the final RSD. Thus, while the interpretation of object colours of a *single* RSD amounts to a partial bijection between addresses (essentially the partial bijection constructed in the work of [1] and [2]), the resulting discipline regarding objects *across a judgement* is slightly different from the notion of non-interference used by [1] and [2]. In particular, the preservation of address colours in the unary proof system enables the specification of policies which, for example, guarantee that the result of a method is the value passed in a certain argument position, or that the result points to a freshly allocated object.

[1] Having completed the work outlined in this paper, we learned that our technique is similar to *resource Kripke semantics*, a technique employed by Birkedal and Yang for modelling the semantics of higher-order separation logics [4].

This extended abstract is based on a complete formalisation of the calculus in Isabelle/HOL which comprises the definition of an operational semantics for the chosen language fragment, the formal derivation of all proof rules, and the example verification of selected program fragments. The latter includes the verification of a method for copying (the spine of) heap-allocated lists, for non-cyclic lists of arbitrary length. The specification of such data types is defined using abstract representation predicates similar to the concrete predicates used in Reynolds' introductory paper on Separation logic [5]. A proof using only unary rules establishes that a single execution indeed creates a copy of the list's spine (with shared abstract pointers to the content-representing objects), while the relational proof system is used to establish the correlation between different variations of the code. These variations arise from each other by simple peephole transformations. Depending on the choice of security levels in the administrative components, these judgements may either be read as witnesses for the semantic admissibility of the transformations or as a non-interference statement asserting that a program environment cannot discover which of the two branches of a high conditional has been executed if each branch contains a different variation of the copying routine.

Acknowledgements

This work was supported by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project, and the DFG-funded project InfoZert, grant number Be 3712/2-1. We are grateful to the members of both projects for discussions on type systems and program logics for information flow.

References

1. Anindya Banerjee and David A. Naumann. Stack-based access control and secure information flow. *J. Funct. Program.*, 15(2):131–177, 2005.
2. Gilles Barthe, David Pichardie, and Tamara Rezk. A Certified Lightweight Non-Interference Java Bytecode Verifier. In Rocco De Nicola, editor, *Proceedings of 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 125–140. Springer, 2007.
3. Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004)*, pages 14–25. ACM Press, 2004.
4. Lars Birkedal and Hongseok Yang. Relational parametricity and separation logic. In Helmut Seidl, editor, *Proceedings of the 10 International Conference on Foundations of Software Science and Computational Structures (FOSSACS 2007)*, volume 4423 of *LNCS*, pages 93–107. Springer, March 2007.
5. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74. IEEE Computer Society, 2002.
6. Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications – special issue on Formal Methods for Security*, 21(1):5 – 19, January 2003.

Property Driven Program Slicing

Sukriti Bhattacharya, Agostino Cortesi
Dipartimento di Informatica
Università Ca' Foscari di Venezia

Program slicing is a technique to extract program parts with respect to some special computation. Since Mark Weiser [1] first proposed the notion of slicing in 1979, many applications of this technique have been studied in the literature. Slicing was first developed to facilitate debugging, but it has been found helpful in many aspects of the software development life cycle, including software testing, software measurement, program comprehension, software maintenance, software reengineering, program parallelization and so on [2]. Different tools have also been developed, like CodeSurfer, Unravel, Indus, Bandera, and Kaveri, just to name a few.

Generally speaking, by applying a slicing technique on a program P with a slicing criterion C (i.e. a line of code in P), we get a program P′ that behaves like P when focussing only on the variables in C. The sliced program P′ is obtained through backward computation from P by removing all the statements that do not affect neither directly nor indirectly the values of the variables in C.

The aim of the present paper is to further refine the traditional slicing technique by combining it with abstract interpretation and data flow analysis [3,4]. This results into a deeper insight on the strong relation between slicing and property based dependency.

The idea can be summarized as follows. Very often, we are interested on a specific property of the variables in the slicing criterion C, not on their exact actual values. For instance, when condering security properties, it may be sufficient to consider just the parity of the variable containing a cyphertext, or just its sign, or just if it stays into a given range. Therefore, when performing slicing, not only the statements that do not affect Vars(C) can be discarded, but also the statements that do not have any impact, respectively, on their parity, on their sign or on their range.

The resulting proposal is a fixed point computation where each iterate has two phases. First, by a forward (dataflow) static analysis, the control flow analysis of the program is enhanced with information about the state of variables with respect to the property of interest. Then, following a backward computation, we enhance the slicing technique by using such additional information to identify relevancy of the statement to be kept (or removed) in the sliced program.

Given a program P, we denote by Vars(P) the set of program variables in P and by exp(Vars(P)) the set of expressions in P. We denote the expression evaluation by eval: exp(Vars(P))$\rightarrow$ Values, A property, ψ is a decidable boolean function $\wp$((Vars(P)),eval) $\rightarrow$ {True, False}.

Given two programs P_1 and P_2 and a set K$\in$ Vars(P_1) $\cap$Vars(P_2), P_1 and P_2 are said to be K-congruent with respect to ψ if for each execution of P_1 and P_2 with the same input, the final value of the variables in K are the same.

Consider for instance P_1: *begin input(x); x 2*x; y 5; z x+y*; *end*, and P_2: *begin input(y); x 4; y x/3+2; z x+1*; *end.* $P_1$is {x,y,z} congruent to P_2 with respect to *parity*, since at the end of the computation both programs behave the same wrt parity: x is even, y is odd and z is odd.

Let P be a program, s be a statement of P, and V$\subseteq$ Vars(s). For statement s and a given property ψ, the slice $P^{'}$ of program P with respect to the slicing criterion (s,V) and ψ, is an executable program such that:

1. $P^{'}$can be obtained by deleting zero or more statements from P.
2. $P^{'}$ is V-congruent to P with respect to ψ.

Consider, for instance, the following C program fragment , We want to slice P with respect to the slicing criteria $<5;\{c\}>$ and the *sign* property.

Stmt. NO.	Original Program (P)	Sliced program
1	scanf(%d,"&b"); /* b≠0 */	scanf(%d,"&b");
2	b b*(-2);	b b*(-2);
3	c b*2;	c b*2;
4	d d+1;	
5	d c/b;	d c/b;
6	c d;	c d;

Statement 4 in P is removed in the slice P', since it is no more relevant when considering the *sign* property and the slicing criterion, as the sign of c in line 6 is not affected by such statemet.

At the macro level, our algorithm is fixed point iteration calternating analysis and slicing, starting form a program P and a slicing criterion C and a property ψ:

```
loop {
        P_old  P;
        A      Analysis(P, ψ);
        P'     Slicer(P, C, ψ, A);
      }until P'  P_old
```

More in detail, *Slicer* considers P as an ordered set of statements (all statements that should not be removed), and the set N to hold all variables that have been used. *Slicer* performs a fixpoint iteration: by using information collected in the analysis phase.

For any statement s, *STMT(s)* denotes the succeeding statements.After the inizialization phase,statements possibly affecting a variable in N are added to P, and the used variables of these statements are added to N, until these sets do not grow any more. Clearly, this process terminates since there are only a finite number of variables and assignments in a program.Variables may be read or written through pointers. We assume, for each statement s and pointer variable p, that there is a points-to [5] set *PTS(p, s)* which contains the set of variables possibly pointed to by p in the statement s. The *DEFS* function calculates a set of locations where the result of an assignment may be stored. If the left-hand side of the assignment contains de-referenced pointers, then *DEFS* uses *PTS* to calculate possible locations where the result might be stored.$LIVE_s^{out}$ holds the variables live [4], after the execution of statement s.

Each program flow graph edge has an associated flag, the *ExecutableFlag*. This flag is initially *FALSE* for all edges. ExecutableFlag value of program flow graph edges are marked *TRUE* by symbolically executing the program, beginning with the start node. Whenever an assignment node is executed, the *ExecutableFlag* value of the out-edge in the program flow graph leaving that node is marked as *TRUE* and added to the worklist. Whenever a conditional node is executed, the expression controlling the conditional is evaluated and we determine which branch (es) may be taken. If the expression evaluates to $\perp$ (not predictable) then all branches may be taken. The *ExecutableFlag* value corresponding to these branches are set to be *TRUE*. Otherwise, only one branch can be taken, and the associated *ExecutableFlag* value is set to be *TRUE*.

The Slicer algorithm can be sketched as follow:

Slicer(P_0, C, ψ, A {PTS,DFG*})
INPUT:
P_0: The code to be sliced (set of statements)
C: The slicing criterion <n,V>
ψ: A given property
PTS: Mapping from pointer variables and statement numbers to points-to-sets
DFG*: Dataflow graph of the program where each node associated with
    $\Pi_v^{in}(\psi_s)$ (variables' value before executing statement s wrt ψ)
    $\Pi_v^{out}(\psi_s)$ (variables' value after executing statement s wrt ψ)
OUTPUT:
P: A program slice which is V-congruent with ψ and s.
BEGIN
N V
P {n}
N_{old} {$\emptyset$}
while N $\not\supseteq$Nold do
  N_{old} N
  for each statement s from n to 1 in P_0 do
    if ExEflg(STMT(s)) TRUE then
      for each v $\in$DEFS (PTS,s,STMT(s)) do

if v∈ N $\wedge \Pi_v^{in}(\psi_s) \neq \Pi_v^{out}(\psi_s)$ ∧ v ∈ $LIVE_s^{out}$ then
P   P∪{s}
N   N ∪ USES(PTS, s, STMT(s))
endif
endfor
endif
endfor

This process terminates since there are only a finite number of variables and assignments in a program and at each step of the while loop the set N is strictly increasing and is bounded by Vars(P).

This work combines static analysis and program slicing, i.e it refines slicing with respect to a property of interest. It can also be combined with the abstract slicing approach in [6]. An implementation of the complete algorithm for C programs, where simple properties like sign and partity are considered, has already been developed, and preliminary experimental results confirm the scalability of the algorithm.

References

[1] M. Weiser. *Program slicing.* In ICSE '81: Proceedings of the 5th international conference on Software engineering, pages 439-449, Piscataway, NJ, USA, 1981. IEEE Press.

[2] F. Tip. *A survey of program slicing techniques.* Journal of Programming Languages, vol. 3, pages 121-189, 1995.

[3] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag New York, Inc., 1999.

[4] Patrick Cousot, Radhia Cousot. *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.* In Conference Record of the Sixth Annual ACM SIGPLANSIGACT Symposium on Principles of Programming Languages, pages 238-252, Los Angeles, California, 1977. ACM Press, New York.

[5] Lars Ole Andersen . *Program Analysis and Specialization for the C Programming Language* .Ph.D. Thesis , DIKU, University of Copenhagen,May 1994, pages 111-120.

[6] I. Mastroeni, D. Zanardini, *Data Dependencies and Program Slicing: from Syntax to Abstract Semantics*, Proc. "ACM SIGPLAN 2008 Workshop on Partial Evaluation and Program Manipulation", San Francisco, CA, USA , 7-8 Gennaio 2008 , 2008 , ACM press, pp. 125-134 .

A Certifiable Formal Semantics of C

Maksym Bortin Christoph Lüth Dennis Walter

1 Introduction

This paper presents a formalisation of a subset of the C programming language, and a corresponding verification calculus, in the theorem prover Isabelle. There are of course many and varied approaches to the verification of safety-critical programs. The characteristics of our approach stem from the application domain[1]: the certification of control software for autonomous mobile robots [3]. This means that firstly, our verification techniques need to stand up to certification by an external agency such as the TÜV; secondly, we can restrict ourselves to a subset of C tailored for safety-critical applications, such as MISRA C [5] (in fact, this is even required by the relevant standard IEC 61508); and thirdly, the algorithms to be verified are comparatively sophisticated for a safety function, involving the calculation of safety zones from a model of the braking behaviour of the robot.

Our verification is based on a formalisation of a subset of MISRA C in the theorem prover Isabelle in typed higher-order logic (HOL). Using Isabelle is crucial to our approach: based on the C semantics, we can build a proof calculus and verify its correctness inside Isabelle. Thus, the validation of our verification can focus on the semantics of C as presented here. Further, using Isabelle allows us to use higher-order logic to express our specifications, so we are not tied to a specific specification language.

2 A formal semantics for a C subset

Overall, our model is close to a text-book semantics as found in e.g. [7], but of course some extensions are necessary to cover features of the C programming language like pointers and their arithmetic, nested structures, arrays and expressions with side-effects.

2.1 The memory model

A *state* represents a program's memory. In contrast to the usual memory model as a stack of local variables and a heap containing allocated objects, we use a flat model where all objects are given a *base location*. The state maps these base locations to object representations. Fig. 1 depicts the structure of a state.

Concretely, a state is a partial function $\Sigma : \mathit{BaseLoc} \rightharpoonup (\mathit{Type} \times (\mathbb{N} \rightharpoonup \mathit{Val}))$ mapping base locations to representations of scalar or structured values and their (run-time) type *Type*. Objects are represented as sequences of what the C standard calls *scalar* values, i.e. integer and floating-point numbers and addresses. These sequences are modelled as partial functions from $\mathbb{N}$ to *Val*, the type of scalar values. To access scalar values (possibly inside structures or arrays) we use *locations*, which are pairs $(\mathit{BaseLoc} \times \mathbb{N})$. Thus, locations represent addresses. They allow a limited form of arithmetic, as defined in the standard: we can add and subtract the offsets of locations sharing the same base location.

Our model precludes the use of structured values in expressions, so they cannot occur as arguments to assignment or functions. The basic operations on states are allocation, deallocation, reading and writing:

$$\mathit{extend} : \mathit{BaseLoc} \to \Sigma \to \Sigma \qquad \mathit{read} : \mathit{Loc} \to \Sigma \to \mathit{Val}$$
$$\mathit{dealloc} : \mathit{BaseLoc} \to \Sigma \to \Sigma \qquad \mathit{update} : \mathit{Loc} \to \mathit{Val} \to \Sigma \to \Sigma$$

Deallocation is currently used for local variables on function exit; *malloc* and *free* could easily be supported by our model as well. State updates always succeed, i.e. at the state level we don't perform type checks, array bounds checks or pointer validity checks. Sanity checks are instead inserted into the semantics of pointer dereferencing and array access.

[1] http://www.sams-project.org/

$BaseLoc_0$	$Type_0$	$Val_{0,0}$	$Val_{0,1}$	$Val_{0,2}$	$\cdots$
$BaseLoc_1$	$Type_1$	$Val_{1,0}$			
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$BaseLoc_n$	$Type_n$	$Val_{n,0}$	$Val_{n,1}$		

Figure 1: A state maps base locations to types and sequences of scalar values

2.2 Language features

This is a brief and incomplete list of supported (•) and unsupported (∘) features of the C language.

- • The address-of operator `&`
- • Arrays and nested structures
- • Pointer offsets and subtraction
- • `sizeof` operator
- • Side-effects in expressions
- ∘ Casts to and from `void *`
- ∘ Function pointers
- ∘ `union`
- ∘ `switch`, `goto`, `break`, `continue`
- ∘ Dynamic memory management (`malloc`)

2.3 State transformer semantics

The abstract syntax of C programs is modelled as a collection of Isabelle/HOL datatypes. We provide a deterministic denotational semantics for the language that identifies all kinds of faults like invalid memory access, non-termination or division by zero as complete failure. Hence, the semantics of functions, statements and expressions map the corresponding data type to partial state transformers:

$$[\![stmt]\!] : \Gamma \to \Sigma \rightharpoonup 1 \times \Sigma$$
$$[\![expr]\!] : \Gamma \to \Sigma \rightharpoonup \mathit{Val} \times \Sigma$$
$$[\![f(x_1, \ldots, x_n)]\!] : \Gamma \to \mathit{Val}^n \to \Sigma \rightharpoonup \mathit{Val} \times \Sigma$$

An environment Γ maps a variable to its allocated base location, and function identifiers to their semantics:

$$\Gamma \cong (\mathit{Id} \rightharpoonup \mathit{BaseLoc}) \times (\mathit{FunId} \rightharpoonup (\mathit{Val}^n \to \Sigma \rightharpoonup \mathit{Val} \times \Sigma))$$

Since expressions can have side-effects, evaluation order is important. However, the MISRA-C guidelines [5, Rule 12.2] require that an expression must yield the same value under every evaluation order. As we only consider MISRA-conformant programs, it is appropriate to fix the evaluation order, proceeding from left to right for both function argument lists and expression trees.

A rather drastic simplification from a theoretical point of view is the lack of support for recursive functions (also a MISRA requirement). This allows us to give semantics to functions in a sequential fashion without the need for a fixed-point operator. Thus, the computational power of our language subset rests on the presence of while-loops, the semantics of which are given in terms of the least number of unfoldings of the loop body.

3 Proving Properties

We use preconditions and postconditions of functions and invariants of while-loops for program specifications. Additionally, as a means to express framing properties — i.e. to specify parts of the state that do not change— we use *assignment lists*. They resemble JML's assignable-clauses [1], and are lists of expression patterns which evaluate to a set of locations that are allowed to be modified.

Very classically, total correctness of a function (or block of statements) means *(i)* the function (block) terminates for all states satisfying the precondition; *(ii)* every state satisfying the precondition is transformed by the semantics of the function (block) to a state satisfying the postcondition; *(iii)* all locations not in the set of modifiable locations have kept their initial values after program execution. Total correctness is derived through a Hoare-style calculus [4]. We have correctness assertions of the form $\Gamma \vdash_{stmt} [P]\ c\ [Q]$, where P and Q are *state predicates*, i.e. functions $\Sigma \to \mathit{bool}$. We decided to embed predicates shallowly because this gives us the full expressiveness of HOL without tying us to a specific specification language. As a consequence, the assignment

rule in our calculus (A) uses state change operations instead of syntactic substitution in the predicate as usual (B), a characteristic shared with e.g. Schirmer's approach [6]:

$$(A) \qquad \frac{\Gamma \vdash_{expr} \; [P] \; E \; [\lambda v S. \; Q \; (update \; (\Gamma \; x) \; v \; S)]}{\Gamma \vdash_{stmt} \; [P] \; x := E \; [Q]} \qquad\qquad \Gamma \vdash_{stmt} \; [P[x/E]] \; x := E \; [P] \qquad (B)$$

An important consequence is that we obtain a rather simple representation and uniform proof goals with regards to aliasing. Terms of the form $read\ l_1\ (update\ l_2\ v\ \Sigma)$ either simplify to v for $l_1 = l_2$, to $read\ l_1\ S$ for $l_1 \neq l_2$, or they lead to a case split for cases where the equality cannot be derived.

Practically, the calculus is used in a weakest precondition fashion. We formulated all proof rules so that they can automatically be applied by a proof procedure that finishes with a number of verification conditions. Isabelle's metavariable mechanism is used to this end. For this to work, in all rules the preconditions in the premisses and the postcondition of the conclusion consists of a single metavariable. The verification conditions are then to be proved by a domain expert.

4 A Verification Environment

The encoding as sketched above forms the core of a verification environment built around it. A syntactic front-end is used to parse and typecheck the C programs, check conformance to the MISRA guidelines, and translate the abstract syntax into the Isabelle datatype; since this is mainly convenience and syntactic translation, it does not impinge on correctness.

The specifications are embedded into the program by using annotations as in JML or Caduceus [2], and translated along with the program. This way, programs and specifications are kept in sync automatically: we can either translate a given program to obtain a running program, or feed it through the verification tool.

References

[1] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[2] J.-C. Filliâtre and C. Marché. Multi-Prover Verification of C Programs. In *Sixth International Conference on Formal Engineering Methods (ICFEM)*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29, Seattle, Nov. 2004. Springer.

[3] U. Frese, D. Hausmann, C. Lüth, H. Täubig, and D. Walter. The importance of being formal. In H. Hungar, editor, *International Workshop on the Certification of Safety-Critical Software Controlled Systems SafeCert'08*, To appear in Electronic Notes in Theoretical Computer Science, 2008.

[4] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[5] MISRA-C: 2004. Guidelines for the use of the C language in critical systems., 2004.

[6] N. Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In F. Baader and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference LPAR 2004*, volume 3452 of *Lecture Notes in Computer Science*, pages 398–414. Springer, 2004.

[7] G. Winskel. *The Formal Semantics of Programming Langauges.* Foundations of Computing Series. The MIT Press, 1993.

Type Inference for a Correspondence Certifying Type System

Morten Dahl and Hans Hüttel
Department of Computer Science, Aalborg University
{dahl, hans}@cs.aau.dk

Introduction We present a type inference algorithm for the correspondence certifying type system presented in [2]. The type system employs dependent types and effects and guarantees that well-typed processes (expressed in a pi-calculus) satisfy the authenticity property expressed by correspondences in the process.

Our sound and complete type inference algorithm separates resolution of type constraints from resolution of effect constraints. Contrary to [2] we do not encode the constraints in a logic but use an unification algorithm for solving type constraints and a saturation algorithm for solving effect constraints. We believe this yields a simpler inference algorithm.

Our work illustrates how to do type inference in the presence of bound names and dependent types with instantiation and abstraction. Using explicit substitutions we find that only the effect solving algorithm needs to deal with instantiation and abstraction.

We have implemented the algorithm in Caml and found that it performs very efficiently in practice.

Process Calculus and Type System Processes are modelled using a standard pi-calculus where messages M, in addition to the standard names and variables, contain pairs and projections. The calculus is extended with correspondence annotations for expressing non-injective authenticity properties in the form of begin- and end-events along with ok-messages. Informally, we say that a process P is safe if in every run of P every end-event is preceded by a begin-event.

Using dependent pair types together with ok effect types the type system guarantees that well-typed processes are safe. The typing rules in Figure 1 show the intuition behind the safety-guarantee: begin-events are collected and stored in the environment by the *begins* function (rule PT-PAR) and whenever an end-event is encountered the environment is required to contain a matching begin-event (rule PT-END). Furthermore, begin-events can be transferred using $\mathsf{Ok}(S)$ types: the type can contain as many begin-events as available in the environment (rule MT-OK) which can be made available again by exercising the ok-message (rule PT-EX).

The last typing rule in Figure 1 illustrates type instantiation and abstraction: pair types $\mathsf{Pair}(x : T_1, T_2)$ are used to type message pairs (M_1, M_2) and bind a variable x in T_2. In rule MT-PAIR, type $T_2 = T_2'\{M_1/x\}$ is the type resulting from substituting x in T_2' with M_1, or equivalently, T_2' is the type resulting from abstracting M_1 out of T_2.

$$\frac{\Gamma, \mathit{begins}(P_2) \vdash P_1 \qquad \Gamma, \mathit{begins}(P_1) \vdash P_2}{\Gamma \vdash P_1 \mid P_2} \text{ PT-PAR} \qquad \frac{l(M) \in \mathit{effects}(\Gamma)}{\Gamma \vdash \mathsf{end}\ l(M)} \text{ PT-END}$$

$$\frac{S \subseteq \mathit{effects}(\Gamma)}{\Gamma \vdash \mathsf{ok} : \mathsf{Ok}(S)} \text{ MT-OK} \qquad \frac{\Gamma \vdash M : \mathsf{Ok}(S) \qquad \Gamma, S \vdash P}{\Gamma \vdash \mathsf{exercise}\ M; P} \text{ PT-EX}$$

$$\frac{\Gamma \vdash M_1 : T_1 \qquad \Gamma \vdash M_2 : T_2 \qquad T_2 = T_2'\{M_1/x\}}{\Gamma \vdash (M_1, M_2) : \mathsf{Pair}(x : T_1, T_2')} \text{ MT-PAIR}$$

Figure 1: Selected typing rules

$T_1 \doteq T_2$	*type equality*	$\dot{S}_1 \doteq \dot{S}_2$	*effect equality*
T generative	*type requirement*	$l(M) \in \dot{S}_1, \dots, \dot{S}_n$	*effect requirement*
		$\dot{S} \sqsubseteq \dot{S}_1, \dots, \dot{S}_n$	*effect bound*

Figure 2: Constraint language

Type Inference For type inference we first introduce an equivalent formulation of the type system using nameless dependent pair types similar to the nameless *de Bruijn* formulation of the lambda calculus. The addition of type variables leads us to an *explicit substitution* [1] type $T\langle M/\omega\rangle$ where ω is a natural number used instead of variables in the nameless formulation and we define a reduction relation matching that of instantiation, e.g. $\mathsf{Pair}(T_1, T_2)\langle M/\omega\rangle = \mathsf{Pair}(T_1\langle M/\omega\rangle, T_2\langle M/\omega + 1\rangle)$.

Our constraint language is shown in Figure 2 and some of the constraint generation rules are listed in Figure 3. Note the strong resemblance between the constraint generation rules and the typing rules in Figure 1; soundness and completeness is an easy consequence. Performing type inference for a given process P under an environment Γ amounts to running the constraint generation rules on P and Γ thereby producing a set C of constraints and then looking for a type and effect variable substitution σ satisfying C. Our procedure for finding a solution is composed of two algorithms: a unification algorithm that first solves all type constraints and a saturation algorithm that finds a solution to the effect constraints. Note that solving the type constraints may produce additional effect constraints.

$$\frac{\begin{array}{c}\Gamma, \mathit{begins}(P_2) \vdash P_1 \rightsquigarrow C_1 \\ \Gamma, \mathit{begins}(P_1) \vdash P_2 \rightsquigarrow C_2\end{array}}{\Gamma \vdash P_1 \mid P_2 \rightsquigarrow C_1 \cup C_2} \text{ PC-PAR} \qquad \frac{\begin{array}{c}\Gamma \vdash M \rightsquigarrow T, C_1 \\ C = \{l(M) \in \mathit{effects}(\Gamma)\}\end{array}}{\Gamma \vdash \mathsf{end}\ l(M) \rightsquigarrow C \cup C_1} \text{ PC-END}$$

$$\frac{C = \{X \doteq \mathsf{Ok}(E),\ E \sqsubseteq \mathit{effects}(\Gamma)\}}{\Gamma \vdash \mathsf{ok} \rightsquigarrow X, C} \text{ MC-OK} \qquad \frac{\begin{array}{c}\Gamma \vdash M \rightsquigarrow T, C_1 \qquad \Gamma, E \vdash P \rightsquigarrow C_2 \\ C = \{T \doteq \mathsf{Ok}(E)\}\end{array}}{\Gamma \vdash \mathsf{exercise}\ M; P \rightsquigarrow C \cup C_1 \cup C_2} \text{ PC-EX}$$

$$\frac{\begin{array}{c}\Gamma \vdash M_1 \rightsquigarrow T_1, C_1 \qquad \Gamma \vdash M_2 \rightsquigarrow T_2, C_2 \\ C = \{X \doteq \mathsf{Pair}(T_1, X_2'),\ T_2 \doteq X_2'\langle M_1/1\rangle\}\end{array}}{\Gamma \vdash (M_1, M_2) \rightsquigarrow X, C \cup C_1 \cup C_2} \text{ MC-PAIR}$$

Figure 3: Selected constraint generation rules

The unification algorithm is standard in most cases except when trying to solve a type constraint on form $T \doteq X\langle M/\omega\rangle$ where T is not a type variable. To solve this constraint, X is assigned an *opening* of T where all variables occurring in T are replaced by fresh variables. For instance, to solve constraint

$$\mathsf{Pair}(X_1, X_2) \doteq X'\langle M/\omega\rangle$$

we assign X' the type $\mathit{open}\big(\mathsf{Pair}(X_1, X_2)\big) = \mathsf{Pair}(X'_1, X'_2)$ and the constraint becomes $\mathsf{Pair}(X_1, X_2) \doteq \mathsf{Pair}(X'_1, X'_2)\langle M/\omega\rangle$. The right hand side of the constraint is reduced to $\mathsf{Pair}(X'_1\langle M/\omega\rangle, X'_2\langle M/\omega+1\rangle)$ in turn yielding constraints $X_1 \doteq X'_1\langle M/\omega\rangle$ and $X_2 \doteq X'_2\langle M/\omega+1\rangle$.

The effect constraint solving algorithm is run when only effect constrains remain. It starts with the solution assigning the empty set to all effect variables and then in turn tries to satisfy all effect requirements by expanding the sets in the solution. The current algorithm can be seen as a fixed-point algorithm but with backtracking: at certain points it has to choose which variable's set to expand but cannot do so in a deterministic manner; backtracking is required if a wrong choice is made. For instance, to solve constraint set

$$l(M) \in E_1, E_2\langle M/1\rangle \quad , \quad E_1 \sqsubseteq \emptyset$$

the algorithm starts with the solution assigning $\emptyset$ to E_1 and E_2. It will then try to satisfy $l(M) \in E_1, E_2\langle M/1\rangle$ and has to choose if the set of E_1 or E_2 should be expanded. Choosing the set of E_1, the updated solution $E_1 = \{l(M)\}$, $E_2 = \emptyset$ is in conflict with constraint $E_1 \sqsubseteq \emptyset$ so backtracking is required. Selecting E_2 instead, solution $E_1 = \emptyset$, $E_2 = \{l(1)\}$ is finally found. Note that M was abstracted out of $l(M)$.

Conclusion, ongoing and future work A simple type inference algorithm using well-known unification has been devised. Although worst-case running time can be as must as exponential, tests with an implementation have shown low running times in practice.

We are currently working on an extension of the algorithm to a type system guaranteeing robust safety [5] i.e. that a process remains safe in the presence of an arbitrary attacker. The calculus upon which [5] is based employs symmetric cryptography and future work include adding support for asymmetric cryptography [3] and injective correspondences [4].

References

[1] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Higher-order unification via explicit substitutions. In *Proceedings of the Tenth Annual Symposium on Logic in Computer Science*, pages 366–374. IEEE Computer Society Press, June 1995.

[2] Andrew D. Gordon, Hans Hüttel, and René Rydhof Hansen. Type inference for correspondence types. In *6th International Workshop on Security Issues in Concurrency (SecCo 2008)*, 2008.

[3] Andrew D. Gordon and Alan Jeffrey. Types and effects for asymmetric cryptographic protocols. In *CSFW*, pages 77–91. IEEE Computer Society, 2002.

[4] Andrew D. Gordon and Alan Jeffrey. Typing one-to-one and one-to-many correspondences in security protocols. In *ISSS*, volume 2609 of *LNCS*, pages 263–282. Springer, 2002.

[5] Andrew D. Gordon and Alan Jeffrey. Authenticity by typing for security protocols. *Journal of Computer Security*, 11(4):451–519, 2003.

Contract Monitoring and Call-by-name Evaluation
— Extended Abstract —

Markus Degen, Peter Thiemann, and Stefan Wehr

Institut für Informatik, Universität Freiburg, Georges-Köhler-Allee 079,
79110 Freiburg, Germany
{degen,thiemann,wehr}@informatik.uni-freiburg.de

Abstract. Contracts are a proven tool in software development. They provide specifications for operations that may be statically verified or dynamically validated by contract monitoring.
Contract monitoring for strict languages has an established theoretical basis. For languages with call-by-name evaluation, several styles of contract monitoring are possible. In this article, we study two such styles: *eager monitoring* enforces a contract when it is demanded, possibly evaluating expressions not touched by the user code, whereas *delayed monitoring* only proceeds as far as the user code itself can observe.
In each case, an effect system ensures that contract monitoring does not change the meaning of a program and guarantees that contract monitoring is idempotent. Our formalization brings forward semantic reasons that favor delayed monitoring for a call-by-name language and comes with a Haskell implementation.

1 Introduction

Design by contract [8] is a methodology for constructing correct software. Each operation is associated with a contract that defines two assertions, a *precondition* and a *postcondition*, for the operation. The contract is fulfilled if the usual partial correctness condition is true: If the input meets the precondition and the operation produces output, then the output is obliged to meet the postcondition. A program run *violates* a contract if any of the pre- and postconditions of the operations involved is false. Thus, a contract provides a (partial) specification of an operation that every implementation of the operation must fulfill.

While contracts can be verified statically, in practice they are often enforced dynamically using *contract monitoring* (cf. Eiffel [7, 6], Java [1, 5], Scheme [10], or Haskell [4]): The implementation of an operation with monitoring checks the precondition before performing its computation and checks the postcondition before returning to its caller. If the precondition of the operation is false, then it raises an exception blaming its caller. Conversely, if the postcondition does not hold, then the operation blames itself.

The semantics of contract monitoring is intricate, and its correct and complete implementation is non-trivial [3]. From a practical point of view, contract

monitoring should guarantee at least meaning preservation (MP) and behave idempotently (IP):

MP: If a program run with contract monitoring enabled has no contract violations, then disabling contract monitoring should not change its meaning.
IP: Applying a contract multiple times is equivalent to applying it once.

The MP property ensures that developers may enable contract monitoring for a test version of their software and safely disable contract monitoring for the release version, without running the risk that the test and the release version behave differently. The IP property ensures a meaningful notion of contract composition.

In the context of call-by-value evaluation, there is only one useful and sensible mode of contract monitoring. This mode corresponds to its implementation in Eiffel, Java, and Scheme and is the one we just described.

In the context of call-by-name evaluation, there are at least two options, *eager monitoring* and *delayed monitoring*. Eager monitoring enforces an assertion when it is demanded. That is, it checks the precondition when the function demands its argument and it checks the postcondition when the caller demands the function's result. This eager strategy sometimes leads to undesirable behavior which violates the IP property as pointed out by Hinze et al. [4].

We have developed a formalization which precisely pinpoints where eager monitoring imposes too many restrictions on expressions subject to a contract. Hence, we propose delayed monitoring as an alternative form of contract monitoring for languages with call-by-name evaluation. Delayed monitoring places no restrictions on expressions subject to a contract and enjoys the MP and IP properties. It defers enforcement of an assertion until all values that it depends on are evaluated by user code. Thus, monitoring only proceeds as far as the user code itself can observe. Violations that the user code cannot observe, yet, are considered to be invisible. Perhaps surprisingly, this delayed interpretation has a logical foundation: While call-by-value monitoring checks properties according to classical logic, lazy monitoring relies on a three-valued logic.

Contributions

We have developed a semantic framework for specifying and comparing contract monitoring in functional languages [2]. The basis of the framework is an extended version of Moggi's monadic metalanguage [9] with a fixed monad providing for nontermination, mutable state, and exceptions[1] and an effect system for keeping track of the effects.

We have developed the semantics of two styles of contract monitoring for impure functional languages with call-by-name evaluation, eager and delayed monitoring. Both are defined by translation into the metalanguage.

[1] Usually, call-by-name languages provide non-termination as the only effect. However, there is often a back door that allows other effects to creep in. In Haskell, this back door is called `unsafePerformIO`.

The semantics enables us to formally prove (or disprove) the MP and IP properties. The semantics also explains and helps fixing a problem with eager monitoring observed by Hinze et al. [4]. We define and prove correct a criterion that fixes the problem by imposing a suitable typing discipline.

We also provide a prototype implementation of delayed monitoring in Haskell [2].

References

1. P. Abercrombie and M. Karaorman. jContractor: Design by contract for Java. `http://jcontractor.sourceforge.net/`, 2003.
2. M. Degen, P. Thiemann, and S. Wehr. Contract monitoring and call-by-name evaluation. Technical Report 243, Institut für Informatik, Universität Freiburg, http://proglang.informatik.uni-freiburg.de/projects/contracts/, Oct 2008. Full paper and implementation.
3. R. B. Findler and M. Felleisen. Contract soundness for object-oriented languages. In *Proc. 16th ACM Conf. OOPSLA*, pages 1–15, Tampa Bay, FL, USA, 2001. ACM Press, New York.
4. R. Hinze, J. Jeuring, and A. Löh. Typed contracts for functional programming. In *Proc. Eighth International Symposium on Functional and Logic Programming FLOPS 2006*, Fuji Susono, Japan, Apr. 2006. Springer.
5. R. Kramer. iContract — the Java design by contract tool. In *TOOLS 26: Technology of Object-Oriented Languages and Systems*, pages 295– 307, Los Alamitos, CA, USA, 1998.
6. B. Meyer. Applying “design by contract”. *Computer*, 25(10):40–51, Oct. 1992.
7. B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.
8. B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.
9. E. Moggi. Notions of computations and monads. *Information and Computation*, 93:55–92, 1991.
10. The PLT Group. *PLT MzLib: Libraries Manual*. Rice University, University of Utah, Dec. 2005. Version 300.

Type Systems for the Termination of Mobile Processes

Romain Demangeon

ENS Lyon, Université de Lyon, CNRS, INRIA – France

Termination is a key property in programming theory. It is not only desirable in itself, but also as a prequisite for other properties (for instance, *lock-freedom* [KS08]). In this abstract, we address termination of concurrent systems, and especially systems allowing the definition of dynamically evolving structures: typically, new servers can be spawned at run-time, names newly created somewhere are sent elsewhere. Deciding termination for such systems is a challenging task.

We present several type systems for termination in the π-calculus. The soundness of such systems ensures that a typable term is terminating. We describe several results coming from joint work with D. Hirschkoff, D. Sangiorgi and N. Kobayashi [DHKS07,DHS08a,DHS08b]; these studies rely on [DS06] as starting point.

1 Weight-based Type Systems

We use as a formalism the standard polyadic π-calculus with only replicated inputs. We denote a polyadic input action by $a(\widetilde{x})$, a polyadic output action by $\overline{a}\langle\widetilde{v}\rangle$ and a replicated input by $!a(\widetilde{x}).P$. When we examine the semantics of the π-calculus, we notice that the size of a process performing a communication involving non replicated terms, like, e.g., $\overline{a}\langle\widetilde{v}\rangle.P \mid a(\widetilde{x}).Q \to P \mid Q[\widetilde{v}/\widetilde{x}]$ strictly decreases. Actually, the replication is the sole source of divergence for π processes: in a replicated communication $\overline{a}\langle\widetilde{v}\rangle.P \mid !a(\widetilde{x}).Q \to P \mid Q[\widetilde{v}/\widetilde{x}] \mid !a(\widetilde{x}).Q$, the persistence of $!a(\widetilde{x}).Q$, which allows us to model the behavior of a server, can let the size of the whole process increase. Terms P_0 and P_1 of Fig. 1 represent the kind of processes we want to reject, as they contain dangerous replications, which lead to diverging behaviours.

