Proceedings of the 16th

Nordic Workshop on Programming Theory

October 6–8, 2004
Uppsala, Sweden

Paul Pettersson and Wang Yi (Eds.)

Department of Information Technology
Uppsala University
Box 337, SE-751 05 Uppsala, Sweden

Technical report 2004-041
October 2004
ISSN 1404-3203

Preface

The objective of the Nordic Workshop on Programming Theory is to bring together researchers from (but not limited to) the Nordic and Baltic countries interested in programming theory, in order to improve mutual contacts and cooperation.

The 16*th* Nordic Workshop on Programming Theory took place at the Uppsala University, Sweden, 6-8 October 2004. The previous workshops were held in Uppsala (1989 and 1999), Aalborg (1990), Gothenburg (1991 and 1995), Bergen (1992 and 2000), Turku (1993, 1998, and 2003), Aarhus (1994), Oslo (1996), Tallinn (1997 and 2002), and in Lyngby (2001).

There were 39 regular presentations at the workshop, arranged in two parallel sessions. In addition the following five invited speakers gave presentations in plenary sessions: Erik Hagersten (Uppsala Univ., Sweden), Neil D. Jones (Copenhagen Univ., Denmark), Kim G. Larsen (Aalborg Univ., Denmark), P.S. Thiagarajan (National University of Singapore), and Michael Williams (Ericsson, Sweden).

Programme Committee

Michael R. Hansen	Techn. U. of Denmark, Denmark
Magne Haveraaen	Univ. of Bergen, Norway
Hannu-Matti Jarvinen	Tampere Univ of Tech., Finland
Kim G. Larsen	Aalborg Univ., Denmark
Bengt Nordstrom	Univ. of Gothenburg, Chalmers Univ of Tech., Sweden
Olaf Owe	Univ. of Oslo, Norway
Kaisa Sere	Abo Akademi University, Finland
Tarmo Uustalu	Inst. of Cubernetics, Estonia
Juri Vain	Tallinn Technical University, Estonia
Wang Yi	Uppsala Univ., Sweden

Local Organizing Committee

Ulrika Andersson, Anders Hessel, Patrik Johansson, Paul Pettersson, Wang Yi

Table of Contents

The size-change approach to program termination analysis
Neil D. Jones 1

Memory Usage Estimation for Java Cards
Gerardo Schneider 2

Stage-Preserving Embeddings of Languages
Todd L. Veldhuizen 5

Signals and Comonads
Tarmo Uustalu and Varmo Vene 9

Simulation-Based Iteration of Tree Transducers
Parosh Abdulla, Axel Legay, Julien d'Orso, and Ahmed Rezine 11

Heuristic Guided Model-Checking of Real-Time Systems
Henning Dierks 14

Minimal DBM Substraction
Alexandre David, John Håkansson, Kim G. Larsen, and Paul Pettersson 17

The threads are comming: New programming challenges for the masses
Erik Hagersten 21

Color-blind Behavioral Specifications for Transformations of Reactive Synchronous Programs
Andrzej Wasowski, Ulrik Larsen, and Kim G. Larsen 22

A formal approach to model-driven engineering
Dubravka Ilic and Elena Troubitsyna 25

Cofree Coalgebras for Signature Morphisms
Uwe Wolter 27

Transfinite Corecursion
Härmel Nestra 30

Scenario(MSC)-Based Specifications: a Survey
P.S. Thiagarajan 33

Compositional Specification and Verification of UML Models
Marcel Kyas, and Frank S. de Boer 34

Refining UML interactions
Ragnhild Kobro Runde 36

An Analysis Tool for UML SPT Models
John Håkansson and Leonid Mokrushin 39

The Controlled Linear Programming Problem
Henrik Björklund, Olle Nilsson, Ola Svensson, and Sergei Vorobyov . 45

Programming Languages Capturing Complexity Classes
Lars Kristiansen och Paul J. Voda 48

Coin Games
Rafal Somla 52

Automated Black-Box Testing of Functional Correctness using Function Approximation
Karl Meinke 53

Specifying Test Cases Using Observer Automata
Johan Blom, Anders Hessel, Bengt Jonsson, and Paul Pettersson . . . 55

Modeling and Simulating an Industrial Robot
Johan Andersson, Pavel Krčál, Leonid Mokrushin, Christer Norström, Anders Wall, and Wang Yi 58

A simulator approach to data race detection
Karl Marklund and Björn Victor 61

Encoding the Applied π-calculus in the π-calculus
Anders Bloch, Morten V. Frederiksen, and Bjørn Haagensen 62

A Timed Mobile Calculus
Jing Chen 65

Axioms and Algorithms for True Concurrency Equivalences
Michael Kannellos, and Mila Majster-Cederbaum 68

Slicing CSP-OZ Specifications
Ingo Brückner 71

Generic implementations of process calculi in Isabelle
Jesper Bengtsson 74

Inference of Timed Transition Systems
Olga Grinchtein, Bengt Jonsson, and Martin Leucker 79

Comparative Linear-Time Semantics on Action-Labelled Continuous-Time Markov Chains
Verena Wolf and Christel Baier 80

A Robust Interpretation of Duration Calculus
Martin Fränzle and Michael R. Hansen 83

Implementing stronger encapsulation for Java-like languages
Ville Leppänen and Sami Mäkelä 86

Object Calculus and Self-Application
Neil Ghani, and Johan Glimming 88

A Hoare Logic for Distributed Objects with Asynchronous Method Calls
Johan Dovland, Einar Broch Johnsen, and Olaf Owe 90

Optimal Scheduling and Priced Timed Automata
Kim G. Larsen 93

Abstraction Based Analysis and Arbiter Synthesis: Radar Memory Interface Card Case Study Revisited
Juhan Ernits 94

Timed Traces and Strand Spaces
Robin Sharp and Michael R. Hansen 96

Intelligent Sensory Using Compact Data Structures and Bayesian Networks
Jens A. Hansen 99

Universal Semi-local Election Protocol Using Forward Links
Dobieslaw Wróblewski 102

Towards programming logics for low-level languages
Ando Saabas, Gilles Barthe, Lilian Burdy, and Tamara Rezk 105

A Program Inverter LRinv and its Structure
Masahiko Kawabe and Robert Glück 108

Extensions of Event Based B for Development of Grid Systems
Pontus Boström and Marina Waldén 110

Sequent Calculi for Temporal Logics of Common Knowledge and Belief
Jüraté Sakalauskaité 113

The size-change approach to program termination analysis

Neil D. Jones

DIKU, University of Copenhagen

Abstract

The "size-change termination" principle for a language with well-founded data is: a program terminates on all inputs if every infinite call sequence (following program control flow) would cause an infinite descent in some data values. Although conceptually simple, many programs terminate for this reason, including programs with mutual recursion and with parameter swapping.

Size-change analysis is based only on local approximations to parameter size changes derivable from program syntax. The condition on tracing data flow along infinite call sequences turns out to be decidable, and not difficult to automate. There is no need for human invention, e.g., of lexicographic or similar orders on data and functions.

First developed for first-order functional programs (POPL'01), the method has since been extended to ensure termination of a partial evaluator (GCSE/SAIG'02), and the untyped lambda-calculus (RTA'04 with N. Bohr). Current work with D. Sereni analyses the termination of programs in a substantial subset of ML.

Memory Usage Estimation for Java Cards

Gerardo Schneider*

IRISA/CNRS, Campus de Beaulieu F-35042 Rennes, France
{gerardo@irisa.fr}

1 Introduction

Embedded systems, i.e. special-purpose computer systems built into larger devices, are widespread and can be found in simple gadgets like coffee machines and in more complex ones like mobile phones and smart cards. Programmable smart cards are small personal devices provided with a microprocessor capable of manipulating confidential data, allowing the owner of the card to have secure access to chosen applications. Applications, called applets, can be downloaded and executed in these small communicating devices, raising major security issues. Indeed, without appropriate security measures, a malicious applet could be installed in smart cards destroying data, disclosing private information over the network or performing non-authorised credit card transactions.

The massive use of such small communicating devices is imminent. Hence, it is essential that their platforms, as well as the applets that will run on them, can provide some minimal security guarantee in order to preserve confidentiality, integrity and availability of information. To guarantee availability of the services offered by small devices the management and control of resources (e.g. memory) is a crucial issue, mainly due to their limitation on size.

Concerning memory management on Java smart cards, we quote Chen: "Because memory in a smart card is very scarce, neither persistent nor transient objects should be created willy-nilly" [3, Section 4.4]. Chapter 13 of the same book provides programming tips on how to optimise applet creation considering the memory limitations. Here are some of the recommendations for applet programmers related to our work: "You should also limit nested method invocations, which could easily lead to stack overflow. In particular, applets should not use recursive calls." ([3, Section 13.3]). Later, on Section 13.4, it says: "An applet should always check that an object is created only once". Even though the above tips are followed by industry when programming applets and are in general respected, mistakes regarding memory usage –like the creation of an object inside a loop– both accidentally or intentionally, may have non-desirable consequences. In fact, there is nothing in the standards which prevents a(n) (intentionally) badly written applet to allocate all persistent memory on a card. Hence, the detection of recursive methods and loops in byte codes for embedded systems (smart cards in particular) is a crucial issue to take into account.

As far as we are aware there is no available tool for detecting the dynamic creation of objects inside cycles and/or recursive functions for Java smart cards. The byte code verifier [4], for instance, does not verify properties related to memory usage. There are, undoubtedly, many ways of writing algorithms for detecting syntactic loops, mutually recursive methods and over-approximating the dynamic memory used for an assembler-like language. In this work, however, we propose a constraint-based algorithm, presented as a set of rules –one for each instruction– for solving the above-mentioned memory-related concerns. The formalism chosen was not governed by mere aesthetic reasons: it is compositional and it allows to obtain a certified analyser using the program-as-proof paradigm (i.e., following the Curry-Howard isomorphism) to extract a program directly from the proof of its correctness. One feature of our algorithm is that it works directly

* Partially supported by the *RNTL* French project, CASTLES (*Conception d'Analyses Statiques et de Tests pour le Logiciel Embarqué Sécurisé*).

on the byte code and there is no need to build extra data structures (e.g., a control-flow graph). Moreover, the byte code considered is arbitrary, i.e. we do not have any assumption about the byte code being produced by a "good" compiler. Our approach includes both intra- and inter-procedural analysis. Furthermore, the formalism used is compatible with existing works done at IRISA [2], which will allow to reuse the constraint solver, the fix-point algorithm and some of the lattice libraries "implemented" in Coq [7].

The language being considered is JCVMLe [8], a *byte code* language that models the Java Card Virtual Machine Language. It manipulates (dynamic) objects and arrays and besides the usual stack and register operations it comprises interesting features like virtual method calls, (mutually) recursive methods, (un)conditional jumps and subroutines. We assume there is no garbage collector, which is the case for Java Card upto version 2.1. Starting with Java Card 2.2 the machine includes a garbage collector which may be activated invoking an API function at the end of the execution of the applet, not during execution.

Our technique lies within the scope of *static analysis* [6], which analyses some run-time properties of a program without executing it. The implementation of static analysis is generally based on the solution of constraints through the computation of fix-points.

2 A constraint-based algorithm

Given a program P, we associate a set of constraints to each program instruction (applying rules like the ones shown in Fig. 1). Essentially, our algorithm add program points (m, pc) to a certain context Γ for each instruction (m, pc) : `new` o, if it is not in a loop nor in a mutually recursive method. If such instruction is inside a cycle and/or a (mutually) recursive method, then $<!>_{(m,pc)}$ is added to Γ instead.

The least solution to the set of constraints associated to P is obtained by computing the least fix-point of a certain functional F arising from such set. By a corollary of Tarski's Fixed Point theorem, the solution may be obtained as the limit of the stabilising sequence $(F^n(\bot))_{n\in\mathbb{N}}$.

Because of lack of space we only present in this abstract some of the rules of our main algorithm and we explain only the first rule (see Fig. 1). If the current instruction at program counter pc in method m is `new` o, and the instruction is inside a loop (detected by the predicate $Loop$) or a (mutually) recursive method (predicate Rec) then the rule propagates the *warning* type $<!>_{(m,pc)}$ to $\Gamma(m, pc + 1)$. Thus, at program point $(m, pc + 1)$, Γ will contain at least everything it had at (m, pc), plus $<!>_{(m,pc)}$. The content of Γ is thus "propagated" by the constraints associated to each instruction, as defined by each rule. The rules for `invokevirtual` and subroutine calls are similar to the `invokedefinite`; for any other instruction, $\Gamma(m, pc) \sqsubseteq \Gamma(m, pc + 1)$.

We have defined, in a similar way, rules for computing the predicate $Loop$ –which detects intra-procedural cycles and instructions reachable from such cycles through method calls– and Rec, which computes all the mutually recursive methods. In fact, the technical difficulties are mainly presented in the computation of these last two predicates. In particular, detecting loops in a byte code program is more difficult than in a high-level language like Java, mainly due to the lack of structure, allowing jumps from (and to) the middle of a cycle.

3 Final Discussion

Besides the advantages mentioned in the introduction, the modularity of our algorithm allows the analyser itself, as well as the predicates $Loop$ and Rec, to be reused by other constraint-based analysers as predicates (programs) which have been proved correct.

$$\frac{(m,pc) : \texttt{new}\ o \qquad Loop(m,pc) \vee Rec(m,pc)}{\Gamma(m,pc) \cup \{<!>_{(m,pc)}\} \sqsubseteq \Gamma(m,pc+1)} \qquad \frac{(m,pc) : \texttt{new}\ o \qquad \neg Loop(m,pc) \wedge \neg Rec(m,pc)}{\Gamma(m,pc) \cup \{(m,pc)\} \sqsubseteq \Gamma(m,pc+1)}$$

$$\frac{(m,pc) : \texttt{if}\ c\ \texttt{goto}\ pc'}{\begin{array}{l}\Gamma(m,pc) \sqsubseteq \Gamma(m,pc+1) \\ \Gamma(m,pc) \sqsubseteq \Gamma(m,pc')\end{array}} \qquad \frac{(m,pc) : \texttt{invokedefinite}\ m'}{\begin{array}{l}\Gamma(m,pc) \sqsubseteq \Gamma(m',1) \\ \Gamma(m',\mathrm{END}_{m'}) \sqsubseteq \Gamma(m,pc+1)\end{array}}$$

$$\frac{(m,pc) : \texttt{return}}{\Gamma(m,pc) \sqsubseteq \Gamma(m,\mathrm{END}_m)} \qquad \frac{(m,pc) : \texttt{goto}\ pc'}{\Gamma(m,pc) \sqsubseteq \Gamma(m,pc')}$$

Fig. 1. Rules of the main algorithm

Current Work. We have proved the termination of the main algorithm and of the two predicates $Loop$ and Rec. To prove termination, we have defined (finite) lattices defined on the range of the program counter numbers and method names. We are currently working on a hand-written proof of soundness w.r.t. to an abstraction of the operational semantics.