The restriction operator ν, allowing the dynamical creation of new names, contributes importantly to the expressiveness of the π-calculus. However, it is the source of several technical complications in the analyses we describe below. For the sake of clarity, we will not address here name restriction.

The first type system we present, called S1, is introduced in [DS06]. The main idea is to assign a level (a natural number) to every name used in the process. The main typing rule, to control replicated processes, is as follows (here and below, we give a simplified presentation of the typing rules, to ease readability):

$$\frac{\Gamma, \widetilde{x} : \widetilde{T} \vdash P : m \qquad m < n}{\Gamma \vdash !a^n(\widetilde{x}).P : 0} \qquad \text{(S1)}$$

For a process to be type-checked, the levels of the names appearing in output position in the continuation (P) must be smaller than the level of a. Here a^n means that a is given level n. The judgment $\Gamma \vdash Q : l$ means that Q is typable in the context Γ, and that the outputs in Q which do not appear under a replication are of level at most l.

Soundness of the type system is established as follows: take as a measure on processes the multiset of levels of all names appearing in output position not under a replication. One proves that for a type-checked process, this measure strictly decreases at each reduction step: this is insured by the above typing rule.

Remark 1.1 (Complexity of inference). It is easy to see that the inference for S1 is polynomial, as it boils down to searching for cycles in a graph of domination (an edge between a, b represents the constraint $a > b$ between levels assigned to names). This algorithm is implemented in [Kob07], and in [Bou08], where a more expressive variant of S1, described in [DHS08a] is also implemented.

$$P_0 = !a(x).\overline{a}\langle x\rangle \qquad P_1 = !a(x).\overline{b}\langle x\rangle \mid !b(y).\overline{a}\langle y\rangle \qquad P_2 = !a(x).b(y).\overline{a}\langle x\rangle \qquad P_3 = !a(x,y).x.(\overline{a}\langle x,y\rangle \mid \overline{y})$$

$$P_4 = !a(x,y,z).x.(\overline{a}\langle x,y,z\rangle \mid \overline{y} \mid \overline{z}) \qquad P_5 = a(X).(\overline{a}\langle X\rangle \mid X) \qquad P_6 = \overline{a}\langle X \mapsto (X \mid X)\rangle \mid a(F).b(Y).F\lfloor Y\rfloor$$

Fig. 1. Motivating examples.

2 Allowing Forms of Recursion

The main disadvantage of S1, from the point of view of expressiveness, is that we reject a process as soon as it contains a recursion: a replicated input $!a(\widetilde{x}).P$ whose continuation contains an output on a cannot be type-checked, For example, the process P_2 of Fig. 1 is rejected by our type system (the level of a should to be strictly greater than itself), although this process is not intrinsically divergent (it requires an input on a and an input on b to produce a single output on a). Several approaches have been explored to increase expressiveness by allowing some controlled recursions (processes like P_0 should still be rejected).

Multisets of names. The paper [DS06] proposes a more complex system, obtained by considering (replicated) input sequences as a whole: to type-check a process of the form $!a_1(\widetilde{x_1}).\dots.a_k(\widetilde{x_k}).P$ we compare the levels of the multiset of names $\{a_1,\dots,a_k\}$ (called the weight of $\{a_1,\dots,a_k\}$) with the weight the multiset of names appearing in output position (again, not under a replication) in P .

The main typing rule of this system, that we call S2, is as follows:

$$\frac{\Gamma, \widetilde{x_1} : \widetilde{T_1}, \dots, \widetilde{x_k} : \widetilde{T_k} \vdash P : M \qquad \{n_1,\dots,n_k\} >_{lex} M}{\Gamma \vdash !a_1^{n_1}(\widetilde{x_1}).\dots.a_k^{n_k}(\widetilde{x_k}).P : \emptyset} \qquad \text{(S2)}$$

Here $>_{lex}$ is a lexicographical comparison between multisets of integers, and the judgment $\Gamma \vdash Q : N$ means that Q is typable in the context Γ and that N is the multiset of the levels of all outputs not appearing under a replication in Q.

Remark 2.1 (Complexity of inference). We prove in [DHKS07] that the type inference problem for this system is NP-complete. The fact that we are lexicographically comparing two multisets of integers leads to a combinatorial number of possible level assignments and allows us to reduce the problem **1in3 SAT** to the problem of the type inference. We also give in [DHKS07] a variant of system S2, at least as expressive, which uses algebraic comparisons between multisets of names, instead of multiset comparisons. Type inference for this variant is polynomial.

Partial orders. However system S2 is still not expressive enough to type complex processes mimicking the behavior of list-like (a *symbol table* example is detailed in [DS06]) or tree-like (we give an example in [DHS08a]) data structures. The action of propagating a request in such data structures is modelled in the π-calculus by processes like P_3 and P_4 respectively (Fig. 1) (in these examples, we omit the arguments received and sent on this node, for the sake of clarity): a firing of a replication trades an input on a name x (modelling the request on a node of the structure) for outputs on names y, z (y modelling a request on its successor in the list structure, and y, z modelling requests on its children in the tree structure). In a π-calculus encoding of such a dynamically evolving structure, the types of x, y and z are necessarily the same, and so are their levels.

In the case of the list sructure, in such a replication, the weight consumed is equal to the weight released. This motivates the definition of a more refined type system, S3, in [DS06]. System S3 uses a well-founded partial order between names. In a typed replication, either we have $\{n_1,\dots,n_k\} > M$ (as in S2), or $\{n_1,\dots,n_k\} = M$ and the partial order between names decreases: we are trading inputs for outputs of the same level, but going down in the partial order. In the case of P_3, the partial order will state that the name a dominates the name b.

In [DHS08a], we define a type system which is expressive enough to type-check processes modelling the behaviour of the tree structure. In that case, the type system has to be modified in a non trivial way, since the weight associated to the process can *increase* when a replication is fired. This introduces some technicalities, in particular in relation with he control of the restriction operator.

3 Higher-order Mobile Calculi

We are currently working on adapting the ideas exposed above to the higher-order paradigm, where the form of recursion is different. The principles of the previous type system are no longer be applied, as they are based on controlling the replication operator; in higher-order calculi like HOPI_2, a version of the π-calculus where values carried on channels are processes, this operator is not required to obtain divergence. This is illustrated, e.g., by the diverging process $Q_5 = \overline{a}\langle P_5 \rangle \mid P_5$ (P_5 is defined in Fig. 1).

We prove in [DHS08b] that if we forbid recursive outputs, that is, the possibility for a channel a to carry processes containing outputs on a (Q_5 contains such a recursive output), we obtain a terminating system. Indeed, we can assign levels to names like in S1 and forbid the presence of outputs on levels higher than l in a message emitted on a name of level l. Here is a simplified version of the main typing rule of this system:

$$\frac{\Gamma \vdash P : k \qquad \Gamma \vdash Q : m \qquad k < n}{\Gamma \vdash \overline{a}^n \langle P \rangle . Q : \max(m, n)} \qquad \text{(S4)}$$

We write the judgment $\Gamma \vdash Q : l$ when Q is typable in the context Γ and the names in output position in Q are at most of level l.

Termination is enforced as there exists a measure (the multiset of levels of all names at top-level, i.e., not appearing in a process in message position) which decreases at every reduction step: the several copies of the message process spawned contain only outputs whose levels are smaller than the level output consumed.

We notice some similarities between this typing rule for output actions and the typing rule for replicated processes in S1. Indeed, we prove in [DHS08b] that every typable HOPi_2 process is encoded (using the standard encoding from [SW01]) into a first order π-calculus process, which is typable in system S1.

We are however able to adapt system S4 to more complex calculi, such as HOPi_ω, where values carried on channels can be higher-order functions, whose codomain is the set of processes. An example is given by process P_6 from Fig. 1, where a function that duplicates a processes is transmitted on channel a. Other extensions of S4, to calculi combining higher-order features and concurrency, are currently being studied.

References

[Bou08] P. Boutillier. Implementation of a hybrid type system for termination in the π-calculus. Training period report, ENS Lyon, 2008.

[DHKS07] R. Demangeon, D. Hirschkoff, N. Kobayashi, and D. Sangiorgi. On the complexity of termination inference for processes. In *Proc. of TGC'07*, volume 4912 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 2007.

[DHS08a] R. Demangeon, D. Hirschkoff, and D. Sangiorgi. Static and dynamic typing for the termination of mobile processes. In *Proc. of TCS'08*. Springer Verlag, 2008.

[DHS08b] R. Demangeon, D. Hirschkoff, and D. Sangiorgi. Termination in higher-order concurrent calculi. in preparation, 2008.

[DS06] Y. Deng and D. Sangiorgi. Ensuring Termination by Typability. *Information and Computation*, 204(7):1045–1082, 2006.

[Kob07] N. Kobayashi. TyPiCal: Type-based static analyzer for the Pi-Calculus. available from `http://www.kb.ecei.tohoku.ac.jp/~koba/typical/`, 2007.

[KS08] N. Kobayashi and D. Sangiorgi. A Hybrid Type System for Lock-Freedom of Mobile Processes. In *Proc. of CAV'08*, 2008. to appear.

[SW01] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes.* Cambridge University Press, 2001.

Lazy Behavioral Subtyping *

Johan Dovland, Einar Broch Johnsen, Olaf Owe, and Martin Steffen

Dept. of Informatics, University of Oslo, Norway
{johand,einarj,olaf,msteffen}@ifi.uio.no

Abstract. Late binding allows flexible code reuse but complicates formal reasoning significantly, as a method call's receiver class is not statically known. This is especially true when programs are incrementally developed by extending class hierarchies. This talk presents a novel method to reason about late bound method calls. In contrast to traditional behavioral subtyping, reverification is avoided without restricting method overriding to fully behavior-preserving redefinition. The approach ensures that when analyzing the methods of a class, it suffices to consider that class and its superclasses. Thus, the full class hierarchy is not needed, and *incremental* reasoning is supported. We formalize this approach as a calculus which lazily imposes context-dependent subtyping constraints on method definitions. The calculus ensures that all method specifications required by late bound calls remain satisfied when new classes extend a class hierarchy. The calculus does not depend on a specific program logic, but the examples use a Hoare-style proof system. We show soundness of the analysis method.

1 Motivation

Late binding of method calls is a central feature in object-oriented languages and contributes to flexible code reuse. A class may extend its superclasses with new methods, possibly overriding the existing ones. This flexibility comes at a price: It significantly complicates reasoning about method calls as the binding of a method call to code cannot be statically determined; i.e., the binding at run-time depends on the actual class of the called object. In addition, object-oriented programs are often designed under an *open world assumption*: Class hierarchies are extended over time as subclasses are gradually developed and added. In general, a class hierarchy may be extended with new subclasses in the future, which will lead to new potential bindings for overridden methods.

To control this flexibility, existing reasoning and verification strategies impose restrictions on inheritance and redefinition. One strategy is to ignore openness and assume a "closed world"; i.e., the proof rules assume that the complete inheritance tree is available at reasoning time (e.g., [9]). This severely restricts the applicability of the proof strategy; for example, libraries are designed to be extended. Moreover, the closed world assumption contradicts inheritance as an object-oriented design principle, which is intended to support incremental development and analysis. If the reasoning relies on the world being closed, extending the class hierarchy requires a costly reverification.

* This research is partially funded by the EU project IST-33826 CREDO: Modeling and analysis of evolutionary structures for distributed services (http://credo.cwi.nl).

An alternative strategy is to reflect in the verification system that the world is open, but to constrain how methods may be redefined. The general idea is that to avoid reverification, any redefinition of a method through overriding must *preserve* certain properties of the method being redefined. An important part of the properties to be preserved is the method's contract; i.e., the pre- and postconditions for its body. The contract can be seen as a description of the promised behavior of all implementations of the method as part of its interface description, the method's *commitment*. Best known as *behavioral subtyping* (e.g, [1,7,8,10]), this strategy achieves incremental reasoning by limiting the possibilities for code reuse. Once a method has committed to a contract, this commitment may not change in later redefinitions. That is overly restrictive and often violated in practice [11]; e.g., it is not respected by the standard Java library definitions.

2 Contribution

In this work, we relax the property preservation restriction of behavioral subtyping, while embracing the open world assumption of incremental program development. The basic idea is as follows: given a method m declared with p and q as the method's pre- and postcondition, there is no need to restrict the behavior of methods overriding m and require that these adhere to that specification. Instead it suffices to preserve the "part" of p and q actually *used to verify* the program at the current stage. Specifically, if m is used in the program in the form of a method call $\{r\}\ e.m()\ \{s\}$, the pre- and postconditions r and s at that call-site constitute m's *required* behavior and it is those weaker conditions that need to be preserved to avoid reverification. Thus, we distinguish declaration-site specifications, which need not be enforced on redefinitions, from call-site requirements, which are in fact enforced on redefinitions. This distinction leads to behavioral subtyping "by need". We call the corresponding analysis strategy *lazy behavioral subtyping*. This strategy may serve as a blueprint for integrating a flexible system for program verification of late bound method calls into object-oriented program development and analysis tools environments [2–4].

The presentation uses an object-oriented kernel language, based on Featherweight Java [6], and Hoare-style proof outlines. We formalize lazy behavioral subtyping as a syntax-driven inference system in which the analysis of a class is done in the context of a *proof environment* constructed during the analysis. The proof environment keeps track of the context-dependent requirements on method definitions, derived from late bound calls. The strategy is incremental; for the analysis of a class C, only knowledge of C and its superclasses is needed. Proofs derived in the context of superclasses are never violated by later extensions to the class hierarchy. We show the soundness of the proposed analysis strategy. The talk builds on previously published work by the authors [5], but extends this work with methodological aspects and applications in the context of multiple inheritance.

References

1. P. America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, pages 60–90. Springer, 1991.
2. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Intl. Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.
3. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software. The KeY Approach*, volume 4334 of *LNAI*. Springer, 2007.
4. L. Burdy, Y. Cheon, D. R. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, , and E. Poll. An overview of JML tools and applications. In T. Arts and W. Fokkink, editors, *Proceedings of FMICS '03*, volume 80 of *ENTCS*. Elsevier Science Publishers, 2003.
5. J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Lazy behavioral subtyping. In J. Cuellar and T. Maibaum, editors, *Proc. 15th Intl. Symposium on Formal Methods (FM'08)*, volume 5014 of *LNCS*, pages 52–67. Springer, May 2008.
6. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.
7. G. T. Leavens and D. A. Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. Technical Report 06-20a, Department of Computer Science, Iowa State University, Ames, Iowa, 2006.
8. B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, Nov. 1994.
9. C. Pierik and F. S. de Boer. A proof outline logic for object-oriented programming. *Theoretical Computer Science*, 343(3):413–442, 2005.
10. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *8th European Symposium on Programming Languages and Systems (ESOP'99)*, volume 1576 of *LNCS*, pages 162–176. Springer, 1999.
11. N. Soundarajan and S. Fridella. Inheritance: From code reuse to reasoning reuse. In P. Devanbu and J. Poulin, editors, *Proc. Fifth International Conference on Software Reuse (ICSR5)*, pages 206–215. IEEE Computer Society Press, 1998.

Detection of Conflicts in Electronic Contracts*

Stephen Fenech, Gordon J. Pace
Dept. of Computer Science
University of Malta, Msida, Malta
{sfen002,gordon.pace}@um.edu.mt

Gerardo Schneider
Dept. of Informatics
University of Oslo, Norway
gerardo@ifi.uio.no

1 Introduction

Today's trend towards service-oriented architectures, in which different decoupled services distributed not only on different machines within a single organisation but also outside of it, provides new challenges for reliability and trust. Since an organisation may need to execute code provided by third parties, it requires mechanisms to protect itself. One of such mechanisms is the use of *contracts*.
Since services are frequently composed of different sub-services, each with its own contract, there is a need to guarantee that each single contract is conflict-free. Moreover, one needs to ensure that the conjunction of all the contracts is also conflict-free —meaning that the contracts will never lead to conflicting or contradictory normative directives.
$\mathcal{CL}$ [5] is a formal language to specify deontic electronic contracts. A trace semantics for the language was presented in [4], useful for runtime monitoring of $\mathcal{CL}$ contracts. Such semantics, however, lacks the deontic information concerning the obligations, permissions and prohibitions of the involved parties in the contract, and thus it is not suitable for conflict analysis.
We present here an extension of the trace semantics of $\mathcal{CL}$ given in [4] to support conflict analysis. Based on that semantics we have developed a decision procedure to automatically detect conflicts in contracts written in $\mathcal{CL}$. We have implemented such an algorithm into an *ad hoc* model checker. Due to space restriction we only present in what follows the $\mathcal{CL}$ syntax, the extended trace semantics, and a brief discussion on the automata construction basis of our model checker.

2 Deontic Logic and $\mathcal{CL}$

Deontic logic [6] enables reasoning about non-normative and normative behaviour (e.g., obligations, permissions and prohibitions), including not only the ideal behaviours but also the exceptional and actual behaviours. One of the main problems of the logic is the difficulty theoreticians have to define a consistent yet expressive formal system, free from paradoxes.
Instead of trying to solve the problem of having a complete paradox-free deontic logic, $\mathcal{CL}$ has been designed with the aim to be used on a restricted application domain: electronic contracts. In this way the expressivity of the logic is reduced, resulting in a language free from most classical paradoxes, but still of practical use. $\mathcal{CL}$ is based on a combination of deontic, dynamic and temporal logics, allowing the representation of obligations, permissions and prohibitions, as well as temporal aspects. Moreover, it also gives a mean to specify *exceptional* behaviours arising from the violation of obligations (what is to be demanded in case an obligation is not fulfilled) and of prohibitions (what is the penalty in case a prohibition is violated). These are usually known in the deontic community as *Contrary-to-Duties* (CTDs) and *Contrary-to-Prohibitions* (CTPs) respectively.
$\mathcal{CL}$ contracts are written using the following syntax:

$$\begin{array}{rcl} C & := & C_O|C_P|C_F|C \wedge C|[\beta]C|\top|\bot \\ C_O & := & O_C(\alpha)|C_O \oplus C_O \\ C_P & := & P(\alpha)|C_P \oplus C_P \\ C_F & := & F_C(\delta)|C_F \vee [\alpha]C_F \\ \alpha & := & 0|1|a|\alpha\&\alpha|\alpha \cdot \alpha|\alpha + \alpha \\ \beta & := & 0|1|a|\beta\&\beta|\beta \cdot \beta|\beta + \beta|\beta^* \end{array}$$

A contract clause C can be either an obligation (C_O), a permission (C_P) or a prohibition (C_F) clause, a conjunction of two clauses or a clause preceded by the dynamic logic square brackets. $O_C(\alpha)$ is interpreted as the obligation to perform α in which case, if violated, then the reparation contract C must be executed (a CTD). $F_C(\alpha)$ is interpreted as forbidden to perform α and if α is performed then the reparation C must be executed (a CTP). $[\beta]C$ is interpreted as if action β is performed then the contract C must be executed — if β is not performed, the contract is trivially satisfied. Compound actions can be constructed from basic ones using the operators $\&$, $\cdot$, $+$ and * where $\&$ stands for the actions occuring concurrently, $\cdot$ stands for the actions to occur in sequence, $+$ stands for a choice between actions and * is the Kleene star. It can be shown that every action expression can be transformed into an equivalent representation where $\&$ appears only at the innermost level. This representation is referred to as the canonical form. In the rest of this paper we assume that action expressions have been reduced to this form. 1 is an action expression matching any action, while 0 is the impossible action. In order to avoid paradoxes the operators combining obligations, permissions and prohibitions

*Partially supported by the Nordunet3 project COSoDIS: "Contract-Oriented Software Development for Internet Services".

are restricted syntactically. See [5, 4] for more details on $\mathcal{CL}$.

As a simple example, let us consider the following clause from an airline company contract: 'When checking in, the traveller is obliged to have a luggage within the weight limit — if exceeded, the traveller is obliged to pay extra.' This would be represented in $\mathcal{CL}$ as $[checkIn]O_{O(pay)}(withinWeightLimit)$.

2.1 Trace Semantics

The trace semantics presented in [4] enables checking whether or not a trace satisfies a contract. However, deontic information is not preserved in the trace and thus it is not suitable to be used for conflict detection. By a conflict we mean for instance that the contract permits and forbids performing the same action at the same time (see below for a more formal definition of conflict). We present in what follows an extension of the trace semantics given in [4].

We will use lower case letters $(a, b \ldots)$ to represent atomic actions, Greek letters $(\alpha, \beta \ldots)$ for compound actions, and Greek letters with a subscript & $(\alpha_{\&}, \beta_{\&}, \ldots)$ for compound concurrent actions built from atomic actions and the concurrency operator &. The set of all such concurrent actions will be written $A_{\&}$. We use # to denote mutually exclusive actions (for example, if a stands for 'opening the check-in desk' and b for 'closing the check-in desk', we write $a\#b$).

In order for a sequence σ to satisfy an obligation, $O_C(\alpha_{\&})$, $\alpha_{\&}$ must be a subset or equal to $\sigma(0)$ or the rest of the trace satisfies the reparation C, thus for the obligation to be satisfied all the atomic actions in $\alpha_{\&}$ must be present in the first set of the sequence. For a prohibition to be satisfied, the converse is required, that is, not all the actions of $\alpha_{\&}$ are executed in the first step of the sequence. One should note that permission is not defined in this semantics since a trace cannot violate a permission clause. An important observation is that the negation of an action is defined as performing any other action except the negated action.

In order to enable conflict analysis, we start by adding deontic information in an additional trace, giving two parallel traces — a trace of actions (σ) and a trace of deontic notions (σ_d). Similar to σ, σ_d is defined as a sequence of sets whose elements are from the set D_a which is defined as $\{O_a \mid a \in A\} \cup \{F_a \mid a \in A\} \cup \{P_a \mid a \in A\}$ where O_a stands for the obligation to do a, F_a stands for the prohibition to do a and P_a for permission to do a.

Also, since conflicts may result in sequences of finite behaviour which cannot be extended (due to the conflict), we reinterpret the semantics over finite traces. A conflict may result in reaching a state where we have only the option of violating the contract, thus any infinite trace which leads to this conflicting state will result not being accepted by the semantics. We need to be able to check that a finite trace has not yet violated the contract and then check if the following state is conflicting. We use a semicolon (;) to denote catenation of two sequences, and len to return the length of a finite sequence. Two traces are pointwise (synchronously) joined using the combine operator where we will use the $\cup$ symbol and defined: $(\sigma \cup \sigma')(n) = \sigma(n) \cup \sigma'(n)$. Furthermore, if α is a set of atomic actions then we will use O_α to denote the set $\{O_a \mid a \in \alpha\}$.

The extended trace semantics for $\mathcal{CL}$ is given below, where $\sigma, \sigma_d \vDash_f C$ can be interpreted as 'finite action sequence σ and deontic sequence σ_d do not violate contract C':

$\sigma, \sigma_d \nvDash_f C$ if $len(\sigma) \neq len(\sigma_d)$

$\sigma, \sigma_d \vDash_f \top$ if $len(\sigma) = 0$ or $\forall i \sigma_d(i) = \emptyset$

$\sigma, \sigma_d \nvDash_f \bot$

$\sigma, \sigma_d \vDash_f C_1 \wedge C_2$ if $\sigma, \sigma'_d \vDash_f C_1$ and $\sigma, \sigma''_d \vDash_f C_2$ and $\sigma_d = \sigma'_d \cup \sigma''_d$

$\sigma, \sigma_d \vDash_f C_1 \oplus C_2$ if $\sigma, \sigma_d \vDash_f C_1$ or $\sigma, \sigma_d \vDash_f C_2$

$\sigma, \sigma_d \vDash_f [\alpha_{\&}]C$ if $len(\sigma) = 0$ or $\sigma_d(0) = \emptyset$ and $(\alpha_{\&} \subseteq \sigma(0)$ and $\sigma(1..), \sigma_d(1..) \vDash_f C,$ or $\alpha_{\&} \nsubseteq \sigma(0)))$

$\sigma, \sigma_d \vDash_f [\beta; \beta']C$ if $\sigma, \sigma_d \vDash_f [\beta][\beta']C$

$\sigma, \sigma_d \vDash_f [\beta + \beta']C$ if $\sigma, \sigma_d \vDash_f [\beta]C \wedge [\beta']C$

$\sigma, \sigma_d \vDash_f [\beta^*]C$ if $\sigma, \sigma_d \vDash_f C \wedge [\beta][\beta^*]C$

$\sigma, \sigma_d \vDash_f [C_1?]C_2$ if $\sigma, \sigma_d \nvDash_f C_1,$ or $\sigma, \sigma_d \vDash_f C_1 \wedge C_2$

$\sigma, \sigma_d \vDash_f O_C(\alpha_{\&})$ if $len(\sigma) = 0$ or $\sigma_d(0) = O\alpha$ and $((\alpha_{\&} \subseteq \sigma(0)$ and $\sigma(1..), \sigma_d(1..) \vDash_f \top)$ or $\sigma(1..), \sigma_d(1..) \vDash_f C)$

$\sigma, \sigma_d \vDash_f O_C(\alpha; \alpha')$ if $\sigma, \sigma_d \vDash_f O_C(\alpha) \wedge [\alpha]O_C(\alpha')$

$\sigma, \sigma_d \vDash_f O_C(\alpha + \alpha')$ if $\sigma, \sigma_d \vDash_f O_\bot(\alpha)$ or $\sigma, \sigma_d \vDash_f O_\bot(\alpha')$ or $(\sigma_d(0) = (O\alpha$ or $O\alpha')$ and $\sigma, \emptyset; \sigma_d(1..) \vDash_f [\overline{\alpha + \alpha'}]C)$

$\sigma, \sigma_d \vDash_f F_C(\alpha_{\&})$ if $len(\sigma) = 0$ or $\sigma_d(0) = F\alpha$ and $((\alpha_{\&} \nsubseteq \sigma(0)$ and $\sigma(1..), \sigma_d(1..) \vDash_f \top)$ or $(\alpha_{\&} \subseteq \sigma(0)$ and $\sigma(1..), \sigma_d(1..) \vDash_f C))$

$\sigma, \sigma_d \vDash_f F_C(\alpha; \alpha')$ if $\sigma_d(0) = F\alpha$ and $(\sigma, \sigma_d \vDash_f F_\bot(\alpha)$ or $\sigma, \sigma_d \vDash_f [\alpha]F_C(\alpha'))$

$\sigma, \sigma_d \vDash_f F_C(\alpha + \alpha')$ if $\sigma, \sigma_d \vDash_f F_C(\alpha) \wedge F_C(\alpha')$

$\sigma, \sigma_d \vDash_f [\overline{\alpha_{\&}}]C$ if $\sigma_d(0) = \emptyset$ and $((\alpha_{\&} \nsubseteq \sigma(0)$ and $\sigma(1..), \sigma_d(1..) \vDash_f C)$ or $\alpha_{\&} \subseteq \sigma(0))$

$\sigma, \sigma_d \vDash_f [\overline{\alpha; \alpha'}]C$ if $\sigma, \sigma_d \vDash_f [\overline{\alpha}]C \wedge [\alpha][\overline{\alpha'}]C]$

$\sigma, \sigma_d \vDash_f [\overline{\alpha + \alpha'}]C$ if $\sigma_d(0) = \emptyset$ and $(\sigma\sigma_d \vDash_f [\overline{\alpha}]C$ or $\sigma, \sigma_d \vDash_f [\overline{\alpha'}]C)$

$\sigma, \sigma_d \vDash_f P(\alpha)$ if $len(\sigma) = 0$ or $\sigma_d(0) = P\alpha$ and $\sigma(1..), \sigma_d(1..) \vDash_f \top$

$\sigma, \sigma_d \vDash_f P(\alpha; \alpha')$ if $\sigma, \sigma_d \vDash_f P(\alpha) \wedge [\alpha]P(\alpha')$

$\sigma, \sigma_d \vDash_f P(\alpha + \alpha')$ if $\sigma, \sigma_d \vDash_f P(\alpha) \wedge P(\alpha')$

Note that the conditions for a trace containing a permission not to violate the contract are defined on σ_d rather than on the trace of actions. So, for any σ there exists a σ_d which will not violate a permission clause. Also note

that in the absence of deontic notions the corresponding element in σ_d is the empty set. We have proved that the infinite and finite trace semantics are sound and complete with respect to each other.

3 Conflict Analysis

Conflicts in contracts arise for 4 different reasons. First, we can be obliged and forbidden to do the same action, and second, we can be permitted and forbidden to perform the same action. In the first conflict we would end up in a state where whatever we do we will violate the contract. The second conflict situation would not result in having a trace that violates the contract since in the trace semantics permissions cannot be broken, however, since we are augmenting the original trace semantics with the deontic notions we can still identify these situations. The remaining two cases correspond to obligations (and permissions and obligations) of mutually exclusive actions. Freedom from conflict can be defined formally as follows (recall that $a\#b$ if a and b are mutually exclusive actions):

Definition 3.1 *A contract C is said to be conflict free if for all traces σ_f and σ_d satisfying $\sigma_f,\ \sigma_d \vDash_f C$, there is no conflict in σ_d, meaning that it is not the case that any of the following are true:*

1. $\exists i \cdot Oa \in \sigma_d(i)$ *and* $Fa \in \sigma_d(i)$
2. $\exists i \cdot Pa \in \sigma_d(i)$ *and* $Fa \in \sigma_d(i)$
3. $\exists i \cdot Oa \in \sigma_d(i)$ *and* $Ob \in \sigma_d(i)$ *and* $a\#b$
4. $\exists i \cdot Oa \in \sigma_d(i)$ *and* $Pb \in \sigma_d(i)$ *and* $a\#b$

By unwinding a $\mathcal{CL}$ formula according to the finite trace semantics, we create an automaton which accepts all non violating traces, and such that any trace resulting in a violation ends up in a violating state. Furthermore, we label the states of the automaton with deontic information provided in σ_d, so we can ensure that a contract is conflict free simply through the analysis of the resulting reachable states (non-violating states).
States of the automaton contain a set of formulae still to be satisfied, following the standard sub-formula construction (as done for instance for CTL). Each transition is labelled with the set of actions that are to be performed in order to move along the transition. From the canonical form assumption we can look at an action as a disjunction of actions that must occur now and for each of these a compound action that needs to occur in the next step. This view is very helpful when processing the actions since a compound action α can be seen as an array of possibilities α_i where for each entry we have the atomic actions which need to hold now ($\alpha_i.now$) and the possibly compound or empty actions that need to follow next ($\alpha_i.next$).
Once the automaton is generated we can go through all the states and check for the four types of conflicts. If there is a conflict of type one or three, then all transitions out of the state go to a special violation state. In general we might need to generate all possible transitions before processing each sub-formula, resulting on a big automaton. In practice, we improve the algorithm in such a way that we create all and only those required transitions reducing the size considerably.
Conflict analysis can also be done on-the-fly without the need to create the complete automaton. One can process the states without storing the transitions and store only satisfied subformulas (for termination), in this manner, memory issues are reduced since only a part of the automaton is stored in memory.

4 Final Remarks

In this paper, we have presented a finite trace semantics for $\mathcal{CL}$ augmented with deontic information, and sketched how it can be used for automatic analysis of contracts for conflict discovery. The automaton we have created here could also be used as a basis for other kinds of analysis not just conflict analysis. These include the possibility of performing queries, the detection of unreachable clauses, and the identification of superfluous clauses. Based on the construction presented in this paper, we have implemented a model checker for detecting conflicts in $\mathcal{CL}$ [1]. In other ongoing work using the semantics presented in this paper, we have implemented a translation from the automaton created from $\mathcal{CL}$ contracts into the runtime verification tool LARVA [2]. This enables us to write contracts about Java programs and automatically obtain monitors to ensure conformance to the contracts at runtime. More detailed trace semantics, the conflict analysis algorithm (including proof of soundness, completeness and termination), as well as a description of the tool can be found in [3].

References

[1] CLAN. CL ANalyser – A tool for Contract Analysis. Available from `www.cs.um.edu.mt/~svrg/Tools/CLTool/`.

[2] C. Colombo, G. J. Pace, and G. Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *FMICS 2008*, LNCS, 2008.

[3] S. Fenech. Conflict analysis of deontic contracts. Master's thesis, Dept. of Computer Science, Univ. of Malta, 2008.

[4] M. Kyas, C. Prisacariu, and G. Schneider. Runtime monitoring of electronic contracts. In *ATVA'08*, LNCS. Springer-Verlag, Oct. 2008. To appear.

[5] C. Prisacariu and G. Schneider. A Formal Language for Electronic Contracts. In *FMOODS'07*, volume 4468 of *LNCS*, pages 174–189. Springer, June 2007.

[6] G. von Wright. Deontic logic. *Mind*, (60):1–15, 1951.

Higher-order attribute semantics of flat declarative languages

Pavel Grigorenko
Institute of Cybernetics, Tallinn University of Technology, Estonia
pavelg@cs.ioc.ee

Let us consider attribute semantics of a traditional programming language as defined originally in [3] and explained in terms of attribute models in [6]. If we look at an attribute model of a production of the language, or at an attribute model of a syntax tree of a text written in this language, we can see that it is just a collection of variables bound by functional dependencies. In other words — it is a functional constraint network representing the meaning of a production or a text. In the present work we extend the attribute models by allowing attribute dependencies to be, beside functional dependencies, also higher-order functional dependencies. This gives us a possibility to express more control of computations in an attribute model itself. We consider declarative languages where a text can be not only a specification of a single program, but it can be also a description of a device or a system (its model) that allows one to ask different questions (cf. Prolog). This means that a program can be obtained from a declarative specification and a goal (a problem statement describing what is needed). We have restricted the set of specification languages considered here to structurally very simple languages that we call flat languages.

A *flat language* is a declarative language suitable for composing typed objects into descriptions of concepts and/or systems (a mathematical model in a broad sense) by connecting their components by equalities. The meaning of a text in a flat language is hidden in the types of objects and in the way the objects are connected. Types of objects describe the ways of possible computations with objects, and their possible values. Good examples of flat languages are visual languages where specifications are schemes, e.g. the language of class diagrams of UML and many simulation languages. Even many popular special purpose specification languages like, for instance, VHDL are in essence flat languages, although they have some features that are difficult to express through local connections of objects.

First, we give a syntax and intuitive semantics of a flat language by defining statements of the *core language* and its extensions. We call conventional attribute models as defined in [6] *simple attribute models*, introduce *computational problems* and use the *value propagation* as a procedure of attribute evaluation on simple attribute models. We define the semantics of a specification in the core language as a set of all algorithms that can be composed on the attribute model of the specification for solving computational problems.

Let U and V be two sets of attributes of an attribute model M. We denote by $U \to V$ a *computational problem* on the attribute model M, and say that U is a set of input attributes (or just inputs) and V is a set of output attributes (or outputs) of a computational problem. The computational problem states a goal that, given values of attributes from U, requires to find values of attributes of V using attribute dependencies of M.

Let A be a set of attributes and P a set of computational problems with inputs and outputs from A.

Higher-order attribute dependency (*hoad*) is a functional dependency that has inputs from $A \cup P$ and outputs from A. Inputs from P are called *subtasks*.

Higher-order attribute model is a pair $\langle A, R\rangle$ where A is a set of attributes and R is a set of attribute dependencies that includes some higher-order attribute dependencies on the set of attributes A. Higher-order attribute models are so expressive that enable one to synthesize recursive, branching and cyclic programs where respective control structures, i.e. recursion, branching and loops are preprogrammed and represented as higher-order attribute dependencies.

Often only one higher-order attribute dependency is used in a specification. The evaluation strategy is quite obvious in this case: first use only conventional attribute dependencies and at the end the higher-order one. Thereafter, if still needed, use simple attribute dependencies again. Time complexity of the search remains linear in this case.

In a general case, when an attribute model M contains several higher-order attribute dependencies, the evaluation strategy is as follows.

First the procedure of simple value propagation is done using only attribute dependencies that are not higher-order. If this does not solve the problem (does not give values of all outputs of the problem),

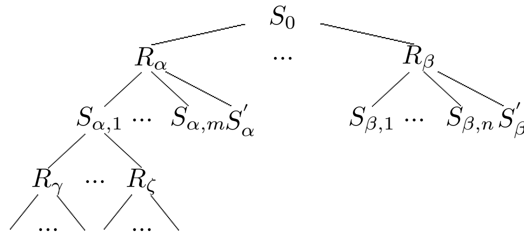


Figure 1: *And-or* search tree for attribute evaluation on higher-order attribute model

then a *hoad* is applied, if it is applicable. A *hoad* is applicable if and only if all its inputs are given and all its subtasks are solvable and it computes values of some attributes that have not been evaluated yet. A sequence of applicable attribute dependencies obtained in this way is called *maximal linear branch* (*mlb*). It contains one *hoad* at the end of the sequence. There are three possible outcomes of this procedure:

1. After constructing the *mlb* the problem is solvable (like in the case of a single *hoad*).
2. A *mlb* cannot be found and the problem is unsolvable.
3. A *mlb* can be found and the initial problem $U_1 \rightarrow V_1$ is reduced to a smaller one $U_2 \rightarrow V_2$, $U_2 = U_1 \cup Y$ and $V_2 = V_1 \backslash Y$, where Y is the set of outputs of the *hoad*.

This procedure (construction of *mlb*) is repeatedly applied until the problem is solved or no more *mlb* can be constructed.

It is important to notice that for applying a *hoad* we have to solve all its subtasks. This means that the whole procedure of problem solving must be applied for every subtask. This requires a search on an *and-or* tree of problems (subtasks) on the attribute model. The root of a tree corresponds to the initial problem, and it is an *or*-node, because there may be several possible *mlb*s for this problem. *And*-nodes correspond to higher-order attribute dependencies and have one successor for its each subtask, plus one successor for the reduced task that has to be solved after applying the *mlb*. *Or*-nodes of the tree correspond to the subtasks that have to be solved for their parent *and*-node.

Let us label *hoad*s with R_i, $i = 1, \ldots, i_{max}$, where i_{max} is the number of all *hoad*s in the model and abbreviate S for a subtask. Then a *hoad* R_i has the form:

$$S_{i,1}, ..., S_{i,r},\ x_1, ..., x_k \rightarrow\ y_1, ..., y_l\{f\},$$

where $x_s, y_t \in A$ are attributes and $S_{i,j} \in P$ are subtasks.