Future Work. Our approach allows the rules to be fed into the proof assistant Coq [1]. We intend to prove the correctness of our algorithm in the constructive logic of Coq and to use its extraction mechanism to automatically obtain a certified analyser. Besides its application on smart cards, we believe we can apply our analyser to mobile phone technology, where the detection of loops and recursive methods in applets is crucial, to avoid for instance the saturation of communication by sending SMS' indiscriminately.

Related Work. Our approach is inspired by the work in [2] where the technique sketched here has been applied with success for extracting a data flow analyser for Java card byte code programs. Even though the motivations and the techniques are not the same, it is worth mentioning the work done in the Mobile Resource Guarantees project [5], which applies ideas from proof-carrying code for solving the problem of resource certification for mobile code,

References

1. Y. Bertot and P.Casteran. *Interactive Theorem Proving and Program Development. Coq'Art : the Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.
2. D. Cachera, T. JeMobile Resource Guarantees projectnsen, D. Pichardie, and V. Rusu. Extracting a data flow analyser in constructive logic. In *ESOP*, LNCS, pages 385–400, 2004.
3. Zhiqun Chen. *Java Card technology for Smart Cards: architecture and programmer's guide*. Java series. Addison Wesley, 2000.
4. Xavier Leroy. Bytecode verifi cation on java smart cards. *Softw. Pract. Exper.*, 32(4):319–340, 2002.
5. MRG. Mobile Resource Guarantees project. See `http://groups.inf.ed.ac.uk/mrg/`.
6. Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., 1999.
7. David Pichardie. Coq sources of the development corresponding to the paper [2]. See `http://www.irisa.fr/lande/pichardie/CarmelCoq/`.
8. I. Siveroni and C. Hankin. A proposal for the JCVMLe operational semantics. Research report SECSAFE-ICSTM-001-2.2, October 2001.

STAGE-PRESERVING EMBEDDINGS OF LANGUAGES

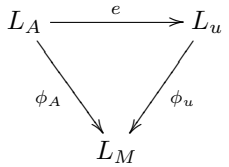
TODD L. VELDHUIZEN, CHALMERS INSTITUTE OF TECHNOLOGY

Embeddings of languages into one another are useful in studying their relative power and, sometimes, finding languages that are *universal* in some sense. Examples include Turing-reducibility for studying computability, poly-time reductions for subrecursive languages [14], and structure-preserving embeddings for expressiveness [4, 5, 13, 12]. To further a search for languages suitable for "domain-specific embedded languages" and "active libraries," [7, 3] we propose stage-preserving embeddings as a tool to study languages in which some evaluation or simplification is guaranteed to take place at compile-time. Such guarantees can be wielded to realize domain-specific optimizations. We prove sufficient conditions for a language to be "stage-universal," in the sense that there is a stage-preserving embedding of any staged language into it.

Assume a class of languages that are all being compiled to the same implementation language L_M — the subscript M here standing for *machine*. A compiler for some source language L_A is a partial map $\phi_A : L_A \rightharpoonup L_M$; partial because for many interesting staged languages (e.g., C++, MetaML [15]) the compiler may not terminate. To speak of embeddings being semantics-preserving, we stipulate an equivalence $\sim$ capturing some appropriate behavioural notion of program equivalence; the details are unimportant.

Definition 1. An embedding $e : L_A \to L_u$ is a total, decidable function that is semantics-preserving: $\phi_A(p) \sim \phi_u(ep)$ for all $p \in L_A$.

The typical scenario we consider is illustrated by this diagram:



We have two languages L_A and L_u, compilers ϕ_A and ϕ_u for them, and we consider an embedding $e : L_A \to L_u$. We ask ask when embeddings that preserve properties of interest (semantics, staging, safety) exist. The scenario of special interest is when L_u is some language purporting to be 'universal.'

0.1. **The kernel of a compiler.** We use *staging* to address compile-time computations (cf. [10, 8, 15]). We are interested in embeddings that are stage-preserving: if a computation occurs at compile time in language L_A, then it occurs at compile time in language L_u. This can be conveniently addressed using the *kernel* of the compiler. Recall that the kernel of a map ϕ is:

$$\ker(\phi) = \{(p_1, p_2) \mid \phi(p_1) = \phi(p_2)\}$$

The kernel is an equivalence relation on programs; every program in an equivalence class is compiled to the same program. Kernels capture staging: for example, a language whose compile-time evaluations are defined by a rewrite relation $\rightarrow$ must satisfy $\rightarrow \subseteq \ker(\phi)$, where ϕ is its compiler.

We can view the kernel as a *staging specification*,[1] and use it to formalize the notion of a stage-preserving embedding.

Definition 2. An embedding $e : L_A \rightarrow L_u$ is stage-preserving when it satisfies $(p_1, p_2) \in \ker(\phi_A) \;\Rightarrow\; (ep_1, ep_2) \in \ker(\phi_u)$.

The kernel of a compiler gives us a measure of its staging power, that is, its ability to perform computations at compile time. If one language has greater staging power than another, then it can subsume more languages with staging capabilities. Defn. 2 effectively says: to increase the staging power of a language, make its kernel larger. But at what point is a kernel "big enough" to subsume any staged language?

Let us write $L_A \leq_S L_B$ to mean there exists a stage-preserving embedding $e : L_A \rightarrow L_B$. The relation $\leq_S$ is a preorder. The obvious question is whether there might exist maximal elements; we call such languages *stage-universal*.

Definition 3. A language is stage-universal when there is a stage-preserving embedding of any other language into it.

The term stage-complete would do equally well. The usual notion of Turing- or Σ_1^0-completeness can be used to give necessary conditions for such languages. We stipulate that all languages and compilers under discussion are Σ_1^0, i.e. computably enumerable under suitable coding. We make the standard assumption that there is an effective coding $\ulcorner \cdot \urcorner$ of the languages L_A, L_M as terms of L_u (e.g., [2, 1]). If $p \in L_A$ is a program then $\ulcorner p \urcorner$ may be thought of as a representation of p by its parse tree, as a string of characters, or (more traditionally) a very large natural number; the particulars do not matter so long as the encoding is unique and computable. We will moreover assume that L_u permits the construction of functions over codes (e.g., parse trees) sufficient to realize (for example) interpreters, and will write $F(c)$ to mean the application of such a function F to a code c.

It is useful to distinguish between functions implemented in L_u, e.g., maps over codes, and *programs* that can take such codes and produce behaviour. For a program P taking as argument some code x, we write $P[x]$.

Definition 4. An *interpreter* for the machine language L_M in the language L_u is a program $I_M[\cdot]$ such that for every machine-language program $p_m \in L_M$, the interpreted version of p_m is equivalent to p_m:

$$\phi_u(I_M[\ulcorner p_m \urcorner]) \sim p_m$$

That is, if we take some machine-language program p_m and 'code' it as (for example) a syntax tree $\ulcorner p_m \urcorner$ and give it to the interpreter I_M, then I_M running $\ulcorner p_m \urcorner$ behaves the same way as the program p_m. The existence of such an interpreter ensures that the language L_u does not lose basic capabilities of the language L_M, such as the

[1] The kernel is related to, but different from, binding-time specifications (cf. [8, 9]): the kernel indicates which programs will compile to the same target program, whereas binding-times indicate which terms are replaceable by constants. These two ideas coincide in some situations, e.g., when programs are terms, the compilation map is compositional, and only partial evaluation is taking place.

ability to interact with the operating system and so forth. This is of concern when dealing with interactive systems (a.k.a. processes, reactive systems, etc.) rather than purely functional programs. More formally, it guarantees that ϕ_u is *onto* the equivalence classes $L_u/\sim$ giving the possible *behaviours* (e.g., [11]) of L_M programs. That is, for every "machine-language" program $p_m \in L_M$, there is a program $p_u \in L_u$ such that $\phi_u(p_u) \sim p_m$, i.e., p_m and $\phi_u(p_u)$ have the same behaviour.

What we need next is some vocabulary to discuss compile-time computations.

Definition 5. A partial function f is *realizable in the kernel* of ϕ_u if there exists an L_u-language function F such that for any program P taking as argument a code, and x, y such that $y = f(x)$:

$$\phi_u(P[F(\ulcorner x \urcorner)]) = \phi_u(P[\ulcorner y \urcorner])$$

Or, equivalently, $(P[F(\ulcorner x \urcorner)], P[\ulcorner y \urcorner]) \in \ker(\phi_u)$.

This means, more or less, that the partial function F is evaluated at compile time.

We now give a sufficient condition for stage-universality, inspired by ideas from partial evaluation, in particular Jones-optimality [9] and the Futamura projections [6]. The proof is boilerplate computability theory and partial evaluation. We rely on the assumption (stated earlier) that compilers are Σ_1^0 functions.

Theorem 1. *If (1) there is an interpreter $I_M[\cdot]$ for L_M in L_u; and (2) any Σ_1^0 function f is realizable in the kernel of ϕ_u, then the language L_u is stage-universal.*

Proof. Pick a language and compiler L_A and ϕ_A. Since ϕ_A is Σ_1^0, by (2) there is a L_u-function Φ_A realizing it such that if $p_m = \phi_a(p_a)$ then $\phi_u(P[\Phi_A(\ulcorner p_a \urcorner)]) = \phi_u(P[\ulcorner p_m \urcorner])$ for any program P taking a code-argument.

Consider the embedding $e : L_A \to L_u$ given by:

$$e(p_a) = I_M[\Phi_A(\ulcorner p_a \urcorner)]$$

where $I_M[\cdot]$ is the L_m interpreter whose existence is ensured by (1). Recall from Defn. 2 that e is stage preserving when $(p_1, p_2) \in \ker(\phi_a) \Rightarrow (ep_1, ep_2) \in \ker(\phi_u)$. Choose p_1, p_2 such that $(p_1, p_2) \in \ker(\phi_a)$. Then there is a p_m such that $\phi_a(p_1) = \phi_a(p_2) = p_m$, and from the choice of Φ_A,

$$\phi_u(I_M[\Phi_A(\ulcorner p_1 \urcorner)]) = \phi_u(I_M[\ulcorner p_m \urcorner]) \qquad \text{and}$$
$$\phi_u(I_M[\Phi_A(\ulcorner p_2 \urcorner)]) = \phi_u(I_M[\ulcorner p_m \urcorner])$$

Therefore $\phi_u(ep_1) = \phi_u(ep_2)$, or $(ep_1, ep_2) \in \ker(\phi_u)$, and the embedding e is stage-preserving. Since such an embedding exists for any language L_A, the language L_u is stage-universal. □

The construction in the proof above is not of immediate practical use; there is no guarantee that an interpreted program $\phi_u(I_M[\Phi_A(\ulcorner p \urcorner)])$ will run anywhere near as fast as $\phi_A(p)$ (cf. Jones-optimality [9]). It does, however, give sufficient conditions for languages to be stage-universal. The construction above would be useful if ϕ_u found programs that were 'optimal.' That is, if the compiler ϕ_u were to find fastest, smallest, etc. programs, then the construction $\phi_u(I_M[\Phi_A(\ulcorner p \urcorner)])$ *would* be practical. Finding optimal programs is Δ_2^0 hard, so this goal is not reachable. However, if we find programs that are near to optimal, then approaches nearing the construction of Theorem 1 might be practical. In [16] we describe a method for realizing such compilers, via "Guaranteed Optimization," a new compiler design technique.

References

[1] Ash, C. J., and Knight, J. *Computable structures and the hyperarithmetical hierarchy*. North-Holland Publishing Co., Amsterdam, 2000.

[2] Boolos, G. S., and Jeffrey, R. C. *Computability and Logic: Third Edition*. Cambridge University Press, 1989.

[3] Czarnecki, K., Eisenecker, U., Glück, R., Vandevoorde, D., and Veldhuizen, T. Generative programming and active libraries (extended abstract). In *Generic Programming '98. Proceedings* (2000), M. Jazayeri, D. Musser, and R. Loos, Eds., vol. 1766 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 25–39.

[4] de Boer, F. S., and Palamidessi, C. Embedding as a tool for language comparison. *Information and Computation 108*, 1 (Jan. 1994), 128–157.

[5] Felleisen, M. On the expressive power of programming languages. In *Selected Papers of ESOP '90*, N. D. Jones, Ed., vol. 17. Elsevier, Dec. 1991.

[6] Futamura, Y. Partial evaluation of computation process - an approach to a compiler-compiler. *Systems, Computers, Controls 2*, 5 (1971), 45–50.

[7] Hudak, P. Building domain-specific embedded languages. *ACM Computing Surveys 28*, 4es (Dec. 1996), 196–196.

[8] Jones, N. D. An introduction to partial evaluation. *ACM Computing Surveys 28*, 3 (Sept. 1996), 480–503.

[9] Jones, N. D., Gomard, C. K., and Sestoft, P. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[10] Jørring, U., and Scherlis, W. L. Compilers and staging transformations. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (1986), ACM Press, pp. 86–96.

[11] Kurz, A. Coalgebras and modal logic. ESSLLI 2001 course notes, 2001.

[12] Matsushita, T. *Expressive Power of Declarative Programming Languages*. PhD thesis, University of York, Department of Computer Science, October 1998.

[13] Mitchell, J. C. On abstraction and the expressive power of programming languages. *Science of Computer Programming 21*, 2 (Oct. 1993), 141–163. Selected papers of the Conference on Theoretical Aspects of Computer Software (TACS '91) (Sendai, 1991).

[14] Royer, J. S., and Case, J. *Subrecursive Programming Systems: Complexity and Succinctness*. Birkhauser, 1994.

[15] Taha, W., and Sheard, T. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science 248*, 1–2 (Oct. 2000), 211–242.

[16] Veldhuizen, T. L. *Active Libraries and Universal Languages*. PhD thesis, Indiana University Computer Science, May 2004.

Signals and Comonads (Abstract)

Tarmo Uustalu[1] and Varmo Vene[2]

[1] Institute of Cybernetics, Tallinn University of Technology
Akadeemia tee 21, EE-12618 Tallinn, Estonia, tarmo@cs.ioc.ee
[2] Dept. of Computer Science, University of Tartu
J. Liivi 2, EE-50409 Tartu, Estonia, varmo@cs.ut.ee

Following the seminal work of Moggi and Wadler [8, 12], it has become standard in programming semantics and functional programming to analyse various notions of computation with side-effects as *monads*. Recently, however, it has also been discussed whether there is a need for finer and more permissive mathematical abstractions beyond monads to get a better uniform grip on the very diverse function-like concepts one encounters in programming. Notably for our topic of interest here, Hughes [6] has promoted *arrows* as an abstraction which is especially handy in programming with signals or flows. His idea has been taken up in functional reactive programming [9] and there exists by now not only an arrow library in Haskell but even specialized syntax [10].

The exists however also a considerably more standard and elementary categorical concept of *comonad* which can be used to capture notions of values-in-context. This fact has received some attention [3, 7], but in general comonads have not found extensive use in semantics or programming. We claim that this unfair and argue that comonads can be a helpful tool in programming with signals and in the semantics of corresponding specialized programming languages. In the paper, we take a closer look at streams as signals in discrete time.

Stream programming is about stream-transforming functions $\mathsf{Str}\,A \to \mathsf{Str}\,B$ where $\mathsf{Str}\,A = \nu X.\, A \times X \cong \mathsf{Nat} \Rightarrow A$ is the type of streams (i.e., infinite sequences) over a given type A. One can either allow arbitrary stream functions or, more interestingly, only those that are causal in the sense that the nth element in the output stream must be determined by the elements of the input stream up to the nth position. Obviously the causal approach is more realistic, if streams are meant to represent signals. Both non-causal and causal stream functions are equivalent to Kleisli arrows of a comonad. The non-causal comonad D is $DA = \mathsf{Str}\,A \times \mathsf{Nat} \cong (\mathsf{Nat} \Rightarrow A) \times \mathsf{Nat} \cong \mathsf{NEList}\,A \times \mathsf{Str}\,A$ whereas the causal comonad D^c is $D^cA = \mathsf{NEList}\,A$ where $\mathsf{NEList}\,A = \mu X.\, (1 + X) \times A$ is the type of non-empty (snoc-)lists over A. The idea is to use non-empty lists to represent histories (inclusive of the present moment) and strings to represent futures. The causal comonad is special in being the cofree recursive comonad over the functor $1 + (-)$ where a recursive comonad is the dual of a completely iterative monad in the sense of [1] (cf. also [11]). This implies that for any map $e : D^c(B \times A) \to B$ factoring in the obvious canonical way through a map $(1 + D^c(B \times A)) \times A \to B$ (a "guarded equation") there exists a unique map $s : D^cA \to B$ satisfying $s = e \circ \langle s, \varepsilon_A \rangle^\dagger$ (the "solution"), which really gives a feedback combinator. Notably, not only the potential solutions, but also the guarded equations are arrows, so also they make good representations of causal stream functions.

We show Haskell implementations of stream programming combinators for both comonads, compare the comonad-based disciplines of programming with streams to the arrow-based counterparts, and discuss semantic description of intensional languages like Lucid [2] and synchronous dataflow languages like Lustre and Lucid Synchrone (Lustre+ML) [5, 4] in terms of the non-causal resp. causal comonad.

Acknowledgements This work in progress is being partially supported by the Estonian Science Foundation under grant No. 5567.

References

1. P. Aczel, J. Adámek, S. Milius, and J. Velebil. Infinite trees and completely iterative theories: A coalgebraic view. *Theoret. Comput. Sci.*, 300(1–3):1–45, 2003.

2. E. A. Ashcroft and W. W. Wadge. *LUCID, The Dataflow Programming Language*. Academic Press, New York, 1985.
3. S. Brookes and S. Geva. Computational comonads and intensional semantics. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, editors, *Applications of Categories in Computer Science*, vol. 177 of *London Math. Society Lecture Note Series*, pp. 1–44. Cambridge University Press, Cambridge, 1992.
4. P. Caspi and M. Pouzet. Synchronous Kahn networks. In *Proc. of 1st ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'96*, pages 226–238. ACM Press, New York, 1996. Also in *SIGPLAN Notices*, 31(6):226–238, 1996.
5. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proc. of the IEEE*, 79(9):1305–1320, 1991.
6. J. Hughes. Generalising monads to arrows. *Sci. of Comput. Program.*, 37(1–3):67–111, 2000.
7. R. Kieburtz. Codata and comonads in Haskell. Unpublished manuscript, July 1999.
8. E. Moggi. Notions of computation and monads. *Inform. and Computat.*, 93(1):55–92, 1991.
9. H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Proc. of 2002 ACM SIGPLAN Wksh. on Haskell, Haskell'02*, pp. 51–64. ACM Press, New York, 2002.
10. R. Paterson. A new notation for arrows. In *Proc. of 6th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'01*, pp. 229–240. ACM Press, New York, 2001. Also in *SIGPLAN Notices*, 36(10):229–240, 2001.
11. T. Uustalu and V. Vene. The dual of substitution is redecoration. In K. Hammond and S. Curtis, editors, *Trends in Functional Programming 3*, pp. 99–110. Intellect, Bristol / Portland, OR, 2002.
12. P. Wadler. The essence of functional programming. In *Conf. Record of 19th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'92*, pp. 1–14. ACM Press, New York, 1992.

Simulation-Based Iteration of Tree Transducers

Parosh Aziz Abdulla[1], Axel Legay[2*], Julien d'Orso[3], and Ahmed Rezine[1]

[1] Dept. of Information Technology
Uppsala University
P.O. Box 337
SE-751 05 Uppsala, Sweden
{parosh,rahmed}@it.uu.se
[2] Université de Liège
Institut Montefiore, B28
4000 Liège, Belgium
legay@montefiore.ulg.ac.be
[3] IRCCyN - UMR CNRS 6597
BP 92 101
44321 Nantes CEDEX 03, France
julien.dorso@irccyn.ec-nantes.fr

"Regular model checking" is the name of a family of techniques for analyzing infinite-state systems in which states are represented by words, sets of states by finite automata on these objects, and transitions by finite automata operating on pairs of state, i.e. finite-state transducers. In this context, the central problem is then to compute the iterative closure of a finite-state transducer. Such a representation allows to compute the set of reachable states of the system (which is useful to verify safety properties) and to detect loops between states (which is useful to verify liveness properties). However, computing transitive closure is in general undecidable (it can be shown that a transducer is Turing powerful); consequently any method for solving the problem is necessarily incomplete. One of the goals of Regular model checking is to provide semi-algorithms which terminate on many practical applications. Such semi-algorithms have already successfully been applied to parameterized systems with linear topologies, and to systems that operate on linear unbounded data structures such as queues, stacks, integers, reals, hybrid systems [BJNT00,BLW03,BLW04,DLS01], and recently on heterogenous systems [Leg].

This work aims at extending the paradigm of Regular model checking to verify systems which operate on tree-like architectures. This include many interesting communication protocols such as the percolate protocol ([KMM+01]) or the Tree-arbiter protocol ([ABH+97]). The work is mainly motivated by the existence of systems for which it is not possible to express the set of reachable states with a "regular-word" encoding but possible with a "regular-tree" encoding.

To verify such systems, we use the extension of "regular model checking" called "tree regular model checking", which was first introduced in [AJMd02] and [BT02]. In "tree regular model checking", states of the systems are represented by trees, sets of states by tree automata, and transitions by tree automata operating on pairs of state, i.e. tree transducers. As for the case with "regular

* This author is supported by a F.R.I.A grant.

model checking", the central problem is to provide semi-algorithms for computing the transitive closure of a tree transducer. This problem was already considered in [AJMd02,BT02], however the proposed algorithms are most of the time inefficient or non-implementable.

In this work, we propose an efficient and implementable semi-alogrithm for computing the transitive closure of a tree transducer. Starting from a tree transducer D, describing the set of transitions of the system, we derive a transducer, called the *history transducer* whose states are *columns* (words) of states of D. The history transducer characterizes the transitive closure of the rewriting relation corresponding to D. The set of states of the history transducer is infinite which makes it inappropriate for computational purposes. In this talk we present a method for computing a finite-state transducer which is an abstraction of the history transducer. The abstract transducer is generated on-the-fly by a procedure which starts from the original transducer D, and then incrementally adds new transitions and merges equivalent states. To compute the abstract transducer, we define an equivalence relation on columns (states of the history transducer). To be a *good* equivalence relation (i.e. an equivalence relation that can be used by our on-the-fly algorithm), a relation on columns of the history transducer must at least satisfy the following sufficient conditions.

- *Soundness and completeness:* we show that merging two equivalent columns will not add any traces which are not present in the history transducer. Consequently, the abstract transducer accepts the same language as the the history transducer (and therefore characterizes exactly the transitive closure of D).
- *Computability of the equivalence relation:* this allows on-the-fly merging of equivalent states during the generation of the abstract transducer.

We provide an equivalence relation induced by two simulation relations (but other ways are possible); namely a *downward* and an *upward* simulation relation, both of which are defined on tree automata and computed by efficient algorithms derived from those of Henzinger *et al.* ([HHK95]) for finite words. Moreover, we show an example of a downward and an upward simulation which satisfy the sufficient conditions needed to induce the equivalence relation.

We also show that our technique can be directly adapted in order to compute the set of reachable states of a system without computing its entire transitive closure. When checking for safety properties, such an approach is in general more efficient.

We have implemented our algorithms, and report the results of their application on a number of protocols including a Two-Way Token protocol, the Percolate Protocol ([KMM+01]), a parametrized version of the Tree-arbiter protocol ([ABH+97]), and a tree-parametrized version of a leader election protocol. The results are very encouraging but a lot of implementation work is still needed to obtain better ones.

References

[ABH+97] R. Alur, R.K. Brayton, T.A. Henzinger, S. Qadeer, and S.K. Rajamani. Partial-order reduction in symbolic state space exploration. In O. Grumberg, editor, *Proc. 9^{th} Int. Conf. on Computer Aided Verification*, volume 1254, pages 340–351, Haifa, Israel, 1997. Springer Verlag.

[AJMd02] Parosh Aziz Abdulla, Bengt Jonsson, Pritha Mahata, and Julien d'Orso. Regular tree model checking. In *Proc. 14^{th} Int. Conf. on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, 2002.

[BJNT00] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In Emerson and Sistla, editors, *Proc. 12^{th} Int. Conf. on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 403–418. Springer Verlag, 2000.

[BLW03] Bernard Boigelot, Axel Legay, and Pierre Wolper. Iterating transducers in the large. In *Proc. 15^{th} Int. Conf. on Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 223–235, 2003.

[BLW04] Bernard Boigelot, Axel Legay, and Pierre Wolper. Omega regular model checking. In *Proc. TACAS '04, 10^{th} Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, 2004. to appear.

[BT02] Ahmed Bouajjani and Tayssir Touili. Extrapolating Tree Transformations. In *Proc. 14^{th} Int. Conf. on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, 2002.

[DLS01] D. Dams, Y. Lakhnech, and M. Steffen. Iterating transducers. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, 2001.

[HHK95] M. Henzinger, T. Henzinger, and P. Kopke. Computing simulations on finite and infinite graphs. In *Proc. 36^{th} Annual Symp. Foundations of Computer Science*, pages 453–463, 1995.

[KMM+01] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. *Theoretical Computer Science*, 256:93–112, 2001.

[Leg] A. Legay. Heterogenous infinite state systems. report.

Heuristic Guided Model-Checking of Real-Time Systems*

Henning Dierks

September 3, 2004

Abstract

We present an approach to model-check real-time systems by the cost-optimising version of Uppaal. The additional features for heuristic guidance of the cost-optimising version improve the performance in finding error traces. An important precondition for successful heuristics are contextual information about the system. To demonstrate our approach we introduce a mutual exclusion problem of a real-time system specified in terms of PLC-Automata. Finding errors by model-checking the semantics in terms of timed automata is substantially improved by heuristics guidance. Our heuristics are derived by exploiting the contextual information that the timed automata system to be checked represents the semantics of PLC-Automata.

1 Introduction

A main advantage of model-checking is that there is always a result, either a positive one or a negative one together with a counter-example provided that there are sufficient resources of time and space. In the usual verification context the positive answer is desired and to this end a model-checker has to explore the full state space. Therefore, model-checkers typically do not provide special features to find counter-examples fast. However, there are other contexts in which it is desirable to find the counter-example fast:

- When a model-checker is applied to find (optimal) schedules or plans. Recent approaches of this kind are published in [BFH+01, LBB+01, DBL02]. Here, the counter-examples found represent feasible (optimal) schedules or valid plans.
- In an abstraction refinement loop [CGJ+00, HJMS02]. Here the verification of a complex model starts with a very coarse abstraction. Model-checking this abstraction typically yields an abstract counter-example which is expected to be spurious. That means it cannot be found in the concrete model. The benefit of spurious counter-examples is that they provide information in which way the abstraction can be refined. This refined abstraction is model-checked again. As long as spurious counter-examples are the output the process is iterated. Clearly, this approach can only be useful if the model-checker in use can provide counter-examples fast.
- When an abstract counter-example is available. If an abstraction of system has been model-checked and the result is a counter-example, the obvious question is whether this counter-example is spurious or not. In this case the full model is checked against a test automaton which models the abstract counter-example such that reaching a certain state yields a trace that describes a concrete counter-example.

*This research was partially supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See `www.avacs.org` for more information.

We present an approach to find counter-examples faster in systems of timed automata. To this end we use the cost-optimising version of Uppaal [BFH+01] which allows the user to assign costs to both transitions and time. It is also possible to manipulate the search order by heuristic functions. The main observation is that this can reduce the time needed to find a counter-example. A precondition for such improvements is additional contextual information about the system. This information is not exploitable in the standard version of Uppaal because it is not expressible in the input language. The features of the cost-optimising Uppaal allow us to add this information, which sharpens Uppaal's focus on the critical points in the system, viz. on traces which are likely to be error traces.

We demonstrate this approach in the context of PLC-Automata [Die99], a specification language for distributed real-time systems. PLC-Automata have a timed automata semantics. When the semantics of a PLC-Automaton is fed into the standard version of Uppaal the model-checker does not get the information that it checks a PLC-Automaton. Hence, it handles all transitions and automata equally weighted. The core idea of our approach is that a model-checker benefits from guidance especially when there are criteria to investigate some parts of the state space earlier than others or to prefer some transitions of the model. For example, in the case of PLC-Automata it makes sense to avoid transitions that change variables which are under control of the environment. This distinction between "interesting" transitions and "less interesting" ones cannot not be done in the standard version of Uppaal. Note that the role of PLC-Automata is that of a guinea pig. Each formalism that is translated into a timed automata semantics is likely to have similar choices of interesting and less interesting transitions. We will present two ideas how to add contextual information of PLC-Automata by assigning costs and defining heuristic functions to sort the search space.