Figure 1 shows a part of an *and-or* search tree for solving a problem S_0 on a higher-order attribute model. The *and*-nodes are the *hoads* $R_\alpha, \ldots, R_\beta$ with $\alpha, \ldots, \beta \in \{1, \ldots, i_{max}\}$ that can be tried first for solving a problem of their parent node. The successors of a *hoad* node R_i are its subtasks $S_{i,1}, \ldots, S_{i,j}$ and the reduced problem $S_i^{'}$ that remains to solve after applying the *hoad*. The search on the *and-or* tree is depth-first search with backtracking. In fact, our attribute evaluation procedure only constructs an algorithm — a tree of applicable attribute dependencies, the values of attributes are computed only during the execution of a program extracted from this tree. Unlike Prolog, backtracking in our case does not involve unnecessary evaluation of variables.

It is useful to notice that types of attribute dependencies (and higher-order attribute dependencies) can be considered as propositional formulas where arrows denote implications and commas denote conjunctions. Building an attribute evaluation algorithm for a particular computational problem with inputs $u_1, \ldots, u_m$ and outputs $v_1, \ldots, v_n$ corresponds then to a derivation of the formula $u_1 \wedge \ldots \wedge u_m \supset v_1 \wedge \ldots \wedge v_n$ in the intuitionistic propositional calculus (IPC). This is justified by the Curry-Howard isomorphism [2]. This gives also an algorithm of proof search for IPC, although some transformation of propositional formulas to the suitable form will be needed in the general case, see [5].) An unpleasant consequence of this fact is that the proof search for IPC is PSPACE-complete [7], hence the higher-order attribute evaluation has exponential time complexity. However sufficiently good search strategies help to reduce the search in practical cases [5].

We have implemented a flat language in a Java-based visual programming framework CoCoViLa [1]. The experience of using CoCoViLa has shown that higher-order attribute semantics is a practically useful instrument for implementing domain specific languages. CoCoViLa has been successfully used for the automatic composition of web services [4].

Acknowledgements This work has been partially supported by the Estonian Science Foundation grant No.6886.

References

[1] Pavel Grigorenko, Ando Saabas, Enn Tyugu. Cocovila - compiler-compiler for visual languages. *Electr. Notes Theor. Comput. Sci.*, 141(4):137–142, 2005.

[2] William A. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, New York, 1980. Reprint of 1969 article.

[3] Donald Knuth. Semantics of context-free grammars. *Mathematical Systems Theory*, 2:127–145, 1968.

[4] Riina Maigre, Pavel Grigorenko, Peep Küngas, Enn Tyugu. Stratified composition of web services. In: *Knowledge-based software engineering : Proceedings of the Eighth Joint Conference on Knowledge-Based Software Engineering: (Toim.) Virvou, Maria; Nakamura, Taichi.* Amsterdam: IOS Press, 2008, (Frontiers in Artificial Intelligence and Applications; 180), 49 − 58, 2008.

[5] Mihhail Matskin, Enn Tyugu. Strategies of structural synthesis of programs and its extensions. *Computing and Informatics*, 20:1–25, 2001.

[6] Jaan Penjam. Computational and attribute models of formal languages. *Theoretical Computer Science*, 1990.

[7] Richard Statman. Intuitionistic propositional logic is polynomial-space complete. *Theoretical Computer Science*, 9:67–72, 1979.

Quantification of Information Flow for Value-passing Process Algebra ⋆

Damas P. GRUSKA

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

Abstract. A quantification of an information flow is defined and studied in the framework of value-passing process algebras. It is based on Shannon's information theory and on a concept of noisy channels.

Keywords: information flow, entropy, security, process algebra, value-passing

1 Introduction

Many security properties are based on an absence of information flow or non-interference (see [GM82]) between private and public system data or activities. 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 for such formalisms as Petri nets, process algebras, imperative languages etc. The absence of information flow or non-interference is a qualitative property - either there is or there is not. It does not take into account an amount of information which can be gained by an intruder from system's observations. For example, if we tray to login to a system even by typing an incorrect password we gain some information since the space of possible passwords was in such a way reduced by one. In spite of that, an existence of such the interference the system is considered to be secure if the set of admissible passwords is large enough.

The aim of this paper is to precisely quantify an amount of (private) information which can be gained by an intruder. As a basic formalism we take a variant of Milners CCS with value-passing and guarded processes ($[be]P$). To quantify an amount of information regarding private system's activities or data which could be gained we will use Shannon's information theory. We will study concepts as *entropy* and *mutual information* between public inputs, public outputs and private data. The concepts will be compared with other security notions known in the literature.

As regards related works there is a number of papers devoted to quantification of information flow in the framework of imperative languages (see [CHM07] for an overview). In [L02] an information flow is studied in case of process algebras. Particularly, it is investigated how much information i.e. a number of

⋆ Work supported by the grant VEGA 1/3105/06.

bits can be transmitted by observing some timed system activities. In [Gru08] a quantitative information flow is formalized by means of observation functions (which can hide some system activities) and by rather general security property called opacity (see [BKMR06]) in framework of timed process algebras. A predicate over sequences of system actions is opaque if from an observation of system activities an observer cannot deduce whether the predicate holds or it does not hold. Shannon's information theory is exploited for quantification of an amount of information regarding validity of given predicate ϕ. As the basic formalism Timed CCS without value-passing was used.

In this paper we will work with a variant of CCS with value-passing and with guarded processes (process $[be]P$ behaves like P if boolean-expression be is evaluated to true otherwise as Nil). As we show later the choice of the boolean expression can significantly influence resulting quantity of possible information flow. We split the set of channels to two disjoint sets of private and public ones with names $h, h_1, h_2, \ldots$ and $l, l_1, l_2, \ldots$, respectively.

Let us consider a simple access control process $\mu X.l(x).h(y)([x = y].\bar{l}_1\texttt{LogIn}.X +[x \neq y].\bar{l}_1\texttt{IncPassw}.X)$ which check whether a user types the correct password. Clearly, there is some information flow between privata input $h(y)$, public input $l(x)$ and possible public outputs $\{\texttt{LogIn}, \texttt{IncPassw}\}$. If a password is, for example, a random string of 8 characters each chosen from set of 2^7 elements, i.e. roughly 10^{20} possibilities, the corresponding information flow is very low. On the other side, if the password is a word from English dictionary (with size, say 10^6, the corresponding information flow is significantly increased. This is due to smaller set of possible private inputs. Now let us consider another example. Let us assume processes $\mu X.l(x).h(y)([x = y].\bar{l}_1\texttt{Accepted}.X + [x \neq y].\bar{l}_1\texttt{Refused}.X)$ which simulates whether an user types a correct 4 digit pin code (out of 10^4 possibilities) and similar process $\mu X.l(x).h(y)([x < y].\bar{l}_1\texttt{Accepted}.X + [x \not< y].\bar{l}_1\texttt{Refused}.X)$ for which there is a very high information flow. Actually due to different choice of guarding boolean expression one needs just 14 attempts to obtain private value transmitted via h. To express quantity of an information flow we will exploit Schannon's information theory (see [Sch48]). Let X be a discrete random variable and let x ranges over the set of values which X may take. By $p(x)$ we will denote probability that X takes the value x. The information entropy (also called self-information or a measure of uncertainty) of the variable X is denoted $\mathcal{H}(x)$ and is defined as the following: $\mathcal{H}(X) = \sum_x p(x). \log_b \frac{1}{p(x)}$. We define $p(x). \log_b \frac{1}{p(x)} = 0$ if $p(x) = 0$. We will work with the base b of $\log_b$ equal to 2 and hence the unit of the information entropy will be one bit. Given two discrete random variables X and Y, the mutual information between them, written $\mathcal{I}(X;Y)$, is defined as follows: $\mathcal{I}(X;Y) = \sum_x \sum_y p(x,y). \log \frac{p(x,y)}{p(x).p(y)}$. It can be easily shown that $\mathcal{I}(X;Y) = \mathcal{H}(X) + \mathcal{H}(Y) - \mathcal{H}(X,Y) = \mathcal{H}(X) - \mathcal{H}(X|Y) = \mathcal{H}(Y) - \mathcal{H}(Y|X)$ where $\mathcal{H}(X|Y)$ is the conditional entropy of X given knowledge of Y. Clearly, if X and Y are independent then $\mathcal{I}(X;Y) = 0$.

Let us consider process P. Without loss of generality we will be interested only in an information flow between data which P receives from one private channel h and one public input/output channel l. Note that there might be other

channels which are out of interests. Note that the following definitions could be extended to tuples of private/public channels. We will treat P as a noisy communication channel (more noisy = more secure) with two inputs (actions $h(x), l(y)$) and one output (actions $\bar{l}v$). So we will omit all other actions and we will consider sequences s such that $P \stackrel{s}{\rightarrow}$ and $s = s_1.h(x).s_2.l(x).s_3.lv$ or $s = s_1.l(x).s_2.h(x).s_3.lv$ such that $s_1, s_2, s_3, s_4 \in (Act \setminus \{h(x), l(x), \bar{l}v\})^\star$.

Suppose that P can receive data from this input channels according distribution given by discrete random variables H_{in}, L_{in} and P produce as the output discrete random variables L_{out}.

We define information flow between private inputs and public outputs (knowing public inputs) as follows: $\mathcal{F}(H \rightsquigarrow L) = \mathcal{I}(H_{in}, L_{out}|L_{in})$ (note that this definition can be naturally extended to the case of several private and public channels). If there is no information flow i.e. random variables H_{in} and $L_{out}|L_{in}$ are independent then $\mathcal{F}(H \rightsquigarrow L) = 0$. In general, higher values of $\mathcal{F}(H \rightsquigarrow L)$ represent higher information flows and less secure processes.

As it was clear from the above example, in the case of process $[be]P$ an amount of information flow depends also on boolean expression be. Suppose that be contains private variables. We define entropy of be as the entropy of corresponding discrete random variable obtained from be which is viewed as the function of private variables with parameters given by public variables. For example, for boolean expression $[x = y]$ from the above mentioned example (x being private and y public variable), we have entropy close to zero $(9999/10000. \log(10000/9999) + 1/10000. \log(10000))$ for every y. On the other side, for boolean expression $[x < y]$ we can increase entropy by suitable choice of y to 1 (what corresponds to a binary choice).

Now suppose that we know the entropy of be and $\mathcal{F}(H \rightsquigarrow L)$ for processes P and Q. From the above mentioned information we could tray to obtain an evaluation of $\mathcal{F}(H \rightsquigarrow L)$ for $[be]P$ and $P + Q$ or to obtain other compositional results.

References

[BKMR06] 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

[CHM07] Clark D., S. Hunt and P. Malacaria: A Static Analysis for Quantifying the Information Flow in a Simple Imperative Programming Language. The Journal of Computer Security, 15(3). 2007.

[GM82] Goguen J.A. and J. Meseguer: Security Policies and Security Models. Proc. of IEEE Symposium on Security and Privacy, 1982.

[Gru08] Gruska D.P.: Quantifying Information Flow in Process Algebras, in proceedings of Concurrency, Specification and Programming, 2008.

[L02] Lowe G.: Quantifying information flow". In Proc. IEEE Computer Security Foundations Workshop, 2002.

[Sch48] Shannon, C. E.: A mathematical theory of communication. Bell System Technical Journal, vol. 27, 1948.

Improving Scalability of Model-Checking for Minimizing Buffer Requirements of Synchronous Dataflow Graphs

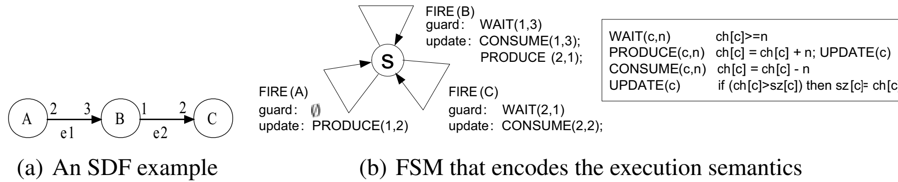
Nan Guan[1], Wang Yi[2], Zonghua Gu[3], and Ge Yu[1]

[1] Northeastern University, Shenyang, China
[2] Uppsala University, Uppsala, Sweden
[3] Hong Kong University of Science and Technology, Hong Kong, China

1 Introduction

Synchronous Dataflow (SDF) is a widely-used model of computation for signal processing applications that allows for powerful static analysis and synthesis techniques. Since signal processing and multimedia applications are often implemented on resource constrained embedded systems, it is important to minimize their memory size. The data buffer minimization problem of SDF is known to be NP-complete [1]. Some authors have used model-checking to obtain the optimal solution [2][3] by exploring the entire state space. One key limitation of model-checking is its lack of scalability due to state space explosion. In this paper, we present several techniques for reducing the state space and improving scalability of model-checking when applied to the SDF buffer minimization problem. We focus on the NuSMV model-checker, but our techniques are at the application-level, and independent of the specific model-checker used.

A SDF graph is a directed graph $G = (V, E)$, where V is the set of nodes representing actors and E is the set of edges. An edge $e \in E$ has a source actor $src(e)$ that produces $p(e)$ tokens on e at each invocation (referred to as an actor *firing* in the rest of the paper), and a sink actor $snk(e)$ that consumes $c(e)$ tokens on e at each firing. $d(e)$ denotes the initial number of tokens on e, also called the *initial delay*.



(a) An SDF example (b) FSM that encodes the execution semantics

Fig. 1. SDF graph Example 1.

A *feasible schedule* of a SDF graph is a finite actor firing sequence, which when executed by the SDF graph, returns the edge buffer token states to the initial state. The

feasible schedule can be repeatedly executed at run time without any deadlock or buffer overflow. For example, the balance equation of the SDF graph in Fig. 1-(a) is:

$$\begin{cases} r_A * 2 = r_B * 3 \\ r_B * 1 = r_C * 2 \end{cases}$$

where r_A is the number of firings of actor `A`. Solving the balance equation yields the repetition vector $(r_A, r_B, r_C) = (3, 2, 1)$, which means that any minimum-length feasible schedule must consist of 6 actor firings in sequence, including 3 firings of actor `A`, 2 firings of actor `B` and 1 firing of actor `C`. Different feasible schedules may have different data buffer size requirements. In this paper, we assume that each edge has its own dedicated buffer space without any buffer sharing. Our objective is to find the minimum total buffer size requirement for a SDF graph to have at least one feasible schedule. Any buffer size less than that will cause the SDF graph to run into a deadlock for any possible schedule.

In order to use model-checking to find the minimum-buffer size schedule of a SDF graph, we transform the SDF graph into a Finite State Machine (FSM) encoding its execution semantics, e.g., the FSM in Fig. 1-(b) corresponds to the SDF graph in Fig. 1-(a). Each FSM transition models an actor firing and its effects on the input and output buffers of the actor. `ch[c]` denotes the current number of tokens on edge `c`, and `sz[c]` denotes the buffer size requirement of edge `c`, i.e., the maximum number of tokens on edge `c` throughout the entire schedule. The guard `WAIT(c,n)` encodes the condition that an actor can be invoked only if there are enough tokens on its input edge(s). The actions `PRODUCE(c,n)` (`CONSUME(c,n)`) encodes the semantics that each actor firing produces (consumes) a certain number of tokens on its output (input) edges. Our objective is to find a feasible schedule with minimum `SUM=sz[c1]+sz[c2]+...`, i.e., the sum of all edge buffer size requirements.

However, scalability is a key limiting factor in using model-checking due to state space explosion. In the following, we present several techniques for improving scalability of model-checking by exploiting problem-specific properties of SDF models.

2 Tight Edge Buffer Size Upper Bounds

It is desirable to obtain correct and tight *Buffer Size Upper Bound* (BSUB) value for each edge, since tight BSUB values help reduce the system state space as well as guarantee the minimal buffer requirement can be found. The following theorem tells that we can get a buffer size upper bound for each edge much tighter than the coarse upper bound in [2], and avoid [3]'s disadvantage of losing of optimality.

Theorem 1. *For a given SDF graph $G = (V, E)$, let s denote any feasible schedule s with buffer size requirement $R(s)$; let $R(s, e_i)$ denote the buffer requirement of edge e_i for schedule s; let s_{opt} denote an optimal schedule with minimum buffer size requirement $R(s_{opt})$. Then for each edge $e_i \in E$, we have:*

$$R(s_{opt}, e_i) \leq R(s) - \sum_{e_j \in E - e_i} BSLB(e_j) \qquad (1)$$

where $BSLB(e_j)$ is the buffer size lower bound of edge e_j obtained from Equation (2) in [3].

We also propose techniques for tighten the BSUB of so-called *Heavy Edges*.

Definition 1 (Heavy Edge) *An edge e_f is a* Forward Heavy Edge (FHE) *if it is the only input edge to its sink actor $snk(e_f)$, and*

$$c(e_f) > \sum_{src(e_j)=snk(e_f)} p(e_j) \tag{2}$$

An edge e_b is a Backward Heavy Edge (BHE) *if it is the only output edge from its source actor $src(e_b)$, and*

$$p(e_b) > \sum_{snk(e_j)=src(e_b)} c(e_j) \tag{3}$$

Theorem 2. *In any optimal feasible schedule s_{opt} of a SDF graph, any FHE e_f must satisfy*

$$R(s_{opt}, e_f) < \max(p(e_f) + c(e_f), d(e_f)) + c(e_f) \tag{4}$$

and any BHE e_b must satisfy

$$R(s_{opt}, e_b) < \max(p(e_b) + c(e_b), d(e_b)) + p(e_b) \tag{5}$$

3 Graph Decomposition

We can decompose the SDF graph into several subgraphs connected by bridges, analyze each subgraph separately, and then obtain the buffer size requirement of the whole SDF graph from that of each subgraph.

Theorem 3. *If $G = (V, E)$ is a SDF graph consisting of the set C of the minimal subgraphs connected by the set of bridges B.*

$$R(s_{opt}, G) = \sum_{G_i \in C} R(s^i_{opt}, G_i) + \sum_{e_i \in B} BSLB(e_i) \tag{6}$$

in which s_{opt} is an optimal schedule of G, and s^i_{opt} is an optimal schedule of G_i.

To use Theorem 3 to improve the efficiency of model-checking for the buffer requirement optimization of SDF G, we should find all subgraphs and bridges of the graph G, and compute the minimal buffer requirement of each subgraph separately, then compute the buffer requirement of S with Theorem 3. If a subgraph is cyclic, or it is acyclic with non-zero delay tokens, then we can use model-checking to derive its minimal buffer requirement and schedule. For acyclic and delayless subgraphs, we can use the algorithm in [4] to derive them analytically. Finding bridges of a graph can be done by the depth-first search with a complexity of $O(m + n)$ [5], where m is the number of actors and n is the number of edges.

Table 1. Performance evaluation.

No. Actors	4	6	8	10	12	14	16	18	20	22
Buff. Size.	32	36	56	64	188	168	222	430	324	471
Peak memory of NuSMV with the original approach in [3]										
`BOUND` (MB)	19.6	96.6	22.4	62.1	156.7	–	–	–	–	–
`BOUND+1` (MB)	19.6	86.9	22.5	62.9	223.6	–	–	–	–	–
Running Time of NuSMV with the original approach in [3]										
`BOUND` (s)	0.6	11.2	1.2	15.2	327.3	–	–	–	–	–
`BOUND+1` (s)	0.1	21.2	1.2	26.5	562.5	–	–	–	–	–
Peak memory of NuSMV with the optimized approach in this paper										
`BOUND` (MB)	2.5	26.1	13.6	25.8	36.6	17.2	21.3	12.6	16.7	56.8
`BOUND-1` (MB)	2.5	2.5	2.5	2.5	26.7	2.5	2.5	2.5	2.5	22.4
Running Time of NuSMV with the optimized approach in this paper										
`BOUND` (s)	0.1	0.1	0.2	0.1	0.6	0.3	0.4	0.2	0.3	2.2
`BOUND-1` (s)	0.1	0.1	0.1	0.1	0.3	0.1	0.1	0.1	0.1	0.4

4 Performance Evaluation

We used the software tool SDF^3 [6] to generate random SDF graphs. Since there exists fast and optimal algorithms [7] for deriving minimum buffer size requirements for acyclic and delayless SDF graphs, which makes it unnecessary to apply model-checking for these types of graphs, we ensured that any generated SDF graph contains cycles or initial delay tokens, or both. In our experiments, the minimum, maximum and average total input-output degree of each actor in the SDF graph are 1, 3 and 2, respectively. The experiments are run on a Linux PC with an Intel dual-core 2.83GHz 64-bit processor and 2GB of main memory. We use a utility program *Memtime* from the UPPAAL group to measure the running time and peak memory usage. Table 1 shows the performance evaluation results. We set a timeout limit of 2 hours. If a model-checking session did not finish within 2 hours, then we denote it with ”−” in the table. As shown in Table 1, the optimization techniques presented in this paper can result in significant reduction in both the memory size and running time of model-checking.

References

1. P. K. Murthy and S. S. Bhattacharyya, *Memory Management for Synthesis of DSP Software*. CRC Press, 2006.
2. M. Geilen, T. Basten, and S. Stuijk, “Minimising buffer requirements of synchronous dataflow graphs with model checking.” in *DAC*, 2005, pp. 819–824.
3. Z. Gu, M. Yuan, N. Guan, M. Lv, X. He, Q. Deng, , and G. Yu, “Static scheduling and software synthesis for dataflow graphs with symbolic model-checking,” in *RTSS*, 2007.
4. M. Cubric and P. Panangaden, “Minimal memory schedules for dataflow networks.” in *CONCUR*, 1993.
5. R. Tarjan, “Depth-first search and linear graph algorithms,” *SIAM Journal of Computer*, 1972.
6. S. Stuijk, M. Geilen, and T. Basten, “Sdf^3: Sdf for free.” in *ACSD*, 2006.
7. S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

Translating Interaction Nets to C

Abubakar Hassan, Ian Mackie, and Shinya Sato

[1] Department of Computer Science, University of Sussex, Falmer, Brighton, U.K.
[2] LIX, CNRS UMR 7161, École Polytechnique 91128 Palaiseau Cedex, France
[3] Faculty of Econoinformatics, Himeji Dokkyo University
5-7-1 Kamiohno, Himeji-shi, Hyogo 670-8524, Japan

Abstract. This paper is about a new implementation technique for interaction nets–a visual programming language based on graph rewriting. We compile interaction nets to C, which offers a robust and efficient implementation, in addition to portability. In the presentation of this work we extend the interaction net programming paradigm to introduce a number of features which make it a practical programming language.

1 Introduction

Interaction nets [6] are a graph rewrite system where programs are represented as graphs and computation is graph rewriting. They enjoy nice properties such as strong confluence, Turing completeness and locality of reduction. For these reasons, optimal [3, 7] and efficient [9] $\lambda-$calculus evaluators based on interaction nets have evolved. Indeed, interaction nets have proved to be very fruitful in the study of the dynamics of computation. However, they still remain useful only for theoretical investigations.

In this note, we take a step towards developing a practical language for interaction nets which allows them to be used in practice. In the same way that functional languages are based on the λ-calculus, logic languages based on horn clauses or the pict [10] language based on the π-calculus, here we present a language based on this very simple graph rewrite system.

There are several implementations of interaction nets [11, 8, 4, 5], but they all suffer at least from one or more drawbacks: execution speed, lack of modern language constructs such as built-in types, input/output etc. The goal of this paper is to address these issues so that we can shift the use of interaction nets from theoretical investigations to a practical programing paradigm. Firstly, we develop a textual syntax for interaction nets with higher level constructs that provide programing comfort. We then show how this language can be compiled down to native codes via the language C [12]. C is a machine independent low-level language that is well suited as a portable target language for the implementation of programming languages. Over the years, C compilers have gone through many improvements to generate optimised machine code. By compiling to C, we also benefit in the improvements of C code generation. In addition, we gain instant portability because C is implemented on a variety of platforms. Many languages [14, 1, 13] have benefited from this line of compilation.

To summarise, the main contributions of this paper are as follows: We extend the definition of interaction nets to allow: built in data types and conditional rewrite rules; states and state transformers; and we define a compiler from interaction nets to native codes via the language C.

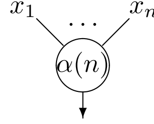
In our previous work [4] we defined a textual language for interaction nets and described how it is transformed into an intermediate language *pin*. We then defined an abstract machine that executes pin instructions. This paper is concerned with: 1) developing a richer language for interaction nets 2) extending the pin language to cater for the introduced source language constructs and 3) compiling pin into C code.

For the rest of this abstract, we just introduce the formalism, and give an example.

2 Interaction nets

Here we review the basic notions of interaction nets. We refer the reader to [6] for a more detailed presentation. Interaction nets are specified by the following,

A set Σ of *symbols.* Elements of Σ serve as *agent* (node) labels. Each symbol has an associated arity ar that determines the number of its *auxiliary ports.* If $ar(\alpha) = n$ for $\alpha \in \Sigma$, then α has $n+1$ *ports:* n auxiliary ports and a distinguished one called the *principal port.* Each agent may have attributes. In this paper, we will restrict attributes to just base types: integers and booleans, and we write the attribute in brackets after the name.



We can represent this agent textually as $x_0 \sim \alpha(n)[x_1, \ldots, x_n]$, where x_0 is the principal port.

A *net* built on Σ is an undirected graph with agents at the vertices. The edges of the net connect agents together at the ports such that there is only one edge at every port. A port which is not connected is called a *free port.* A set of free ports is called an *interface.*

Two agents $(\alpha, \beta) \in \Sigma \times \Sigma$ connected via their principal ports form an *active pair* (analogous to a redex). An interaction rule $((\alpha, \beta) \Longrightarrow N) \in \mathcal{R}$ replaces the pair (α, β) by the net N. All the free ports are preserved during reduction, and there is at most one rule for each pair of agents. The following diagram illustrates the idea, where N is any net built from Σ.

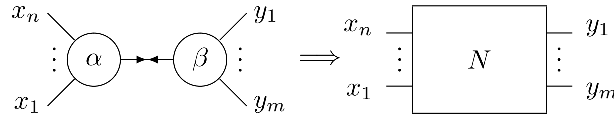


Figure 1 gives a simple example of an interaction net system that encodes the addition operation. We represent numbers using the agents S to represent the successor function $(n \mapsto n+1)$ and Z to represent the number 0. Figure 2 gives an example reduction sequence that shows how a net representing $1+1$ is reduced to 2.

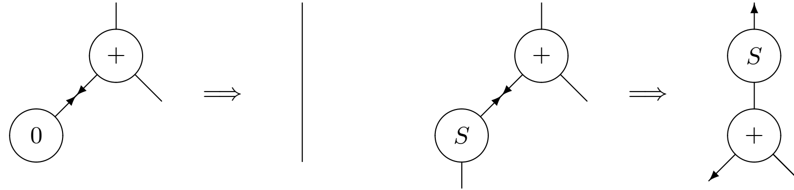


Fig. 1. Rules for addition

3 The source language and compilation

Following [2], an interaction net system can be described as a configuration $c = (\Sigma, \Delta, \mathcal{R})$, where Σ is a set of symbols, Δ is a multiset of active pairs, and $\mathcal{R}$ is a set of rules. A language for interaction nets needs to capture each component of the configuration, and provide ways to structure and organise the components. Starting from a calculus for interaction nets we build a core language. A core language can be seen both as a programming language and as a target language where we can compile high-level constructs. Drawing an analogy with functional programming, we can write programs in the pure λ-calculus and can also use it as a target language to map high-level constructs. In this way, complex high-level languages can be obtained which by their definition automatically get a formal semantics based on the core language.

We write nets textually as a comma separated list of agents. This just corresponds to a flattening of the net, and there are many different (equivalent) ways to do this depending on the order the agents are enumerated. As an example, we write the initial net in Figure 2 as:

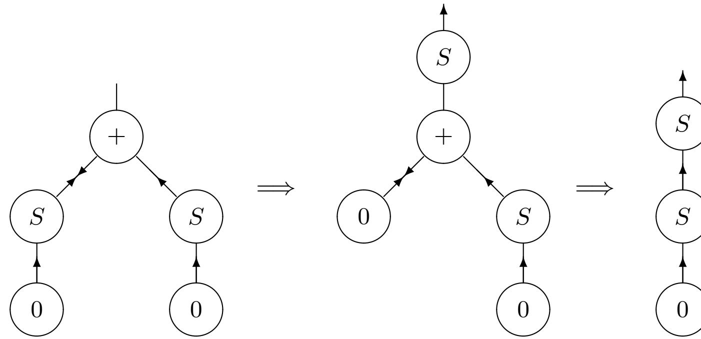


Fig. 2. Example reduction sequence

```
a∼Add[x,y], a∼S[b], b∼Z[], y∼S[v], v∼Z[].
```

This can be simplified by replacing equals for equals:

```
S[Z[]]∼Add[x,S[Z[]]].
```

In this notation the general form of an active pair is $\alpha[\ldots] \sim \beta[\ldots]$. From now on, we shall omit the brackets `[]` when the arity of an agent is 0. We give a more complete example below, which introduces many of the novelties of this paper.

```
int counter;
Fact[result] ><
 {
    Num(int x)[] =>
      counter = counter + 1;
     if(x < 0)
       result∼Error[];
     else  if(x == 0)
       result∼Num(1)[];
     else
       Fact[Mult(x)[result]]∼Num(x-1)[] ;
 }
Mult(int x)[res] ><
 {
    Num(int y)[] =>
      counter = counter + 1;
      res∼Num(x*y)[];
}
main(){
  counter = 0;
  Fact[result]∼Num(6)[];
  print ‘‘total number of interactions ’’,counter;
}
```

We conclude this short abstract by showing the structure of part of the code generated for the main net given in the example program given above. In the full paper we give the formal translation of the language into C.

```
void MAIN(){
  counter = 0;
  Agent Fact = mkAgent("Fact",1);   Agent result = mkVar("result");
  connect(Fact, 1, result, 1)
  Agent Num = mkAgent("Num",0);     Num->value[1] = 6;
```

```
 connect(Fact,0,Num,0);  eval(Fact,Num);
 printf("total number of interactions");  printf(counter);
}

int main(){   MAIN(); }
```

References

[1] Daniel Diaz. Wamcc: Compiling prolog to c. In *In 12th International Conference on Logic Programming*, pages 317–331. MIT PRess, 1995.

[2] Maribel Fernández and Ian Mackie. A calculus for interaction nets. In G. Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, number 1702 in LNCS, pages 170–187. Springer-Verlag, September 1999.

[3] Georges Gonthier, Martín Abadi, and Jean-Jacques Lévy. The geometry of optimal lambda reduction. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL'92)*, pages 15–26. ACM Press, January 1992.

[4] Abubakar Hassan, Ian Mackie, and Shinya Sato. Interaction nets: programming language design and implementation. In *Proceedings of the seventh international workshop on Graph Transformation and Visual Modeling Techniques*, March 2008.

[5] M.Vilaca J.B. Almeida, J.S.Pinto. A tool for programming with interaction nets. Technical report, University of Minho, 2006.

[6] Yves Lafont. Interaction nets. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 95–108. ACM Press, January 1990.

[7] John Lamping. An algorithm for optimal lambda calculus reduction. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 16–30. ACM Press, January 1990.

[8] S. Lippi. in^2 : A graphical interpreter for interaction nets. In Sophie Tison, editor, *Rewriting Techniques and Applications (RTA'02)*, volume 2378 of *Lecture Notes in Computer Science*, pages 380–386. Springer, 2002.

[9] Ian Mackie. YALE: Yet another lambda evaluator based on interaction nets. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 117–128. ACM Press, September 1998.

[10] Benjamin C. Pierce and David N. Turner. Pict: a programming language based on the pi-calculus. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction*, pages 455–494. The MIT Press, 2000.

[11] Jorge Sousa Pinto. Parallel evaluation of interaction nets with mpine. In *RTA*, pages 353–356, 2001.

[12] Dennis M. Ritchie. The c programming language, 1988.

[13] David Tarditi, Peter Lee, and Anurag Acharya. No assembly required: Compiling standard ml to c. Technical report, ACM Letters on Programming Languages and Systems, 1990.

[14] Geo Wong. Compiling erlang via c. Technical report, 1998.

A Type System for Usage of Software Components. Extended Abstract*

Dag Hovland

Department of Informatics, The University of Bergen,
PB 7803, N-5020 Bergen, Norway
dagh@ii.uib.no

Abstract. The aim of this article is to support component-based software engineering by modelling exclusive and inclusive usage of software components. Truong and Bezem describe in several papers abstract languages for component software with the aim to estimate bounds for the number of instances of components. Their language includes primitives for instantiating and deleting instances of components and operators for sequential, alternative and parallel composition and a scope mechanism. The language is here supplemented with the primitives `use`, for inclusive usage, and `lock` and `free` for exclusive usage. The main contribution is a type system which guarantees the safety of usage, in the following way: When a well-typed program executes a subexpression `use`$[x]$ or `lock`$[x]$, it is guaranteed that an instance of x is available.

1 Introduction

The idea of "Mass produced software components" was first formulated by McIlroy [2] in an attempt to encourage the production of software routines in much the same way industry manufactures ordinary, tangible products. The last two decades "component" has got the more general meaning of a highly reusable piece of software. According to Szyperski [4] (p. 3), "(. . .) software components are executable units of independent production, acquisition, and deployment that can be composed into a functioning system". We will model software that is constructed from such components, and assume that during the execution of such a program, instances of the components can be created, used and deleted.

Efficient component software engineering is not compatible with programmers having to acquire detailed knowledge of the internal structure of components that are being used. Components can also be constructed to use other components, such that instantiating one component, could lead to several instances of other components. This lack of knowledge in combination with nested dependencies weakens the control over resource usage in the composed software.

The goal of this article is to guarantee the safe usage of components, in the following two ways: when `use`$[x]$ is executed there is an instance of x available, and when `lock`$[x]$ is executed, there is an instance which is exclusively available

* This research was supported by the Research Council of Norway.

for the current thread of execution. In [1,5,6], Truong and Bezem describe abstract languages for component software with the aim of estimating the number of instances of components existing during and remaining after execution of a component program. Their languages include primitives for instantiating and deleting instances of components and have operators for sequential, alternative and parallel composition and a scope mechanism. The first three operators are well-known, and have been treated by for example Milner [3] (where alternative composition is called *summation*). The scope mechanism works like this: Any component instantiated in a scope has a lifetime limited to the scope. Furthermore, from inside a scope, only instances in the local store of the same scope can be deleted. The types count the maximum number of active component instances during execution and remaining after execution of a component program.

These languages lack a direct way of specifying that one or more instances of a component must exist at some point in the execution. In this paper we have added the primitives `use`, `lock` and `free` in order to study the usage of components. The first (`use`) is used for "inclusive usage", that is, when a set of instances must be available, but these instances may be shared. The other form (`lock` and `free`) is used when the instances must exclusively be available for this execution thread. The difference between exclusive and inclusive usage can be seen by comparing the expressions $\texttt{new}\,x(\texttt{use}\,[x] \parallel \texttt{use}\,[x])$ and $\texttt{new}\,x(\texttt{lock}\,[x]\texttt{free}\,[x] \parallel \texttt{use}\,[x])$. The first expression is safe to execute, while executing the latter expression can lead to an error if x is locked, but not freed, by the left thread before it is used by the right thread.

2 Language

An *expression* in the language is defined inductively as a list where the elements are of the form $\texttt{new}\,x, \texttt{del}\,x, \texttt{lock}\,M, \texttt{free}\,M, \texttt{use}\,M, \texttt{nop}, (\mathit{Expr} + \mathit{Expr}), (\mathit{Expr} \parallel \mathit{Expr})$ or $\{M, \mathit{Expr}\}$, where M is a bag of component names, x is a component name, and *Expr* is an expression. Furthermore, $+$ denotes alternative composition, $\parallel$ denotes parallel composition, $\{\}$ denotes a local scope and `nop` means "no operation". A *component program* P is a comma-separated list starting with `nil` and followed by zero or more *component declarations*, which are of the form $x \prec \mathit{Expr}$.

3 Examples

We assume that a program is executed by executing $\texttt{new}\,x$, where x is the last component declared in the program, starting with empty stores of component instances. Examples of programs that will execute properly and will be well-typed are

$$x \prec \texttt{nop}, y \prec \texttt{new}\,x\,\texttt{use}\,[x]\,\texttt{lock}\,[x]\,\texttt{free}\,[x]$$
$$x \prec \texttt{nop}, y \prec \texttt{new}\,x\,\texttt{new}\,x\,\{[\,], (\texttt{use}\,[x] \parallel \texttt{lock}\,[x])\}\,\texttt{free}\,[x]$$

Examples of programs that can, for some reason, produce an error are:

$$x \multimap \texttt{nop}, y \multimap \texttt{new}\,x\,\texttt{new}\,x\,\{[\,], (\texttt{use}\,[x] \parallel \texttt{lock}\,[x])\}$$
$$x \multimap \texttt{nop}, y \multimap \texttt{new}\,x\,\texttt{lock}\,[x]\,\texttt{use}\,[x]\,\texttt{free}\,[x]$$
$$x \multimap \texttt{nop}, y \multimap \texttt{new}\,x\,\{[\,], (\texttt{use}\,[x] \parallel \texttt{lock}\,[x])\}\,\texttt{free}\,[x]$$
$$x \multimap \texttt{nop}, y \multimap \texttt{new}\,x\,\texttt{free}\,[x]\,\texttt{lock}\,[x]$$
$$x \multimap \texttt{nop}, y \multimap \texttt{new}\,x\,\{[\,], (\texttt{use}\,[x] + \texttt{lock}\,[x])\}\,\texttt{free}\,[x]$$

The first program leaves one instance of x locked after execution. The second will get stuck as no x will be available for use by the `use`-statement. The third might also get stuck. Note that there exists an error-free execution of the third program, where the left branch of $(\texttt{use}\,[x] \parallel \texttt{lock}\,[x])$ is executed before the right one. But as we do not wish to make any assumptions about the scheduling of the parallel execution, we consider this an error. The fourth program tries to free a component instance that is not locked. The fifth program can be run such that when $\texttt{free}\,[x]$ is executed no instance of x has been locked.