Because Uppaal is optimised for reachability properties it is often necessary to construct test automata that break a complex temporal property down to a reachability problem, e.g. the complex property $\mathcal{P}$ is satisfied iff a certain state q in the test automaton $\mathcal{T}(\mathcal{P})$ is reachable. In our setting test automata are generated from Constraint Diagrams [Die96, Kle00]. Again, the translation into test automata does not represent some structural information. If the model-checker learns a bit about this structure by costs and heuristics, the counter-examples are found significantly faster. The standard version of Uppaal cannot be taught which automata of the current system represent test automata and which represent the model.

2 Contents of the Presentation

In case of acceptance we will introduce briefly the notion of PLC-Automata which serves as an example of a language that has a semantics in terms of timed automata. We will demonstrate a set of ideas how to guide Uppaal to find error traces in erroneous systems faster. To this end we will show that this is especially effective provided that test automata are involved. We will also present the results for case studies ranging from toy examples to medium size[1]. In most cases we found errors at least 10 times faster and sometimes more than 100 times faster. Often the speedup cannot be computed since only the heuristic guided version terminated successfully whereas the standard version of Uppaal exceeded memory limits.

References

[BFH+01] G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Pettersson, and J. Romijn. Efficient Guiding Towards Cost-Optimality in Uppaal. In T. Margaria and Wang Yi, editors, *Pro-*

[1] For example, one real world case study consists of 10 automata with 3 clocks and 57 variables. There are also successes with models consisting of 8 automata, 8 clocks, and 18 variables.

ceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, volume 2031 of *LNCS*, pages 174–188. Springer, 2001.

[CGJ+00] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[DBL02] H. Dierks, G. Behrmann, and K.G. Larsen. Solving Planning Problems Using Real-Time Model-Checking (Translating PDDL3 into Timed Automata). In F. Kabanza and S. Thiebaux, editors, *AIPS-Workshop Planning via Model-Checking*, pages 30–39, April 2002.

[Die96] C. Dietz. Graphical Formalization of Real-Time Requirements. In B. Jonsson and J. Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1135 of *LNCS*, pages 366–385, Uppsala, Sweden, September 1996. Springer.

[Die99] H. Dierks. *Specification and Verification of Polling Real-Time Systems*. PhD thesis, University of Oldenburg, July 1999.

[HJMS02] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, pages 58–70. ACM Press, 2002.

[Kle00] C. Kleuker. *Constraint Diagrams*. PhD thesis, University of Oldenburg, December 2000.

[LBB+01] K. Larsen, G. Behrmann, E. Brinksma, A. Fehnker, T. Hune, P. Pettersson, and J. Romijn. As Cheap as Possible: Efficient Cost-Optimal Reachability for Priced Timed Automata. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of CAV 2001*, volume 2102 of *LNCS*, pages 493–505. Springer, 2001.

Minimal DBM Substraction

Alexandre David[1], Johan Håkansson[2], Kim G. Larsen[1], and Paul Pettersson[2]

[1] Department of Computer Science, Aalborg University, Denmark
{adavid,kgl}@cs.auc.dk.
[2] Department of Information Technology, Uppsala University, Sweden
{johnh,paupet}@it.uu.se.

Abstract. In this paper we present an algorithm to compute DBM substractions with a guaranteed minimal number of splits and disjoint DBMs to avoid any redundance. The substraction is one of the few operations that result in a non-convex zone, and thus, requires splitting. It is of prime importance to reduce the number of generated DBMs in an exploration loop, e.g., for reachability, because the result is propagated and serves to compute further successors later.

1 Introduction

DBMs (difference bound matrices) [6, 4] are efficient data structure to represent clock constraints in timed automata [1]. However, some operations require splitting, e.g. substraction or some extrapolation algorithms [2] because DBMs can not represent non-convex zones. The resulting DBMs of a splitting are parts of successor states that will be used to compute further successors. Reducing splitting means to reduce the state explosion.

The new DBM library of the model-checker UPPAAL[1] supports substractions and federations of DBMs to represent non-convex zones. There are other representations of zones that can deal with non-convex zones, such as CDDs [3] or CRDs [7]. In this paper we are concerned about how to solve the substraction for the DBMs when the DBMs are already used in a model-checker. Depending on the operation, a given representation may be more or less efficient, we do not address this issue.

2 DBM Substraction

Given two DBMs A and S, we want to substract S from A. The resulting zone Z can be defined as the zone satisfying the constraints of A and $\neg S$. Intuitively, a point $p \in \neg S$ iff $\neg(p \in S)$. Computing the result $Z = A \wedge \neg S$ is done by constraining A with the negated constraints of S. Figure 1 illustrates the basic substraction algorithm in two dimensions, i.e., with two clocks. In the worst case, for a DBM of dimension n, there are n^2 splits where each split costs a copy. The algorithm complexity is $O(n^4)$.

[1] http://www.uppaal.com

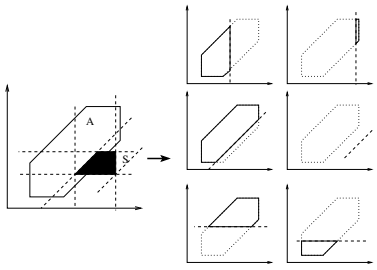


Fig. 1. Basic substraction algorithm.

3 Reducing Splitting

The first observation from the basic algorithm is that some splits may be avoided by taking into account only the edges that belong to the minimal graph [5] of the DBM. A DBM can be seen as a directed graph between clocks with the constraints on the edges. Figure 2 shows the reduced substraction by using only these constraints. The complexity for computing the minimal graph is $O(n^3)$ but it is worth doing since it is more important to reduce the result.

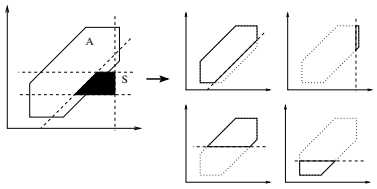


Fig. 2. Substraction using the constraints part of the minimal graph.

4 Substraction with Minimal Split

To further reduce the number of splits, let us consider the following four cases that arise on the constraints belonging to the minimal graph of the DBM to substract:

1. The (negated) constraint reduces A to an empty zone: we ignore it.
2. The (negated) constraint has no effect on A: because A is convex, this means that the DBM S to substract is outside of A so we stop and the result is A.
3. The (negated) constraint is on a facet of S that does not intersect A: we ignore it.
4. Otherwise we compute a split.

The third case is the contribution of our algorithm: the decision procedure is linear in the number of clocks and can rule out constraints that do not affect the result. The main argument for ruling out these constraints comes from the convexity of A and S: if a facet of S does not intersect A, then it has no effect on the substraction. We argue that this algorithm is sound and complete. As a remark, the first case is redundant with the third case but it is a simple test used to rule out simple cases. The complexity of the additional test is $O(n)$ per constraint, which is, $O(n^3)$ in total: we do not make the substraction worse.

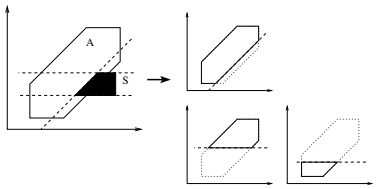


Fig. 3. Substraction with minimal splitting.

5 Minimal Substraction

Having the minimal number of DBMs as the result from a substraction may not be enough: if the successor states are going to be arguments for further substractions then there should be no overlapping between them otherwise future substractions will be redundant. The bad thing could be that future generated DBMs may not be simply comparable with respect to inclusion checking, which means, that even if some of them are redundant, they will be kept. We propose a simple procedure to reduce the size of the resulting DBMs to make them disjoint. The ordering of the splitting procedure affects the result but it is still guaranteed to be minimal and disjoint. This procedure costs a copy per split. The complexity is $O(n^4)$ in total. The substraction is still in $O(n^4)$.

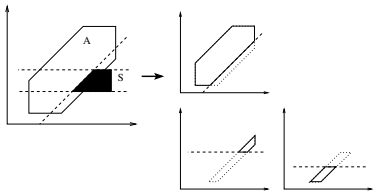


Fig. 4. Substraction with minimal splitting and disjoint result.

6 Conclusion

The algorithm we propose has been implemented and tested with other functions of the DBM library that use substraction as a sub-function. The library has an extensive set of tests that can be used as benchmarks. It turns out that it is more important to have a reduced result rather than a cheap and redundant one. The reason comes from the propagation of the result in the exploration loop of the model-checker, e.g., for reachability, or in the tests we made on the propagation of partial results: the resulting DBMs are used later to compute further successors and substractions.

References

1. Rajeev Alur and David L. Dill. Automata for modeling real-time systems. In *Proc. of Int. Colloquium on Algorithms, Languages, and Programming*, volume 443 of *LNCS*, pages 322–335, 1990.
2. Johan Bengtsson. *Clocks, DBMs and States in Timed Systems*. PhD thesis, Uppsala University, 2002.
3. Kim G. Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Clock difference diagrams. *Nordic Journal of Computing*, 6(3):271–298, 1999.
4. Kim G. Larsen, Paul Pettersson, and Wang Yi. Model-checking for real-time systems. In *Proc. of Fundamentals of Computation Theory*, number 965 in Lecture Notes in Computer Science, pages 62–88, August 1995.
5. Fredrik Larsson, Kim G. Larsen, Paul Pettersson, and Wang Yi. Efficient verification of real-time systems: Compact data structures and state-space reduction. In *Proc. of the 18*th *IEEE Real-Time Systems Symposium*, pages 14–24. IEEE Computer Society Press, December 1997.
6. Tomas Gerhard Rokicki. *Representing and Modeling Digital Circuits*. PhD thesis, Stanford University, 1993.
7. Farn Wang. RED: Model-checker for timed automata with clock-restriction diagram. In Paul Pettersson and Sergio Yovine, editors, *Workshop on Real-Time Tools, Aalborg University Denmark*, number 2001-014 in Technical Report. Uppsala University, 2001.

The threads are comming: New programming challenges for the masses

Erik Hagersten

Uppsala University, Sweden

Abstract

The long awaited introduction of multi-threaded execution inside mainstream microprocessors can become the next challenge for software technology. This talk will cover the technical reasons for running multiple threads per CPU chip, identify some technology possibilities and challenges that come as a direct result of this paradigm shift and finally make some predictions for the future.

Color-blind Behavioral Specifications for Transformations of Reactive Synchronous Programs

Andrzej Wąsowski*
Department of Innovation, IT University of Copenhagen, Denmark
wasowski@itu.dk

Ulrik Larsen Kim G. Larsen
Center for Embedded Software Systems (CISS), Aalborg University, Denmark
{ulrikl,kgl}@cs.aau.dk

Abstract

We describe a language-based approach to derivation of software product lines. A single general model, described as an I/O-alternating transition system, is used as a description of the available functionality. Hierarchically organized behavioral specifications define the actual family members by restricting input and output abilities of the general model. I/O alternating transition systems are used to model semantics of both systems and environments. Our environments are novel in that they not only restrict possible input traces, but also exhibit inabilities in distinguishing output traces. Some outputs are indistinguishable for a given environment in the same way as a color-blind person cannot distinguish some colors. Color-blindness can be used to model surprisingly many aspects of realistic environments (for example causality between firing and timing-out of a stop-watch, boolean memory flags, or use of a single actuator in place of two). The environments which are formalized as color-blind I/O-alternating transition systems, can also describe dynamic properties such as; an output that is ignored only after a certain set of other outputs.

An I/O alternating transition system can be transformed according to a behavioral specification of its environment, and individually optimized, during code generation, for the particular environment and purpose. The specialized members of the product family are obtained by requiring that the general model and the specialized models should be indistinguishable in the given environment. The use of environments enables new compiler optimizations, vastly exceeding usual reductions. If the general model defines much functionality that is not needed in all versions of the product, this functionality can be removed, even if it is defined in the general model interweaved with needed functionality.

We present the semantics of a specification language for environments of reactive synchronous systems, together with a powerful notion of context-dependent refinement based on color-blindness. This refinement relation is more liberal than usual in allowing some mutations to program outputs, instead of bare reductions. This refinement relation also preserves deadlock properties in a flexible way. We demonstrate how partial specifications of behaviors can be composed and used to define families of products.

The framework is demonstrated in two ways, first by discussing adaptations to realistic engineering design languages, second by presenting an example of a product line. Two classes of realistic engineering languages are handled: Languages with set based output and languages with sequence based output. The example is given in the modeling language of State/Event machines, which fits in the class of set based output.

In [11] a static framework for specifying environments for reactive models is presented, which relies solely on state independent properties. The present paper provides a theoretical foundation for a product line management setup similar to the one of [11], but based on behavioral properties.

Relativized simulation has been originally introduced by Larsen [6, 5, 4]. Our framework is modeled after this work, rephrased in the setting of I/O alternating transition systems and extended with the notion of color-blindness. In Larsen's formulation, based on simple labeled transition systems [8], it was impossible to express environment's inability to distinguish outputs.

The study of behaviors of systems embedded into execution contexts is relatively mature [5, 1, 7, 9, 3]. Our work stems out from this series, by its extended support for observability specifications via color-blindness. This support is needed, if the tools based on this frame-

*Work partly performed during author's visit at CISS.

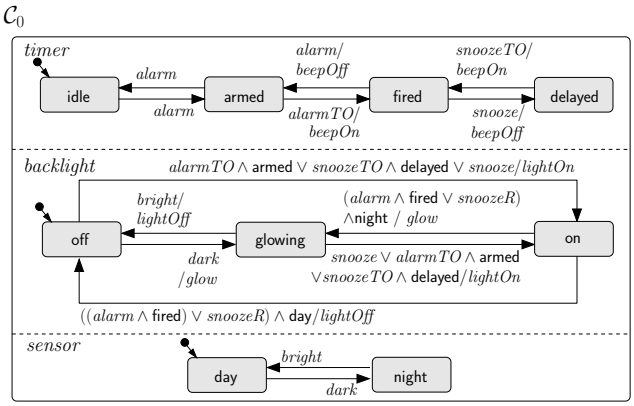


Figure 1: State/event model of an alarm clock.

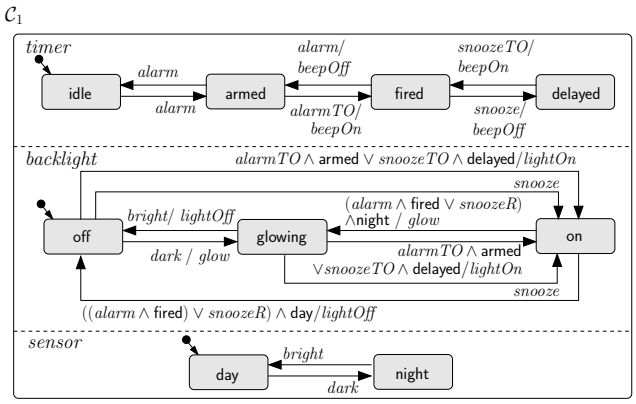


Figure 2: A specialized model, $\mathcal{C}_1$ of the alarm clock.

work, are to be useful for development of product lines of embedded systems.

The framework is designed for use in an upcoming tool [10] for compact code generation and product line derivation for discrete control embedded systems. Our specifications shall be used as preconditions for advanced, context-aware model optimizers/specializers. This prototype tool is supposed to be compatible with an industrial development environment for embedded systems [2], which will allow realistic case studies.

1 Example

The following example is included with the abstract to give a better understanding of the ideas involved in using color-blind environments for software product lines. This example of course lacks all the formal definitions of the full article.

Figure 1 depicts a state/event model of an alarm clock. This model consists of three state/event machines. The essential features of the alarm clock are handled by the *timer* machine. If the *timer* is in the armed state and the hardware sends an alarm time out event, *alarmTO*, then the beeper is turned on. If the user wants to postpone the alarm he has to press the snooze button (event *snooze*), which allows him to continue sleeping until the snooze timer times out (event *snoozeTO*). The *backlight* machine controls whether the backlight in the alarm clock should be: off, glowing or on. Only a faint light is displayed in the glowing state, such that the clock display can be read in the dark. The full light is on (state on) while the alarm is beeping or the snooze button is being pressed. The *snoozeR* event denotes the releasing of the snooze button. The *sensor* machine models a memory cell storing information about the level of light in the surroundings of the alarm clock. Proper events (*dark*, *bright*) are generated by the sensor driver whenever the ambient light passes above or below some threshold.

We would like to support automatic derivation of variants for discrete control systems like the alarm clock. One such variant $\mathcal{C}_1$, which does not activate the backlight in reaction to the *snooze* button, is depicted on Fig. 2. Note the simplification of guards and two new transitions in the *backlight* state machine. What is the relation between the two models? Both models are indistinguishable for some execution environment, namely the one, which becomes blind for the *lightOn* action immediately after producing the *snooze* event.

In order to structure our specifications, we shall firstly state general requirements, which should hold for all the environments used to execute the alarm clock. These general requirements usually reflect the physical nature of actuators and sensors. In our case we state that *dark*/*bright* and *snooze*/*snoozeR* are always generated in an alternating fashion.

$$\mathcal{E}_0 = \mathit{Interleave\ snooze\ snoozeR} \wedge \mathit{Interleave\ dark\ bright}$$

The environment for $\mathcal{C}_1$which can formally be defined as $\mathcal{E}_1 = \mathcal{E}_0 \wedge \mathcal{E}'$, where $\mathcal{E}'$ is depicted in Fig. 3.

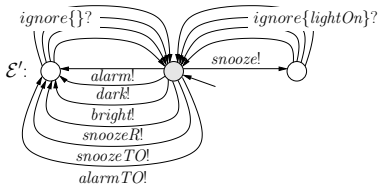


Figure 3: Environment $\mathcal{E}'$ ignoring the *lightOn* output produced in reaction to the *snooze* button.

Consider a new alarm clock variant $\mathcal{C}_2$, which is devoid of the actual snooze function (Fig. 4). The user of this clock can still press the snooze button, but the only effect it has is turning the backlight off for a short while.

We can say that this user (environment) becomes blind to *beepOn* and *beepOff* actions initiated by the *snooze* and *snoozeTO* events. Formally $\mathcal{E}_2 = \mathcal{E}_0 \wedge \mathcal{E}''$, where $\mathcal{E}''$ is defined in Fig. 5.

In this example we have shown two different environments for the general alarm clock model, and their respectively specialized alarm clocks. Both being dynamic properties, the first adding new transitions to the alarm clock, while simplifying some guards, the second only removing transitions and states from the original model.

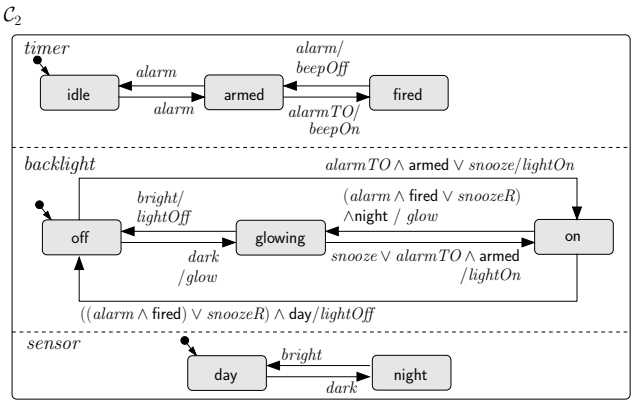


Figure 4: An alarm clock without the snooze function obtained by removing outputs/guards that are never observed/satisfied in $\mathcal{E}_2$, and then unifying states made indistinguishable (fired and snooze).

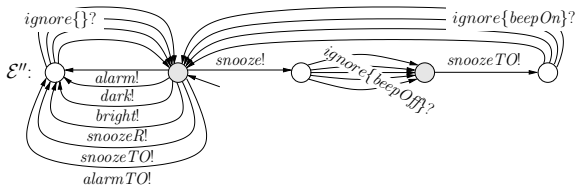


Figure 5: Environment $\mathcal{E}''$ ignoring the snooze function of the clock. Generation transitions to the blind observer **b** are not shown.

References

[1] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE)*, pages 109–120, Vienna, Austria, September 2001. ACM Press.

[2] IAR Inc. IAR visualSTATE®. `http://www.iar.com/Products/VS/`.

[3] A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. In *POPL 2001*. ACM Press.

[4] K.G. Larsen and R. Milner. A compositional protocol verification using relativized bisimulation. *Information and Computation*, 99(1):80–108, 1992.

[5] Kim G. Larsen. *Context Dependent Bisumulation Between Processes*. PhD thesis, Edinburgh University, 1986.

[6] Kim G. Larsen. A context dependent equivalence between processes. *Theoretical Computer Science*, 49:184–215, 1987.

[7] Nancy Lynch. I/O automata: A model for discrete event systems. In *Annual Conference on Information Sciences and Systems*, pages 29–38, Princeton University, Princeton, N.J., 1988.

[8] Robin Milner. *Communication and Concurrency*. Prentice Hall International Series in Computer Science. Prentice Hall, 1989.

[9] Sriram K. Rajamani and Jakob Rehof. Conformance checking for models of asynchronous message passing software. In Ed Brinksma and Kim Guldstrand Larsen, editors, *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *LNCS*, pages 166–179, Copenhagen, Denmark, 2002. Springer-Verlag.

[10] Scope: a statechart compiler. `http://www.mini.pw.edu.pl/~wasowski/scope`.

[11] Andrzej Wąsowski. Automatic generation of program families by model restrictions. In *Software Product Line Conference (SPLC)*, LNCS, Boston, USA, August/September 2004. Springer-Verlag.

A formal approach to model-driven engineering

Dubravka Ilic and Elena Troubitsyna

Turku Centre for Computer Science (TUCS),
Department of Computer Science, Åbo Akademi University,
FIN-20520 Turku, Finland.

{Dubravka.Ilic, Elena Troubitsyna}@abo.fi

Abstract

Model **D**riven **A**rchitecture (MDA) [5] - a recent initiative of OMG [6] - is gaining increasing popularity in industry. The main idea behind MDA is to develop software-intensive systems by transforming their models expressed in Unified Modeling Language (UML) [8]. Though such an approach facilitates developing systems in a structured way and potentially improves their design, it is yet insufficient for ensuring correctness of the constructed system. In contrast, formal methods have proved to be invaluable for ensuring system correctness but still are rather reluctantly accepted by industry practitioners. In this paper we propose an approach to formalizing MDA-style model transformations in the B Method.

The B Method [1] (further referred to as B) is an approach for the industrial development of correct software. The method has been successfully used in the development of several complex real-life applications [4]. It enables specification, verification and development of a system in a rigorous way. The tool support available for B provides us with the assistance for the entire development process. For instance, Atelier B [3], one of the tools supporting the B Method, has facilities for automatic verification and code generation as well as documentation, project management and prototyping. The high degree of automation in verifying correctness improves scalability of B, speeds up development and, also, requires less mathematical training from the users.

The development methodology adopted by B is based on stepwise refinement [2]. While developing a system by refinement, we start from an abstract formal specification and transform it into an implementable program by a number of correctness preserving steps, called refinements. The top-down design approach advocated by stepwise refinement coincides with the idea of MDA development by model transformations. By integrating these two development paradigms we enhance dependability of the developed systems without losing the visual appeal of diagrammatic UML notations.

In this paper we propose an approach aiming at developing specification and development patterns generic to ad-hoc mobile networks. At first we create templates for modeling ad-hoc networks by expressing the models in UML. Then we translate and verify them in B. We propose a generic development process based on transformations of corresponding models. We verify the correctness of our development by establishing refinement between the corresponding B models.

Our approach allows us not only to reason about correctness of system under construction but also helps in understanding and structuring complex system requirements. Besides, visual nature of the diagrams allows us to easily navigate through the design space and simplifies the process of incorporating changing or emerging requirements into the system. To validate the proposed approach we conducted a case study - model-driven development of routing protocol for ad hoc networks, **A**d hoc **O**n-Demand **D**istance **V**ector (AODV) [7] protocol.

References

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

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

[3] ClearSy, Aix-en-Provence, France. *Atelier B - User Manual*, Version 3.6, 2003.

[4] *MATISSE Handbook for Correct Systems Construction*. EU-project MATISSE: Methodologie and Technologies for Industrial Strength Systems Engineering, IST-199-11345, 2003.

Available at http://www.esil.univ-mrs.fr/~spc/matisse/Handbook.

[5] The OMG Model Driven Architecture Official Website: http://www.omg.org/mda/.

[6] The Official Object Management Group Website: http://www.omg.org.

[7] C. E. Perkins, E. M. Belding-Royer, and I. Chakeres. Ad Hoc on Demand Distance Vector (AODV) Routing. *IETF Internet draft*, 2003.

Available at http://moment.cs.ucsb.edu/pub/draft-perkins-manet-aodvbis-00.txt.

[8] J. Rumbaugh, I. Jacobson and G. Booch. *Unified Modeling Language Reference Manual*. Addison Wesley, 1999.

Cofree Coalgebras for Signature Morphisms*

Uwe Wolter
Department of Informatics, University of Bergen
wolter@ii.uib.no

August 27, 2004

It is an old observation that unsorted signatures used in Universal Algebra and Algebraic Specifications can be coded by functors $\mathcal{F} : \mathbf{Set} \to \mathbf{Set}$. $\mathcal{F}$-algebras are given in this setting by a carrier A and a map $\alpha : \mathcal{F}(A) \to A$. A corresponding unsorted signature morphism is modeled by a natural transformation $\tau : \mathcal{F} \Rightarrow \mathcal{G}$ and gives rise to a forgetful functor $U_\tau : \mathbf{Alg}(\mathcal{G}) \to \mathbf{Alg}(\mathcal{F})$, where $\mathbf{Alg}(\mathcal{F})$ denotes the category of all $\mathcal{F}$-algebras and all homomorphisms between them.

Free algebras w.r.t. U_τ and the corresponding free functor $T_\tau : \mathbf{Alg}(\mathcal{F}) \to \mathbf{Alg}(\mathcal{G})$, i.e., the functor left-adjoint to U_τ, play an important rôle in Algebraic Specifications, especially for compositionality, i.e., parametrization, structuring, and modularization [2, 3, 4, 6].

On the other hand, it has become evident, in the last few years, that coalgebras, the categorical dual of algebras, provide a unifying framework for formal specification of dynamical and behavioural aspects of systems [5, 7, 9]. $\mathcal{F}$-coalgebras are given by a carrier A and, dually, by a map $\alpha : A \to \mathcal{F}(A)$. An unsorted signature morphism $\tau : \mathcal{F} \Rightarrow \mathcal{G}$ gives rise to a functor $U_\tau^c : \mathbf{Alg}^c(\mathcal{F}) \to \mathbf{Alg}^c(\mathcal{G})$ between the corresponding categories of coalgebras. Those "co-forgetful" functors have been, e.g., successfully used in establishing a hierarchy of probabilistic systems types [1]. In contrast the corresponding cofree functors have not been investigated and applied up to now. But, taking into account the importance of free algebras in Algebraic Specifications, we should be curious about the rôle of cofree coalgebras in structuring and modularizing (dynamic) System Specifications.

In the paper we take a first step and investigate the construction of cofree coalgebras for unsorted signature morphisms. Due to the perfect categorical duality between the traditional concept of equations and the concept of coequations developed in [8] we can fully take profit of the methodological power of Category Theory [2] and follow a clean two-step-strategy: In a first, most demanding, step we analyse the traditional BIRKHOFF construction of free algebras: For a given $\mathcal{F}$-algebra (α, A) the free $\mathcal{G}$-algebra $T_\tau(\alpha, A) = (\iota_{\tau,\alpha}, T_\tau(A))$ arises as a quotient algebra of the $\mathcal{G}$-termalgebra $(\iota_{\mathcal{G},A}, T_\mathcal{G}(A))$ over the carrier A. The BIRKHOFF

*Research partially supported by the Norwegian NFR project MoSIS/IKT.

construction provides this quotient by constructing the $\mathcal{G}$-subalgebra generated by A of the product of an appropriated family of $\mathcal{G}$-algebras representing the "theory" of (α, A).

We develop a systematic categorical description of this construction thus we can in a second, easy step dualize this construction in a quite formal categorical way: We obtain a cofree functor $P_\tau : \mathbf{Alg}^c(\mathcal{G}) \to \mathbf{Alg}^c(\mathcal{F})$ right-adjoint to the co-forgetful functor $U_\tau^c : \mathbf{Alg}^c(\mathcal{F}) \to \mathbf{Alg}^c(\mathcal{G})$. Given a $\mathcal{G}$-coalgebra (A, α) the cofree $\mathcal{F}$-coalgebra $(P_\tau(A), \phi_{\tau,\alpha})$ arises as a subcoalgebra of the $\mathcal{F}$-processcoalgebra $(P_\tau(A), \phi_{\tau,\alpha})$ over the carrier A. The dual BIRKHOFF construction provides this subcoalgebra by constructing the quotient $\mathcal{F}$-coalgebra induced by A of the coproduct of an appropriated family of $\mathcal{F}$-coalgebras representing the "theory" of (A, α).