4 Results

The execution of a program is formalized in a small-step operational semantics. The rules operate on *states*, pairs consisting of an expression in the language and of a *global store* of component instances. Further a type system is presented, which can assign types to expressions and programs. The type of an expression is a 6-tuple of multisets over component names. The type of a program is a (partial) mapping from component names to expression types. The main lemmas, type preservation, progress and termination are formulated and proved in terms of the operational semantics and the type system. The main result is a theorem stating that if a program is well-typed in the system, then all execution traces of the program are finite and safe.

References

1. Marc Bezem and Hoang Truong. A type system for the safe instantiation of components. *Electronic Notes in Theoretical Computer Science*, 97:197–217, 2004.
2. Malcolm Douglas McIlroy. Mass produced software components. In P. Naur and B. Randell, editors, *Software Engineering: Report of a conference sponsored by the NATO Science Committee*, pages 79–87. Scientific Affairs Division, NATO, October 1968.
3. Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.
4. Clemens Szyperski. *Component Software—Beyond Object–Oriented Programming.* Addison–Wesley / ACM Press, 2nd edition, 2002.
5. Hoang Truong. Guaranteeing resource bounds for component software. In Martin Steffen and Gianluigi Zavattaro, editors, *FMOODS*, volume 3535 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2005.
6. Hoang Truong and Marc Bezem. Finding resource bounds in the presence of explicit deallocation. In Dang Van Hung and Martin Wirsing, editors, *Proceedings ICTAC*, volume 3722 of *Lecture Notes in Computer Science*, pages 227–241. Springer, 2005.

Secrecy in Mobile Ad-hoc Networks

Hans Hüttel[1] and Willard Thór Rafnsson[2]

[1] Department of Computer Science, Aalborg University, Denmark
hans@cs.aau.dk
[2] School of Computer Science, Reykjavík University, Iceland
willard@ru.is

We propose *a framework for automated verification of secrecy properties of MANET protocols*, consisting of a *formal language* and a proven sound *verification technique* which can be automated. We start off by presenting the distributed applied pi calculus with broadcast (DAπ_β), whereafter we summarise our procedure for generating Horn clauses from a DAπ_β model expressing control flow in the model. We then present our soundness result, from which it follows that the generated Horn clauses can be used to reason about secrecy in the source model, in an automated manner.

1 The Calculus DAπ_β

Our calculus is an extension to the calculus of Abadi and Blanchet [1] with arbitrary term rewrite systems, with an extended process layer as in the applied pi calculus [3], with a network layer similar to that of the distributed pi calculus [6], and with connectivity graphs akin to those of CBS$^\sharp$ [4].

Let $\mathcal{A}$, $\mathcal{X}$, and $\mathcal{U} = \mathcal{A} \cup \mathcal{X}$ be the infinite sets of names, variables, and identifiers, ranged by $a, b, c, k, \ldots, t \in \mathcal{A}$, $a, b, c, k, \ldots, t \in \mathcal{X}$, and $u, v, w \in \mathcal{U}$, and let $\mathcal{A}$ and $\mathcal{X}$ be disjoint. Let $\mathcal{F}$ and $\mathcal{G}$ be the sets of constructors and destructors, ranged over by f and g, respectively. The syntax of DAπ_β networks is given by the following grammar specification.

$$
\begin{aligned}
T &::= u \mid \mathsf{f}(T, \ldots, T) \\
P &::= \mathbf{0} \mid u(x) \,.\, P \mid \overline{u}\langle T \rangle \,.\, P \mid\, !P \mid \nu a \,.\, P \mid P \parallel P \mid \mathbf{let}\ x = \mathsf{g}(\tilde{T})\ \mathbf{in}\ P\ \mathbf{else}\ P \\
A &::= A \parallel A \mid \nu u \,.\, A \mid P \mid \{^T/_x\} \\
N &::= N \parallel N \mid \nu u \,.\, N \mid \mathbf{0} \mid l[A] \mid \{^T/_x\}
\end{aligned}
$$

A DAπ_β network is a parallel composition of sequential message-passing processes residing at a location, which pass terms as values. *Terms* $(T, U, V \in \mathcal{T})$ can either be an identifier, or a construction of terms. For instance, suppose $\mathsf{enc}, \mathsf{pair} \in \mathcal{F}$, both of arity 2. Then $\mathsf{enc}(\mathsf{pair}(x, y), k)$ is a term. *Primitive Processes* $(P, Q, R \in \mathcal{P})$ are identical to the processes of the pi calculus, except for the presence of variables, and the "let" operator; $\mathbf{let}\ x = \mathsf{g}(\tilde{T})\ \mathbf{in}\ P\ \mathbf{else}\ Q$ binds some (nondeterministically chosen) T to x in P, if a T exists such that $\mathsf{g}(\tilde{T})$ reduces to T, written $\mathsf{g}(\tilde{T}) > T$. If no such T exists, then the process proceeds as Q. A classic rewrite rule is that of synchronous key cryptography; for $\mathsf{dec} \in \mathcal{G}$,

we let def(dec), the set of rewrite rules for dec, be $\{\mathsf{dec}(\mathsf{enc}(z_1, z_2), z_2) > z_1\}$. Here, $\mathsf{dec}(\mathsf{enc}(\mathsf{pair}(x, y), k), k) > \mathsf{pair}(x, y)$ holds. As usual, perfect encryption is assumed. *Extended Processes ($A, B \in \mathcal{B}$)* are exactly like in [3]. *Networks ($M, N, O \in \mathcal{N}$)* are like the syntactic category A, with the addition of $l[A]$, representing A residing at location l. We assume in every N that each bound location name is pairwise syntactically different from any other location name.

We adopt the notion of connectivity graphs from [4], with some adjustments. A directed graph $G = (V, E)$, where V and E are the vertices and edges of G, is a *connectivity graph* if V is a finite set, $V \subseteq \mathcal{A}$, and $\forall l \in V \,.\, (l, l) \in E$ holds[3]. G is *admissible* on N if V contains all locations in N. A *network topology* τ is a set of connectivity graphs. τ is *admissible* to a network N if each graph in τ is.

The semantics of DAπ_β is defined in terms of structural equivalence $\equiv$, internal reduction $\rightarrow_G$, and labelled reduction $\rightarrow_G^{(l,\alpha)}$, which are derived from [3], and extended to take into account τ and the source location of each message. $\rightarrow_G$ is the least preorder on $\mathcal{N}$ closed by $\equiv$ and evaluation contexts, satisfying

$$\prod_{i\in I} l_i[a(x) \,.\, P_i] \parallel l[\overline{a}\langle x\rangle \,.\, P] \rightarrow_G \prod_{i\in I} l_i[P_i] \parallel l[P], \text{ if } \forall i \in I \exists (l, l_i) \in \mathsf{E}(G)$$

$$l[\textbf{let } x = \mathsf{g}(\tilde{T}) \textbf{ in } P \textbf{ else } Q] \rightarrow_G l[\nu x \,.\, (\{^T/_x\} \parallel P)], \text{ if } \mathsf{g}(\tilde{T}) > T$$

$$l[\textbf{let } x = \mathsf{g}(\tilde{T}) \textbf{ in } P \textbf{ else } Q] \rightarrow_G l[Q], \text{ if } \neg\exists T \,.\, \mathsf{g}(\tilde{T}) > T$$

$$l[!P] \rightarrow_G l[P \parallel !P].$$

Example 1 (Unauthenticated route). A common interest in MANET protocol analysis is checking the possibility of routes, particularly unauthenticated ones, such as in the ARAN protocol [5]. Let l_{att} be an unauthenticated location in

$$N \stackrel{\text{def}}{=} l_s[\nu a_{\text{secret}} \,.\, \overline{c}\langle a_{\text{secret}}\rangle \,.\, P_s] \parallel l_r[c(x) \,.\, \nu a_{\text{oops}} \,.\, \overline{c}\langle \mathsf{pair}(x, a_{\text{oops}})\rangle \,.\, P_r].$$

We construct a network topology $\tau = \{G_1, G_2\}$ that keeps l_s and l_r separate.

G_1: (l_s) $(l_{\text{att}}) \leftrightarrows (l_r)$, G_2: $(l_s) \leftrightarrows (l_{\text{att}}) \leftrightarrows (l_r)$

If N reveals $\mathsf{pair}(a_{\text{secret}}, a_{\text{oops}}) =: m$, an unauthenticated route has been established. Here, $N \rightarrow_{G_1}$, the resulting network after N performs reduction $\rightarrow_{G_1}$, is not in a position to reveal m to the environment, while $N \rightarrow_{G_2}^{(l_s, \overline{c}\langle a_{\text{secret}}\rangle)}$ can be, as l_{att} is in a position to route a_{secret} to l_r.

2 Analysis

We have developed a Horn clause generation procedure expressing control flow in our networks, which bears close resemblance to that of [1, 2]. For our example above, the set $\mathsf{H}(N, \tau)$ of clauses generated from N and τ would include

$$\mathsf{msg_{out}}(x, y, z) \wedge \mathsf{connected}(z, z') \implies \mathsf{msg_{in}}(x, y, z'), \quad \mathsf{msg_{out}}(c, a_{\text{secret}}, l_s)$$
$$\mathsf{msg_{in}}(c, x, l_r) \implies \mathsf{msg_{out}}(c, \mathsf{pair}(x, a_{\text{oops}}), l_r), \quad \mathsf{connected}(l_s, l_{att}),$$

[3] We usually make this last detail implicit in our graph specifications.

$$\mathsf{att}(x) \wedge \mathsf{att}(y) \wedge \mathsf{att}(z) \implies \mathsf{msg}_{\mathsf{out}}(x, y, z), \quad \mathsf{connected}(l_{att}, l_r),$$
$$\mathsf{msg}_{\mathsf{out}}(x, y, z) \wedge \mathsf{connected}(z, z') \wedge \mathsf{att}(x) \wedge \mathsf{att}(z') \implies \mathsf{att}(y).$$

We then get an affirmative result when performing SLD resolution on $\mathsf{H}(\tau)$ with the goal clause $\mathsf{msg}_{\mathsf{out}}(c, \mathsf{pair}(a_{\text{secret}}, a_{\text{oops}}), l_r)$. We denote this *deduction* by $\mathsf{H}(N, \tau) \vdash \mathsf{msg}_{\mathsf{out}}(c, \mathsf{pair}(a_{\text{secret}}, a_{\text{oops}}), l_r)$, which expresses that *in topology* τ, N will output $\mathsf{pair}(a_{\text{secret}}, a_{\text{oops}})$, which indicates the existence of a false route.

By making syntactic behaviour-preserving transformations on a network specification, we obtain, for any N and τ admissible to N, that

Theorem 1. *There is a* syntactic bijective relationship *between process prefixes of* N *ending in an output, and* $cl \in \mathsf{H}(N, \tau)$ *with* $\mathsf{msg}_{\mathsf{out}}$ *in their conclusion.*

We use this result to derive two key results in our framework.

Theorem 2. *i)* $\vdash$ *is sound.* *ii) If* $\mathsf{H}(N, \tau) \nvdash \mathsf{att}(s)$*, then* s *is secret in* N*.*

Thus, deducing facts from $\mathsf{H}(N, \tau)$ is sound; while deduction can yield false positives regarding control flow due to overapproximation, false negatives never arise. Also, $\mathsf{H}(N, \tau)$ can be used to reason about secrecy in N. A useful feature is the "pluggable" nature of τ; we are capable of proving that some protocols are sound when mobility is constrained, as is the case of the ARAN protocol.

At last, a central consequence of Theorem 1 is that $\vdash$ can easily be implemented in ProVerif [7] and the Succinct Solver [8]. As such, these tools can reason soundly about the behaviour of *broadcasting processes*, which to our knowledge has not been demonstrated before.

References

1. Martín Abadi and Bruno Blanchet: Analyzing Security Protocols with Secrecy Types and Logic Programs. Journal of the ACM 52, pp. 102–146 (2005)
2. Bruno Blanchet. From Secrecy to Authenticity in Security Protocols. In: SAS '02: Proceedings of the 9th International Symposium on Static Analysis, pp. 342–359, Springer-Verlag, London, UK (2002)
3. Martín Abadi, Cédric Fournet: Mobile Values, New Names, and Secure Communication. In: 28th ACM Symposium on POPL 2001, pp. 104–115, (2001)
4. Sebastian Nanz, Chris Hankin: A Framework for Security Analysis of Mobile Wireless Networks. Electronic Notes in Theor. Comp. Sci., 367:207–227 (2006)
5. Jens Chr. Godskesen: Formal Verication of the ARAN Protocol Using the Applied Pi-calculus. In Proceedings of the Sixth International IFIP WG 1.7 Workshop on Issues in the Theory of Security, 99–113 (2006)
6. Matthew Hennessy, James Riely: Resource Access Control in Systems of Mobile Agents. In: Proceedings of HLCL'98, volume 16(3) of ENTCS. Elsevier (1998)
7. Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In: 14th IEEE Computer Security Foundations Workshop (CSFW14), pp. 82–96, IEEE Computer Society, Cape Breton, Nova Scotia, Canada. (2001)
8. F. Nielson, H. Riis Nielson, H. Sun, M. Buchholtz, R. R. Hansen, H. Pilegaard, and H. Seidl: The Succinct Solver Suite. In: Proc. TACAS'04, volume 2988 of Lecture Notes in Computer Science, pp. 251–265. Springer-Verlag (2004)

On building a Supercompiler for GHC

Peter A. Jonsson Johan Nordlander

Luleå University of Technology

{pj, nordland}@csee.ltu.se

Abstract

Supercompilation is a program transformation that removes intermediate structures and performs program specialization. We discuss problems and necessary steps for building a supercompiler for GHC.

1. Introduction

Supero [4] is a supercompiler for Haskell that has achieved runtime improvements of 16% for a subset of the imaginary part of the nofib suite compared to GHC. This is a remarkable result considering that it is doing a single program transformation beyond those of GHC and that GHC is a mature optimizing Haskell compiler. Not only is Supero showing great results, but the theoretical underpinnings of supercompilation have been investigated quite thoroughly as well. We know that it is possible to supercompile both strict [2] and lazy [9] languages, that the algorithms preserve the semantics of the program [7] and that the algorithms will terminate [8].

If the theory of supercompilers is well investigated, and runtime improvements are impressive – why does not every compiler include the optimization? Mitchell and Runciman [4] list three main areas in the future work for Supero:

Runtime performance For certain benchmarks earlier versions of Supero had better results. This is both evidence that there are improvements to be made, and it also highlights how small changes to the algorithm can have large effects on the result of the optimization.

Compilation speed Profiling Supero showed that 90% of the time was spent to ensure termination (the homeomorphic embedding test), which was done in a naïve way.

More benchmarks Supero was applied to a subset of the imaginary part of nofib. No one has ever benchmarked supercompiled real world Haskell programs.

We set out to build a supercompiler for GHC in order to tackle these three problems (Sections 3 and 4). Using GHC gives the additional benefit of support for many Haskell extensions for free, since they are all translated to System F_c, GHC's typed intermediate language [10]. We start with some examples of a supercompiler in action to try to convey the intuition behind the algorithm (Section 2).

2. Examples

We have left out many of the technical details of the algorithm due to space constraints and instead try to convey the intuition via a series of examples of how a supercompiler behaves. Readers who are interested in the gory details are encouraged to read some of the related work already cited, in particular Sørensen et al. [9] for a call-by-name algorithm, the work on Supero by Mitchell and Runciman [4], or Jonsson and Nordlander [2] for a call-by-value algorithm.

Supercompilation is a program transformation, closely related to deforestation [11], that both removes intermediate structures and performs program specialization. Removing intermediate structures will make the program allocate less memory and thus put less strain on the garbage collector. The program specialization will in practice remove many of the higher order functions, replacing them with a specialized first order variant. Higher order languages usually represent functions as closures on the heap, and these closures need to be garbage collected. Having removed them at compile-time reduces the amount of garbage collection necessary, and also avoids an indirect jump to the function pointer in the closure at runtime.

Our first example is transformation of $sum\ (map\ square\ ys)$. The functions used in the examples are defined as:

$$
\begin{array}{lll}
square\ x & = & x * x \\
map\ f\ xs & = & \mathbf{case}\ xs\ \mathbf{of} \\
 & & \quad [] \rightarrow ys \\
 & & \quad (x : xs) \rightarrow f\ x : map\ f\ xs \\
sum\ xs & = & \mathbf{case}\ xs\ \mathbf{of} \\
 & & \quad [] \rightarrow 0 \\
 & & \quad (x : xs) \rightarrow x + sum\ xs
\end{array}
$$

We start our transformation by allocating a new fresh function name (h_0) to this expression, inlining the body of sum and substituting $map\ square\ ys$ into the body of sum:

$$
\begin{array}{l}
\mathbf{case}\ map\ square\ ys\ \mathbf{of} \\
\quad [] \rightarrow 0 \\
\quad (x' : xs') \rightarrow x' + sum\ xs'
\end{array}
$$

After inlining map and substituting the arguments into the body the result becomes:

$$
\begin{array}{l}
\mathbf{case}\ (\ \mathbf{case}\ ys\ \mathbf{of} \\
\qquad\qquad [] \rightarrow [] \\
\qquad\qquad (x' : xs') \rightarrow (square\ x') : map\ square\ xs')\ \mathbf{of} \\
\quad [] \rightarrow 0 \\
\quad (x' : xs') \rightarrow x' + sum\ xs'
\end{array}
$$

We duplicate the outer case in each of the inner case's branches, using the expression in the branches as head of that case-statement. Continuing the transformation on each branch with ordinary reduction steps yields:

$$
\begin{array}{l}
\textbf{case}\ ys\ \textbf{of} \\
\quad [\,] \rightarrow 0 \\
\quad (x' : xs') \rightarrow \mathit{square}\ x' + \mathit{sum}\ (\mathit{map}\ \mathit{square}\ xs')
\end{array}
$$

Now inline the body of the first square and observe that the second argument to $(+)$ is similar to the expression we started with. We replace the second parameter to $(+)$ with $h_0\ xs'$. The result of our transformation is $h_0\ ys$, with h_0 defined as:

$$
\begin{array}{l}
h_0\ ys = \textbf{case}\ ys\ \textbf{of} \\
\qquad\qquad [\,] \rightarrow 0 \\
\qquad\qquad (x' : xs') \rightarrow x' * x' + h_0\ xs'
\end{array}
$$

This new function only traverses its input once, and no intermediate structures are created. If the expression $\mathit{sum}\ (\mathit{map}\ \mathit{square}\ xs)$ or a renaming thereof is detected elsewhere in the input, a call to h_0 will be inserted there instead.

The following examples are due to Ohori and Sasano [5]:

$$
\begin{array}{ll}
\mathit{mapsq}\ xs & = \textbf{case}\ xs\ \textbf{of} \\
 & \qquad [\,] \rightarrow [\,] \\
 & \qquad (x' : xs') \rightarrow (x' * x') : \mathit{mapsq}\ xs' \\
f\ xs & = \textbf{case}\ xs\ \textbf{of} \\
 & \qquad [\,] \rightarrow [\,] \\
 & \qquad (x' : xs') \rightarrow (2 * x') : g\ xs' \\
g\ xs & = \textbf{case}\ xs\ \textbf{of} \\
 & \qquad [\,] \rightarrow [\,] \\
 & \qquad (x' : xs') \rightarrow (3 * x') : f\ xs'
\end{array}
$$

Transforming $\mathit{mapsq}\ (\mathit{mapsq}\ xs)$ will inline the outer mapsq, substitute the argument in the function body and inline the inner call to mapsq:

$$
\begin{array}{l}
\textbf{case}\ (\ \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad [\,] \rightarrow [\,] \\
\qquad\qquad (x' : xs') \rightarrow (x' * x') : \mathit{mapsq}\ xs')\ \textbf{of} \\
\quad [\,] \rightarrow [\,] \\
\quad (x' : xs') \rightarrow (x' * x') : \mathit{mapsq}\ xs'
\end{array}
$$

As previously, we duplicate the outer case in each of the inner case's branches, using the expression in the branches as head of that case-statement. Continuing the transformation on each branch by ordinary reduction steps yields:

$$
\begin{array}{l}
\textbf{case}\ xs\ \textbf{of} \\
\quad [\,] \rightarrow [\,] \\
\quad (x' : xs') \rightarrow (x' * x' * x' * x') : \mathit{mapsq}\ (\mathit{mapsq}\ xs')
\end{array}
$$

This will encounter a similar expression to what we started with, and create a new function h_1. The final result of our transformation is $h_1\ xs$, with the new residual function h_1 that only traverses its input once defined as:

$$
\begin{array}{l}
h_1\ xs = \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad [\,] \rightarrow [\,] \\
\qquad\qquad (x' : xs') \rightarrow (x' * x' * x' * x') : h_1\ xs'
\end{array}
$$

For an example of transforming mutually recursive functions, consider the transformation of $\mathit{sum}\ (f\ xs)$. Inlining the body of sum, substituting its arguments in the function body and inlining the body of f yields:

$$
\begin{array}{l}
\textbf{case}\ (\ \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad [\,] \rightarrow [\,] \\
\qquad\qquad (x' : xs') \rightarrow (2 * x') : g\ xs')\ \textbf{of} \\
\quad [\,] \rightarrow 0 \\
\quad (x' : xs') \rightarrow x' + \mathit{sum}\ xs'
\end{array}
$$

We now move down the outer case into each branch, and perform reductions until we end up with:

$$
\textbf{case}\ xs\ \textbf{of}\ \{[\,] \rightarrow 0;\ (x' : xs') \rightarrow (2 * x') + \mathit{sum}\ (g\ xs')\ \}
$$

We notice that unlike in previous examples, $\mathit{sum}\ (g\ xs')$ is not similar to the expression we started with. For space reasons, we focus on the transformation of the rightmost expression in the last branch, $\mathit{sum}\ (g\ xs')$, while keeping the functions already seen in mind. We inline the body of sum, perform the substitution of its arguments and inline the body of g:

$$
\begin{array}{l}
\textbf{case}\ (\ \textbf{case}\ xs'\ \textbf{of} \\
\qquad\qquad [\,] \rightarrow [\,] \\
\qquad\qquad (x'' : xs'') \rightarrow (3 * x'') : f\ xs'')\ \textbf{of} \\
\quad [\,] \rightarrow 0 \\
\quad (x' : xs') \rightarrow x' + \mathit{sum}\ xs'
\end{array}
$$

We now move down the outer case into each branch, and perform reductions:

$$
\begin{array}{l}
\textbf{case}\ xs'\ \textbf{of} \\
\quad [\,] \rightarrow 0 \\
\quad (x'' : xs'') \rightarrow (3 * x'') + \mathit{sum}\ (f\ xs'')
\end{array}
$$

We notice a familiar expression in $\mathit{sum}\ (f\ xs'')$, and fold when reaching it. Adding it all together gives a new function h_2:

$$
\begin{array}{l}
h_2\ xs = \textbf{case}\ xs\ \textbf{of} \\
\qquad [\,] \rightarrow 0 \\
\qquad (x' : xs') \rightarrow (2 * x') + \textbf{case}\ xs'\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad [\,] \rightarrow 0 \\
\qquad\qquad\qquad\qquad\qquad\qquad (x'' : xs'') \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (3 * x'') + h_2\ xs''
\end{array}
$$

Kort [3] studied a ray-tracer written in Haskell, and identified a critical function in the innermost loop of a matrix multiplication, called vecDot:

$$
\mathit{vecDot}\ xs\ ys = \mathit{sum}\ (\mathit{zipWith}\ (*)\ xs\ ys)
$$

This is simplified by our positive supercompiler to:

$$
\begin{array}{ll}
\mathit{vecDot}\ xs\ ys & = h_1\ xs\ ys \\
h_1\ xs\ ys & = \textbf{case}\ xs\ \textbf{of} \\
 & \quad (x' : xs') \rightarrow \textbf{case}\ ys\ \textbf{of} \\
 & \qquad\qquad\qquad (y' : ys') \rightarrow \\
 & \qquad\qquad\qquad\quad x' * y' + h_1\ xs'\ ys' \\
 & \qquad\qquad\qquad _ \rightarrow 0 \\
 & \quad _ \rightarrow 0
\end{array}
$$

The intermediate list between *sum* and *zipWith* is transformed away, and the complexity is reduced from $2|xs|+|ys|$ to $|xs|+|ys|$ (since this is matrix multiplication $|xs| = |ys|$).

3. Towards a Supercompiler in GHC

The first step towards a supercompiler in GHC is to construct a supercompilation algorithm for System F_c, the typed intermediate language found in GHC [10]. The conversion of the algorithm is quite straightforward and in our experience it is rare to accidentally introduce non-termination or unsound transformation steps. The challenge is rather to achieve the desired transformation effects in the presence of the casts (▶) that might propagate inside expressions in System F_c. Previous work on supercompilation has been for untyped languages. Since GHC assumes well-typed expressions once the type-checker pass is done it is necessary to prove that the supercompiler preserves types as well

Mitchell and Runciman lists three choices that need to be made during optimization:

- Which function to inline.
- What termination criterion to use.
- What generalisation to use.

Our current implementation makes a fixed choice for function to inline (left-most), what termination criterion to use (the homeomor-

phic embedding), and what generalisation to use (the most specific generalisation, or simple splitting in the case where no common terms are found).

4. Analysis of Problems

4.1 Code Explosion

It is possible to construct examples where a module exports two mutually recursive functions, and if one supercompiles both these functions independently it might lead to code duplication. We have sidestepped this issue by only supercompiling one function, *main*, which makes our supercompiler unusable for libraries at the moment. However, upon compiling the entire program with the library present the desired optimization should occur. We expand on issues with whole program compilation in Section 4.3.

Our current strategy to always inline the left-most function is not always beneficial for performance since it might lead to code explosion, and possibly evaluating the same expression multiple times. An example is the following Haskell-program:

```
main = do
         x  ← getArgs
         xs ← readFile (x !! 2)
         ys ← readFile (x !! 3)
         doCompute xs ys
         return ()
```

which will give two copies of readFile: one $readFile_{!!2}$ and one $readFile_{!!3}$, something that is not necessarily faster than simply calling *readFile* directly with the different parameters.

Inlining itself is a difficult problem, which is not that well studied. The inliner of GHC has been investigated previously [6], and we expect many results from that investigation to carry over to our supercompiler. We find it likely that more work is necessary for inlining efficiently in our supercompiler, possible directions for work include, but are not limited to:

- Investigate how inliners of other compilers than GHC work.
- Characterize what a good or bad inlining is.
- Create a partial order between inlinings.

With the above knowledge, it should be possible to design a heuristic that works well in practice.

4.2 Compilation Speed

We propose to make the homeomorphic embedding test on a smaller part of the tree, which still preserves termination of the algorithm. Any improvements in the implementation of the homeomorphic embedding test will be of benefit both to our approach and to Supero. Changing the test makes it impossible to extend our algorithm to distillation [1], which removes more intermediate structures. This a trade-off we are prepared to accept considering that supercompilation is still an improvement over what is currently in use today. Should we change our mind in the future it should not be a problem to go back to the kind of test Supero uses. It is still too early to tell whether this change is enough, or if there are other bottlenecks that will show when supercompiling larger programs.

4.3 Whole Program Compilation

Currently, our experiments have only been on complete programs defined in one module, avoiding to import the Prelude. To gain the most out of supercompilation on real world programs GHC needs to be tweaked to handle whole program compilation. This can be done by removing the current constraint that expressions placed in the interface files (.hi) must be "small", and by annotating loop-breakers in the interface files. The size increases to the interface files from these changes need to be measured.

Whole program compilation has been used in MLton and they manage to compile programs larger than 100k lines [12]. If this number carries over to GHC and Haskell it makes whole program compilation a viable approach for a majority of the known Haskell programs.

5. Conclusions and Future Work

We have reported on our current work on building a supercompiler for GHC. Many problems that we intend to tackle have been mentioned already, among them a proof of type preservation of the algorithm, investigating inlining to avoid code explosion, and measuring the effects of making the interface files contain entire modules to achieve whole program compilation. We are however certain that problems which we have not foreseen will surface as we make progress on our supercompiler.

Acknowledgements

We thank Simon Peyton Jones for suggesting several of the examples and explaining how to achieve whole program compilation with GHC. Max Bolingbroke has patiently explained large parts of the GHC internals to us and deserves a special thanks.

References

[1] G. W. Hamilton. Distillation: extracting the essence of programs. In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 61–70, New York, NY, USA, 2007. ACM.

[2] P. A. Jonsson and J. Nordlander. Positive supercompilation for a higher-order call-by-value language. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, 2009. ACM. To appear.

[3] J. Kort. Deforestation of a raytracer. Master's thesis, University of Amsterdam, 1996.

[4] N. Mitchell and C. Runciman. A supercompiler for core Haskell. In O. Chitil et al., editor, *IFL 2007*, volume 5083 of *Lecture Notes in Computer Science*, pages 147–164. Springer-Verlag, 2008.

[5] A. Ohori and I. Sasano. Lightweight fusion by fixed point promotion. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 143–154, New York, NY, USA, 2007. ACM.

[6] S. L. Peyton Jones and S. Marlow. Secrets of the glasgow haskell compiler inliner. *J. Funct. Program*, 12(4&5):393–433, 2002.

[7] D. Sands. Proving the correctness of recursion-based automatic program transformations. *Theoretical Computer Science*, 167(1–2):193–233, 30 October 1996.

[8] M.H. Sørensen. Convergence of program transformers in the metric space of trees. *Sci. Comput. Program*, 37(1-3):163–205, 2000.

[9] M.H. Sørensen, R. Glück, and N.D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

[10] M. Sulzmann, M. M. T. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *TLDI '07: Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66, New York, NY, USA, 2007. ACM.

[11] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, June 1990.

[12] S. Weeks. Whole-program compilation in MLton. In *ML '06: Proceedings of the 2006 workshop on ML*, pages 1–1, New York, NY, USA, 2006. ACM. http://mlton.org/pages/References/attachments/060916-mlton.pdf.

A Semantics for a Real-Time Actor Language

István Knoll Anders P. Ravn Arne Skou
Dept. of Computer Science, Aalborg University
S. Lagerlöfs Vej 300, DK-9220 Aalborg Øst
{iknoll, apr, ask}@cs.aau.dk

Abstract

In order to develop simulators and analysis tools for an actor based real-time language, we define its semantics. The semantics is interesting in itself, as it models the functional, communication, and timing aspects separately, allowing several variants of the language to be investigated.

1 Introduction

Development tools for hybrid embedded real-time systems often need to be combined with external verification tools to support a variety of model checking approaches. An example is COMDES-II [9], an integrated development environment that supports the model-driven paradigm by providing a graphical interface for designing embedded real-time controllers, using an extended function block diagram notation with three syntactic levels. Function blocks (*level 1*) are grouped into actors (*level 2*), which form a network communicating through channels (*level 3*).

Fig. 1 shows an example of an embedded system model in COMDES-II.

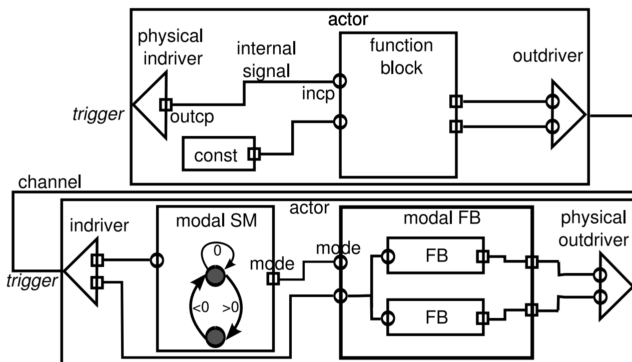


Figure 1: A COMDES-II system model example.

*Actor*s are triggered periodically by timing events in accordance with distributed timed multitasking [1], or by other, already triggered actors. Communication among actors takes place through *channel*s, carrying composite data types, connecting input and output drivers (*indriver*, *outdriver*) in the actors. One input driver which *trigger*s the actor upon receiving data. The functionality of the actors is given by function block (*fb*) diagrams. Besides basic function blocks, there are modal state machine function blocks that provide discrete (reactive) functionality by controlling a modal function block. Which contained function block in a modal function block is executed is determined by a computed *mode*[1]. Data transfer among function blocks happen through internal signals (*isignal*), which connect ingoing and outgoing connection points (*incp*, *outcp*) in function blocks.

Comparable notations with distinguished syntactic level representations include IF [3, 4] and Ptolemy [7]. *IF* supports three syntactic levels; the *specification level*, expressing the system, which is translated to the *intermediate level*, which is the IF internal format, and the labelled transition system (*LTS*) representation, the *semantic model level*, serving verification and simulation purposes using external, linked tools [3]. In *Ptolemy*, architectural and behavioral design is distinguished, and various models of computations (semantic domains) are supported [5]. Ptolemy supports actor oriented design, and supports timed multitasking [10], as well as continuous time, and discrete time semantic domains [7].

Verification of COMDES-II models is not supported, however, semantic anchoring to Uppaal [2] has been investigated for individual actors, where functionality is abstracted away, and the intent is to investigate scheduling [8].

Our contribution is a complete semantics for the COMDES-II modeling language itself. It is a foundation for building semantically correct translators to the input languages of a variety of verification tools and frameworks like IF, Spin [6], and Uppaal.

Here, we present the three level semantics for COMDES-II: (1) The semantics of actor functionality, parametrized by simple function blocks and finite state machines, (2) semantics for actor communication and coordination, and (3) semantics for timed systems. Based on the semantics, we report on the development of a semantics derived simulator, which is used for demonstrating potential nondeterminism and race conditions in the current COMDES-II specification.

2 COMDES-II Semantics

The behaviour of a COMDES-II system is on an outermost level determined by the flow of values over the channels connecting the actors. An actor that is enabled, and has been triggered, computes a new value for its local state variables and an output for its output drivers. The entire system can thus be seen as a network of Mealy machines, where each actor takes exactly one step each time it is triggered. With this structure in mind, we define the semantics

[1]Additionally, COMDES-II has a *composite function block* type, which can be used to abstract away complex aggregates of function blocks (not shown in Fig. 1)

in two steps. First we give an operational semantics to the network and then a denotation for each actor corresponding to the definition for a Mealy machine's transition and output functions.

Network semantics We begin by defining the following syntactic domains: $a \in \mathbf{Actor}$, the set of actors, $i \in \mathbf{Indriver}$, the set of input drivers, $o \in \mathbf{Outdriver}$, the set of output drivers, $c \in \mathbf{Channel} = \mathbf{Outdriver} \rightarrow \mathcal{P}(\mathbf{Indriver})$, the set of channels, $t \in \mathit{Trigger} : \mathbf{Actor} \rightarrow \mathbf{Indriver}$, the input driver that triggers a given actor, and $f \in \mathbf{FB}$, the set of function blocks, The set of drivers is $\mathbf{Driver} = \mathbf{Indriver} \cup \mathbf{Outdriver}$.

The mapping $\mathit{indrivers} : \mathbf{Actor} \rightarrow \mathcal{P}(\mathbf{Indriver})$ returns the input drivers that are contained in a given actor, while $\mathit{outdrivers} : \mathbf{Actor} \rightarrow \mathcal{P}(\mathbf{Outdriver})$ returns the output drivers, and $\mathit{fb} : \mathbf{Actor} \rightarrow \mathcal{P}(\mathbf{FB})$ returns the function blocks for a given actor. A number of well-formedness conditions ensure that drivers and function blocks belong to a unique actor.

We shall not be particularly concerned with the values communicated and computed, but assume that for the drivers, connection points, and states, there is a universal flat value domain Val, with a designated bottom element $\perp$ that denotes the undefined value.

Configurations A configuration C in the transition system is a triple: $\langle D, S, A \rangle$, where $D \in DV = \mathbf{Driver} \rightarrow \mathbf{Val}$ maps all input and output drivers to their current values, $S \in SV = \mathbf{FB} \rightarrow \mathbf{Val}$ maps the state variables in the function blocks to their current values, and $A \subseteq \mathbf{Actor}$ is the set of enabled (triggered) actors.

In the following, DV_a and SV_a denote the projection of DV respectively SV on the drivers or function block states for a given actor a. Due to the well-formedness conditions, these projections form partitions of the corresponding configuration components. We leave initial configurations unspecified.

Transitions A transition of the global system depends on two functions defined for actors: $\sigma : \mathbf{Actor} \rightarrow (DV \times SV) \rightarrow SV$, that computes a next state, and $\pi : \mathbf{Actor} \rightarrow (DV \times SV) \rightarrow DV$, that computes the outputs. These functions are defined below.

A global transition, $\langle D, S, A \rangle \longrightarrow \langle D', S', A' \rangle$, selects a subset $A_t \subseteq A$ of the enabled actors to take a step. It updates the state for function blocks of the selected actors (1).

$$S' = S \uplus [(\sigma(a)(D, S))_a \mid a \in A_t] \tag{1}$$

Note that the states of unselected actors are left unchanged.

Then, it updates the output drivers of the selected actors (2),

$$D^o = D \uplus [(\pi(a)(D, S))_a \mid a \in A_t] \tag{2}$$

and completes by updating the input drivers with the values of the corresponding output drivers (3).

$$D' = D^o \uplus [i \mapsto D^o(d) \mid i \in c(d) \wedge d \in \mathit{outdriver}(A_t)] \tag{3}$$

Finally, the actors triggered by updated input drivers have to be set after removing the selected actors (4).

$$\begin{aligned} A' &= A \setminus A_t \cup \{a \mid t(a) \in \mathit{Updated}\}, \text{ where} \\ \mathit{Updated} &= \{i \in c(d) \mid d \in \mathit{outdriver}(A_t)\} \end{aligned} \tag{4}$$

The semantics is deliberately very nondeterministic in the selection of actors for a step. It leaves a maximum of freedom for later constraints that comes from the timing properties that are superimposed on the actors. A system will be deterministic, modulo stuttering steps where no actor is selected, only when at most one actor is triggered at a time.