As a running example (that is beyond the traditional unsorted signature morphisms) we consider the identy functor $\mathcal{I} : \mathbf{Set} \to \mathbf{Set}$ and the diagonal functor $\mathcal{D} : \mathbf{Set} \to \mathbf{Set}$ with $\mathcal{D}(A) := A \times A$ for all sets A and $\mathcal{D}(f) := f \times f : A \times A \to B \times B$ for all maps $f : A \to B$ thus the assignments $A \mapsto \Delta_A : A \to A \times A$ with $\Delta_A(a) := (a, a)$ for all $a \in A$ define a natural transformation $\Delta : \mathcal{I} \Rightarrow \mathcal{D}$.

The analysis of the corresponding sample free and cofree constructions allows for a first intuitive interpretation of cofree constructions: For a given unsorted signature morphism $\tau : \mathcal{F} \Rightarrow \mathcal{G}$ the forgetful functor $U_\tau : \mathbf{Alg}(\mathcal{G}) \to \mathbf{Alg}(\mathcal{F})$ forgets some 'algebraic structure' thus the corresponding free functor $T_\tau : \mathbf{Alg}(\mathcal{F}) \to \mathbf{Alg}(\mathcal{G})$ constructs new 'algebraic structure' and 'identifies certain constructions' afterwards. Coalgebraically, or in terms of system behaviour, the co-forgetful functor $U_\tau^c : \mathbf{Alg}^c(\mathcal{F}) \to \mathbf{Alg}^c(\mathcal{G})$ interprets a simple, restricted behaviour as a special case of a more sophisticated, free behaviour. Therefore, the coresponding cofree functor $P_\tau : \mathbf{Alg}^c(\mathcal{G}) \to \mathbf{Alg}^c(\mathcal{F})$ restricts 'behaviour' and 'separates certain restrictions' afterwards, i.e., reduces a system by 'separating' the part of the system obeying the restricted pattern of behaviour.

A generalization of the results to morphisms between (co)equational specifications (or even conditional (co)equational specifications) will be notationally very tedious, but should not cause principal problems, because the (dual) Birkhoff construction is based on products (coproducts) and subalgebras (partitions) and the corresponding classes of (co)algebras are closed under these constructions [7, 8].

Probably the next steps should focus on the analysis of more comprehensive examples as, for instance, CSP [9], to reach a better intuition and understanding of cofree constructions and their potential rôle in System Specification.

References

[1] F. Bartels, A. Sokolova, and E. de Vink. A hierarchy of probabilistic sytem types. *TCS*, 2004. Submitted.

[2] H. Ehrig, M. Große–Rhode, and U. Wolter. Applications of category theory to the area of algebraic specification in computer science. *Applied Categorical Structures*, 6(1):1–35, 1998.

[3] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, Berlin, 1985.

[4] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*, volume 21 of *EATCS Monographs on Theoretical Computer Science*. Springer, Berlin, 1990.

[5] B. Jacobs and J. Rutten. A tutorial on (Co)Algebras and (Co)Induction. *Bulletin of the EATCS*, 62:222–259, June 1997.

[6] H. Reichel. *Initial Computability, Algebraic Specifications, and Partial Algebras*. Oxford University Press, Oxford, 1987.

[7] J.J.M.M. Rutten. Universal coalgebra: A theory of systems. *TCS*, 249:3–80, 2000. First appeared as Report CS-R9652, CWI, Amsterdam 1996.

[8] U. Wolter. On Corelations, Cokernels, and Coequations. In H. Reichel, editor, *Third Workshop on Coalgebraic Methods in Computer Science (CMCS'2000), Berlin, Germany, Proceedings*, volume 13 of ENTCS, pages 347–366. Elsevier Science, 2000.

[9] U. Wolter. CSP, Partial Automata, and Coalgebras. *TCS*, 280:3–34, 2002.

Transfinite Corecursion

(Extended abstract)

Härmel Nestra

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

Standard semantics consider computations doing at most ω steps (ω is the least infinite ordinal). By *transfinite semantics*, one means a semantics according to which computation may continue after an infinite number of steps from some limit state determined somehow by the infinite computation performed.

Transfinite semantics have been studied first for functional programming, see [5]. Recently, transfinite semantics of imperative languages are also been investigated. The main motivation has been reasoning about correctness of program transformation. In [4], transfinite semantics are developed with the aim of overcoming the so-called *semantic anomaly* of program slicing which arises when infinite loops are sliced away [2]. The idea was proposed already in [3].

In our work in progress, we look for mathematically rigorous specification of slicing using transfinite trace semantics, trying to go beyond the results of [4]. Transfinite trace semantics of a program is basically a set of transfinite sequences of states or configurations satisfying certain conditions. Transfinite sequence is a function whose domain is a segment of ordinals.

This paper makes a step towards this aim providing corecursion theorems for justifying some kinds of definitions of transfinite trace semantics.

Like recursion, *corecursion* is a special kind of way of defining functions. The soundness of both recursive and corecursive definitions in the case of their simplest forms is obvious; however, proving it is surprisingly hard. There are works trying to give proofs to corecursion theorems at very general level; the reader being interested in is recommended to study [1].

Our aim here is to give corecursion theorems for being able to check easily the soundness of transfinite semantics. As hinted before, the idea is to provide ways to compute limit states into which computation happens to fall after an infinite number of steps.

An ordinal is called a *limit ordinal* if it has no immediate predecessor; otherwise, it is called a *successor ordinal*. The least limit ordinals are 0, ω, $\omega \cdot 2$ etc. (in the literature, 0 is often excluded). It is clear that something new has to be carried in for limit ordinals only since, for any successor ordinal greater than ω, one can perform analogously to finite successor ordinals. Moreover, it is easy to see that, if one has defined limit values to any subsequence of length ω, one does not have to do anything more about $\omega \cdot 2$, $\omega \cdot 3$ etc. because the limit values of such sequences are already determined by the subsequence of the last ω elements. This is analogous to that in finite case, the computation step to be done does not depend on how many computation steps there are already

performed. So the next lengths of sequence for which one has to provide a limit function are ω^2, ω^3, etc. This gives rise to the following notion.

Definition 1. *We call an ordinal $\gamma > 0$ selfish if, for every $\alpha < \gamma$ and x,*

$$\alpha + x = \gamma \iff x = \gamma.$$

In other words, γ is selfish iff the well-order of the part remaining when cutting away any initial segment from the well-order represented by γ is isomorphic to γ itself. Selfish are those non-zero ordinals that cannot be represented as the sum of two ordinals both less than it.

For example, ω, ω^2, ω^3, etc. are selfish; $\omega \cdot 2 = \omega + \omega$ is not since represented as a sum of two ordinals both less than it. This notion captures precisely the kind of ordinals for which a corecurrent definition must provide a computing rule. Note that 1 is also selfish — the least among them.

The following result follows from the well-known laws of ordinal arithmetic.

Proposition 1. (i) *Every ordinal $o > 0$ is uniquely representable in the form $o = \alpha + \gamma$ where γ is selfish and α is the least for which this equality holds.*

(ii) *Every ordinal $o > 0$ is uniquely representable in the form $o = \gamma + \beta$ where γ is selfish and $\beta < o$.*

Let $\propto$ be an ordinal and let $\mathrm{TList}\, A$ denote the set of all transfinite lists over A of length less than $\propto$. Let $\mathbb{O}_o$ denote the set of all ordinals less than o. Let $\mathrm{take}\, o\, l$ and $\mathrm{drop}\, o\, l$ denote the transfinite list which is obtained from l by taking and dropping, respectively, the first o elements from it.

Theorem 1. *Let X, A be sets. Let*

$$\varphi : X \to 1 + A,$$
$$\psi : \left(\bigcup_{\substack{\gamma < \propto \\ \gamma \text{ selfish}}} (\mathbb{O}_\gamma \to A) \right) \to X.$$

Assume that, for any selfish ordinals λ, γ such that $\lambda < \gamma < \propto$, and transfinite list $l \in (\mathbb{O}_\gamma \to A)$, the equality

$$\psi(l) = \psi(\mathrm{drop}\, \lambda\, l)$$

holds. Then there is a unique function $f : X \to \mathrm{TList}\, A$ such that

1. *if $\varphi(x) = a \in A$ then $\mathrm{head}(f(x)) = a$,*
   if $f(x) \in 1$ then $f(x) = \varepsilon$ (ε denotes the empty sequence);
2. *if $|f(x)| \geqslant \gamma$ and $\gamma < \propto$ is selfish then*

$$\mathrm{drop}\, \gamma\, (f(x)) = f(\psi(\mathrm{take}\, \gamma\, (f(x)))).$$

Here, ψ plays the role of "limit operator" but it is also used for $\gamma = 1$, i.e. for ordinary steps. Note that, taking $\gamma = 1$, one obtains conditions similar to stream corecursion in the sense that the result list is defined in two parts: one determines its head and the other, corecursively, its tail.

This was the corecursion for deterministic semantics; the following theorem for non-deterministic semantics is analogous.

Theorem 2. *Let X, A be sets. Let*

$$\varphi : X \to 1 + A,$$
$$\psi : \left(\bigcup_{\substack{\gamma < \propto \\ \gamma \ selfish}} (\mathbb{O}_\gamma \to A) \right) \to \wp X.$$

Assume that, for any selfish ordinals λ, γ such that $\lambda < \gamma < \propto$, and transfinite list $l \in (\mathbb{O}_\gamma \to A)$, the equality

$$\psi(l) = \psi(\mathrm{drop}\, \lambda\, l)$$

holds. Then there is a largest function $f : X \to \wp(\mathrm{TList}\, A)$ such that

1. *if $\varphi(x) = a \in A$ then, for every $l \in f(x)$, $\mathrm{head}\, l = a$,*
   if $f(x) \in 1$ then $f(x) = \{\varepsilon\}$;
2. *for any selfish γ and transfinite list $l \in (\mathbb{O}_\gamma \to A)$ such that there exists an $m \in f(x)$ for which $\mathrm{take}\, \gamma\, m = l$, the equality*

$$\{m \in f(x) \mid \mathrm{take}\, \gamma\, m = l \bullet \mathrm{drop}\, \gamma\, m\} = \bigcup_{z \in \psi(l)} f(z)$$

holds.

These theorems have some more variants which capture more concretely the situation of transfinite semantics.

At first sight, our corecursion theorems are relatively straitening in the sense that any single component of any sequence constructed by the functions definable determines all the following components uniquely. In the case of usual corecursive definitions of streams, it is not so. It is easy to see however that this is not a fundamental restriction as, if a "more general" kind of transfinite sequence is desired, one can tie the "additional information" together with "elements" into pairs and define transfinite sequences of these pairs.

References

1. Barwise, J., Moss, L.: Vicious Circles. CSLI Lecture Notes No 60. CSLI Publications (1996)
2. Cartwight, R., Felleisen, M.: The Semantics of Program Dependence. ACM SIGPLAN Notices, Proceedings of the ACM SIGPLAN 1989 Conference on programming language design and implementation, **24**(7) (1989) 13–27
3. Cousot, P.: Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. Electronic Notes in Theoretical Computer Science, **6** (1997) 25 pp.
4. Giacobazzi, R., Mastroeni, I.: Non-Standard Semantics of Program Slicing. Higher-Order Symbolic Computation, **16** (2003) 297–339
5. Kennaway, R., Klop, J. W., Sleep, R., Vries, F.-J. de.: Transinite Reductions in Orthogonal Term Rewriting Systems. Information and Computation, **119**(1) (1995) 18–38

Scenario(MSC)-Based Specifications: a Survey

P.S. Thiagarajan

National University of Singapore

Abstract

Message Sequence Charts (MSCs) are an appealing visual notation for specifying patterns of interactions between processes and is used by a number of design communities.

A key role of MSCs is to capture system requirements in terms of "good" scenarios the implementation should exhibit and "bad" scenarios it must avoid. A good part of research on MSCs has been driven by this use. Thus in this line of work, the focus is on mechanisms for specifying collections of MSCs, ways of analyzing such collections and studying their relationship to implementation models.

On the other hand, a broad -and in some sense, a branching time- extension of MSCs called Live Sequence Charts (LSCs)has proposed by David Harel and Werner Damm to serve as an executable specification language. Further, David Harel and co-workers have developed a simulation engine called the Play-Engine to prototype system designs based on LSCs. From a conceptual standpoint, the model called Communicating Transaction Processes (CTPs) also falls within this framework.

We will survey research on MSCs covering both these themes.

Compositional Specification and Verification of UML Models*

Marcel Kyas† Frank S. de Boer‡

September 6, 2004

1 Motivation

Today, UML 2.0 [3] is an accepted modeling language for object-oriented software systems. It provides a range of diagrammatic notations for modeling and specifying different aspects of a system's structure and behavior. Quite a lot of methods and tools are developed to enable the formal analysis of these models. But most of these methods are concerned with a particular implementation of the model. These methods do not scale well or they enrich the model with notations outside of the UML, because UML does not provide any means to specify a systems behavior *compositionally* yet.

We present a compositional specification language based on OCL 2.0 [4] extended with histories (see [1]). The compatibility predicate, which asserts the compatibility of specifications, of [1] is extended to a a compositional proof rule, which allows to derive global properties of the model. This results in a trace logic for object oriented programs, where we investigate the composition of *class invariants*.

Among the main challenges for this specification language is ensuring the distinction between local specifications, which describe the behavior of the objects, and global specifications, which define the possible object structures. Global specifications are similar to UML's architecture diagrams. Another challenge concerns unbounded object creation.

As a case study we specify the Sieve of Eratosthenes in our trace logic and derive the correctness using the PVS theorem prover [5]. The translation from OCL to PVS is automated [2].

*Part of this work has been financially supported by IST project Omega (IST-2001-33522) and NWO/DFG project Mobi-J (RO 1122/9-1, RO 1122/9-2)

†Institute for Computer Science and Applied Mathematics, Christian-Albrechts-Universität zu Kiel, Germany, e-mail: mky@informatik.uni-kiel.de

‡CWI Amsterdam, The Netherlands, e-mail: frb@cwi.nl

2 Trace Logic

In this work we investigate a *trace logic* for object-oriented programs. The programming language chosen for this is a subset of UML 2.0 consisting of classes with operations and state machines which communicate by asynchronous and synchronous signals.

A program in our setting is given by a set of classes C with typical element c, where each class specifies attributes describing its data, and a state machine describing its behavior. We assume that the attributes specified in a class are private to the object, i.e., no object may read or write to another object's attributes. This ensures that communication between objects is only by sending signals. Finally, each program specifies a root class. For sake of simplicity, we assume that all state machines are flat.

To achieve compositionality, we equip each object with a *history variable* χ, which can be specified using a class invariant I_c. This invariant has to be local, i.e., it must not refer to the state or the history of any other object.

Given a specification we derive the verification condition

$$\bigwedge_{c \in C} \forall z_c : I_c[z_c/self] \to \phi$$

where the substitution $[z_c/self]$ transforms the local assertion I_c to a global assertion on the object z_c and ϕ is a property of the *global* trace of events we want to prove.

We are concerned with safety properties and assume that all interaction between objects is either by asynchronous or synchronous messages. There-

fore it is sufficient to use finite sequences of *events* as traces:

- $\text{send}(s, r, \text{sig}, \vec{v})$ states that s enqueues a signal sig in r's input queue.
- $\text{recv}(s, r, \text{sig}, \vec{v})$ denotes that r dequeues a signal sig from s from its input queue.
- $\text{return}(s, r, \text{sig}, \vec{v}, v')$ states that r returns from a synchronous signal.

If a communication is asynchronous, only send and recv is observed. If it is synchronous, only send and return is observed, because sending and receiving a message coincides, and the return is observed simultaneously by the sender and receiver. This means, that synchronous communication uses a rendez-vouz mechanism.

We specify the behavior of all instances of a class by an invariant I_c on its communication history. This logic is (essentially) defined by the language[1]

$$\varphi ::= t \sqsubseteq t' \mid t \preceq t' \mid t = t' \mid \neg\varphi \mid \varphi \wedge \varphi' \mid \exists v.\varphi$$
$$t ::= t \downarrow \lambda v.\varphi(v) \mid r.\chi \mid t_i$$

The predicate $t \sqsubseteq t'$ states that t is a subsequence of t', $t \preceq t'$ states that t is a prefix of t', $r.\chi$ denotes the local history of r, where r is some reference name, most of the time *self*, the term $t \downarrow \lambda v.\varphi(v)$ designates the *projection* onto φ, i.e., largest subsequence of t for which φ holds on each position, and t_i refers to the ith event in t. A *local* class invariant is a predicate on the object's trace such that *self* is either sender or receiver in each event.

We do not need to observe the creation of a new object, because the first communication with an object in the global trace establishes its creation.[2]

Finally, traces have to satisfy the following axioms:

- An object cannot be created by two objects. There can only be one object which sends a message with a new object identity to another object *spontaneously*. After this, each object needs to have received the message from another one.
- There are no cycles in the creation order.
- Reading from an event queue is done in a first-in first-out order.
- A local history is obtained by projection on the global history.

[1] We omitted the sub-language for integer arithmetic.

[2] This only holds if we can create an unlimited number of objects.

3 Tool Support

We have implemented a prototype tool which translates the extended OCL specifications (a syntax for the trace logic) into the PVS language. We have specified an object-oriented version of the Sieve of Eratosthenes using our trace logic and proved its correctness in PVS. The specification is quite concise and captures the essence of the systems behavior. To our mind this is an important property of a specification language.

In the future we want to apply our method to larger examples. We also need to investigate the features needed to allow compositional specifications for models using a richer subset of UML.

References

[1] M. Kyas and F. S. de Boer. On message specification in OCL. In F. S. de Boer, M. Bonsangue, and B. Josko, editors, *Compositional Verification in UML*, ENTCS. Elsevier, 2003.

[2] M. Kyas, H. Fecher, F. S. de Boer, M. van der Zwaag, J. Hooman, T. Arons, and H. Kugler. Formalizing UML models and OCL constraints in PVS. In *Workshop on Semantic Foundations of Engineering Design Languages*, ENTCS. Elsevier, 2004.

[3] Object Management Group. *UML 2.0 Superstructure Specification*, 2003. `http://www.omg.org/cgi-bin/doc?ptc/03-08-02`.

[4] Object Management Group. *UML 2.0 OCL Specification*, 2004. `http://www.omg.org/cgi-bin/doc?ptc/03-10-14`.

[5] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *Proc. CADE*, number 607 in LNAI, pages 748–752, Saratoga, NY, June 1992. Springer.

Refining UML interactions

Ragnhild Kobro Runde
Department of Informatics, University of Oslo, Norway

Abstract

STAIRS is an approach to the compositional development of interactions, supporting the specification of mandatory as well as potential behaviour. The classical notion of refinement is supported both by using a formal denotational trace semantics and by more practical transformation rules together with pragmatic constraints.

Introduction

STAIRS [HS03] is an approach to the compositional development of UML interactions, such as sequence diagrams and interaction overview diagrams. The process of developing the interactions is viewed as a process of learning through describing. From a fuzzy, rough sketch, the aim is to reach a precise and detailed description applicable for formal handling.

An essential feature in STAIRS is the ability to distinguish between mandatory and potential behaviour. By potential behaviour, we mean that a specification may give several alternative behaviours serving the same overall purpose, and that fulfilling only some of them is acceptable for an implementation to be correct. On the other hand, mandatory behaviours are alternatives that must all be present in a valid implementation.

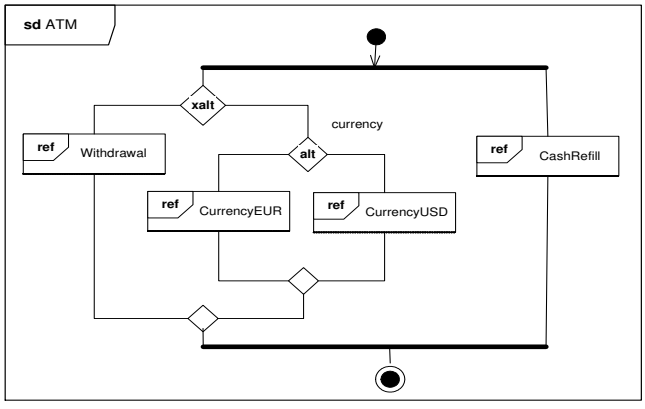


Figure 1: Mandatory (xalt) and potential (alt) alternatives

As an example, consider the interaction overview diagram in figure 1, specifying the behaviour of an ATM (Automatic Teller Machine). An ATM offers the customer withdrawal of money or the purchase of a number of foreign currencies (in addition to having cash refill). The alt-operator is used to specify that for an implementation of the ATM it is optional to offer either euros or US dollars, or both. We also want to specify that any ATM *must* offer both withdrawal of native money and at least one foreign currency. These mandatory alternatives are expressed using the novel operator xalt, introduced in [HS03].

Semantics of interactions

To explain the meaning of interactions, STAIRS uses denotational trace semantics. A trace is a sequence of input/output events, representing one execution of the specified system. In general, an interaction is viewed as specifying a set of positive and/or negative traces. Traces not specified as either positive or negative are called inconclusive.

A central notion in STAIRS is that of an interaction obligation, used to capture mandatory behaviour. An interaction obligation is a pair (p, n) of sets of traces where the first set is interpreted as the set of positive traces and the second set is the set of negative traces.

An interaction is given its semantics as a set of interaction obligations. Any valid implementation of the interaction must contain at least one positive trace from each of the interaction obligations. Different interaction obligations are specified syntactically by using the novel operator xalt, where each operand gives rise to one obligation. The positive traces within one obligation are alternatives that *may* be present in an implementation, thus supporting specification of potential behaviour. (At the specification level this is, among other things, achieved by the alt-operator.)

Formally, the semantics of interactions is defined by a function that for any interaction d yields a set $[\![d]\!]$ of interaction obligations. For instance, we have

$$
\begin{array}{lcl}
[\![d_1 \,\mathsf{alt}\, d_2]\!] & \stackrel{\mathsf{def}}{=} & \{(p_1 \cup p_2, n_1 \cup n_2) \mid (p_1, n_1) \in [\![d_1]\!] \wedge (p_2, n_2) \in [\![d_2]\!]\} \\
[\![d_1 \,\mathsf{xalt}\, d_2]\!] & \stackrel{\mathsf{def}}{=} & [\![d_1]\!] \cup [\![d_2]\!]
\end{array}
$$

For more details, see [HHRS04].

Refinement of interactions

An important part of STAIRS is to explain the classical notion of refinement in the setting of interactions. This is done by distinguishing between three main kinds of system development steps: supplementing, narrowing, and detailing. Supplementing categorizes inconclusive behaviour as either positive or negative, while narrowing reduces the set of positive behaviours. Detailing means to introduce more detailed descriptions, while maintaining the essential behaviour.

Formally, we have that an interaction obligation (p_2, n_2) is a refinement of an interaction obligation (p_1, n_1), written $(p_1, n_1) \rightsquigarrow_r (p_2, n_2)$, iff $n_1 \subseteq n_2 \wedge p_1 \subseteq p_2 \cup n_2$. An interaction d' is a refinement of an interaction d iff $\forall o \in [\![d]\!] : \exists o' \in [\![d']\!] : o \rightsquigarrow_r o'$.

Supplementing and narrowing are special cases of the general notion of refinement. An interaction obligation (p_2, n_2) supplements an interaction obligation (p_1, n_1) iff $(n_1 \subset n_2 \wedge p_1 \subseteq p_2) \vee (n_1 \subseteq n_2 \wedge p_1 \subset p_2)$. An interaction obligation (p_2, n_2) narrows an interaction obligation (p_1, n_1) iff $p_2 \subset p_1 \wedge n_2 = n_1 \cup (p_1 \backslash p_2)$. Detailing corresponds to interface refinement as defined in e.g. FOCUS [BS01].

In [HHRS04] we prove that refinement as defined above is transitive and monotonic with respect to the most common interaction operators. Transitivity is important, as it ensures that successive refinement steps results in a valid refinement of the original specification. In the same way, monotonicity allows for the different parts of an interaction to be refined independently.

Refinement calculus

The semantic details as presented in [HHRS04] are quite involved, and it is unlikely that most users of interactions will be interested in working at that level. Thus, there is a need for a more practical, high-level approach to dealing with refinement of interactions. We are currently working on constructing a refinement calculus for STAIRS. In this first phase, the calculus consists of two kinds of rules:

Transformation rules describing syntactical modifications on interactions, modifications that will result in valid refinements.

Pragmatic constraints to assist in developing *useful* refinements, not only valid ones. (For instance, it is very easy to refine a consistent specification into an inconsistent one, thus making it not implementable.)

References

[BS01] M. Broy and K. Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement.* Springer, 2001.

[HHRS04] Ø. Haugen, K. E. Husa, R. K. Runde, and K. Stølen. Why timed sequence diagrams require three-event semantics. Technical Report 309, Department of Informatics, University of Oslo, September 2004.

[HS03] Ø. Haugen and K. Stølen. STAIRS – Steps to analyze interactions with refinement semantics. In *Sixth International Conference on UML*, number 2863 in Lecture Notes in Computer Science, pages 388–402. Springer, 2003.

An Analysis Tool for UML Models with SPT Annotations

John Håkansson and Leonid Mokrushin

Uppsala University
Department of Information Technology
P.O. Box 337, SE-751 05 Uppsala, Sweden
Email: {johnh,leom}@it.uu.se

Abstract. In this paper, we describe a plug-in for the Rhapsody tool, which demonstrates how UML models with SPT annotations can be analysed using the Times tool — a tool for modelling, schedulability analysis, and code generation for timed systems. The plug-in takes as input an UML model consisting of an assembly of components whose behaviours are specified by statecharts. Operations may be annotated with SPT timing parameters for their execution time, deadline, priority etc. The output is a network of timed automata extended with tasks that can be analysed using the Times tool. In particular, the Times tool will show whether the operations invoked from the UML model are guaranteed to meet their deadlines or not. We describe how this has been done in a case study where an SPT annotated UML model of an adaptive cruise controller is studied.

1 UML with SPT Annotations

We use a subset of UML with SPT annotations (the UML profile for Scheduling, Performance and Time [11]) as an input language for our tool. The models that can be analysed are hierarchical static structure diagrams containing objects representing components or assemblies of components. The behaviour of each component is described by an associated statechart, each executing in its own thread of control. Figure 1 shows an example which can be analysed with the tool presented in this paper. Component A is composed of subcomponents B and C whose corresponding statecharts are shown to the right of the structure diagram.

The operations of components represent tasks being scheduled for execution. We apply the SPT stereotype SAAction for operations to annotate them with parameters such as priority, computation time, deadline, etc. Parameters of the operation foo of C are shown in Figure 1. Operations are triggered by statecharts, and operations triggered by the same statechart are executed synchronously.

Components can have ports that can be connected with directed links allowing for interaction between them. There are two types of ports, namely data ports and trigger ports. Data ports are defined using DataPort interface which has two methods get and set. Links between data ports are one place buffers that

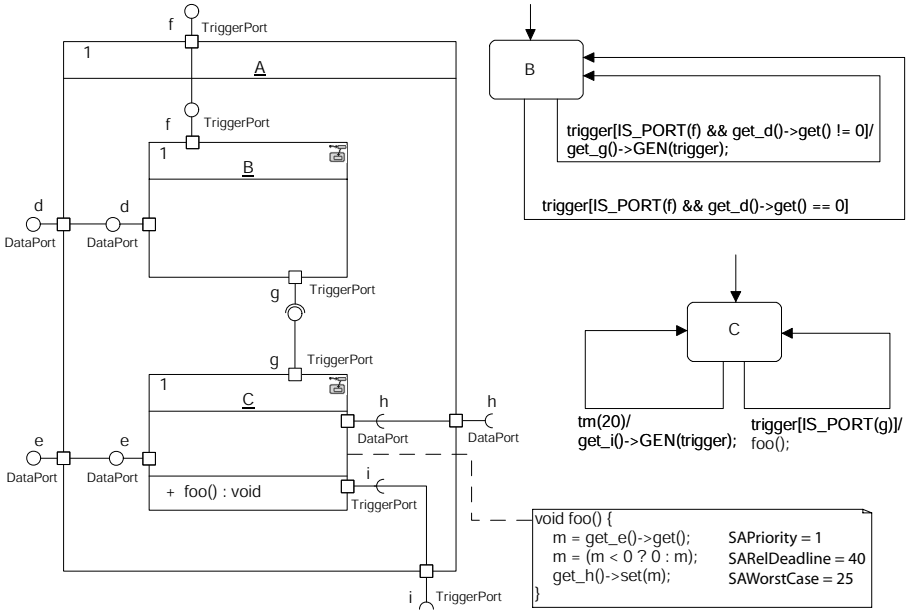


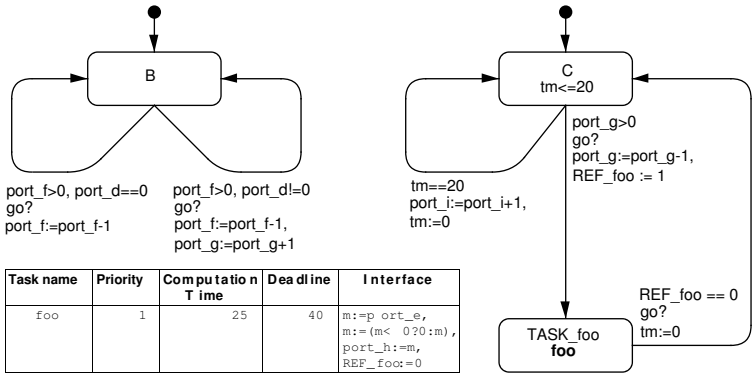
Fig. 1. An example UML model with SPT annotations

can have one sender and many receivers. A data item in a one place buffer is observable by receivers until overwritten by the sender. Trigger ports are defined using TriggerPort interface. Untyped events can be sent over the link using the GEN method. Events are stored in a buffer of a predefined size until consumed by the receiver.