Actor semantics Recall that an actor specifies one step of a Mealy machine, so the task is to define the two functions: σ that computes a next state, and π that computes the outputs.

During the evaluation of the function blocks, values are passed through connection points, so we introduce the syntactic domain: $cp \in \mathbf{CP}$, the set of connection points. As usual, these are private for each function block or driver, such that the sets are partitionend by projection on these. The connection points have values, so we define the auxiliary domain: $IV = \mathbf{CP} \rightarrow \mathit{Val}$ and have through the syntactic binding of points to indrivers and outdrivers an initialisation function and an extraction function $\mathit{init} : DV \rightarrow IV$ and $\mathit{exit} : IV \rightarrow DV$.

We have also $s \in \mathbf{Signal} = \mathbf{CP} \rightarrow \mathcal{P}(\mathbf{CP})$, the set of signals, going from an outgoing to ingoing connection points. These are private for each actor.

Since the graph defined by the signals of an actor a is acyclic, we can define a precedence relation $\prec$ on the function blocks FB_a by $f \prec f'$ just when a signal connects a cp of f to a cp of f'. Since it is a precedence relation on a finite set, a non-empty subset of function blocks $FB'_a \subseteq FB_a$ will always have an element that precedes all others or is unrelated to all others.

Semantics of function blocks The step and output functions of the individual function blocks are assumed to be specified by a COMDES-II framework. Thus we expect to be given a next state function $\sigma^f : \mathbf{FB} \rightarrow (IV \times SV) \rightarrow SV$, and an output function $\pi^f : \mathbf{FB} \rightarrow (IV \times SV) \rightarrow IV$.

The computation for an actor a is now defined by iterating over its function blocks in precedence order. This is done by the auxiliary function $\mathit{iterate}$ (5):

$$\mathit{iterate} : (\mathcal{P}(\mathbf{FB}) \times (IV \times S)) \rightarrow (IV \times S) \tag{5}$$

Now, let $(IV', S') = \mathit{iterate}(\mathit{fb}(a), (\mathit{init}(D), S))$, then the main semantic functions are defined by: $\sigma(a)(D, S) = S'$, and $\pi(a)(D, S) = D \uplus [d \mapsto \mathit{exit}(IV')(d) \mid d \in \mathit{outdriver}(a)]$.

The iteration function is defined inductively. The empty set of function blocks gives the identity as result (6). In (7) f is selected so it precedes all other elements of $\mathbf{FB}$.

$$\mathit{iterate}(\emptyset, (IV, S)) = (IV, S) \tag{6}$$

$$\mathit{iterate}(\mathbf{FB}, (IV, S)) = \mathit{iterate}(\mathbf{FB} \setminus f, \mathit{step}(f, IV, S)) \tag{7}$$

The step function (8) just applies the definition of the function block and signals from the outgoing connection points.

$$\begin{aligned} \mathit{step}(f, S, IV) = \; & (\sigma^f(f)(IV, S), IV) \\ & \uplus [cp_{in} \mapsto \pi^f(f)(IV, S)(cp_{out}) \\ & \mid cp_{in} \in s(cp_{out}) \wedge cp_{out} \in CP_f] \end{aligned} \tag{8}$$

It is clear that the iteration terminates, because the finite set of function blocks is reduced by one for every step. It is also clear

that the function is defined, because each function block is executed at most once, and furthermore no function block is executed before its ingoing connection points are defined (to the extent that the indrivers are defined).

The non-determinism in selecting a next function block reflects an interleaving semantics for parallel execution of unrelated function blocks.

Towards timed semantics The timing of the system is given in terms of *deadlines*. To be able to measure the time passing during the execution of the actors, and thereby to know when they have reached their deadlines, we introduce a clock (*Clk*). With the added clock, the configuration becomes $\langle D, S, A, Clk \rangle$, where $Clk : Actor \to \mathbb{N}$. During global transitions, the value of the clock is updated (9).

$$Clk' = [Clk + 1] \uplus [Clk(a) \mapsto 0 \mid a \in A_t] \tag{9}$$

A deadline is defined as $deadline : Actor \to \mathbb{N}$. We can use the deadline to select the subset of actors that are to be executed in a given global transition (10),

$$A_t = \{a \mid a \in A \wedge Clk(a) \sim deadline(a)\} \tag{10}$$

where $\sim$ is any relation defined to select the subset of actors, based on the values of their deadlines.

3 Conclusion

To support validation of models designed with COMDES-II, a development tool for embedded real-time systems, we have defined a semantics for its modeling language. As a demonstration of the functionality expressed, we have developed a semantics-derived simulator, which accepts COMDES-II models expressed in an input language with syntax derived from the abstract syntax[2]. Simulations of simple example systems have shown that COMDES-II models may exhibit nondeterministic behavior, as the execution order of actors is not defined explicitly *in the model*[3].

By adding semantics to the timing behavior, nondeterministic behavior can be reduced, as the actor execution sequence is further constrained. To address the nondeterministic behavior, two approaches are proposed: (1) Extending the COMDES-II specification syntax with actor *priorities*, and requiring updates on incoming channels before actor execution, or (2) model checking the discrete and timing behavior of the model can detect and enable the system designer to address nondeterministic issues.

The provided semantics for COMDES-II enables the creation of semantically correct translators to other representations with clearly defined semantics, including the input languages of model checkers. Verification of the discrete behavior, represented by the modal state machines in COMDES-II model, and the timing behavior thus becomes possible.

In further work we will focus on creating a translator for the timed automata based model checker, Uppaal, and interfacing the simulator to existing function block implementations.

[2]ANTLR v3 [11] was used to define the concrete syntax, and Java 6 was used for implementing the simulator.

[3]Prototype implementations of the COMDES-II IDE have implementation specific actor execution sequence, namely the appearance order in the compiled controller software. This is, however, clearly not part of the formal specification, and even so may lead to unspecified behavior when executed on multiple processors.

References

[1] Christo Angelov and Jesper Berthing. Distributed timed multitasking - a model of computation for hard real-time distributed systems. In Bernd Kleinjohann, Lisa Kleinjohann, Ricardo Jorge Machado, Carlos Eduardo Pereira, and P. S. Thiagarajan, editors, *DIPES*, volume 225 of *IFIP International Federation for Information Processing*, pages 145–154. Springer–Verlag, 2006.

[2] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, 2004.

[3] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm, and Laurent Mounier. IF: An intermediate representation and validation environment for timed asynchronous systems. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *World Congress on Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 307–327. Springer–Verlag, 1999.

[4] Marius Bozga, Susanne Graf, and Laurent Mounier. IF-2.0: A validation environment for component-based real-time systems. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 343–348. Springer–Verlag, 2002.

[5] Holger Giese and Stefan Henkler. A survey of approaches for the visual model-driven development of next generation software-intensive systems. *J. Vis. Lang. Comput.*, 17(6):528–550, 2006.

[6] Gerard J. Holzmann. Design and validation of protocols: A tutorial. *Computer Networks and ISDN Systems*, 25(9):981–1017, 1993.

[7] Christopher Hylands, Edward A. Lee, Jie Liu, Xiaojun Liu, Stephen Neuendorffer, Yuhong Xiong, Yang Zhao, and Haiyang Zheng. Overview of the Ptolemy project. Technical Report UCB/ERL M03/25, University of California at Berkeley, July 2003.

[8] Xu Ke, Paul Pettersson, Krzysztof Sierszecki, and Christo Angelov. Verification of COMDES-II systems using UPPAAL with model transformation. In *14th International IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE Computer Society Press, August 2008.

[9] Xu Ke, Krzysztof Sierszecki, and Christo Angelov. COMDES-II: A component-based framework for generative development of distributed real-time control systems. In *Proc. of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications RTCSA*, pages 199–208. IEEE Computer Society Press, 2007.

[10] J. Liu and E. A. Lee. Timed multitasking for real-time embedded software. *IEEE Control Systems Magazine: Advances in Software Enabled Control*, pages 65–75, 2003.

[11] Terence J. Parr and Russell W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience*, 25(7):789–810, 1995.

A Specification-Driven Interpreter for Testing Asynchronous Creol Components*

Marcel Kyas[1], Andries Stam[2], Martin Steffen[1], and Arild B. Torjusen[1]

[1] University of Oslo, Norway
[2] Almende, The Netherlands

1 Motivation

Software testing [9] is an established practice to ensure the quality of programs and systems. Hosts of different testing approaches and frameworks have be proposed and put to (good) use over the years. Formal methods and program language theory have proven valuable to render testing practice a more formal, systematic discipline (cf. e.g. [5]). In itself not a new proposal —first inspirations to put testing on more formal grounds can be dated back as early as the seminal Nato conference on "Software Engineering" [8]— formal approaches to testing have gained momentum in recent years, as for instance witnessed by the trend towards model-based testing [4,1]. In this paper we propose and explore a formal approach for black-box testing of asynchronously communicating components in open environments.

Creol: a language for asynchronously communicating, active objects Creol [3,7] is a high-level, object-oriented language for distributed systems, that features active objects. Creol is formally defined and especially its operational semantics is implemented in rewriting logic, using Maude [2] as execution platform. Its communication model is based on exchanging messages *asynchronously.* This is in contrast with object-oriented languages based on multi-threading, such as *Java* or $C^\sharp$, which use "synchronous" message passing in which the calling thread inside one object blocks and control is transferred to the callee. Exchanging messages asynchronously decouples caller and callee, which makes that mode of communication advantageous in a distributed setting. On the other hand, the asynchronicity makes validating and testing of programs more challenging.

Behavioral interface description language Abstracting from internal executions, the black-box behavior of components is given by interactions at their

* Part of this work has been supported by the EU-project IST-33826 *Credo: Modeling and analysis of evolutionary structures for distributed services* and the German-Norwegian DAAD-NWO exchange project *Avabi* (Automated validation for behavioral interfaces of asynchronous active objects).

interface. We formalize the interface specification language over communication trace labels to specify components in terms of traces of observable behavior.

In the specification language, a clean separation of concerns between interaction under the control of the component or coming from the environment is central, which leads to an assumption-commitment style description of a component's behavior. The assumptions *schedule* the order of inputs, whereas the outputs as commitments are *tested* for conformance. To ensure the mentioned separation of responsibilities, we define well-formedness conditions which in addition assure that only "meaningful" traces, i.e., those corresponding to possible behavior, can be specified. The specification language is characterized by two other salient features: it allows to specify freshness of communicated values and furthermore, it respects the asynchronous nature of communication in Creol: Due to asynchronous communication, the order in which outgoing messages from a component are observed by an external observer does not necessarily reflect the order in which they where actually sent. We take this into account by only considering trace specifications up-to an appropriate notion of *observational equivalence.* The specification language is a simple recursive trace language designating sets of finite traces over communication labels.

A second point to stress is that the specification language is designed to be efficiently executable on Creol's executing platform and thus be used for testing a component. We define an operational semantics for the specification language. This is done by synchronising the execution of the specifications with that of the component for the purpose of both generating the required input to the component and at the same time testing that the output behavior of the component conforms to the specification, up-to observational equivalence.

2 Results

The paper extends the technical report [6], which concentrates on the formalization, and contains the following contributions:

Formalization: We formalize the interface behavior of a concurrent, object-oriented, language plus a corresponding behavioral interface specification language. This gives the basis for testing active Creol objects, where a test environment can be simulated by execution of the specifications.

Implementation: The existing Creol interpreter, realized in rewriting logic on the Maude platform, is extended with the implementation of the specification language. This yields a specification-driven interpreter for testing asynchronous Creol components.

Case study: As a case study, we apply the test method to a model of an industrial software system which is inherently multi-threaded and based on asynchronous communication.

References

1. *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.
2. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 system. In R. Nieuwenhuis, editor, *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA 2003)*, volume 2706 of *Lecture Notes in Computer Science*, pages 76–87. Springer-Verlag, June 2003.
3. The Creol language. `http://heim.ifi.uio.no/creol`, 2007.
4. S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the 1999 International Conference on Software Engineering, 1999*, pages 285–294, 1999.
5. M.-C. Gaudel. Testing can be formal, too. In P. D. Mosses, M. Nielsen, and M. I. Schwarzbach, editors, *Proceedings of TAPSOFT '95*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer-Verlag, 1995.
6. I. Grabe, M. Steffen, and A. B. Torjusen. Executable interface specifications for testing asynchronous Creol components. Technical Report 375, University of Oslo, Dept. of Computer Science, July 2008. A shorter version has been submitted for conference proceedings.
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. A. I. Llewelyn and R. F. Wickens. The testing of computer software. In P. Naur and B. Randell, editors, *Software Engineering: A Report on a Conference sponsored by the NATO science committee*, pages 189–199. NATO, Jan. 1969.
9. R. Patton. *Software Testing*. SAMS, second edition, July 2005.

A method for modelling probabilistic object behaviour for simulations of formal specifications

Timo Nummenmaa
Department of Computer Science
University of Tampere
timo.nummenmaa@cs.uta.fi

Extended Abstract

1. Introduction

Formal specifications are a powerful method for modelling system behaviour. The specifications should precisely state what the eventual piece of software is supposed to do [Diller, 1990]. One popular way to study the specifications is by simulations. Although random simulations may, in principle, exhibit all potential behaviours, for certain purposes they do not appear to be realistic.

This work presents a method for modelling probabilistic object behaviour, to be used in simulations of formal specifications. Our work assumes that the specifications are composed of objects and actions, where the objects may participate in the actions provided that certain conditions evaluate to true.

Our approach is based on the idea that when an object is a potential participant for an action, the object will decline the participation with a predefined probability specific to that object. The objects also have individual eagerness intervals. When an object is needed to participate in an action, an eagerness value is randomly assigned for each potential object from their respective intervals, and the object with the greatest eagerness value is chosen. Using these probabilistic components it is possible to model individual probabilistic strategies for objects.

We have implemented our method in the DisCo software package [DisCo,2008]. The DisCo software package includes a language for writing specifications and a simulation engine. We added the probabilistic modelling features to the DisCo package and changed the simulation engine to use the probabilistic information in simulations.

Our research was utilized to find out if formal methods could be used to gain the advantages provided by them in game design, an area where formal specifications are not generally used. Such advantages from mathematics would be precision, conciseness, clarity, abstraction, independence from natural language, and proofs, [Lightfoot 1991]. However, for the purpose of modeling player strategies, the execution model for simulating the specifications would need to take into account probabilities.

To evaluate the usefulness of the method in practice, a formal specification of an existing pervasive massively mobile on-line game called Mythical: The Mobile Awakening [MythicalMobile, 2008]. was created using the proposed probabilistic

execution model. A massively multiplayer game suited the purpose of this work well because of the higher requirements on software design quality they have [Garriott, 2003] compared to other types of games. The particular game was chosen as there was detailed design data available from previous work on the project that created the game by the author [Nummenmaa, 2008]. Several test runs with different goals were realised using with the specification that was created.

2. Examples

Test runs were used to validate the applicability of specifications with our probabilistic execution model. A particular test run was used to analyze the success of different player strategies. The test was made to find out if players could be more successful by concentrating on certain areas of game play. In the game Mythical: the Mobile Awakening, players play two main sections of the game: rituals and encounters. The players have control on how much they participate in each of the parts, but players must complete certain rituals and encounters to progress into later sections of the game.

The test was designed as a very synthetic test, where all players start playing at exactly the same time. Moreover, if a small number of players play, it means the following: the players would only interact with each other and only with the personalities of those other players. The structure of the game makes it possible to obtain useful results with tests containing a small number of players, as the players become very active, and the game does not present many limitations on how many activities the players can undertake at the same time.

When the test was run, the players were given eagerness values to describe how interested they were to participate in encounters and rituals. The test results clearly showed that the players who concentrated considerably more on only one of these sections performed the worst. Thus, according to this indication, the players should avoid a strategy concentrating only on rituals or only on encounters.

3. Conclusions and future research

A novel approach to use probabilities in simulating formal specifications is presented. In the approach, it is possible to define probabilities by which objects decline to participate in the execution of an action, and probabilities on how eager they are to take part in that action compared to other objects. Using these probabilities, it is possible to control the behaviour of the objects to model probabilistic strategies. The approach was implemented by modifying the execution model of the DisCo language and was tested by executing several executions that simulated how the game would run with different kinds of players.

It was possible to create the functionality of a real working multiplayer game using the DisCo language on a generalized level, based on the actions and objects identified in

the real game. The resulting specification resembled the original game enough that the world content of the real game could be imported from the games' database and used for running executions.

Having the possibility to specify eagerness properties for players of the game made a huge difference in the executions of the specification. As different players had a different playing style and had different interests in different play modes, they performed differently. Having all players play randomly would clearly make executions of the game more unrealistic as there are very few games, where all players are completely equal. Now players had different activity profiles, as had been defined in the starting state for the executions.

Based on data from the executions, DisCo specifications with the modified execution model can be used as an aid in game development. Executing the specification can point to issues in the design and can answer questions that a designer might have on the design. One such question is if the player needs to repeat a certain aspect of the game too many times. It might, however, be hard for designers with no previous programming ability to create such specifications.

In addition to game design, there are other potential application areas for the enhanced method to model specifications and simulate them. One obvious area, where the method can be tried out, is transaction management.

Key words and phrases

Formal specification, DisCo, probabilistic simulation, game design research.

References

[Diller, 1990] Diller, A., *Z - An introduction to formal methods*, John Wiley & Sons, 1990.

[DisCo, 2008] The DisCo project WWW page. At URL *http://disco.cs.tut.fi* on the World Wide Web. [Accessed 7 May 2008].

[Garriott, 2003] Richard Grarriott, Foreword. In: Thor Alexander (ed), *Massively Multiplayer Game Development*. Charles River Media, 2003, xi-xiv.

[Lightfoot, 1991] Lightfoot, D., *Formal Specification using Z*, Macmillan, 1991.

[MythicalMobile, 2008] The Mythical: The Mobile Awakening description page. At URL http://www.pervasive-gaming.org/iperg_games10.php on the World Wide Web. [Accessed 24 September 2008].

[Nummenmaa, 2008] Timo Nummenmaa. Adding probabilistic modeling to executable formal DisCo specifications with applications in strategy modeling in multiplayer game design. Master's thesis, University of Tampere, June 2008. Available at URL http://tutkielmat.uta.fi/pdf/gradu03189.pdf.

Extending Kleene Algebra with Synchrony: Completeness and Decidability*
(extended abstract)**

Cristian Prisacariu

Department of Informatics, University of Oslo,
P.O. Box 1080 Blindern, N-0316 Oslo, Norway.
E-mail: cristi@ifi.uio.no

1 Motivation

The work reported here investigates the introduction of synchrony into Kleene algebra. The resulting algebraic structure is called *synchronous Kleene algebra*. Models are given in terms of regular sets of concurrent strings and finite automata accepting concurrent strings. The extension of synchronous Kleene algebra with Boolean tests is presented together with models on regular sets of guarded concurrent strings and the associated automata on guarded concurrent strings. Completeness w.r.t. the standard interpretations is given in each case. Decidability follows from completeness. A comparison with Mazurkiewicz traces is made which yields their incomparability with the synchronous Kleene algebra (one cannot simulate the other). Applications to respectively deontic logic of actions and to propositional dynamic logic with synchrony are sketched.

Kleene algebra is a formalism used to represent and reason about programs, and it formalizes axiomatically the structures of regular expressions and finite automata (see work of J.H. Conway and D. Kozen). Kleene algebra with tests combines Kleene algebra with a Boolean algebra and can encode the propositional Hoare logic, therefore it can express while programs. In one form or another, Kleene algebras appear in various formalisms in computer science: relation algebras, logics of programs and in particular propositional dynamic logic (PDL), regular expressions and formal language theory.

Synchrony is a model for concurrency which was introduced in the process algebra community in R. Milner's SCCS, but detaches form the general interleaving approach. On the other hand it is a concept which does not belong to the partial order model of true concurrency either. The synchrony concept proves highly expressive and robust; SCCS can represent CCS (i.e. asynchrony) as a subcalculus, and a great number of synchronizing operators can be defined in terms of the basic SCCS-Meije operators.

The notion of *synchrony* has different meanings in different areas of computer science. Here we take the distinction between *synchrony* and *asynchrony* as presented in

* Partially supported by the Nordunet3 project "COSoDIS – Contract-Oriented Software Development for Internet Services" (http://www.ifi.uio.no/cosodis/).

** One may like to check details in the technical report: C. Prisacariu, "Extending kleene algebra with synchrony – technicalities", Dept. Informatics, Univ. Oslo, 2008.

the SCCS calculus and later implemented in e.g. the Esterel (Meije) synchronous programming language. We understand *asynchrony* as when two concurrent systems execute at indeterminate relative speeds (i.e. their actions may have different noncorelated durations); whereas in the *synchrony* model each of the two concurrent systems execute instantaneously a single action at each time instant. The SCCS concurrency operator $\times$ over processes is different from the classical $||$ of CCS.

The perfectly synchronous concurrency model takes the assumption that time is discrete and that basic actions are instantaneous (i.e. take zero time and represent the time step). Moreover, at each time step all possible actions are performed. The reasoning is governed by the assumption of a global clock which provides the time unit for all the actors in the system. Note that for practical purposes this is a rather strong assumption which offends the popular relativistic view from process algebras. On the other hand the mathematical framework of the synchronous calculus is much cleaner and more expressive than the asynchronous model, and the experience of the Esterel implementation and use in industry contradict the general believe.

The motivation for adding synchrony to Kleene algebra spawns from the need to reason about actions (where Kleene algebra is the equational tool of choice in conjunction with PDL) that can be executed in a truly concurrent fashion. On the one hand we do not need such a powerful concurrency model like the ones based on partial orders; on the other hand the low level interleaving model is not expressive enough. The synchrony model has appealing equational representation and thus it is natural to integrate into the Kleene algebra. Moreover, the reasoning power and expressivity that synchrony offers is enough for the applications listed below.

We have successfully used synchronous Kleene algebra in the context of deontic logic of actions, and secondly we used the extension of synchronous Kleene algebra with tests in the context of PDL with synchrony. A third application that we currently work on is to give a semantics for Java threads. More general, wherever one uses Hoare logic to reason about programs one can use the more powerful Kleene algebra with tests which has also tool support (the KAT-ML prover). Moreover, one may safely choose the synchronous Kleene algebra with tests where in addition reasoning about concurrent executions is needed (similar to some extent to the current work of C.A.R. Hoare). In these contexts (synchronous) Kleene algebra proves more powerful and general than classical logical formalisms.

2 Results

Synchronous Kleene algebra ($\mathcal{SKA}$) adds to the Kleene algebra the $\times$ operator of SCCS. $\mathcal{SKA}$ is a σ-algebra with signature $\sigma = \{+, \cdot, \times, ^*, \mathbf{0}, \mathbf{1}, \mathcal{A}_B\}$ which gives the action operators and the basic actions $\mathcal{A}_B$. The operators of $\mathcal{SKA}$ are defined over a carrier set of *compound actions* denoted $\mathcal{A}$. The non-constant functions of σ are: "$+$" for *choice* of two actions, "$\cdot$" for *sequence* of two actions (or concatenation), "$\times$" for *concurrent composition* of two actions, and "*" to model *recursive execution* of one action.

We give the *standard interpretation* of the actions by defining a *homomorphism* $\hat{I}_{\mathcal{SKA}}$ which takes any action of the $\mathcal{SKA}$ algebra into a corresponding *regular concurrent set* and preserves the structure of the actions given by the operators. A *concurrent*

set is a subset of strings from $\mathcal{P}(\mathcal{A}_B)^*$. The important construction on concurrent sets is the one corresponding to $\times$: $A \times B \triangleq \{u \times v \mid u \in A, v \in B\}$, where $u, v \in \mathcal{P}(\mathcal{A}_B)^*$ are concurrent strings and $u \times v$ is defined as (where $x_i, y_j \in \mathcal{P}(\mathcal{A}_B)$):

$$u \times v \triangleq (x_1 \cup y_1)(x_2 \cup y_2) \dots (x_n \cup y_n) y_{n+1} \dots y_m.$$

Theorem 1 (completeness). *For any actions $\alpha, \beta \in \mathcal{A}$ then $\alpha = \beta$ is a theorem of $\mathcal{SKA}$ iff the regular concurrent sets $\hat{I}_{\mathcal{SKA}}(\alpha)$ and $\hat{I}_{\mathcal{SKA}}(\beta)$ are the same.*

The proof of completeness follows the ideas of D. Kozen only that we use a combinatorial argument. We use the translation of actions into automata that accept regular concurrent sets (and need to prove an equivalent of Kleene's representation theorem). The main ideas are that the alphabet of the automata is $\mathcal{P}(\mathcal{A}_B)$ and that we require a special product construction. Decidability in P-time follows then naturally from completeness and decidability of the equivalence problem for regular concurrent sets.

Synchronous Kleene algebra with tests $\mathcal{SKAT} = (\mathcal{A}, \mathcal{A}^?, +, \cdot, \times, ^*, \neg, \mathbf{0}, \mathbf{1})$ is an order sorted algebraic structure which combines $\mathcal{SKA}$ with a Boolean algebra in a special way. The structures $(\mathcal{A}^?, +, \cdot, \neg, \mathbf{0}, \mathbf{1})$ and $(\mathcal{A}^?, +, \times, \neg, \mathbf{0}, \mathbf{1})$ are Boolean algebras and the Boolean negation operator $\neg$ is defined only on Boolean elements (called *tests*) of $\mathcal{A}^? \subseteq \mathcal{A}$. The intuition behind tests is that for an action $\phi \cdot \alpha$ it is the case that action α can be performed only if the test ϕ succeeds.

Actions of $\mathcal{SKAT}$ are interpreted over what we call regular sets of *guarded concurrent strings* (an extension of guarded strings). Special two levels finite automata are defined which accept these regular sets; fusion product and concurrent composition are particular operations on these automata.

We compared the synchrony notion of $\mathcal{SKA}$ with Mazurkiewicz traces. Basically the Mazurkiewicz traces have defined a *global* and *partial* independence relation. If we take the similar view in $\mathcal{SKA}$ we need a *local* and *total* independence relation. The locality comes from the perfect synchrony model we adopted, where all the concurrent actions are executed at each tick of a universal clock. The totality comes from our view of concurrent basic actions as forming a set.

We also related to the *shuffle* operator over regular languages which has been used to model concurrency, with a position between the interleaving approach and the partial orders approach. Shuffle is a generalization of interleaving but it does not take into considerration any other relation on the actions(events) which it interleaves. If we were to ignore the branching information in our actions then we can view $\times$ as an *ordered shuffle*. The shuffling of two sequences of actions in $\mathcal{SKA}$ walks step by step (on the $\cdot$ operation) and shuffles the basic actions found (locally).

Our work is also related to Q-algebra which is a two idempotent semirings structure. The difference is in our introduction of the notion of synchrony and the relation of the $\times$ operator with the sequence operator. Also related to our algebraic structure is the language mCRL2. This is a too complex formalism (and tool set) to be naturally incorporated in the applications to the logics that we mentioned in the beginning.

Finding Errors of Hybrid Systems by Optimising an Abstraction-Based Quality Estimate

Stefan Ratschan[1] and Jan-Georg Smaus[2]

[1] Academy of Sciences of the Czech Republic, stefan.ratschan@cs.cas.cz
[2] University of Freiburg, Germany, smaus@informatik.uni-freiburg.de

1 Introduction

Hybrid systems are a formalism for modelling embedded systems. An important problem is to ensure correctness, i.e., *verification*. However, during the design process, hybrid systems are usually not correct yet, and hence *error detection* is equally important. We address here the problem of automatically finding error trajectories that lead the system from an initial to an unsafe state. In contrast to previous works [1, 3], we consider systems with deterministic evolution. Moreover, we do not assume a-priori that our system is incorrect, but rather, we interleave verification, using abstractions of the system [4], and falsification attempts.

We define a real-valued function (the *quality estimate*) that approximates the notion of a given point being close to an initial point of an error trajectory. Then we use numerical optimisation to search for an optimum of this function. The function is computed from a simulation of the hybrid system, using information from the abstraction. For each simulation, the point at which it is cancelled depends on a quality estimate computed on-the-fly. The accuracy of the quality estimate improves as the abstraction is refined.

Analysing the related work, one sees that methods designed for non-deterministic systems [1, 3] try to fill the state space as much as possible (according to some measure) with simulations. As a result, they would start a huge number of simulations in parallel—either from a grid (similar to our naïve algorithm from Sec. 3) or from random sample points. In the case of highly non-deterministic systems, such a strategy is promising since the probability of hitting upon an error trajectory is high. However, for systems with only a small amount of non-determinism, and especially, completely deterministic evolution, this creates a huge number of useless simulations. We avoid this by guiding our search using abstractions in order to quickly arrive at a simulation close to an error trajectory.

Our work has some resemblance with *pure optimisation* problems in artificial intelligence [5]. It is distinctive of our work that the objective *function* itself improves over time. Our work also resembles reinforcement learning [6], because we compute the quality as we do the simulation, and depending on this quality we will do other simulations in the neighbourhood.

2 Hybrid Systems and Abstractions

Hybrid systems are systems with both continuous and discrete state. A hybrid system has a finite set S of *modes* and n continuous variables. In each mode, the behaviour of the variables is controlled by an arithmetic expression involving the variables and their derivatives, called the *Flow* constraint. Each possible transition between modes is controlled by a *Jump* constraint, and there are constraints specifying the initial and unsafe states, respectively. A *trajectory* is a sequence of flows connected by jumps. An *error trajectory* is a trajectory starting in an initial state and ending in an error state. A system is *safe* if it does not have an error trajectory.

An *abstraction* of a hybrid system H is a directed graph whose nodes (the *abstract states*) are subsets of the state-space of H. The transition relation of the abstraction is defined so that each concrete *error* trajectory corresponds to an abstract error trajectory.

Abstractions are useful for verification because if the abstraction is safe, the original system is necessarily also safe. Abstractions can be useful for falsification because the abstract error trajectories narrow down the search space for concrete error trajectories.

We use here a technique that decomposes the state space into hyper-rectangles (*boxes*), as implemented in the tool HSolver [4]. In HSolver, an abstraction that is not fine enough yet to verify the desired property is refined by *splitting* a box.

A *simulation* is an explicitly constructed sequence of points in Φ corresponding to the points of a trajectory at discrete moments in time.

3 The Search Algorithm

We want to find an error trajectory of a hybrid system. Since we focus on deterministic systems, the problem reduces to determining the startpoint of an error trajectory among the initial states. A naïve solution to our problem would be obtained by running simulations of a certain length starting at points lying on a grid covering the entire state space. If no error trajectory is found, the grid width should be reduced and the simulation length increased.

The aim of our work is to find an error trajectory with as little simulation effort as possible. More precisely, we want to: (a) interleave falsification with verification; (b) start simulations at the most promising points; (c) cancel simulations when they do not look promising enough anymore. To address these three aims we designed a search procedure that exploits information available from verification. The procedure uses a quality estimate for simulations to determine which startpoints are the most promising, and when to cancel a simulation.

The quality estimate measures how close the simulation *eventually* gets to an unsafe state. We compute the closeness of all individual simulation points to an unsafe state, and take the optimum of these. Note that this optimum can be easily computed on-the-fly.

For an individual point p, we want to estimate how far p is from an error state along the current simulation. To do so, we use information from the abstraction. We interpret the HSolver abstraction in a geometrical way as illustrated in the figure to the right. Here a_0 is an initial abstract state and a_4 is an unsafe abstract state. The dashed lines are the line segments between connected abstract states. For the point p, the estimated distance is the length of the solid line segment sequence. Note that the abstraction approximates the actual trajectories, and in particular, the present abstraction is sufficiently fine to capture the fact that the trajectories first move away from a_4 before approaching a_4. Thus the quality estimate will capture that moving roughly along the solid line leads towards the unsafe state. The quality estimate will become more faithful as the abstraction becomes refined.

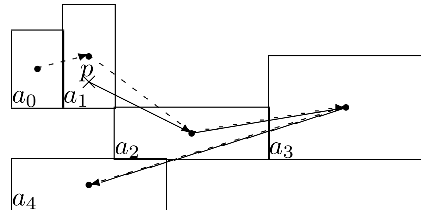


Fig. 1: The distance estimate

The falsification is interleaved with verification. The falsification algorithm is called from the verification after a refinement whenever an initial abstract state is split.

We understand our search problem as the problem of optimising the quality estimate. We use direct search methods [2], specifically the *compass method*. The method takes an initial box I and an n-dimensional cross that fits exactly into I. For the midpoint and each cross tip, we start a simulation and compute the quality estimate f. If f attains an optimum in some cross tip, we move the cross to this cross tip and continue. Otherwise, we halve the size of the cross and continue. The compass method terminates when either the number of cross shrinkings or of cross moves has exceeded a threshold.

We cancel simulations when an unsafe state is hit and thus we have found an error trajectory. Moreover, we cancel a simulation when the quality estimate has not improved for *sim_cnc* steps. There is of course the risk that a simulation is cancelled too early.

	our algorithm					naïve algorithm		
Example	time	ref.	sim.	sim. steps	jumps	time	sim.	sim. steps
convoi	0.5	0	1	7	0	∞	∞	∞
eco *sim_cnc*=400	0.1	0	1	328	2	0.1	1	313
eco	2.8	10	87	29027	2	0.1	1	313
focus	0.1	0	9	2312	0	0.04	1	131
focus *sim_cnc*=20	33.9	434	319	14176	0	0.04	1	131
parabola *sim_cnc*=105	0.0	0	1	201	0	∞	∞	∞
parabola *sim_cnc*=30	18.3	353	113	7665	0	∞	∞	∞

Table 1. Experiments

This risk is countered by the fact that our abstraction is refined over time so that the quality estimate will eventually be faithful enough to tell whether a simulation is "really" improving.

4 Implementation and Experiments

We ran experiments on modifications of various well-known benchmarks from the literature, see `http://hsolver.sourceforge.net/benchmarks/falsification`. Since the benchmarks were mostly safe, we injected an error into those systems.

Table 1 shows the results for a small selection of benchmarks. The table shows the runtime in seconds, the number of abstraction refinements, simulations, the total number of single simulation steps, and the number of jumps of the trajectory that was found, and some figures for the *naïve* algorithm of Sec. 3.

The naïve algorithm performs very well on some apparently easy examples, where the method we propose here also performs well, but on numerous examples it does not terminate within several hours, indicated by ∞.

We did an experiment with `focus` showing that even for a too small value of *sim_cnc*, simulations will eventually "survive" long enough thanks to the refinement of the quality function. The example is extremely easy for HSolver, provided *sim_cnc* is not too small, but hard otherwise. The same effect occurred for `eco`. We have also created an example where we isolate the aspect just mentioned: `parabola`. In this example, the error trajectory looked for is an extremely tight parabola, i.e., at the beginning, one must move away from the unsafe state. If *sim_cnc* is too small and the quality function is not faithful enough yet, then the simulations will be cancelled prematurely.

References

1. A. Bhatia and E. Frazzoli. Incremental search methods for reachability analysis of continuous and hybrid systems. In Rajeev Alur and George J. Pappas, editors, *HSCC'04*, number 2993 in LNCS. Springer, 2004.
2. Tamara G. Kolda, Robert Michael Lewis, and Virginia Torczon. Optimization by direct search: New perspectives on some classical and modern methods. *SIAM Review*, 45(3):385–482, 2003.
3. Erion Plaku, Lydia Kavraki, and Moshe Vardi. Hybrid systems: From verification to falsification. In Holger Hermanns and Werner Damm, editors, *Proceedings of the 19th International Conference on Computer Aided Verification*, volume 4590 of *LNCS*, pages 463–476. Springer-Verlag, 2007.
4. Stefan Ratschan and Zhikun She. Safety verification of hybrid systems by constraint propagation based abstraction refinement. *ACM Transactions in Embedded Computing Systems*, 6(1), 2007.
5. Stuart J. Russell and Peter Norvig. *Artificial Intelligence: a Modern Approach.* Series in artificial intelligence. Prentice Hall, 2003.
6. Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning.* The MIT Press, 1998.

Model-Based Testing of a Web-Based Positioning Application

Rivo Roo[1] and Juhan Ernits[2]

[1] Reach-U, Tartu, Estonia
rivo.roo@reach-u.com
[2] Inst. of Cybernetics / Dept. of Comp. Sci.,
Tallinn University of Technology, Tallinn, Estonia
juhan@cc.ioc.ee

Extended abstract

1 Introduction

We present an experience report of applying model-based testing on a component of a distributed web application - a web-based positioning system called WorkForce Management (WFM). The purpose of the system is to allow subscribers to track the position of their employees for the purpose of, for example, improving the practice of a courier service. The system consists of a number of components that involve the positioning system, the bank and the mobile phone operator.

For modeling we use *model programs*. Model programs is a useful formalism for the modeling of software and for design analysis. Model programs are used as the foundation of tools such as Spec Explorer [3] and NModel [1, 2].

We built the models and the system adapter for model-based testing of WFM in C# using the NModel toolkit.

2 Positioning System WFM

WFM is a system that enables to track the geographical position of employees, send them messages and look at the history of their movement. The system is based on mobile positioning service provided by a mobile service provider.

Application of model-based testing to WFM is motivated by the need to test various aspects of the system and to evaluate the usefulness of the current state of the art of model-based testing for industrial application.

The architecture of the WFM system is outlined in Fig. 1. The web server talks to the backend, which in turn communicates with the billing system, the actual positioning system, and logs the procedures to the history subsystem.

The functional specification was given in terms of causal relationships between different messages on different ports in the system. For the purposes of model-based testing, we modelled the typical positioning scenario, where the operator logs in, selects some cell phone numbers to position and expects the results. We control the system from the web interface, indicated as *web* in Fig. 1. In addition we observe messages on the *web*, *billing*, and *history* interfaces.

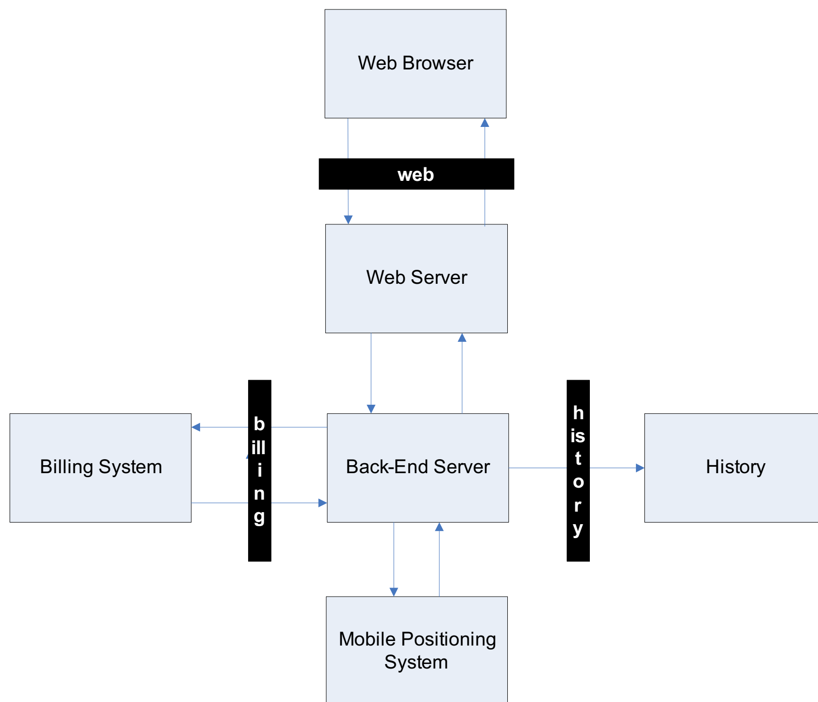


Fig. 1. Architecture of the WorkForce Management system.

3 Results

We modeled the system using NModel model programs. The system was split up into different logical features like `Login`, `Positioning`, `BillingAndHistory`. The interface between the model and the actual system, called *stepper* in NModel, was built using .Net library functions. Observable functions were realized as web based callbacks to a web server embedded into the stepper.

WFM was quite thoroughly tested already before model-based testing was applied on the system. Still, model-based testing revealed some errors in both WFM, and also a minor error in the NModel toolkit.

In WFM we experienced unexpected behaviour when in a scenario where users tried logging in and out to the system very frequently. We also found that in some cases the positioning results were not received by the end user. These errors were reported to the development team and fixed.

In NModel toolkit, the conformance tester behaved unexpectedly with a certain combination of command line parameters.

The ongoing work involves testing the behaviour of the WFM system in situations where the system is shut down and started.

4 Summary

The WFM case study is an excellent example where model-based testing yields useful results. Still, it also demonstrated that a significant amount of time was spent on learning the modeling formalism and last but not least building the test harness, i.e. the adapter between the model and the system under test.

Acknowledgements

We thank Toivo Vajakas for his help and support. This work was supported by the ELIKO Competence Center. In addition, Juhan Ernits was partially supported by the ITEA-2 D-MINT project and the Estonian Science Foundation grant No. 7667.

References

1. J. Jacky, M. Veanes, C. Campbell, and W. Schulte. *Model-based Software Testing and Analysis with C#.* Cambridge University Press, 2007.
2. NModel. NModel web site, 2008. http://www.codeplex.com/NModel.
3. 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. Preliminary version of a book chapter in the forthcoming text book *Formal Methods and Testing*.

A Category-Theoretical Approach to the Formalisation of Version Control in MDE

Adrian Rutle[1], Alessandro Rossini[2], Yngve Lamo[1] and Uwe Wolter[2]
[1]Faculty of Engineering, Bergen University College, Norway
[2]Department of Informatics, University of Bergen, Norway

ABSTRACT

In Model-Driven Engineering models are the primary artefacts of the software development process. Similar to other software artefacts, models undergo a complex evolution during their life cycles. Version control is one of the key techniques which enables developers to tackle this complexity. Traditional version control systems are based on the copy-modify-merge paradigm which is not fully exploited in MDE because of the lack of model-specific techniques. In this paper we give a formalisation of the copy-modify-merge paradigm in MDE. In particular, we analyse how common models and merge models can be defined by means of category-theoretical constructions. Moreover, we show how the properties of those constructions can be used to identify model differences and conflicting modifications.

1. INTRODUCTION AND MOTIVATION

Software models are abstract representations of software systems. These models are used to tackle the complexity of present-day software by enabling developers to reason at a higher level of abstraction. In Model-Driven Engineering (MDE) models are first-class entities of the software development process and undergo a complex evolution during their life-cycles. Therefore, the need for techniques and tools to support model evolution activities such as version control is increasingly growing. Present-day MDE tools offer a limited support for version control of models. Typically, the problem is addressed using a *lock-modify-unlock* paradigm, where a repository allows only one developer to work on an artefact at a time. This approach is workable if the developers know who is planning to do what at any given time and can communicate with each other quickly. However, if the development group becomes too large or too spread out, dealing with locking issues might become a hassle.

On the contrary, traditional version control systems such as Subversion enable efficient concurrent development of source code. These systems are based on the *copy-modify-merge* paradigm. In this approach each developer accesses a repository and creates a local working copy – a snapshot of the repository's files and directories. Then, the developers modify their local copies simultaneously and independently. Finally, the local modifications are merged into the repository. The version control system assists with the merging by detecting conflicting changes. When a conflict is detected, the system requires manual intervention of the developer.

Unfortunately, traditional version control systems are focused on the management of text-based files, such as source code. That is, difference calculation, conflict detection, and source code merge are based on a per-line textual comparison. Since the structure of models is graph-based rather than text- or tree-based [1], the existing techniques are not suitable for MDE.

Research has lead to various outcomes related to model evolution during the last years: [2] for the difference calculation, [3] for the difference representation and [4] for the conflict detection, to cite a few. However, the proposed solutions are not formalised enough to enable automatic reasoning about model evolution. For example, operations such as *add*, *delete*, *rename* and *move* are given different semantics in different works/tools. In addition, concepts such as *synchronisation*, *commit* and *merge* are only defined semiformally. Moreover, the terminology is not precise and unique, e.g. the terms "create", "add" and "insert" are frequently used to refer to the same operations.

Our claim is that the adoption of the copy-modify-merge paradigm is necessary to enable effective version control in MDE. This adoption requires formal techniques which are targeting graph-based structures. The goal of this paper is the formalisation of the copy-modify-merge paradigm in MDE. In particular, we show that common models and merge models can be defined as pullback and pushout, respectively. To achieve this, we use the Diagram Predicate Framework (DPF)[1] [5, 6, 7] which provides a formal approach to modelling based on category theory – the mathematics of graph-based structures. In addition, DPF enables us to define a language to represent model differences and a logic to detect conflicting modifications.

2. VERSION CONTROL IN MDE

First we start with an example to present a usual scenario of concurrent development in MDE. The example is obviously simplified and only the details which are relevant for our discussion are presented. Then, common models, merge models and their computations are analysed in the subsequent sections.

Suppose that two software developers, Alice and Bob, use a version control system based on the copy-modify-merge paradigm. The scenario is depicted in Fig. 1.

Alice checks out a local copy of the model V_1 from the repository and modifies it to V_{1A}, where 1 is a version number and A stands for Alice. This modification takes place in the *evolution step* e_{1A}. Since the model in the repository may have been updated in the mean time, she needs to synchronise her model with the repository in order to integrate

[1]Formerly named Diagrammatic Predicate Logic (DPL).

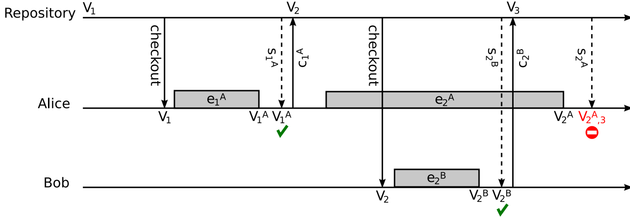


Figure 1: The scenario for the example

her local copy with other developers' modifications. This is done in the *synchronisation* s_{1^A}. However, no modifications of the model V_1 has taken place in the repository while Alice was working on it. Therefore, the synchronisation completes without changing the local copy V_{1^A}. Finally, Alice commits the local copy, which will be labelled V_2 in the repository. This is done in the *commit* c_{1^A}.

Afterwards, Bob checks out a local copy of the model V_2 from the same repository and modifies it to V_{2^B}. Then, he synchronises his model with the repository. Again, the synchronisation completes without changing the local copy V_{2^B}. Finally, Bob commits the local copy, which will be labelled V_3 in the repository.

Alice continues working on her local copy, which is still V_2. This model is not synchronised with the repository which contains Bob's modifications. She synchronises her model with the repository where the last model is V_3. Therefore, the synchronisation computes the *merge model* $V_{2^B,3}$. Now, the version control system may report a conflict in the merge model which forbids the commit c_{2^B}. The resolution of the conflict requires the manual intervention of Alice, who must review the model and decide to adapt it to Bob's modifications, or, adapt Bob's modifications to her own model.

2.1 Common Model

When Alice changes her local copy from V_2 to V_{2^A}, her development environment must keep track over what is common between the two models. The identification of what is common is the same as the identification of what is not modified, which should be feasible to implement in any tool.

Every two model elements which correspond to each other can be identified in a *common model*. For example, the model $C_{2^A,3}$ is the common model of the models V_{2^A} and V_3. The usage of a common model makes the construction of merge models at synchronisation step easier (explained in Sec. 2.2). In some frameworks, what is common between two models is defined implicitly by stating that structurally equivalent elements imply that the elements are equal (*soft-linking*). This approach has the benefit of being general, but its current implementations are too resource greedy to be used in production environment. In other frameworks, elements with equal identifiers are seen as equal elements (*hard-linking*). Unfortunately, this approach is tool-dependent, since the element identification is different for every environment. Our claim is that "recording" which elements are kept unmodified during an evolution step addresses the problems of the soft- and hard-linking approaches. That is, these equalities are specified explicitly in common models as in the following definition.

Definition 1. A model C_{i,i^U} together with the injective morphism inj_{V_i} and the inclusion morphism $inc_{V_{i^U}}$ is a common model for V_i and V_{i^U}.

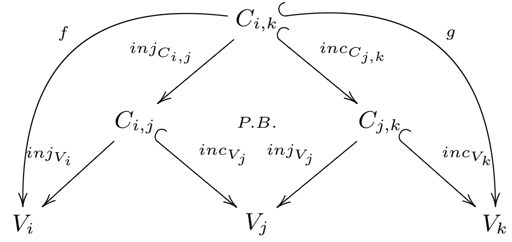


Figure 2: Common models: $C_{i,j}$ and $C_{j,k}$; and the composition: $C_{i,k}$

In order to find the common model between two models which are not subsequent versions of each other, i.e. for which we do not have a direct common model, we can construct the common model by the composition of the common models of their intermediate models. We call this common model for the *composition of commons* or the *normal form*. A possible way to compute this common model is as follows (see Fig. 2):

Proposition 1. Given the diagrams $V_i \xleftarrow{inj_{V_i}} C_{i,j} \xhookrightarrow{inc_{V_j}} V_j$ and $V_j \xleftarrow{inj_{V_j}} C_{j,k} \xhookrightarrow{inc_{V_k}} V_k$, for $j = i+1$ and $k = j+1$, the common model for V_i and V_k is $C_{i,k}$ with the two morphisms f and g where $f = inj_{C_{i,j}}; inj_{V_i}$, $g = inc_{C_{j,k}}; inc_{V_k}$, and, $C_{i,k}$ is the pullback $(C_{i,k},\ inj_{C_{i,j}} : C_{i,k} \to C_{i,j},\ inc_{C_{j,k}} : C_{i,k} \to C_{j,k})$ of the diagram $C_{i,j} \xhookrightarrow{inc_{V_j}} V_j \xleftarrow{inj_{V_j}} C_{j,k}$ such that $inc_{C_{j,k}}$ is an inclusion.

2.2 Merge Model

Recall that when Alice wanted to commit her local copy V_{2^A} to the repository, she had to first synchronise it with the repository. In the synchronisation s_{2^A}, a merge model $V_{2^A,3}$ was created. The merge model must contain the information which is needed to distinguish which model elements come from which model. But this is exactly one of the properties of pushout; therefore, we use pushout construction to compute merge models, as stated in the next proposition. The properties of the pushout are then used to decorate merge models such that added, deleted, renamed and moved elements are distinguished (explained in Sec. 2.4).

Proposition 2. Given the models V_i, V_j and $C_{i,j}$, the merge model $V_{i,j}$ is the pushout $(V_{i,j},\ m_i : V_i \to V_{i,j},\ m_j : V_j \to V_{i,j})$ of the diagram $V_i \xleftarrow{inj_{V_i}} C_{i,j} \xhookrightarrow{inc_{V_j}} V_j$ such that m_j is an inclusion.

2.3 Synchronisation and Commit

Fig. 3 outlines synchronisation and commit operations in the copy-modify-merge paradigm. These operations are defined as follows. In Fig. 3 and in the following definitions and propositions, U stands for a username.

Definition 2. Given the local copy V_{i^U}, the last model in the repository V_j and their merge model $V_{i^U,j}$, the synchronisation $s_{i^U} : (V_{i^U}, V_j) \to V_{j^U}$ is an operation which generates a synchronised local copy V_{j^U} such that
$V_{j^U} := \begin{cases} V_{i^U} & \text{if } i = j; \\ V_{i^U,j} & \text{if } i < j, \text{ and } V_{i^U,j} \notin \mathcal{C}^U \end{cases}$ where $\mathcal{C}^U$ is the set of conflicting merge models.

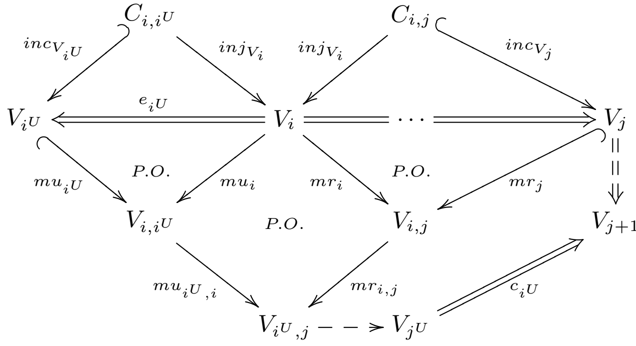


Figure 3: Synchronisation and Commit

Definition 3. Given the synchronisation $s_{i^U} : (V_{i^U}, V_j) \rightarrow V_{j^U}$, the commit $c_{i^U} : V_{j^U} \Rightarrow V_{j+1}$ is an operation which adds the model V_{j^U} to the repository as V_{j+1}.

Whenever a local copy V_{i^U} is synchronised with a model V_j from the repository, if the version numbers are the same, i.e. $i = j$, then a *synchronised* local copy V_{j^U} will be created such that $V_{j^U} = V_{i^U}$. However, if $i < j$, then a merge model $V_{i^U,j}$ will be created such that $V_{j^U} = V_{i^U,j}$ only if $V_{i^U,j}$ is not in a conflict state, i.e. $V_{i^U,j} \notin \mathcal{C}^U$. Finally, the commit operation will add the synchronised local copy V_{j^U} to the repository and will label it V_{j+1}. The next procedure explains the details of our approach to the synchronisation and commit operation.

Given the models V_{i^U}, V_i, C_{i,i^U} and V_j, where $i < j$, the synchronisation $s_{i^U} : (V_{i^U}, V_j) \rightarrow V_{j^U}$ is computed as follows:

1. compute the merge model V_{i,i^U}
2. compute the common model $C_{i,j}$
3. compute the merge model $V_{i,j}$
4. compute the merge model ($V_{i^U,j}$, $mu_{i^U,i}$, $mr_{i,j}$) as the pushout of $V_{i,i^U} \xleftarrow{mu_i} V_i \xrightarrow{mr_i} V_{i,j}$
5. $V_{j^U} := V_{i^U,j}$ only if $V_{i^U,j} \notin \mathcal{C}^U$

2.4 Difference and Conflict

As mentioned, during a synchronisation operation $s_{i^U} : (V_{i^U}, V_j) \rightarrow V_{j^U}$ where $i < j$, the merge model $V_{i^U,j}$ may contain conflicts. To detect these conflicts, we need a way to identify the differences between V_{i^U} and V_j, i.e. the modifications which has occurred in the evolution step(s). Difference identification in the merge model $V_{i^U,j}$ can be done by distinguishing common elements, V_{i^U}-elements and V_j-elements from each other. However, since this is one of the properties of merge models, we only need a language to express the differences.

Since models are graph-based, we need a diagrammatic language to tag the model elements as *common*, *added*, *deleted*, *renamed* and *moved*. Therefore, we use DPF to define such a diagrammatic language. In DPF each modelling language L corresponds to a diagrammatic signature Σ_L and a metamodel MM_L. L-models are represented by Σ_L-specifications which consist of a graph and a set of constraints. The graph represents the structure of the model, and predicates from Σ_L are used to add the constraints to the graph [7].

We define a signature Σ_Δ for our language. Σ_Δ consists of five predicates: `[common]`, `[add]`, `[delete]`, `[rename]`, and `[move]`. The merge models will be decorated by predicates from the signature Σ_Δ in addition to the predicates from Σ_L. For example, an element in V_{i,i^U} will be tagged with `[add]` if it does not exist in V_i, with `[delete]` if it does not exist in V_{i^U}, and with `[common]` if it exists in both.

We have also developed a logic for Σ_Δ which is used for two main purposes: to obtain the synchronised local copy V_{j^U} from $V_{i^U,j}$; and to detect conflicts in the decorated merge model $V_{i^U,j}$. The synchronised local copy is obtained by interpreting the predicates as operations, e.g. if an element is tagged with the predicate `[delete]`, it will not exist in V_{j^U}. The detection of conflicts is done by checking the tagged elements against some predefined set of conflicting modifications, e.g. if an element is tagged with `[rename]` twice, it is in conflict. Although conflicts are context-dependent, we have recognised some situations where syntactic conflicts will arise. The following is a summary of the concurrent modifications which we identify as conflicts:

- adding structure to an element which has been deleted
- renaming an element which has been renamed
- moving an element which has been moved

3. SUMMARY AND FUTURE WORK

The copy-modify-merge paradigm is proven to be an effective solution to tackle the complexity of version control of text-based artefacts. In this paper, we have shown how this paradigm can be exploited for version control of graph-based structures such as software models. We have formalised the concepts of the copy-modify-merge paradigm in MDE, by means of pullback and pushout constructions. In addition, we have defined a proof-of-concept diagrammatic language for the representation of model differences, and a logic for the detection of syntactic conflicts.

In a future work, we will consider semantic conflicts. Moreover, a prototype implementation will be developed to verify the efficiency of the proposed techniques.

4. REFERENCES

[1] L. Baresi and R. Heckel. Tutorial Introduction to Graph Transformation: A Software Engineering Perspective. In *ICGT 2004*, volume 3256 of *LNCS*, pages 431–433. Springer, 2004.

[2] C. Brun, J. Musset, and A. Toulmé. *EMF Compare.* http://www.eclipse.org/emft/projects/compare/.

[3] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. A Metamodel Independent Approach to Difference Representation. *Journal of Object Technology*, 6(9):165–185, October 2007.

[4] T. Mens, G. Taentzer, and O. Runge. Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. *ENTCS*, 127(3):113–128, 2005.

[5] A. Rutle, U. Wolter, and Y. Lamo. Diagrammatic Software Specifications. In *NWPT'06*, October 2006.

[6] A. Rutle, U. Wolter, and Y. Lamo. A Diagrammatic Approach to Model Transformations. In *EATIS 2008*, 2008.

[7] A. Rutle, U. Wolter, and Y. Lamo. A Formal Approach to Modeling and Model Transformations in Software Engineering. Technical Report 48, Turku Centre for Computer Science, Finland, 2008.

Automatic Definition of Model Transformations at Instance Level

Adrian Rutle[1], Alessandro Rossini[2], Yngve Lamo[1] and Uwe Wolter[2]
[1]Faculty of Engineering, Bergen University College, Norway
[2]Department of Informatics, University of Bergen, Norway

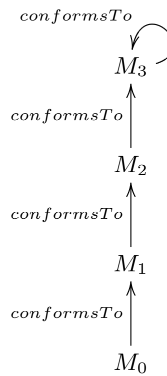
ABSTRACT

A model transformation is the generation of a target model from a source model. Usually, it is defined for metamodels, i.e. models at the meta-level, and executed by a transformation engine to transform instances of those metamodels. In some cases, it is also desired to transform the instances of the transformed models. In this paper, we use the Diagram Predicate Framework to show how model transformations which are defined at the metamodel level can be used as guidelines to automatically define model transformations at the model level. This requires a special relationship between the metamodel and the instances of its instances in order to inherit all the properties of the transformations from the metamodel level. A formalisation of this relationship is outlined in this paper.

1. INTRODUCTION AND MOTIVATION

Software models are abstract representations of software systems. These models are used to tackle the complexity of present-day software systems by enabling developers to reason at a higher level of abstraction. In Model-Driven Engineering (MDE), we refer to four levels of abstraction, which are summarized in the following *modelling hierarchy*:

Models at the M_0-level are called *instances* which represent the running system. These instances must conform to *models* at the M_1-level, which are structures specifying what instances should look like. These models, in their turn, must conform to a *metamodel* at the M_2-level, against which models can be checked for validity. Metamodels correspond to modelling languages, for example the Unified Modeling Language (UML) and Common Warehouse Model (CWM). The highest level of abstraction (as defined by the Object Management Group [2]) is the M_3-level. A model at this level is often called *meta-metamodel*; it conforms to itself and it is used to describe metamodels.



In MDE models are the primary artefacts of the development process. Model transformations play a central role and have many applications in MDE, such as; model integration, model refinement, model evolution, multi-modelling, and code-generation, to mention a few. These transformations are defined at the metamodel level, and executed at the model level. For example, when we want to transform a Java object, we define the transformation for its class. In the same way, when we want to transform a Java class, we define the transformation for the meta-type of Java classes, which is the class `Class`.

An advantage of using the metamodel level is to enable the definition of transformations at a higher level of abstraction. Another advantage is to work with a smaller set of model elements. This is because metamodels are typically smaller and more compact than their instances.

The focus of this paper will be on the automatic definition of transformations at the model level based on the transformation definition at the metamodel level. This kind of automatisation is investigated algebraically using pullbacks [1], however, our approach is more generic and is not bound to a specific algebraic operation. A requirement for the automatisation is the existence of a special relationship between metamodels and the instances of their instances. A short description of this relationship is given in Section 2.

2. DIAGRAM PREDICATE FRAMEWORK

Diagram Predicate Framework (DPF)[1] [3, 4, 5] provides a formal approach to modelling based on category theory – the mathematics of graph-based structures. In DPF each modelling language L corresponds to a diagrammatic signature Σ_L and a metamodel MM_L. L-models are represented by Σ_L-specifications[2] which consist of a graph and a set of constraints. The graph represents the structure of the model, and predicates from Σ_L are used to add the constraints to the graph [5]. Signatures, constraints and diagrammatic specifications are defined as follows:

Definition 1. A (diagrammatic predicate) signature $\Sigma := (\Pi, \alpha)$ is an abstract structure consisting of a collection of predicate symbols Π with a mapping that assigns an arity (graph) $\alpha(p)$ to each predicate symbol $p \in \Pi$.

Definition 2. A constraint (p, δ) in a graph $G(M)$ is given by a predicate symbol p and a graph homomorphism $\delta : \alpha(p) \to G(M)$, where $\alpha(p)$ is the arity of p.

Definition 3. A Σ-specification $M := (G(M), M(\Pi))$, is a graph $G(M)$ with a set $M(\Pi)$ of constraints (p, δ) in G(M) with $p \in \Pi$.

Fig. 1a shows an example of a Σ-specification $M = (G(M), M(\Pi))$. $G(M)$ in Fig. 1b is the graph of M without any

[1]Formerly named Diagrammatic Predicate Logic (DPL).
[2]For the rest of the paper we use the terms "model" and "diagrammatic specification" interchangeably.

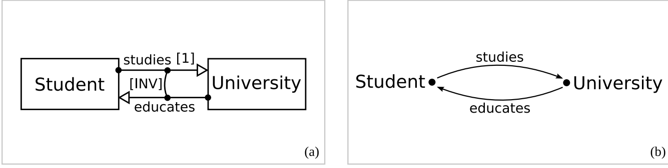


Figure 1: A Diagrammatic Specification: (a) $M = (G(M), M(\Pi))$, (b) its graph $G(M)$.

constraints on it. In M, every university educates *one or more* students; this is forced by the constraint $([total], \delta_1)$ on the arrow *educates*, which is visualised as a filled circle at the beginning of the arrow. Moreover, every student studies at *exactly one* university; this is forced by the constraint $([single\text{-}valued], \delta_2)$ on the arrow *studies*, which is visualised as the marker [1] at the cap of the arrow.

A signature with details of the semantics of its predicates is shown in [5]. Here, only an informal definition of the concept of instances of diagrammatic specifications is given. An instance (ι_M, I) of a diagrammatic specification M is a graph homomorphism $\iota_M : I \to G(M)$, i.e. the elements of I are typed by $G(M)$, the graph of M, such that the constraints in M are satisfied. The set of the instances of a diagrammatic specification M is denoted $Inst(M)$.

As mentioned, in a modelling hierarchy each model at M_i-level conforms to a model at M_{i+1}-level, for $0 \leq i < 3$. In a modelling hierarchy with transitive conformance, if an instance I at M_0-level conforms to a model M at M_1-level, and M in its turn conforms to a metamodel MM at M_2-level, then I also conforms to MM. This property is formalised in the next definition; its importance for the automatic definition of model transformations at the model level based on transformation definition at the metamodel level is explained in Sec. 3.

Definition 4. In a modelling hierarchy with transitive conformance, given any diagrammatic specification MM, for each $(\iota_{MM}, M) \in Inst(MM)$ if $(\iota_M, I) \in Inst(M)$ then $(\iota_M; \iota_{MM}, I) \in Inst(MM)$.

3. MODEL TRANSFORMATIONS

As an example, suppose we want to generate a relational database model from a class model. The class model contains a hierarchy of classes with references between them. The relational database model contains a set of database tables and relationships between them. We can easily develop a model transformation to transform the elements of a specific class model to a database model. Alternatively, we can define a generic transformation which can be used to transform any class model to a database model. Obviously the former procedure is less convenient since it implies that for every class model we have to define a new transformation. To achieve the latter, we define the transformation for the metamodels of class models and database models. Then we use a transformation engine to (automatically) transform any model that conforms to the metamodel of class models, to a model which conforms to the metamodel of relational databases.

Moreover, in some cases, we want also to transform instances of these class models. Thus we have to define a new model transformation between the models themselves. The

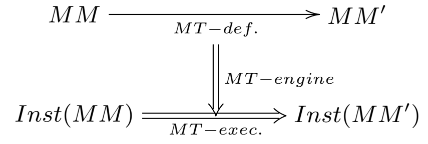


Figure 2: Model transformations: overview.

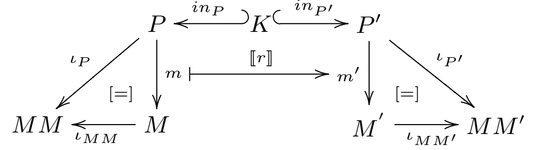


Figure 3: A transformation rule $r : P \to P'$.

results of this paper enables us to reuse the model transformation which was defined at the metamodel level to define the model transformation at the model level. This is outlined in Sec. 3.1. But first, we have to explain some concepts which are used in model transformations.

Fig. 2 shows the overview of model transformations. Each model transformation is given by a transformation definition *MT-def* which describes how instances of a source model MM can be transformed to instances of a target model MM'. The transformation definition is specified in a transformation language, and is executed by a transformation engine. That is, given a model transformation definition, a source model, and a target model, the transformation engine tries to create an instance of the target model for each instance of the source model.

Every transformation definition consists of a set of transformation rules. Each rule defines which elements of the source instances are to be transformed to which elements of the target instances. This is done through an input pattern P and an output pattern P', as shown in Fig. 3. Thus the transformation engine generates a match of the output pattern in the target instance whenever it finds a match of the input pattern in the source instance. Details of how the transformation engine is controlled and how patterns are coordinated (through the coordination set K) are given in [4]. Here, patterns and matches of patterns are defined as follows:

Definition 5. A pattern P over a diagrammatic specification MM is an instance (ι_P, P) of MM, where the items in P are formal parameters or variables.

Definition 6. A match $m : P \to M$ of a pattern P in an instance (ι_{MM}, M) of a diagrammatic specification MM is a graph homomorphism $m : P \to M$ where variables in P are assigned values from M, and $m; \iota_{MM} = \iota_P$.

For a match $m : P \to M$, the analogy is that P is the formal parameter of m and some part of M is the actual parameter. That is, a pattern P can be seen as a scheme and a match m assigns values from M to the variables in P. We use $Match(P)$ to denote the set of all matches of P in $Inst(MM)$, i.e. all matches of P in all instances (ι_{MM}, M) of MM. Moreover, we use $Match^M(P)$ for $M \in Inst(MM)$ to denote the set of all matches of the pattern P in M.

Recall that each transformation definition consists of a set of transformation rules. These rules and their semantics are defined as follows:

Definition 7. A transformation rule r is declared by $r : (\iota_P, MM) \to (\iota_{P'}, MM')$ (abbreviated $r : P \to P'$) where both P and P' are patterns over the models MM and MM', respectively.

Definition 8. The semantics of a transformation rule $r : P \to P'$ is a mapping $[\![r]\!]$ which assigns to each match $m \in Match(P)$ a match $m' \in Match(P')$, i.e. $[\![r]\!] : Match(P) \to Match(P')$.

3.1 Automatisation of Transformations

An overview of the automatic construction of transformation rules at the model level is shown in Fig. 4. Based on the execution of each transformation rule $r : P \to P'$ at the metamodel level, a (set of) transformation rule(s) r^* at the model level will be created. These transformation rules use $\mathcal{P}^M$, where $M \in Inst(MM)$ and $Match^M(P) \neq \emptyset$, as input patterns; and $\mathcal{P}^{M'}$, where $M' \in Inst(MM')$ and $Match^{M'}(P') \neq \emptyset$, as output patterns. These patterns are defined as follows:

Definition 9. $\mathcal{P}^M$ is the set of input patterns such that $\forall m(P) \in Match^M(P) : \exists P^* \in \mathcal{P}^M$, and $\mathcal{P}^{M'}$ is the set of input patterns such that $\forall m'(P') \in Match^{M'}(P') : \exists P^{*'} \in \mathcal{P}^{M'}$.

The input and output patterns in $\mathcal{P}^M$ and $\mathcal{P}^{M'}$ are created by adding a *free variable* to the matches of the patterns of r. In Fig. 5, these constructions are abbreviated as $f : M \Rightarrow \mathcal{P}^M$ and $g : M' \Rightarrow \mathcal{P}^{M'}$. For example, if a transformation rule r takes the pattern $P =$ `x:Class` as input and the pattern $P' =$ `x:Table` as output, then for the input matches `Student:Class`[3] and `University:Class` in the model M, the construction $f^* : Match^M(P) \to \mathcal{P}^M$ will create x_1`:Student:Class` and x_2`:University:Class` as input patterns P_1^* and P_2^* for the rules r_1^* and r_2^*, respectively. Similarly, for the output matches `Student:Table` and `University:Table` in the model M', the construction $g^* : Match^{M'}(P') \to \mathcal{P}^{M'}$ will create x_1`:Student:Table` and x_2`:University:Table` as output patterns $P_1^{*'}$ and $P_2^{*'}$ for r_1^* and r_2^*, respectively.

Definition 10. $f^* : Match^M(P) \to \mathcal{P}^M$ is a construction such that, $\forall m(P) \in Match^M(P)$, $f^*(m(P)) = P^*$ where P^* is a pattern over M. g^* is defined similarly.

Theorem 1. Given a transformation rule $r : P \to P'$ at the metamodel level, it is possible to create a set of transformation rules $\mathcal{R}^* : \mathcal{P}^M \to \mathcal{P}^{M'}$ at the model level.

Corollary 1. The semantics of a transformation rule $r_i^* \in \mathcal{R}^* : \mathcal{P}^M \to \mathcal{P}^{M'}$ is a mapping $[\![r_i^*]\!]$ which assigns to each match $m^* : P^* \in \mathcal{P}^M \to I$ a match $m^{*'} : P^{*'} \in \mathcal{P}^{M'} \to I'$, i.e. $[\![r_i^*]\!] : Match(\mathcal{P}^M) \to Match(\mathcal{P}^{M'})$.

In modelling hierarchies with transitive conformance, the application of automatically generated model transformations at the M_1-level – which are based on model transformations at the M_2-level – will generate models at the M_0-level which are instances of the target metamodel. This feature is important in order to inherit the properties of the model transformations which are defined at a higher level of abstraction. For example, if a model transformation at the M_2-level is correct, then the automatically constructed model transformation at the M_1-level will also be correct.

[3]Notice that (`Student:Class`) is a "user-friendly" notation for the assignment ($\iota_{MM} : Student \mapsto Class$).

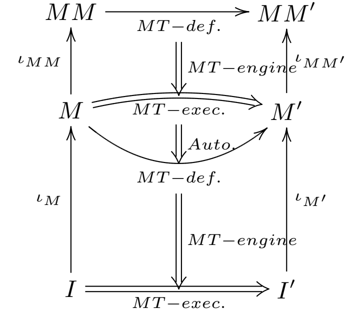


Figure 4: Model transformations: automatisation.

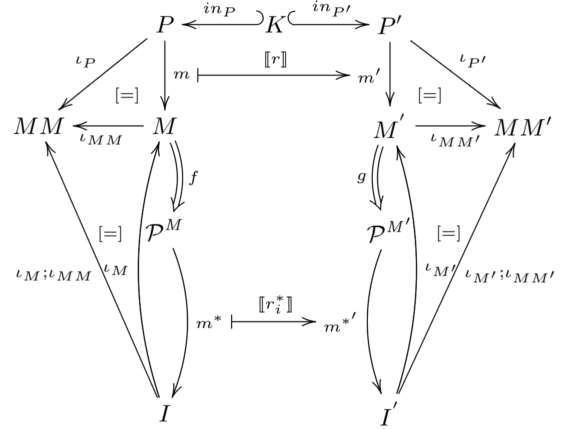


Figure 5: Creation of the transformation rules r^* based on the transformation rule r.

4. SUMMARY

Transformation rules which are defined between metamodels can be used to automatically derive transformation rules between models. Moreover, if this technique is used for modelling hierarchies with transitive conformance, the properties of the model transformations at the M_2-level are preserved in the model transformations at the M_1-level.

5. REFERENCES

[1] Z. Diskin and J. Dingel. A metamodel Independent Framework for Model Transformation: Towards Generic Model Management Patterns in Reverse Engineering. In *ATEM 2006*.

[2] Object Management Group. *Web site*, October 2008. `http://www.omg.org`.

[3] A. Rutle, U. Wolter, and Y. Lamo. Diagrammatic Software Specifications. In *NWPT'06*, October 2006.

[4] A. Rutle, U. Wolter, and Y. Lamo. A Diagrammatic Approach to Model Transformations. In *EATIS 2008*.

[5] A. Rutle, U. Wolter, and Y. Lamo. A Formal Approach to Modeling and Model Transformations in Software Engineering. Technical Report 48, Turku Centre for Computer Science, Finland, 2008.

On Operational Termination of Deterministic Conditional Rewrite Systems*

Felix Schernhammer and Bernhard Gramlich
TU Wien, Austria, {felixs,gramlich}@logic.at

Abstract. We characterize the practically important notion of operational termination of deterministic conditional term rewriting systems (DCTRSs) by context-sensitive termination of a transformed TRS on original terms. Experimental evaluations show that this new approach yields more power when verifying operational termination than existing ones. Moreover, it allows us to disprove operational termination of DCTRSs.

1 Introduction and Overview

Conditional term rewriting systems (CTRSs) are a natural extension of unconditional such systems (TRSs) allowing rules to be guarded by conditions. Conditional rules tend to be very intuitive and easy to formulate and are therefore used in several rule based programming and specification languages, such as Maude or ELAN.

Here we focus on the particularly interesting class of *deterministic* (oriented) CTRSs (DCTRSs) which allows for extra variables in conditions (corresponding to *let-constructs* or *where-clauses* in other functional-(logic) languages) and has been used for instance in proofs of termination of (well-moded) logic programs [4].

When analyzing the termination behaviour of conditional TRSs, it turns out that the proof-theoretic notion of *operational termination* is more adequate than ordinary termination in the sense that practical evaluations w.r.t. operationally terminating DCTRSs always terminate (which is indeed not true for other similar notions like *effective termination* [5]).

We propose the notion of context-sensitive quasi-reductivity ([6]), that will be proved to be equivalent to operational termination of DCTRSs. Furthermore, we use a simple modification of *unraveling* transformations ([7], [9]) that allows us to completely characterize the new property of context-sensitive quasi-reductivity of a DCTRS by means of termination of a context-sensitive (unconditional) TRS *on original terms*.

In the following, we assume familiarity with the basic concepts and notations of (context-free, context-sensitive and conditional) term rewriting (cf. e.g. [2], [6], [9]). In this work we are exclusively concerned with deterministic conditional term rewrite systems (DCTRSs).

2 Proving operational termination of DCTRSs via context-sensitive quasi-reductivity

The main goal of this work is to provide methods for proving *operational termination* of DCTRSs. For that purpose we will now introduce the concept of *context-sensitive quasi-reductivity* which is equivalent to *operational termination* while being practically easier to verify for given systems.

Definition 1. *A DCTRS $\mathcal{R}$ ($\mathcal{R} = (\Sigma, R)$) is called* context-sensitively quasi-reductive *(*cs-quasi-reductive*) if there is an extension of the signature Σ' ($\Sigma' \supseteq \Sigma$), a replacement*

* Preliminary results of our approach have been presented at WST'07 (Paris, France, 2007).

μ (s.t. $\mu(f) = \{1, ..., ar(f)\}$ for all $f \in \Sigma$) and a μ-monotonic, well-founded partial order $\succ_\mu$ on $\mathcal{T}(\Sigma', V)$ satisfying for every rule $l \to r \Leftarrow s_1 \to^* t_1, ..., s_n \to^* t_n$, every $\sigma\colon V \to \mathcal{T}(\Sigma, V)$ and every $i \in \{0, ..., n-1\}$:[1]

- *If $\sigma s_j \succeq_\mu \sigma t_j$ for every $1 \leq j \leq i$ then $\sigma s_{i+1} \prec_\mu^{st} \sigma l$.*
- *If $\sigma s_j \succeq_\mu \sigma t_j$ for every $1 \leq j \leq n$ then $\sigma r \prec_\mu \sigma l$.*

The ordering $\prec_\mu^{st}$ is defined as $(\prec_\mu \cup \lhd_\mu)^+$ where $t \lhd_\mu s$ if and only if s is a proper subterm of t at some position $p \in Pos^\mu(t)$.

Now, we define a transformation from DCTRSs into CSRSs, such that μ-termination of the transformed CSRS implies cs-quasi-reductivity of the original DCTRS. The transformation is actually a variant of the one in [9].

Definition 2. **[9]** *Let $\mathcal{R} = (\Sigma, R)$ be a DCTRS. For every rule $\alpha : l \to r \Leftarrow s_1 \to^* t_1, ..., s_n \to^* t_n$ we use n new functions symbols U_i^α ($i \in \{1, ..., n\}$). Then α is transformed into a set of unconditional rules in the following way:*

$$\begin{aligned} l &\to U_1^\alpha(s_1, Var(l)) \\ U_1^\alpha(t_1, Var(l)) &\to U_2^\alpha(s_2, Var(l), \mathcal{E}Var(t_1)) \\ &\vdots \\ U_n^\alpha(t_n, Var(l), \mathcal{E}Var(t_1), ..., \mathcal{E}Var(t_{n-1})) &\to r \end{aligned}$$

Here $Var(s)$ denotes the sequence of variables in a term s rather than the set. The set $\mathcal{E}Var(t_i)$ is $Var(t_i) \setminus (Var(l) \cup \bigcup_{j=1}^{i-1} Var(t_j))$. Again, abusing notation, by $\mathcal{E}Var(t_i)$ we mean an arbitrary but fixed sequence of the variables in the set $\mathcal{E}Var(t_i)$. Any unconditional rule of $\mathcal{R}$ is transformed into itself. The transformed system $U_{cs}(\mathcal{R}) = ((U(\Sigma), U(R), \mu)$ is obtained by transforming each rule of $\mathcal{R}$ where $U(\Sigma)$ is Σ extended by all new function symbols. Furthermore, the replacement map μ is given by $\mu(f) = \{1\}$ if $f \in U(\Sigma) \setminus \Sigma$ and $\mu(f) = \{1, \ldots, ar(f)\}$ otherwise.

Apart from analyzing operational termination, this transformation can also be used to exactly simulate conditional derivations.

Theorem 1. *Let $\mathcal{R} = (\Sigma, R)$ be a DCTRS and $U_{cs}(\mathcal{R})$ its transformed CSRS. For every $s, t \in \mathcal{T}(\Sigma, V)$ we have $s \to_\mathcal{R}^+ t$ if and only if $s \to_{U_{cs}(\mathcal{R})}^+ t$.*

Unfortunately, and interestingly, cs-quasi-reductivity of a DCTRS $\mathcal{R}$ does not imply $\mu_{U_{cs}(\mathcal{R})}$-termination of $U_{cs}(\mathcal{R})$, cf. [9, Ex. 7.2.51]. However, it implies $\mu_{U_{cs}(\mathcal{R})}$-termination of $U_{cs}(\mathcal{R})$ *on original terms* (i.e., terms over the original signature of $\mathcal{R}$), thus allowing us to characterize cs-quasi-reductivity.

Theorem 2. *Let $\mathcal{R} = (\Sigma, R)$ be a DCTRS. The following properties of $\mathcal{R}$ are equivalent: $\mu_{U_{cs}(\mathcal{R})}$-termination of $U_{cs}(\mathcal{R})$ on $\mathcal{T}(\Sigma, V)$, cs-quasi-reductivity and operational termination.*

From a practical point of view these results yield two contributions for the analysis of operational termination of DCTRSs. First, when reducing the task of proving operational termination of DCTRSs to the task of proving (context-sensitive) termination of TRSs, it is now sufficient to prove termination *on original terms*. In order to exploit this relaxation of the termination property, we developed a dedicated method based on the dependency pair framework of [3] and [1], that allows us to prove termination of CSRSs obtained by the proposed transformation on original terms. First experimental results with a prototype implementation are promising. In particular, we were able to automatically

[1] Note that – in contrast to the definition of quasi-reductivity – the substitution maps variables only into terms over the original signature. This restriction is crucial for some of our main results.

prove termination on original terms of systems that are not terminating in the general sense.

Secondly, our results provide the basis for automatically disproving operational termination of DCTRSs, which was, to the authors' knowledge, impossible with transformational approaches before. On the other hand, as proving non-termination on original terms may be significantly harder than proving ordinary non-termination, the practical benefits of the latter results seem unclear. However, we were able to show that the proposed transformation is sound and complete with respect to *collapse-extended* termination (cf. e.g. [9]), so from (ordinary) non-termination of a transformed system we can at least conclude that the original DCTRS does not enjoy collapse-extended termination.

3 Related Work and Discussion

Our notion of cs-quasi-reductivity provides a new sufficient (in fact, equivalent) criterion for operational termination. Furthermore, cs-quasi-reductivity can be verified by proving termination of the resulting CSRS (on original terms). We have shown that the proposed transformation, which has already been discussed in [8] regarding simulation soundness and completeness and briefly in [10] regarding termination analysis, yields operational termination of strictly more DCTRSs than Ohlebusch's context-free transformation. Furthermore, we developed methods for the termination analysis that are tailored to verify termination on original terms. We implemented a prototype termination prover which, besides other well-known techniques, makes essential use of these methods. Our implementation was able to automatically prove operational termination of several DCTRSs, taken from the standard literature, for which all existing approaches failed. Finally, our work is the first to provide means for disproving operational termination.

References

1. B. Alarcón, F. Emmes, C. Fuhs, J. Giesl, R. Gutiérrez, S. Lucas, P. Scheider-Kamp and R. Thiemann. Improving context-sensitive dependency pairs. Technical Report: Aachener Informatik Berichte (AIB), 2008-13
2. F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, New York, NY, USA, 1998.
3. Jürgen Giesl, René Thiemann, Peter Schneider-Kamp and Stephan Falke. Mechanizing and Improving Dependency Pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.
4. H. Ganzinger and U. Waldmann. Termination proofs of well-moded logic programs via conditional rewrite systems. In *Proc. CTRS'92, Pont-à-Mousson, France, July 1992*, pp. 430–437, LNCS 656, Springer,, 1993.
5. S. Lucas, C. Marchè, and J. Meseguer. Operational termination of conditional term rewriting systems. *Inf. Process. Lett.*, 95(4):446–453, 2005.
6. S. Lucas. Context-sensitive computations in functional and functional logic programs. *J. of Functional and Logic Programming*, 1998(1), January 1998.
7. M. Marchiori. Unravelings and ultra-properties. In *Proc. ALP'96, Aachen*, LNCS 1139, pp. 107–121. Springer, September 1996.
8. N. Nishida, M. Sakai, and T. Sakabe. Partial inversion of constructor term rewriting systems. In J. Giesl, ed., *Proc. RTA'05, Nara, Japan, April 19-21, 2005*, LNCS 3467, pp. 264–278. Springer, 2005.
9. Enno Ohlebusch. *Advanced topics in term rewriting.* Springer-Verlag, London, UK, 2002.
10. F. Durán, S. Lucas, J. Meseguer, C. Marché, and X. Urbain. Proving operational termination of membership equational programs. *Higher-Order and Symbolic Computation*, 21:59–88, 2008.

Secure Open Networks

Neva Slani* Francisco Martins†

Introduction. This paper presents a method to reason about resource usage and the related security issues for a mobile calculus in an open network. By an open network we mean a network that is only partially known at compile time. Hence, the statical enforcement of security policies is just possible for a fragment of the network, contrasting most of the work done in the area that assumes a total knowledge of the network at compile time.

The Dπ-calculus [1] has long been used to model distributed, mobile systems. Its rich type system [1, 2] allows for the static control of resource usage policies prescribed beforehand by a network administrator. However, to enforce type safety all the network source code must be present at compile time, which is impractical for large networks, and even impossible in a realistic scenario.

By lifelike setting one usually has in mind the Internet (or mobile phone network etc.), where the system as a whole is incomprehensible, and new entities constantly emerge. It is a challenge to reason about such systems and establish desired security guarantees. Our goal is to present a model that:

1. reflects the *open* nature of large networks: only a partial perception (of network's locations and resources) is attainable, and new agents and resources can always be revealed;

2. provides mechanisms for managing *correct resource usage*, by all the agents involved, known or unknown, according to permissions prescribed by the owner.

Our work is inspired by Hennessy's and Riely's work, namely [1] and [3]. In the latter the authors ensure *safe resource access* for open networks (meaning that resources, such as printers, servers or web-services do not get spoilt after interactions), but leave open the problem of *authorised resource access* (being usage of resources according to the owner's written permissions), hinting the usage of the type system from [1] as a solution. The model we propose has several advantages over Riely's and Hennessy's work, and tends to overcome what we consider fragilities in [3]:

1. We claim that our networks are "more open" than those in [3]. In that paper static typing is performed with an explicit awareness of untyped sites, called *bad*. On the other side, *good* sites do not acquire essentially new knowledge. Communication with the previously unknown locations (for example a printer that some site might want to reveal to us) is impossible. Also, *good* sites may actually unpunished behave badly.

2. Our proposal performs less run-time type-checks. As opposed to the two-fold checks of migrating processes and communicating values, we check only once, and when possible in Necula's *proof-carrying code* style.

*University of Zagreb, Croatia, nslani@fsb.hr

†LASIGE/DI-FCUL, University of Lisbon, Portugal, fmartins@di.fc.ul.pt

3. The type system in [1] is designed in such a way that security is dispersed along the network, in the sense that resource access capabilities are released to the client sites. Further, it is not obvious how the distribution of such rights is controlled, or to which sites authorisations are granted. We leave space to a different treatment of access policies.

Concurrent model. We start by studying the openness problem in a concurrent setting, using a simplified version of the asynchronous π-calculus. This allows for a clear introduction of the problem in a setting that is rich enough to enable us to understand some of the subtleties of dealing with an open environment. We identify two distinct processes running in parallel, $[P]_\Gamma \mid Q$, that we designate as a *system*. Process P, named *secure process*, is syntactically isolated inside a separate shell and is statically typed by type environment Γ; and process Q, named *untrusted process*, is not type-checked. Both P and Q are standard π-calculus processes.

We carefully define the operational semantics by reductions ($\rightarrow$) on system, and build a model that enjoys type safety at run-time. The rules describing interactions between the secure and the untrusted processes reveal the core of the model (the same is valid for the distributed model described below).

$$\frac{\Gamma \vdash a\colon \mathsf{r}\langle\tau\rangle}{[\overline{a}\langle v\rangle \mid P]_\Gamma \mid a(x).P_1 \mid P_2 \rightarrow [P]_\Gamma \mid P_1\{v/x\} \mid P_2} \qquad \text{(R-DCOMM1)}$$

$$\frac{\Gamma \vdash a\colon \mathsf{w}\langle\tau\rangle \qquad \Gamma \sqcap \{v\colon \tau\} \text{ well-defined}}{[a(x).P \mid P_1]_\Gamma \mid \overline{a}\langle v\rangle \mid P_2 \rightarrow [P\{v/x\} \mid P_1]_{\Gamma \sqcap \{v\colon \tau\}} \mid P_2} \qquad \text{(R-DCOMM2)}$$

Rule R-DCOMM1 regulates the sending of value through a channel named a from the secure to the untrusted region. Channel a must posses read capabilities ($\Gamma \vdash a\colon \mathsf{r}\langle\tau\rangle$; it altogether means that a is a read-write channel in Γ). When an untrusted process wants to write to a channel inside a secure region, rule R-DCOMM2, then we first verify if it is allowed to write on that channel ($\Gamma \vdash a\colon \mathsf{w}\langle\tau\rangle$) and whether value v has a type consistent with what we possibly already might know about v in Γ ($\Gamma \sqcap \{v\colon \tau\}$ well-defined). If this is true than the secure process learns value v and includes it in its knowledge ($\Gamma \sqcap \{v\colon \tau\}$, where $\sqcap$ is the operation of "adding" knowledge). The types used enable control of channels (resources) usage (i.e. capabilities of reading and writing), and they are as in [1], an extension of standard Pierce and Sangiorgi types. Accordingly to the types, we define as a *run-time* the situation where the secure process is trying to read from (or write to) a channel that it is not supposed to read (or write), or even confuses a channel with something else (data). Systems, ranged over by R, are well-typed if environment Γ types P in $[P]_\Gamma \mid Q$; this is true even if $R = (\boldsymbol{\nu} n)R'$, i.e. typing rules and scope extrusion treat carefully restriction operator. The main result is

Theorem 1 (Type Safety) *Well-typed system R after reducing does not produce run-time errors.*

Distributed model. We extend the previous results to a distributed, mobile setting, choosing the Dπ as the referent calculus. The Dπ-calculus comprises a flat network N of sites running in parallel, put together using composition operator $\mid$. Sites represent (physical or logical) computational shells, hosting channels (resources). Code can be sent from one site to another, which is expressed by a migration construct in the calculus. We extend the Dπ distinguishing two kinds of sites: *secure* and *untrusted*, dividing the network (the world) in two areas, as before: the protected one and an outer, untrusted one. The intuition is that trusted sites are those that you administer and therefore possess all the (compile-time) information about their behaviour; for instance, the sites of a private wide-area network. All the other sites that you interact

with and have no means of checking their behaviour, e.g. sites from the Internet, are considered untrusted.

A secure site $k[P]_{\Gamma;\ \Delta}$ consists of the site's name k, a process P running on it, and two type environments Γ and Δ containing information (e.g. security) about sites and their resources. Set Γ contains (global, static) knowledge about secure sites, whereas set Δ contains the knowledge of a particular site instance acquired during its interaction with other sites at run-time. In the Dπ-calculus a site is unique but may appear split (in instances) across the network. An untrusted site k is of the form $k\{P\}$, having sites's name k and hosting process P. During computation secure sites may have an untrusted part resulting from the interaction with untrusted sites. This untrusted part is a kind of sandbox of the site. Therefore, network $k[P]_{\Gamma;\ \Delta} \mid k\{Q\}$ represents a site k running a secure process P and having a process Q that requires special attention, since its origin is from an untrusted location. Communication between those two is achieved by rules analogous to R-DCOMM1 and R-DCOMM2. The rest of the reductions are in Dπ-style, but including local knowledge transfer when the typed sites are sending code one to another.

A network is well-typed when all of its protected sites are typed under the same global knowledge Γ, leaving the untrusted sites untyped. Defining the run-time error as before, we obtain: as it reduces, a well-typed network never incurs in a run-time errors.

Conclusions and future work. In this paper we study resource access control for open networks. We start by investigating the problem within the π-calculus. Collecting type information we protect the typed processes of channel abuses in interaction with the untyped processes; therefore obtain type safety. The same idea is extended to a distributed model of open network, built upon the following principles:

- openness of the network characterized by the possible increment of the number of actors (locations and resources);
- explicit knowledge-guided behaviour, knowledge acquisition and transfer;
- as much as possible security verifications is performed statically, lessening dynamic type-checking.

The result is an open network model with the type system guaranteeing safe resource access. Authorised resource access component is a further work and builds on result of Martins and Vasconcelos [2]. We feel comfortable with joining the two models as we foresee that it is an adequate basis for the continuation of our work—dynamic acquisition and changing of access policies, in an open environment.

References

[1] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Journal of Information and Computation*, 173:82–120, 2002.

[2] F. Martins and V. Vasconcelos. History-based access control for distributed processes. In *Proceedings of TGC'05*, volume 3705 of *LNCS*, pages 98–115. Springer-Verlag, 2005.

[3] J. Riely and M. Hennessy. Trust and partial typing in open systems of mobile agents. In *Proceedings of POPL'99*, pages 93–104, 1999.

Parametricity for Haskell with Imprecise Error Semantics

Florian Stenger* and Janis Voigtländer

Institut für Theoretische Informatik
Technische Universität Dresden
01062 Dresden, Germany

Abstract:

Error raising, propagation, and handling in the functional programming language Haskell can be imprecise in the sense that a language implementation's choice of local evaluation order, and optimising transformations to apply, may influence which of a number of potential failure events hidden somewhere in a program is actually triggered. While this has pragmatic advantages from an implementation point of view, it also complicates the meaning of programs and thus requires extra care when reasoning about them. The proper semantic setup is one in which every erroneous value represents a whole set of potential (but not arbitrary) failure causes [PRH+99]. The associated propagation rules are somewhat askew to standard notions of program flow and value dependence. As a consequence, standard reasoning techniques are cast into doubt, and rightly so. We study this issue for one such reasoning technique, namely the derivation of (equational and inequational) free theorems from polymorphic types [Rey83,Wad89]. Continuing earlier work [JV04], we revise and extend the foundational notion of relational parametricity, as well as further material required to make it applicable. More generally, we believe that our new development and proofs help direct the way for incorporating further and other extensions and semantic features that deviate from the "naive" setting in which reasoning about Haskell programs often takes place. Let us elaborate a bit.

As is well known, functional languages come with a rich set of conceptual tools for reasoning about programs. For example, structural induction and equational reasoning tell us that the standard Haskell functions

$$\begin{array}{ll} \mathit{takeWhile} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha] & \\ \mathit{takeWhile}\ p\ [\,] = [\,] & \\ \mathit{takeWhile}\ p\ (x : y) & \\ \quad \mid p\ x = x : \mathit{takeWhile}\ p\ y & \\ \quad \mid \mathit{otherwise} = [\,] & \end{array}$$

$$\begin{array}{l} \mathit{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\ \mathit{map}\ h\ [\,] = [\,] \\ \mathit{map}\ h\ (x : y) = h\ x : \mathit{map}\ h\ y \end{array}$$

satisfy the following law for appropriately typed p, h, and l:

$$\mathit{takeWhile}\ p\ (\mathit{map}\ h\ l) = \mathit{map}\ h\ (\mathit{takeWhile}\ (p \circ h)\ l)\,. \tag{1}$$

* This author was supported by the DFG under grant VO 1512/1-1.

But programming language reality can be a tough game, leading to unexpected failures of such near-obvious laws. For example, [PRH+99] proposes a design for error handling based on a certain degree of impreciseness. The major implementations GHC and Hugs (as well as one distribution of the language Clean) have integrated this design years ago. However, the attendant semantics betrays law (1) to be wrong. An instantiation showing this is $p = \mathit{null}$, $h = \mathit{tail}$, and $l = [[i] \mid i \leftarrow [1..(\mathit{div}\ 1\ 0)]]$ (or any other immediately failing expression of type list-of-lists), where

$$\begin{array}{ll} \mathit{null} :: [\alpha] \rightarrow \mathsf{Bool} & \mathit{tail} :: [\alpha] \rightarrow [\alpha] \\ \mathit{null}\ [\,] \quad\ \ = \mathsf{True} & \mathit{tail}\ [\,] \quad\ \ = \mathit{error}\ \text{“Prelude.tail: empty list”} \\ \mathit{null}\ (x : y) = \mathsf{False} & \mathit{tail}\ (x : y) = y \end{array}$$

are standard Haskell functions as well. The problem with (1) now is that its left-hand side yields exactly the “divide by zero”-error coming from l, whereas its right-hand side may also yield the “Prelude.tail: empty list”-error. This is so due to the semantics of pattern-matching in the design of [PRH+99] (and also [MLP99]). In short, it prescribes that when pattern-matching on an erroneous value as scrutinee, not only are any errors associated with it propagated, but also are the branches of the pattern-match investigated in “error-finding mode” to detect any errors that may arise there independently of the scrutinee. This is done in order to give the language implementation more freedom in arranging computations, thus allowing more transformations on the code prior to execution. But here it means that when $\mathit{takeWhile}\ (\mathit{null} \circ \mathit{tail})$ encounters an erroneous value, also $(\mathit{null} \circ \mathit{tail})\ x$ is evaluated, with x bound to a special value $\mathit{Bad}\ \emptyset$ that exists only to trigger the error-finding mode. And indeed, the application of *tail* on that x raises the “Prelude.tail: empty list”-error, which is propagated by *null* and then unioned with the “divide by zero”-error from l. In contrast, $\mathit{takeWhile}\ \mathit{null}$ on an erroneous value does not add any further errors, because the definition of *null* raises none. And, on both sides of (1), $\mathit{map}\ h$ only ever propagates, but never introduces errors.

Thus, if we do not want to take the risk of introducing previously nonexistent errors, then in an implementation that builds on the semantics of [PRH+99] we cannot use (1) as a transformation from left to right, even though this might have been beneficial (by bringing p and h together for further analysis or for subsequent transformations potentially improving efficiency). The supposed semantic equivalence simply does not hold. Impreciseness in the semantics has its price, and if we are not ready to abandon the overall design, then we better learn how to cope with it when reasoning about programs.

The above discussion regarding a concrete instantiation of p, h, and l does not provide any positive information about conditions under which (1) actually *is* a semantic equivalence. Moreover, it is relative to the particular definition of *takeWhile* given at the very beginning, whereas laws like (1) are often derived more generally as *free theorems* [Rey83,Wad89] from types alone, without considering concrete definitions. In the study reported here we undertake to develop the theory of free theorems for Haskell with imprecise error semantics. This continues earlier work [JV04] for Haskell with all potential error causes (including

nontermination) conflated into a single erroneous value $\bot$. That earlier work gives that in this setting (1) is a semantic equivalence provided $p \neq \bot$ and h is strict and total in the sense that $h\ \bot = \bot$ and for every $x \neq \bot$, $h\ x \neq \bot$. The task set ourselves involves finding the right generalisations of such conditions for a setting in which not all errors are equal. Questions like the following ones arise:

- From which erroneous values should p be different?
- For strictness, is it enough that h preserves the least element $\bot$, which in the design of [PRH+99] denotes the union of all error causes, including nontermination?
- Or do we need that also every other erroneous value (denoting a collection of only some potential error causes, maybe just a singleton set) is mapped to an erroneous one? To the same one? Or to $\bot$?
- For totality, is it enough that "non-$\bot$ goes to non-$\bot$"?
- Does this allow "non-$\bot$ goes to non-$\bot$ but erroneous"?
- Or do we need "nonerroneous goes to nonerroneous"?

Our investigation is very much goal-directed by studying proof cases of the *(relational) parametricity theorem*, which is the foundation for all free theorems, and trying to adapt the proof to the imprecise error setting. This leads us to discover, among other formal details and ingredients, the appropriate generalised conditions sought above (first as restrictions on relations, then specialised to the function level). In fact, we think that beside our results for the imprecise error setting, our study can also serve as a guide on how to go about extending relational parametricity to new language features and semantic designs in general. We have established both equational and inequational parametricity theorems, including one for the refinement order of [MLP99]. And while we do not deal with error recovery through exception handling in the IO monad, we have made some initial steps into the realm of exceptions as first class citizens by integrating a primitive (Haskell's *mapException*) that allows manipulating already raised errors (respectively, their descriptive arguments) from inside the language.

References

[JV04] P. Johann and J. Voigtländer. Free theorems in the presence of seq. In *Principles of Programming Languages, Proceedings*, pages 99–110. ACM Press, 2004.

[MLP99] A. Moran, S.B. Lassen, and S.L. Peyton Jones. Imprecise exceptions, Co-inductively. In *Higher Order Operational Techniques in Semantics, Proceedings*, volume 26 of *ENTCS*, pages 122–141. Elsevier, 1999.

[PRH+99] S.L. Peyton Jones, A. Reid, C.A.R. Hoare, S. Marlow, and F. Henderson. A semantics for imprecise exceptions. In *Programming Language Design and Implementation, Proceedings*, pages 25–36. ACM Press, 1999.

[Rey83] J.C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing, Proceedings*, pages 513–523. Elsevier, 1983.

[Wad89] P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM Press, 1989.

Consistency Check for Component-based Design of Embedded Systems using SAT-solving

Peter V. B. Sørensen & Jan Madsen, IMM, DTU, {pbs,jan}@imm.dtu.dk

Introduction

Today, embedded applications are increasingly relying on customized heterogeneous multi-processor systems to provide the computational power and flexibility needed to meet the ever increasing demands on performance, features and reliability. Because of tight time-to-market constraints it is imperative that the application and platform are developed in parallel. One way to enable parallel development of application and platform is through the use of a *programming model* [1, 2], which serves as an abstraction layer between the application written in some high-level programming language and the platform. Tools and methods exist that automatically generate an implementation of such a model for some restricted classes of platforms [3, 4], whereas in the case of arbitrary platforms, implementing a programming model is a manual, error prone and time consuming undertaking.

The long term goal of this work is to devise a method to automatically generate an implementation in the context of a component-based design methodology. In such a methodology, a platform is assembled from a library of pre-defined components and combined with a specification of the application. Some components may require access to services provided by other components in order to function properly and, similarly, part of the specification mapped to one processing element may be dependent on having access to other parts which are mapped to different processing elements. Ensuring that a system is *consistent* with respect to such dependencies is a necessary first step toward automatically synthesizing an implementation – this consistency check is the focus here.

The Service Relation Model

We have developed a formalism, called the *service relation model*, useful to analyze the flow and availability of services in a system composed of components. In a service relation model of a system the components represent platform entities (e.g. processors, interconnects, operating systems) as well as entities from the specification (e.g. processes, functions, variables) providing and consuming services. Figure 1 shows a simple streaming application, consisting of three processes (P_i) communicating via two memory regions (MR_j), and a possible mapping onto a simple multi-processor platform. In general, deciding on a mapping is a complex problem and verifying that all inter-application dependencies are satisfied by the mapping is a challenge. A central concept to the service relation model is *service aggregation*. A service s_1 is said to

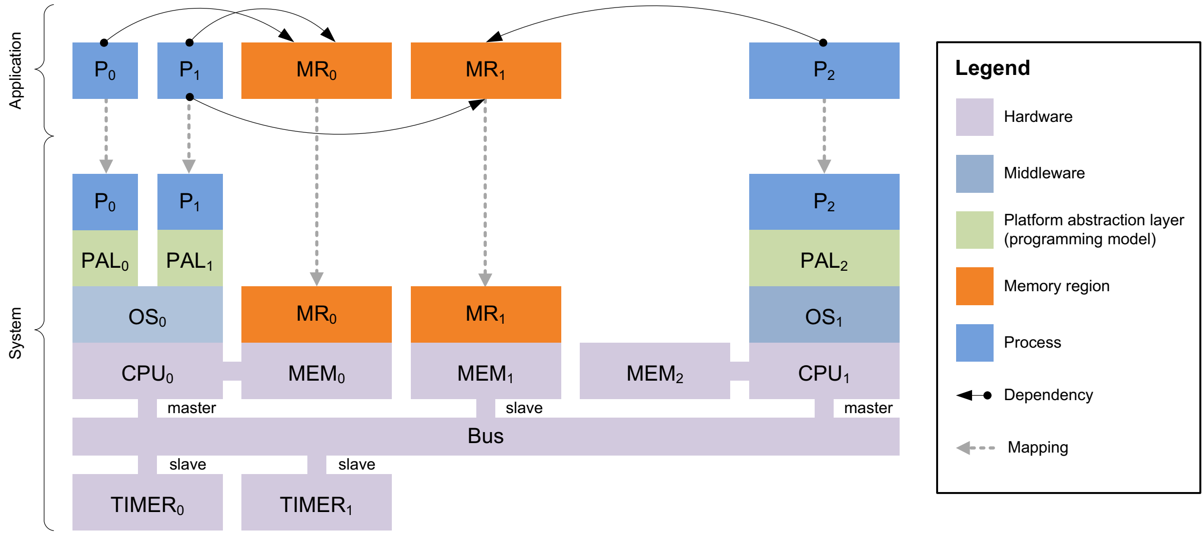


Figure 1: Platform, Platform Abstraction Layer and Application

aggregate another service s_2 if s_2 is accessible through s_1. In the example, the memories (MEM_k) are associated with a service representing the possibility of other components to read and write its content. Connected to the MEM_1 memory is a component representing a memory region MR_1 which has two services associated with it – representing the possibility to read and write it's content. These two services are aggregated by the service of the memory meaning that the memory region can be read and written through the service provided of the memory component. Similarly, the interconnect provides a service (the ability of master attachments to read and write slave attachments) which aggregates the services provided by the slave attachments (MEM_1 and the timers). We also say that an aggregated service is *available* at the aggregating service. Also, we consider any services available at an aggregated service to also be available at the aggregating service. The effect of this, in the example, is that the set of services available at the bus is the union of the services available at the three slave attachments. The platform abstraction layer components (PAL_l), also depicted in figure 1, represents the implementation of the programming model that we aim to synthesize. A system like the one in figure 1 can be captured in the service relation model using a few dosen components at a rather coarse granularity. In a real-world scenario, however, a system will have in the order of hundreds or perhaps thousands of components and services. The literature contains many examples of component models that can be used to analyze various properties of systems composed of components (e.g. [5, 6, 7]). The service relation model differs from these models in it's ability to cope with both hardware, middleware and application specific software components uniformly and, most notably, it's intended use.

Testing Simple Service Dependencies

Given a service relation model of a system a directed (and possibly cyclic) graph called a *service flow graph*, can be extracted. The relationship between the services are explicitly represented in

this graph. We can determine the sets of services available at any point in the system using, for instance, a worklist algorithm [8]. Using this information, testing if a dependency is satisfiable amounts to testing whether or not the service is in the set of available services. This procedure can only be used to test if it is possible to access a given service from a given component and assumes that the service is available in “sufficient quantities”.

Testing Exclusive Service Dependencies

In some cases a service cannot be shared. For example, assume that the two operating systems in the preceding example requires access to a timer in order to do time-sliced preemptive scheduling. In this case the timer resource cannot (in general) be shared and consequently simply testing if a timer service is available for the operating systems will not suffice. To test any such *exclusive service dependencies* we construct a Boolean satisfiability (SAT) problem based on the service flow graph and the service availability information previously mentioned. The constructed SAT problem is an encoding of the different paths between providers and consumers of services where any illegal paths have been removed using the service availability information. By applying a SAT solver to the problem we may determine if the system is consistent with respect to such exclusive service dependencies. In addition, if the problem is satisfiable then a valid allocation of services to service consumers can be extracted from a satisfying variable assignment.

Conclusion & Future Work

We have presented two procedures for checking the consistency of two different kinds of service dependencies – one to test service availability based on a graph worklist algorithm and one to test exclusive service usage using a SAT solver. We have run experiments on a number of different synthetic examples, using the HySAT [9] and Z3 solvers [10], and we are working on including one or more real-world examples. There are numerous interesting extensions that could be explored in the future: We have so far assumed that there is either an infinite amount of service available (availability testing) or only a single entity (exclusive availability testing). In case of memory, for instance, there is a finite quantity of service available (bytes in the memory) that potentially can be shared between multiple consumers. To handle this, a hybrid SAT solver, such as [9], could be applied. We can see no principle difficulties in this extension. Finally, it is interesting to employ optimization methods in order to minimize the cost of using resources.

References

[1] A.A. Jerraya, A. Bouchhima and F. Pétrot, “Programming models and HW-SW interfaces abstraction for multi-processor SoC”, *DAC ’06: Proceedings of the 43rd annual conference on Design automation*, pp. 280–285, 2006.

[2] P. van der Wolf, E. de Kock, T. Henriksson, W. Kruijtzer and G. Essink, “Design and Programming of Embedded Multiprocessors: An Interface-Centric Approach”, *CODES-ISSS ’04 International conference on Hardware/software codesign and system synthesis*, pp. 206–217, 2004.

[3] P.G. Paulin, C. Pilkington, E. Bensoudane, M. Langevin and D. Lyonnard “Application of a Multi-Processor SoC Platform to High-Speed Packet Forwarding”, *DATE ’04: Proceedings of the conference on Design, automation and test in Europe*, 2004.

[4] H. Nikolov, T. Stefanov and E. Deprettere, "Systematic and Automated Multiprocessor System Design, Programming, and Implementation", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol.27, no. 3, pp. 542–555, 2008.

[5] L. Alfaro and T. A. Henzinger, "Interface automata", *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE), ACM*, 2001.

[6] D. Garlan, R.T. Monroe and D. Wile, "Acme: architectural description of component-based systems", *Foundations of component-based systems*, Campridge University Press, pp. 47–67, 2000.

[7] R. van Ommering, F. van der Linden, J. Kramer and J. Magee, "The Koala Component Model for Consumer Electronics Software", *IEEE Computer*, vol. 33, no. 3, pp. 78–85, 2000.

[8] F. Nielson, H. R. Nielson and C. Hankin, "Principles of Program Analysis", Corrected 2nd Printing, Springer-Verlag, ch. 6, pp. 355–392, 2005.

[9] M. Fränzle and C. Herde, "HySAT: An efficient proof engine for bounded model checking of hybrid systems", *Formal Methods in System Design*, Springer, vol. 30, no. 3, pp. 179–198, 2007.

[10] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver", *Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science*, Springer, vol. 4963, pp. 337–340, 2008.

Quantitative Simulations of Weighted Transition Systems

Claus Thrane, Uli Fahrenberg, Kim G. Larsen
Dept. of Computer Science, Aalborg University, Denmark
{crt,uli,kgl}@cs.aau.dk

We present research motivated by the *Challenge on embedded systems design*, posed by Henzinger and Sifakis in [5]. Henzinger and Sifakis express the need for a coherent theory of embedded systems design, where concern for physical constraints is supported by the computational models used to model software, thus achieving a more heterogeneous approach to systems design. Highly distilled, Henzinger and Sifakis call for a new mathematical basis for systems modeling, which facilitates modeling of behavioural properties as well as environmental constraints.

In this work we propose a notion of *weighted transition systems* (WTS), a quantitative extension of the classical computational model of labeled transition systems [7]. Specifically, a notion of weights is added, with the intention to facilitate modeling of extra-functional properties, such as power or resource consumption in general. The proposed formalisms and metrics have been designed for general approximate reasoning about reactive systems, but for now we are especially interested in the subclass generated by *weighted timed automata* [3, 2]. Applications to other real-time formalisms as *e.g.* TCCS, or to probabilistic formalisms, are also possible.

Traditionally, *equivalences* have been employed when comparing systems. Many different *behavioural* and *language* equivalences have been proposed, see *e.g.* the survey provided in [9]. The use of equivalences gives rise to classical decision problems of the form: *is A equal to B*, which may have the specific purpose of expressing that one system, the implementation A, conforms to some desired specification B. In a quantitative setting, such equivalences may be replaced by *metrics* which assign to pairs of systems, *i.e.* initial states, a real valued *distance*; intuitively the problem is lifted such that

$$\{\mathit{true}, \mathit{false}\} \text{ becomes } \mathbb{R}_{\geq 0}\,.$$

Here a distance of 0 (zero) represents a verdict of *true* for the binary decision problem. All values $\varepsilon > 0$ are mapped back to *false*, but are intended to give indication of systems which are not equal, yet related up to some error margin given by their distance ε. Note that problems of computing a distance ε between systems A, B can be reformulated to decision problems $R(A, B, \varepsilon)$.

When generalizing from binary decision problems to distances, one has a number of choices on what is to be measured. In this work we concentrate on three different metrics, *point-wise*, *accumulated*, and *maximum-lead*, but a number of others are also relevant and interesting. Some of these have been considered in the literature; for real-time systems see *e.g.* [6, 4], and also for probabilistic systems there is a body of work on approximate distances. We show that the three metrics we consider are interesting from an application point of view, and that they measure inherently different properties.

More specifically, we show that for all three metrics mentioned above, bisimulation (resp. trace equivalence) may be lifted to a family of ε-bisimulations (resp. ε-trace distances) on weighted transition systems, such that $p \sim q$ and $p =_{\mathcal{L}} q$ are generalized to $p \sim_\varepsilon q$ and $|p, q| = \varepsilon$, where states p and q are related if they have a (bisimulation resp. trace) distance which is no more than ε. We show that also in the quantitative setting, $p \sim_\varepsilon q$ implies $|p, q| = \varepsilon$.

1 Quantitative models

Our notion of *weighted transition system*, or *WTS*, is a simple extension of the well-known labeled transition system formalism. Transition systems have been used to define (operational) semantics for a wide range of systems; similarly, weighted transition systems should be applicable for a number for formalisms. Here we shall concentrate on *timed automata* [1] and *weighted timed automata* [3, 2].

Definition 1 A weighted transition system is a triple $(\mathcal{S}, w, lg)$ where

- $\mathcal{S} = \langle S, s_0, \Gamma, R\rangle$ is a labeled transition system, with states S, initial state s_0, alphabet Γ, and transitions $R \subseteq S \times \Gamma \times S$,
- $w : R \to \mathbb{R}_{\geq 0}$ assigns weights to transitions, and
- $lg : \Gamma \to \mathbb{R}_{\geq 0}$ assigns lengths to labels.

We write $s \xrightarrow{\alpha,\omega} s'$ whenever $(s, \alpha, s') \in R$ and $\omega = w((s, \alpha, s'))$. Moreover, we let $\mathsf{c}(s, \alpha, s') = w((s, \alpha, s')) \cdot lg(\alpha)$ denote the actual cost of taking the transition.

The use both of weights on transitions and lengths of labels is to some extent superfluous; either one could be

omitted without changing expressibility, however we shall see later that there is a good motivation.

1.1 Weighted timed automata

Weighted timed automata [3, 2] is a useful formalism for modeling optimal scheduling and control problems. Their semantics is usually given as a weighted timed transition system, but we shall instead translate them to WTS. Note that by setting prices and rates to zero, one obtains a usual timed automaton, hence our translation also applies to these.

Definition 2 A weighted timed automaton is a tuple $(L, l_0, \mathcal{C}, I, E, p, r)$ where

- L is a finite set of locations, with l_0 initial,
- $\mathcal{C}$ is a finite set of real-valued clocks,
- $I : L \to \Psi(\mathcal{C})$ assigns invariants to locations,
- $E \subseteq L \times \Psi(\mathcal{C}) \times 2^{\mathcal{C}} \times L$ is a set of edges,
- $p : E \to \mathbb{N}$ is a edge price function, and
- $r : L \to \mathbb{N}$ is a location rate function.

Again, we write $l \xrightarrow{\psi, C} l'$ instead of $(l, \psi, C, l') \in E$. The set $\Psi(\mathcal{C})$ of clock constraints above is generated by the grammar

$$\psi ::= x \bowtie k \mid x - y \bowtie k \mid \psi_1 \wedge \psi_2$$

for $x, y \in \mathcal{C}, k \in \mathbb{R}, \bowtie \in \{\leq, <, =, \geq, >\}$.

The semantics of a WTA $\mathcal{A}$ is given by a WTS $\mathbf{W} = (\mathcal{S}, w, lg)$, where $\mathcal{S} = (S, (l_0, v_0), \{\star\} \cup \mathbb{R}_{\geq 0}, T)$ is the (usual) labeled transition system associated with the underlying TA of $\mathcal{A}$, $lg(\star) = 1$, $lg(\delta) = \delta$ for $\delta \in \mathbb{R}_{\geq 0}$, and for $t \in T$,

$$w(t) = \begin{cases} p(e) & \text{if } t = (l, v) \xrightarrow{\star} (l', v') \text{ and } e = l \xrightarrow{\psi, C} l' \in E \\ r(l) & \text{if } t = (l, v) \xrightarrow{\delta} (l, v + \delta) \end{cases}$$

2 Quantifying relations

We propose three different distances between WTS, both in linear and branching versions. Our first is a *point-wise* distance, comparing matching steps without considering the past or future. The second is an *accumulating* distance, using discounting. Finally, an accumulating distance which allows *catch-up* describes the *maximum lead* of accumulated weights.

For the definitions below, we fix a WTS $(\mathcal{S}, w, lg)$ and a discounting factor $\lambda \in [0, 1]$. $\mathsf{Tr}(s)$ denotes the set of (finite or infinite) traces emanating from s, and for a trace σ, $\sigma(i)$ denotes the ith transition in the trace, and $s_i(\sigma) = \sum_{j=0}^{i} lg(\sigma(j))$ denotes its accumulated length.

2.1 Trace distances

The *point-wise trace distance* between states $s, t \in S$ is defined by

$$\|s, t\|_{\bullet} = \tag{1}$$
$$\max \begin{cases} \sup_{\sigma \in \mathsf{Tr}(s)} \inf_{\sigma' \in \mathsf{Tr}(t)} \{\sup_i \lambda^{s_i(\sigma)} |\mathsf{c}(\sigma(i)) - \mathsf{c}(\sigma'(i))|\} \\ \sup_{\sigma \in \mathsf{Tr}(t)} \inf_{\sigma' \in \mathsf{Tr}(s)} \{\sup_i \lambda^{s_i(\sigma)} |\mathsf{c}(\sigma(i)) - \mathsf{c}(\sigma'(i))|\} \end{cases}$$

The *accumulating trace distance* between s and t is

$$\|s, t\|_{+} = \tag{2}$$
$$\max \begin{cases} \sup_{\sigma \in \mathsf{Tr}(s)} \inf_{\sigma' \in \mathsf{Tr}(t)} \{\sum_i \lambda^{s_i(\sigma)} |\mathsf{c}(\sigma(i)) - \mathsf{c}(\sigma'(i))|\} \\ \sup_{\sigma \in \mathsf{Tr}(t)} \inf_{\sigma' \in \mathsf{Tr}(s)} \{\sum_i \lambda^{s_i(\sigma)} |\mathsf{c}(\sigma(i)) - \mathsf{c}(\sigma'(i))|\} \end{cases}$$

The *maximum-lead distance* between s and t is

$$\|s, t\|_{\pm} = \tag{3}$$
$$\max \begin{cases} \sup_{\sigma \in \mathsf{Tr}(s)} \inf_{\sigma' \in \mathsf{Tr}(t)} \{ \sup_j \{\lambda^{s_i(\sigma)} | \sum_{i=0}^{j} \mathsf{c}(\sigma(i)) - \sum_{i=0}^{j} \mathsf{c}(\sigma'(i))|\}\} \\ \sup_{\sigma \in \mathsf{Tr}(t)} \inf_{\sigma' \in \mathsf{Tr}(s)} \{ \sup_j \{\lambda^{s_i(\sigma)} | \sum_{i=0}^{j} \mathsf{c}(\sigma(i)) - \sum_{i=0}^{j} \mathsf{c}(\sigma'(i))|\}\} \end{cases}$$

All of the distances above take the value ∞ whenever $s \neq_{\mathcal{L}} t$ – *i.e.* when states s and t are not *unweighted* trace equivalent.

Hence the point-wise distance measures the largest difference between the cost of corresponding single transitions, whereas the accumulating distance adds up all these differences. The maximum-lead distance allows the traces to "play catch-up"; here, accumulated delays can also decrease instead of only increasing.

2.2 ε-Bisimulation relations

Matching the trace distances above, we define three bisimulation relations $\dot{\sim}_{\varepsilon}$, $\overset{+}{\sim}_{\varepsilon}$, and $\overset{\pm}{\sim}_{\varepsilon}$:

A family of relations $\mathbf{R} = \{\dot{\mathcal{R}}_{\varepsilon} \subseteq S \times S \mid \varepsilon \geq 0\}$ is a *point-wise bisimulation family* if $(s, t) \in \dot{\mathcal{R}}_{\varepsilon} \in \mathbf{R}$ and $\alpha \in \Gamma$ imply that

- if $s \xrightarrow{\alpha, c} s'$ then $t \xrightarrow{\alpha, d} t'$ with $|c - d| \leq \varepsilon / lg(\alpha)$ and $(s', t') \in \dot{\mathcal{R}}_{\varepsilon'} \in \mathbf{R}$ for some d, t' and $\varepsilon' \leq \frac{\varepsilon}{\lambda^{lg(\alpha)}}$,
- if $t \xrightarrow{\alpha, c} t'$ then $s \xrightarrow{\alpha, d} s'$ with $|c - d| \leq \varepsilon / lg(\alpha)$ and $(s', t') \in \dot{\mathcal{R}}_{\varepsilon'} \in \mathbf{R}$ for some d, t' and $\varepsilon' \leq \frac{\varepsilon}{\lambda^{lg(\alpha)}}$.

We write $s \dot{\sim}_{\varepsilon} t$ whenever $(s, t) \in \dot{\mathcal{R}}_{\varepsilon} \in \mathbf{R}$ for some point-wise bisimulation family $\mathbf{R}$.

A family of relations $\mathbf{R} = \{\mathcal{R}_{\varepsilon} \subseteq S \times S \mid \varepsilon \geq 0\}$ is an *accumulating bisimulation family* if $(s, t) \in \mathcal{R}_{\varepsilon} \in \mathbf{R}$ and $\alpha \in \Gamma$ imply that

- if $s \xrightarrow{\alpha, c} s'$ then $t \xrightarrow{\alpha, d} t'$ with $|c - d| \leq \varepsilon / lg(\alpha)$ and $(s', t') \in \mathcal{R}_{\varepsilon'} \in \mathbf{R}$ for some d, t' and $\varepsilon' \leq \frac{\varepsilon - |c-d|}{\lambda^{lg(\alpha)}}$,
- if $t \xrightarrow{\alpha, c} t'$ then $s \xrightarrow{\alpha, d} s'$ with $|c - d| \leq \varepsilon / lg(\alpha)$ and $(s', t') \in \mathcal{R}_{\varepsilon'} \in \mathbf{R}$ for some d, t' and $\varepsilon' \leq \frac{\varepsilon - |c-d|}{\lambda^{lg(\alpha)}}$.

We write $s \overset{+}{\sim}_\varepsilon t$ whenever $(s,t) \in \mathcal{R}_\varepsilon \in \mathbf{R}$ for some accumulating bisimulation family $\mathbf{R}$.

A family of relations $\mathbf{R} = \{\mathcal{R}_{\varepsilon,\delta} \subseteq S \times S \mid \varepsilon \geq 0, -\varepsilon \leq \delta \leq \varepsilon\}$ is a *maximum-lead bisimulation family* if $(s,t) \in \mathcal{R}_{\varepsilon,\delta} \in \mathbf{R}$ and $\alpha \in \Gamma$ imply that

- if $s \xrightarrow{\alpha,c} s'$ then $t \xrightarrow{\alpha,d} t'$ with $\delta + (c-d)lg(\alpha) \leq \varepsilon$ and $(s',t') \in \mathcal{R}_{\varepsilon',\delta'} \in \mathbf{R}$ for some d, t' and $\varepsilon' \leq \frac{\varepsilon}{\lambda^{lg(\alpha)}}$, $\delta' \leq \frac{\delta + (c-d)lg(\alpha)}{\lambda^{lg(\alpha)}}$,
- if $t \xrightarrow{\alpha,c} t'$ then $s \xrightarrow{\alpha,d} s'$ with $\delta + (c-d)lg(\alpha) \leq \varepsilon$ and $(s',t') \in \mathcal{R}_{\varepsilon',\delta'} \in \mathbf{R}$ for some d, t' and $\varepsilon' \leq \frac{\varepsilon}{\lambda^{lg(\alpha)}}$, $\delta' \leq \frac{\delta + (c-d)lg(\alpha)}{\lambda^{lg(\alpha)}}$.

We write $s \overset{\pm}{\sim}_\varepsilon t$ whenever $(s,t) \in \mathcal{R}_{\varepsilon,0} \in \mathbf{R}$ for some maximum-lead bisimulation family $\mathbf{R}$.

3 Results and open problems

We can show that, analogous to the unweighted setting, our trace and bisimulation distances are related in the sense that for each of the three pairs, membership in an ε-bisimulation relation implies a trace distance not greater than ε.

We can also prove that all six distances are metrics in the technical sense, hence each of them gives a metric-space structure on the set of WTS. Perhaps surprisingly, all these spaces are mutually topologically inequivalent, indicating that the metrics are indeed measuring very different properties.

We also have a logical characterization of our point-wise bisimulation distance [8], and we believe that similar can be found for the other two distances.

In [6] it is shown that for timed automata, maximum-lead bisimulation distance is computable in a certain sense. Whether this also holds for weighted timed automata is an open question, as is computability of the other bisimulation distances, both for (weighted) timed automata and for other interesting formalisms.

References

[1] R. Alur and D. Dill. Automata for modeling real-time systems. In *Proc. ICALP'90*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer-Verlag, 1990.

[2] R. Alur, S. La Torre, and G. J. Pappas. Optimal paths in weighted timed automata. In *Proc. 4th Int. Workshop Hybrid Systems: Computation and Control (HSCC'01)*, volume 2034 of *Lecture Notes in Computer Science*, pages 49–62. Springer-Verlag, 2001.

[3] G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, J. Romijn, and F. Vaandrager. Minimum-cost reachability for priced timed automata. In *Proc. 4th Int. Workshop on Hybrid Systems: Computation and Control (HSCC'01)*, volume 2034 of *Lecture Notes in Computer Science*, pages 147–161. Springer-Verlag, 2001.

[4] Vineet Gupta, Thomas A. Henzinger, and Radha Jagadeesan. Robust timed automata. In *Proc. HART'97*, volume 1201 of *Lecture Notes in Computer Science*, pages 331–345. Springer-Verlag, 1997.

[5] Thomas A. Henzinger and Joseph Sifakis. The embedded systems design challenge. In *14th International Symposium on Formal Methods (FM)*, Lecture Notes in Computer Science, pages 1–15. Springer, September 2006.

[6] Thomas A. Henzinger, Rupak Majumdar, and Vinayak Prabhu. Quantifying similarities between timed systems. In *Proc. FORMATS'05*, volume 3829 of *Lecture Notes in Computer Science*, pages 226–241. Springer-Verlag, 2005.

[7] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981. URL `citeseer.ist.psu.edu/plotkin81structural.html`.

[8] Claus Thrane. On weighted labelled transition systems, quantitative relations and logic. Technical Report 1213005970, Dept. of Computer Science, Aalborg University, 2008.

[9] R. van Glabbeek. The linear time – branching time spectrum I: The semantics of concrete, sequential processes. URL `citeseer.ist.psu.edu/328833.html`.

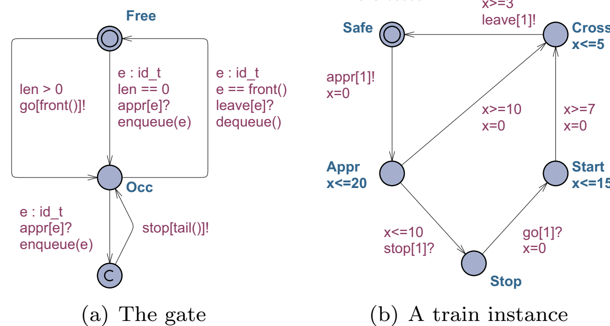
Slicing For UPPAAL

Claus Thrane, Uffe Sørensen and Kim G. Larsen
Dept. of Computer Science, Aalborg University, Denmark
{crt,us,kgl}@cs.aau.dk

We present *slicing* for the model-checking tool UPPAAL [5]. Slicing is a technique based on static analysis used here to reduce the syntactic size of models based on an extension of *Timed Automata* [1]. We show how our approach will obtain reachability preserving reductions of models, possibly improving the performance of the tool. Finally, we present the results of experimental conducted to further validate our claims.

Whereas traditionally, slicing is used for debugging and testing purposes, the goal here is to apply slicing to improve model-checking capabilities of UPPAAL. Using automated slicing in UPPAAL drastically reduces the need for manual adaptation of models for faster verification of individual properties. Moreover, it allows less experienced users, which unknowingly may design models, containing unnecessary large amounts of data, to verify properties which UPPAAL otherwise would have been unable to check. With the rich imperative language of UPPAAL 4.0 – including structured and user-defined types as well as C-style expressions and functions – the need for automated slicing support is becoming increasingly important.

Similar to the mental analysis performed when debugging software, automated slicing is performed by analysing the control-flow of a model, computing dependencies among components. We present an approach which given UPPAAL model will produce a possibly smaller model, containing only elements of interest w.r.t. some specific reachability analysis. Our approach incorporates a graph based data structure known as a *System Dependency Graph* (SDG) used to represent dependencies from the *combined* control-flow of timed automata (TA) and C-style imperative functions, used in updates and guards. Moreover, it entails an analysis in the from of a work-list algorithm for computing the *relevant* components from SDG. Finally, it defines how the sliced model is constructed using only the relevant components.



(a) The gate (b) A train instance

Figure 1: The Timed Automata from the *Train-Gate* Model

1 By Example

In the following we show how slicing reduces an endowed version of the the train-gate model, which is part of the official UPPAAL distribution[1]. It demonstrates how one, may model the design of a railway

[1]Note that the model it self is if no importance in this context

intersection, as TA, in order to verify certain properties, such that; all approaching trains eventually may cross without the possibility of collisions. The agents of the model are expressed as extended timed automata. Fig. 1(a) shows the model of the gate governing the intersection and Figure 1(b) shows an instance of a train approaching.

The agents communicate through (arrays of) *channels* e.g. `go[x]` and `leave[x]` where x a train id. Notice that some locations of the trains are restricted by invariants, limiting its delay in this position (state). As an example, whenever a train has announced its approach, it moves to the location **Appr** where it may wait at most 20 time units ($x \leq 20$) before taking action again. Furthermore, a train takes 3 time units to cross the gate, modeled by a guard ($x \geq 3$) on the outgoing edge from location **cross**.

```
1   id_t list[N+1];
2   int[0,N] len;
3   int noOfTrainsWaitingToCross;
4
5   void enqueue(id_t element)
6   {
7     noOfTrainsWaitingToCross++;
8     list[len++] = element;
9   }
10
11  void dequeue()
12  {
13    int i = 0;
14
15    if(noOfTrainsWaitingToCross > 0)
16    {
17       noOfTrainsWaitingToCross--;
18    }
19
20    len--;
21
22    while(i < len)
23    {
24      list[i] = list[i + 1];
25      i++;
26    }
27
28    list[i] = 0;
29  }
```

Original

```
id_t list[N+1];
int[0,N] len;


void enqueue(id_t element)
{

  list[len++] = element;
}


void dequeue()
{
  int i = 0;






  len--;

  while(i < len)
  {
    list[i] = list[i + 1];
    i++;
  }

  list[i] = 0;
}
```

Sliced

Figure 2: Slicing the C code of the Train-Gate model.

The lefthand side of Fig. 2 shows the imperative code of the Train-gate model. Observe that the presence of the variable `noOfTrainsWaitingToCross` has no impact on the semantics of the `enqueue` and `dequeue` functions. Thus it, has no obvious relevance w.r.t. the model's behavior. But in order to conclude this formally, we must consider the combined control-flow of both automata and code. Using our approach we may show that, it and any reference to it, may indeed be removed in order to reduce the state-space, during model-checking. Resulting code is shown to the right.

2 Experiments

Based on our theory, we have developed a prototype implementation as preprocessor for UPPAAL. The results obtained from running our prototype on two example models can be seen in Table 1. The examples used here is a model of a CAN bus[2] and the Train-Gate Example seen above. The experiments are conducted by monitoring the symbolic states explored, memory consumption and cpu time before and after slicing.

The first half of Table 1, shows results from the train-gate example, which was tested for the absence of deadlocks (`A[]not deadlock`) and that it is possible for all trains to cross the gate (e.g. `E<> Train1.Cross`). The second half, shows test from the CAN Bus Model. Also here a test of showing deadlock freeness was attempted. This example was brought to our attention specifically because UPPAAL could not verify deadlock freeness, verification of the original model thus failed due to resource consumption. After slicing, UPPAAL detected an overflow error in the model, which had otherwise gone undetected. The experiment was restarted after the error had been corrected.

[2]Donated by anonymous students in the dept of Control at AAU

Table 1: The test results of the Train-Gate experiments. VT = Verification Time, MU = Memory Usage, SS = Symbolic States explored, NS = Number of Statements and NV = Number of Variables

Deadlock free	VT	MU	SS	NS	NV
Before Slicing	N/A	4GB+	77636326+	34	14
After Slicing	0.2sec	2848KB	413	28	10
Train may cross	VT	MU	SS	NS	NV
Before Slicing	0.11sec	2856KB	14	34	14
After Slicing	0.10sec	2848KB	14	28	10
Before Slicing	N/A	4GB+	85587630+	12	6
After Slicing	11.12sec ⋆	66572KB	786391	6	3
After Fix	VT	MU	SS	NS	NV
Before Slicing	0.11sec	2862KB	2074	12	6
After Slicing	0.10sec	2852KB	199	6	3

3 Conclusion – Related and Future Work

Work on slicing has previously been produced for a number of imperative languages. J. Hatcliff [2] shows how the traditional slicing concepts described by Wieser [6] and later S. Horwitz, T. Reps and D. Binkley [3] may be extended to slicing a more complicated model of multi-threaded Java programs with JVM concurrency primitives. More closely related is the work by Janowska and Janowski [4], who show how slicing may be used for the formalism of TA defined by Alur and Dill [1]. Although both show that slicing is applicable for non-trivial languages, no one has (to our knowledge) ventured to show that slicing may be performed on a complex hybrid language of imperative code and timed automata, similar to what is found in Uppaal.

Its obvious by the above experiments, that slicing is a very effective tool for optimizing the performance of Uppaal. Nevertheless, the current state of the proved theory only supports truly unnecessary variables, which are then completely removed, a more involved analysis should make attempts at reducing the declared size of variables, which is a more common source of excessive state-space use. Also the control-flow induced by the communication of automata should be studied further.

References

[1] Alur, Courcoubetis, and Dill. Model-checking for real-time systems. In *LICS: IEEE Symposium on Logic in Computer Science*, 1990.

[2] John Hatcliff, James Corbett, Matthew Dwyer, Stefan Sokolowski, and Hongjun Zheng. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In Agostino Cortesi and Gilberto Filé, editors, *Static Analysis*, volume 1694 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 1999.

[3] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 35–46, New York, NY, USA, 1988. ACM Press.

[4] Agata Janowska and Pawel Janowski. Slicing of timed automata with discrete data. pages 181–195. Fundamenta Informaticae, 2006.

[5] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[6] Mark D. Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method.* PhD thesis, The University of Michigan, 1979.

Towards Model Checking Bounded Response in Real-Time Maude (Extended Abstract)

Peter Csaba Ölveczky and Daniela Lepri
Department of Informatics, University of Oslo

I. Introduction

Real-Time Maude [1], [2] is a high-performance tool that extends the rewriting-logic-based Maude [3] system to support the formal modeling and analysis of real-time systems. Real-Time Maude is characterized by its general and expressive, yet simple and intuitive, modeling formalism. Real-Time Maude is particularly useful for specifying and analyzing advanced distributed object-based systems with novel forms of communications and/or complex and unbounded data types. The tool has been successfully applied to a wide array of advanced state-of-the-art applications that are beyond the pale of timed automata, including the CASH scheduling algorithm [4], density and topology control algorithms for wireless sensor networks [5]–[7], commercial embedded car software, a 50+ pages multicast protocol suite for active networks [8], resource-sharing algorithms [9], etc.

Real-Time Maude supports a range of formal analysis methods, including rewriting for simulation, reachability analysis, and *untimed* linear temporal logic model checking. However, there is sometimes a need to analyze also *metric* (or *timed*) temporal logic properties such as

> *"an X-ray should be taken within two seconds after the button has been pushed"*

in which the duration of/between events is crucial.

This paper represents a first step in our investigation of how important classes of metric linear temporal logic (MLTL) properties can be model checked in Real-Time Maude. In particular, Section III defines a metric temporal logic for Real-Time Maude by introducing the well known time-bounded temporal operator $\mathcal{U}_{\leq t}$. Section III also explains that certain classes of MLTL properties can be model checked using the existing features of Real-Time Maude.

One of the most useful classes of MLTL properties is *bounded response*: any p-state must be followed by a q-state within time r (in MLTL: $\Box\,(p \rightarrow \Diamond_{\leq r}\, q)$). The main contribution of this paper is to show how bounded response properties can be model checked in *object-oriented* Real-Time Maude specifications. We use a transformational approach where a bounded response problem is transformed into a reachability problem that can be analyzed using Maude's efficient reachability checker. With this method, we can analyze bounded response for a large class of systems that cannot be modeled as timed automata.

II. Background on Real-Time Maude

A Real-Time Maude *timed module* specifies a *real-time rewrite theory* [10] of the form (Σ, E, IR, TR), where:

- (Σ, E) is a *membership equational logic* [11] theory with Σ a signature[1] and E a set of conditional equations. The theory (Σ, E) specifies the system's state space as an algebraic data type.
- IR is a set of *conditional instantaneous rewrite rules* specifying the system's *instantaneous* (i.e., zero-time) local transitions, each of which is written `crl` t `=>` t' `if` *cond*. The rules are applied *modulo* the equations E.[2]
- TR is a set of *tick (rewrite) rules*, written with syntax
  `crl {`t`} => {`t'`} in time` τ `if` *cond* `.`
  that model time elapse. τ is a term of sort `Time` that denotes the *duration* of the rewrite.

The initial states must be ground terms that are reducible to terms of the form $\{t\}$ using the equations in the specifications.

In object-oriented modules, a *class* declaration

`class` C `|` att_1 `:` s_1`, ... ,` att_n `:` s_n `.`

declares a class C with attributes att_1 to att_n of sorts s_1 to s_n. An *object* of class C in a given state is represented as a term $< O : C \mid att_1 : val_1, ..., att_n : val_n >$ where O is the object's *identifier*, and where val_1 to val_n are the current values of the attributes att_1 to att_n. In a distributed object-oriented system, the state is a term of the sort `Configuration`. It has the structure of a *multiset* made up of objects and messages. Multiset union for configurations is denoted by a juxtaposition operator (empty syntax) that is declared associative and commutative, so that rewriting is *multiset rewriting* supported directly in Real-Time Maude. The dynamic behavior of a system is axiomatized by specifying each of its concurrent transition patterns by a rewrite rule. For example, the rule

```
rl   m(O,w)    < O : C | a1 : x, a2 : O' >
  =>
   < O : C | a1 : x + w >    m'(O') .
```

defines a family of transitions in which a message `m`, with parameters `O` and `w`, is read and consumed by an object `O` of class `C`. The transitions have the effect of altering the attribute `a1` of the object `O` and of sending a new message `m'(O')`.

[1] i.e., Σ is a set of declarations of *sorts*, *subsorts*, and *function symbols*

[2] Operationally, a term is reduced to its E-normal form before any rewrite rule is applied.

In object-oriented real-time systems, time elapse is typically modeled by the tick rule [2]:

```
var C : Configuration .   var T : Time .
crl {C} => {delta(C, T)} in time T
       if T <= mte(C) .
```

The function `delta` defines the effect of time elapse on a configuration, and the function `mte` defines the maximum amount of time that can elapse before some action must take place. These functions distribute over the objects and messages in a configuration.

Real-Time Maude's *search* command uses breadth-first search to analyze all possible behaviors of the system, by checking whether a state matching a *pattern* and satisfying a *condition* can be reached from the initial state t. The command that searches for *one* state satisfying the search criteria has syntax

```
(utsearch [1] t =>* pattern such that cond .)
```

Real-Time Maude also extends Maude's *linear temporal logic model checker* [3] to check whether each behavior, possibly up to a certain duration as explained in [2], satisfies a temporal logic formula. *State propositions* are terms of sort `Prop`, and their semantics should be given by (possibly conditional) equations of the form `{`$statePattern$`} |=` *prop* = *b*, for *b* a term of sort `Bool`, which defines the state proposition *prop* to hold in all states `{`t`}` where `{`t`} |=` *prop* evaluates to `true`. A temporal logic *formula* is constructed by state propositions and temporal logic operators such as `True`, `~` (negation), `/\`, `\/`, `->` (implication), `[]` ("always"), `<>` ("eventually"), and `U` ("until").

III. Metric LTL for Real-Time Maude

As shown in [12], a common way of incorporating timing requirements in a propositional temporal logic formula is to introduce time-bounded versions (such as the time-bounded *until* operator $\mathcal{U}_{\leq r}$) of the temporal operators. This approach has been proposed by Koymans, Vytopil, and de Roever in [13]–[15]. Given a set of *atomic propositions* AP, we can therefore define the formulas of metric linear temporal logic (MLTL) inductively as follows:

$$\phi ::= \texttt{True} \mid p \mid \neg\, \phi \mid \phi_1 \wedge \phi_2 \mid \phi_1\, \mathcal{U}_{\leq r}\, \phi_2$$

where $p \in AP$ and $\mathcal{U}_{\leq r}$ is the time-bounded until operator. Other MLTL operators can be defined in the usual way, for instance $\Diamond_{\leq r}\, \phi = \texttt{True}\,\mathcal{U}_{\leq r}\, \phi$; we can also define unbounded temporal operators: $\mathcal{U}\, \phi = \mathcal{U}_{\leq \texttt{INF}}\, \phi$, $\Diamond\, \phi = \Diamond_{\leq \texttt{INF}}\, \phi$, etc., where `INF` denotes infinity value in Real-Time Maude.

Satisfaction of MLTL formulas in Real-Time Maude is defined in the expected way; details are not given in this abstract. Informally, a bounded until property $\phi_1\, \mathcal{U}_{\leq r}\, \phi_2$ is satisfied along a path if ϕ_2 holds within r time units and ϕ_1 holds in all states before that point.

It is worth noticing that, if P and Q are boolean combinations of atomic propositions, then $\Box_{\leq r} P$, $\Diamond_{\leq r} P$, and $P\, \mathcal{U}_{\leq r}\, Q$ can be directly model checked using Real-Time Maude's *time-bounded* LTL model checker, that analyzes all behaviors up to a given duration r.

IV. Model Checking Bounded Response Properties in Object-Oriented Specifications

We transform the bounded response model checking problem $\mathcal{R}, t_0 \models \Box\,(P \longrightarrow \Diamond_{\leq r}\, Q)$, for P and Q as above, into the untimed model checking problem $\widetilde{\mathcal{R}}, \widetilde{t_0} \models \Box\,(C_{P,Q} \leq r)$, where $C_{P,Q}$ is a "clock" that measures the time since P held without Q holding in the meantime. The general idea is to add a "clock" $C_{P,Q}$ to the system, and update the clock as follows:

1) If the clock $C_{P,Q}$ is turned off, and a state satisfying $P \wedge \neg Q$ is reached, the clock is set to 0 and is turned on.
2) The clock is turned off when a state satisfying Q is reached.
3) A clock that is on is increased according to the elapsed time in the system.

A. An Automatic Way of Model Checking Bounded Response for Flat Object-Oriented Specifications

The method suggested above is fairly general. For the very useful class of "flat" object-oriented specifications specified according to the guidelines in [2]—all advanced Real-Time Maude applications have been so specified—we can automate the transformation from $(\mathcal{R}, L, \{t_0\}, P, Q)$[3] into a $C_{P,Q}$-extension as follows:[4]

1) Add the following class for the "clock:"
```
class BRClock | clock : Time, status : OnOff .
sort OnOff .    ops on off : -> OnOff [ctor] .
```
2) A clock object is added to the initial state $\{t_0\}$, so that the initial state becomes `{`t_0 `<` $c_{P,Q}$ `: BRClock | clock : 0, status :` x `>}`, where x is `on` if $P \in L(\{t_0\})$ and $Q \notin L(\{t_0\})$, and is `off` otherwise. Note that $P \in L(\{t_0\})$ can be checked in Maude by checking whether $\{t_0\}$ `|=` $P = \texttt{true}$.
3) Each *instantaneous* rule t `=>` t' `if` *cond* in $\mathcal{R}$ is replaced by the rules
```
crl {t  REST < cP,Q : BRClock | status : on >}
   =>
    {t'  REST  < cP,Q : BRClock | >}
   if {t' REST} |= Q =/= true ∧ cond

crl {t  REST  < cP,Q : BRClock | status : on >}
   =>
    {t' REST < cP,Q : BRClock | status : off >}
   if {t' REST} |= Q ∧ cond

crl {t  REST  < cP,Q : BRClock | status : off >}
   =>
```

[3] L is the function which gives the set of atomic propositions holding in a state.

[4] We may assume without loss of generality that P and Q are atomic propositions, since if P is, say, $p \wedge \neg q$, it can be defined as an atomic proposition as follows:

```
op P : -> Prop . var S : State .
eq {S} |= P = ({S} |= p) and ({S} |= q =/= true) .
```

```
   {t'  REST
    < c_P,Q : BRClock | clock : 0, status : on >}
  if  {t' REST } |= P and
      {t' REST} |= Q =/= true ∧ cond

crl {t  REST  < c_P,Q : BRClock | status : off >}
  =>
   {t'  REST  < c_P,Q : BRClock | >}
  if  {t' REST} |= Q or
      {t' REST} |= P =/= true ∧ cond.
```

for `REST` a variable of sort `Configuration` that does not appear in the original rule. `REST` matches the "other" objects and messages in the state.

4) We keep the "standard" object-oriented tick rule and define the function `delta` on clocks in the expected way (so that the clock value increases according to the elapsed time when it is `on`) and ensure that `mte` is not affected by the new object:

```
eq delta(< B : BRClock | status : on,
             clock : T >, T') =
          < B : BRClock | status : on,
             clock : T + T' > .
eq delta(< B : BRClock | status : off >, T') =
          < B : BRClock | > .
eq mte(< B : BRClock | >) = INF .
```

The original bounded response property $\Box\,(P \longrightarrow \Diamond_{\leq r}\, Q)$ is equivalent to the clock value being less than r in each state in the transformed module, which can be analyzed by the following search command that searches for a state in which the clock value is greater than r:

```
(utsearch [1]
   {t0 < c_P,Q : BRClock | clock: 0, status : s >}
  =>*
   {C:Configuration
    < c_P,Q : BRClock | clock : T:Time >}
        such that T:Time > r .)
```

The semantic foundations and the correctness proofs of our method will appear elsewhere.

V. Concluding Remarks

We have shown an implementable method for model checking bounded response properties for the important class of object-based Real-Time Maude specifications. This class captures many systems that cannot be specified as timed automata; indeed, all advanced Real-Time Maude applications have been so specified.

The techniques have been successfully applied on a small network of medical devices [16], as well as on a traffic intersection system developed in the context of a Lockheed-Martin-led research project.

We are currently working on implementing our method in Real-Time Maude. Future work includes developing methods for model checking other fragments of MLTL and on applying the techniques on more applications.

References

[1] Ölveczky, P.C., Meseguer, J.: The Real-Time Maude tool. In Ramakrishnan, C.R., Rehof, J., eds.: Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08). Volume 4963 of Lecture Notes in Computer Science., Springer (2008) 332–336

[2] Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation **20**(1-2) (2007) 161–196

[3] Clavel, M., Durán, F., Eker, S., Lincoln, P., Mart-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. Volume 4350 of Lecture Notes in Computer Science. Springer (2007)

[4] Ölveczky, P.C., Caccamo, M.: Formal simulation and analysis of the CASH scheduling algorithm in Real-Time Maude. In Baresi, L., Heckel, R., eds.: Fundamental Approaches to Software Engineering (FASE'06). Volume 3922 of Lecture Notes in Computer Science., Springer (2006) 357–372

[5] Ölveczky, P.C., Thorvaldsen, S.: Formal modeling and analysis of the OGDC wireless sensor network algorithm in Real-Time Maude. In Bonsangue, M.M., Johnsen, E.B., eds.: Formal Methods for Open Object-Based Distributed Systems (FMOODS'07). Volume 4468 of Lecture Notes in Computer Science., Springer (2007) 122–140

[6] Ölveczky, P.C., Thorvaldsen, S.: Formal modeling, performance estimation, and model checking of wireless sensor network algorithms in Real-Time Maude. Theoretical Computer Science (2008) To appear.

[7] Katelman, M., Meseguer, J., Hou, J.: Redesign of the LMST wireless sensor protocol through formal modeling and statistical model checking. In Barthe, G., de Boer, F., eds.: Formal Methods for Open Object-Based Distributed Systems (FMOODS'08). Volume 5051 of Lecture Notes in Computer Science., Springer (2008) 150–169

[8] Ölveczky, P.C., Meseguer, J., Talcott, C.L.: Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. Formal Methods in System Design **29**(3) (2006) 253–293

[9] Ölveczky, P.C., Prabhakar, P., Liu, X.: Formal modeling and analysis of real-time resource-sharing protocols in Real-Time Maude. In: 22nd International Parallel and Distributed Processing Symposium (IPDPS 2008), IEEE Computer Society Press (2008)

[10] Ölveczky, P.C., Meseguer, J.: Specification of real-time and hybrid systems in rewriting logic. Theoretical Computer Science **285** (2002) 359–405

[11] Meseguer, J.: Membership algebra as a logical framework for equational specification. In Parisi-Presicce, F., ed.: Proc. WADT'97. Volume 1376 of Lecture Notes in Computer Science., Springer (1998) 18–61

[12] Alur, R., Henzinger, T.: Logics and models of real time: A survey. In de Bakker, J., Huizing, K., de Roever, W.P., Rozenberg, G., eds.: Real Time: Theory in Practice. Volume 600 of Lecture Notes in Computer Science., Springer (1992) 74–106

[13] Koymans, R., Vytopil, J., de Roever, W.P.: Real-time programming and asynchronous message passing. In: PODC. (1983) 187–197

[14] Koymans, R., de Roever, W.P.: Examples of a real-time temporal logic specification. In Denvir, B.T., Harwood, W.T., Jackson, M.I., Wray, M.J., eds.: The Analysis of Concurrent Systems. Volume 207 of Lecture Notes in Computer Science., Springer (1983) 231–251

[15] Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Systems **2**(4) (1990) 255–299

[16] Ölveczky, P.C.: Towards formal modeling and analysis of networks of embedded medical devices in Real-Time Maude. In: Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2008), IEEE Computer Society (2008) 241–248

Author index

Beringer, L. 11
Bhattacharya, S. 15
Bortin, M. 19
Clarke, D. 7
Cortesi, A. 15
Dahl, M. 22
Danos, V. 8
Degen, M. 25
Demangeon, R. 28
Dovland, J. 31
Ernits, J. 75
Fahrenberg, U. 97
Fenech, S. 34
Fränzle, M. 9
Gramlich, B. 84
Grigorenko, P. 37
Gruska, D. 40
Gu, Z. 43
Guan, N. 43
Hassan, A. 47
Hovland, D. 51
Hüttel, H. 22, 54
Johnsen, E. B. 31
Jonsson, P. A. 57
Knoll, I. 60
Kyas, M. 63
Lamo, Y. 78, 81
Larsen, K. G. 97, 100
Lepri, D. 103
Lüth, C. 19
Mackie, I. 47
Madsen, J. 93
Martins, F. 87
Nordlander, J. 57
Nummenmaa, T. 66
Owe, O. 31
Pace, G. 34
Prisacariu, C. 69
Rafnsson, W. T. 54
Ratschan, S. 72
Ravn, A. P. 60
Roo, R. 75
Rossini, A. 78, 81
Rutle, A. 78, 81
Sato, S. 47
Schernhammer, F. 84
Schneider, G. 34
Skou, A. 60
Slani, N. 87
Smaus, J.-G. 72
Stam, A. 63
Steffen, M. 31, 63
Stenger, F. 90
Sørensen, P. 93
Sørensen, U. 100
Thiemann, P. 25
Thrane, C. 97, 100
Torjusen, A. 63
Veanes, M. 10
Voigtländer, J. 90
Walter, D. 19
Wolter, U. 78, 81
Wehr, S. 25
Ölveczky, P. 103
Yi, W. 43
Yu, G. 43