Every component can be associated with a statechart. Transitions of statecharts are enabled by triggers and/or guards; the associated actions will be executed when the transition is fired. A trigger can be a timeout or the reception of an event from a trigger port. For example in the statechart of component C the left transition has timeout trigger `tm(20)`, and the right transition has event `trigger[IS_PORT(g)]`. The timeout expires 20 ms after state C is reached. Guards are logical conjuncts of integer expression comparisons. Actions can update variables, synchronously execute corresponding operations, generate events on trigger ports, and set the values on data ports.

2 Extracting Timed Models for Analysis

We have developed a tool to extract timed models from UML models with SPT annotations. The extracted model is a network of timed automata [1] extended with tasks [6] and integer variables. Figure 2 shows the timed model extracted from the UML model in Figure 1. The statecharts are converted into timed automata, while the links of the structural diagrams are translated into shared



Task name	Priority	Computation Time	Deadline	Interface
foo	1	25	40	m:=p ort_e, m:=(m< 0?0:m), port_h:=m, REF_foo:=0

Fig. 2. Timed model extracted from example in Figure 1

data for asynchronous communication between automata. Operations with the stereotype SAAction become tasks whose computation time, deadline and priority are given according to the SPT annotations SAWorstCase, SARelDeadline, and SAPriority, respectively.

The ports are represented as shared integer variables that are aliased according to how the links connect them. A variable connecting data ports represent the data item currently residing in the one place buffer. For trigger ports the variable is used to count the number of events queued at the port. The get operation on a data port corresponds to accessing the current value of the variable associated with that port, while the set operation corresponds to updating the variable with a new value. The GEN operation on a trigger port corresponds to increasing the event counter variable, the counter is then decreased when a transition is fired due to an event received at this port.

A transition in a statechart is translated into edges in an automaton, with intermediate locations when needed. An event trigger becomes a guard that is true for non-zero queue lengths of the triggered port. A timeout tm(T) in a statechart i is converted to a guard $tm_i = T$, an invariant $tm_i \leq T$ on the source location, and a reset operation on the clock tm_i for each transition reaching the source location. A call to an SAAction operation becomes an intermediate location where the corresponding task is released for asynchronous execution, the automaton will remain in this location until the task completes.

3 Tool Overview

The tool is a plug-in for analysis of Rhapsody [9] UML models with SPT annotations [11] using the Times [2] tool. The plug-in can be added to the Rhapsody Tool menu, and uses the Rhapsody COM API to access the UML model. The extracted model is generated as Java objects directly into a running instance of the Times tool.

TIMES is a design and analysis tool for real-time embedded systems, based on timed automata [1] extended with tasks [6]. Tasks are temporal abstractions of executable programs represented as their execution time, deadline, priority, etc. Releases of task instances are triggered by timed automata for asynchronous execution according to a given scheduling strategy. The TIMES tool is developed for automated schedulability analysis of the extended models.

4 Case Study

The tools described in this paper have been applied to analyse an adaptive cruise controller (ACC) model. An ACC is an extension of an ordinary cruise controller, with the purpose to control the velocity of a vehicle, while maintaining a minimal distance to any vehicle in front. A radar is used, with an object recognition component, to detect the speed and distance of any vehicle in front.

The ACC model is defined within SaveCCM [8], a component model developed as a part of the SAVE project. The purpose of the project is to develop a component technology for safety-critical vehicle systems. The concepts of data ports and trigger ports in our modelling language comes from SaveCCM. In SaveCCM there are three types of entities, separated using stereotypes: SaveCOMP is used to denote components, Assembly denotes component assemblies, and Switch denotes a special construct for switching triggering events. Statecharts model the triggering of components by first receiving from all input trigger ports, then perform its calculations, and finally generate events on all output trigger ports. Switches differ in that they can generate events on selected output trigger ports.

The ACC assembly of our model is shown in Figure 3. The calculations of each component is modelled by an operation execute, with SPT parameters according to Table 1. The ACC is assembled from three components and a sub-assembly. The component ModeLogics calculates the current operating mode of the ACC. ObjectRecognition applies an algorithm to radar input, in order to determine the velocity and distance of any vehicle in front. HMI_outputs determines the output to be sent to the driver panel. AccControllers is an assembly consisting of two PID controller components and a switch. The switch determines what controllers are active depending on the current operating mode of the ACC. In standard cruise controller mode only the SpeedController is active, but in ACC mode both the SpeedController and DistanceController are active and coupled in a cascaded control loop.

When the model is exported from Rhapsody into TIMES, the resulting TIMES model is schedulable. The response times reported by TIMES are presented in Table 1. The schedulability analysis was performed in less then one second and consumes 3.3 Mb of memory, using an Intel® Celeron® 1.7GHz computer.

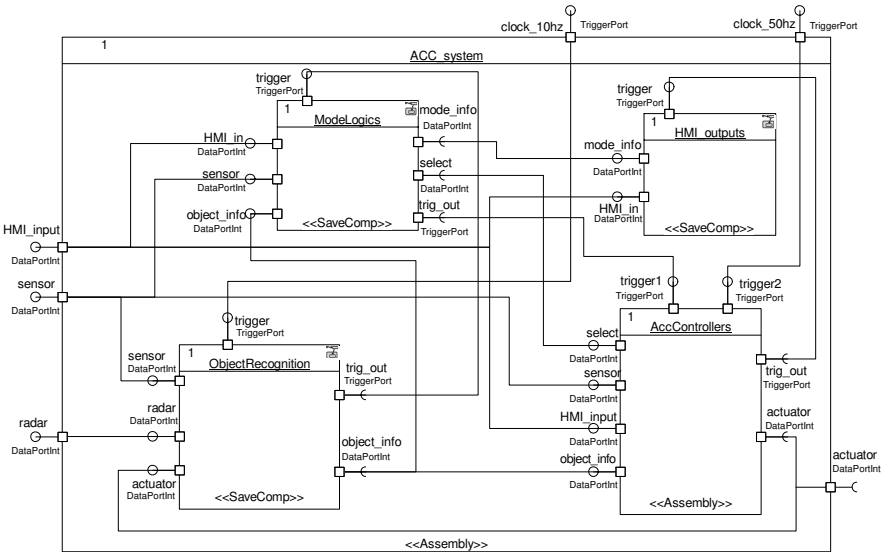


Fig. 3. The ACC subsystem assembly

Component Name	SAWorstCase	SARelDeadline	SAPriority	Worst Case Response Time
SpeedController	5	20	5	5
ObjectRecognition	30	100	4	40
ModeLogics	1	100	3	6
HMI_outputs	2	100	2	2
DistanceController	20	100	1	25

Table 1. The properties of the execute operation for the components.

5 Conclusions

In this ongoing work we demonstrate how UML with the SPT profile can be used to model schedulability problems in a way that is analyzable by the Times tool. Some work remains to be done. The case study model should be extended with more details, for a more reliable validation of the work. To handle a richer class of UML models and data structures, we plan to study and extend the tool with predicate abstraction [13, 4, 3].

Related work: Extracting timed models from UML has been done in [7, 5]. There is a tool [12] by OFFIS for analysis of untimed Rhapsody models. Tri-Pacific has a tool [10] for schedulability analysis of periodic task sets extracted from Rhapsody models. In our work we propose a more generic model

where tasks are triggered not only periodically but also by external events. The exact task release pattern is defined by means of UML statecharts.

References

1. Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
2. Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. TIMES: a tool for schedulability analysis and code generation of real-time systems. In *Proc. of 1*st *International Workshop on Formal Modeling and Analysis of Timed Systems*, Lecture Notes in Computer Science. Springer–Verlag, 2003.
3. Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.
4. E. M. Clarke, O. Grümberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. of the 12th Int. Conf. on Computer Aided Verification*, pages 154–169, 2000.
5. Alexandre David, Oliver Möller, and Wang Yi. Formal verification of uml statecharts with real-time extensions. In Ralf-Detler Kutsche and Herbert Weber, editors, *Proceedings of FASE 2002*, number 2306 in Lecture Notes in Computer Science, pages 218–232. Springer–Verlag, 2002.
6. Elena Fersman, Paul Pettersson, and Wang Yi. Timed automata with asynchronous processes: Schedulability and decidability. In J.-P. Katoen and P. Stevens, editors, *Proc. of the 8*th *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 2280 in Lecture Notes in Computer Science, pages 67–82. Springer–Verlag, 2002.
7. Susanne Graf, Ileana Ober, and Iulian Ober. Model checking of UML models via a mapping to communicating extended timed automata. In S. Graf and L. Mounier, editors, *Proc. of the 11*th *International SPIN Workshop*, number 2989 in Lecture Notes in Computer Science, pages 127 – 145. Springer–Verlag, 2004.
8. Hans Hansson, Mikael Åkerholm, Ivica Crnkovic, and Martin Törngren. SaveCCM - a component model for safety-critical real-time systems. In *Proc. of Euromicro Workshop on Component Models for Dependable Systems*. IEEE Computer Society Press, 2004.
9. David Harel and Eran Gery. Executable object modeling with statecharts. *Computer*, 30(7):31–42, 1997.
10. Paulo Martins. *Integrating Real-Time UML Models with Schedulability Analysis*. Tri-Pacific Software Inc., 2004.
11. Object Management Group Inc. *UML Profile for Schedulability, Performance, and Time Specification*, 2002.
12. OFFIS. *The Rhapsody UML Verification Environment – Installation-Guide and Tutorial*, 2004.
13. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grümberg, editor, *Proc. of the 9th Int. Conf. on Computer Aided Verification*, volume 1254, pages 72–83. Springer–Verlag, 1997.

The Controlled Linear Programming Problem

Henrik Björklund, Olle Nilsson, Ola Svensson, Sergei Vorobyov

Information Technology Department Uppsala University

1 Introduction

We introduce and investigate the new Controlled Linear Programming Problem (CLPP) [1]. In a system of linear constraints of the form $x_i \leq p_i(\bar{x}) + w_i$, where p_i are linear homogeneous polynomials with nonnegative coefficients, some variables are *controlled* and the controller wants to select exactly one constraint for each controlled variable in a way that makes $\max \sum x_i$ as large as possible (over all selections). Intuitively, the CLPP mixes the notion of linear programming with that of game theory.

We investigate several different versions of the problem by varying the restrictions on the polynomials in the constraints. For these instances we then prove optimality conditions which allow for designing subexponential iterative improvement algorithms and proving NP∩coNP membership. Also, we show that a slight generalization of the problem, with negative coefficients in the polynomials, leads to NP-hardness.

A motivation for studying this problem is that many well known and much studied games like parity, mean payoff, discounted payoff, and simple stochastic games are easily reducible to restricted versions of the CLPP. This gives a unifying view and easily explainable subexponential algorithms for such games.

2 Definitions

Let $\mathbf{x} = (x_1, \ldots, x_n)$, $\mathbf{y} = (y_1, \ldots, y_m)$ be two vectors of variables, called *controlled* and *uncontrolled*, respectively. Let $p_i^j(\mathbf{y})$ and $q_k^l(\mathbf{x})$ be homogeneous linear polynomials with *non-negative* real coefficients, and $w_i^j, w_k^{'l} \in \mathbb{R}$, for $i \in \{1, \ldots, n\}$, $j \in \{1, \ldots, n_i\}$, $k \in \{1, \ldots, m\}$, $l \in \{1, \ldots, m_k\}$. A linear constraint is called:

- *optional* if it has form $x_i \leq p_i^j(\mathbf{y}) + w_i^j$, and
- *compulsory* if it has form $y_k \leq q_k^l(\mathbf{x}) + w_k^{'l}$.

Definition 1 (Controlled LP-Problem). *Given the data as above, find a tuple, called a* pure strategy, *$(j_1, \ldots, j_n) \in \prod_{j=1}^{n} \{1, \ldots, n_j\}$ that renders maximal (over all possible such tuples) the value of the objective function $\sum_{i=1}^{n} x_i + \sum_{i=1}^{m} y_i$ subject to the system of constraints $S(j_1, \ldots, j_n)$ consisting of all compulsory and the optional constraints $x_i \leq p_i^{j_i}(\mathbf{y}) + w_i^{j_i}$ for $i = 1, \ldots, n$.* □

For a given pure strategy, denote an optimal solution of the resulting system $S(j_1, \ldots, j_n)$ by $(\mathbf{x}^*, \mathbf{y}^*)$.

Say that a *switch* for x_k from j_k-th to j_k'-th constraint is *attractive*, if

$$p_k^{j_k}(\mathbf{y}^*) + w_k^{j_k} < p_k^{j_k'}(\mathbf{y}^*) + w_k^{j_k'}. \tag{1}$$

Say that a sequence of switches from $(j_1, \ldots, j_n)$ to $(j'_1, \ldots, j'_n)$ is *profitable*, if each variable in the optimal solution of the new linear program has at least the same value, and one has a value strictly greater, as the original value. A strategy λ is admissible if $S(\lambda)$ is feasible. The following theorem underlies the monotonic improvement of several iterative switching algorithms in Section 4.

Theorem 1. *Every sequence of attractive switches is profitable and transforms an admissible strategy into an admissible one.* □

Say that an admissible strategy is *stable* if it does not have attractive switches. For many important cases of the CLPP, discussed below, we have the following polynomial-time optimality test.

Theorem 2. *Every stable strategy is optimal.* □

The above definitions and theorems have all been relative to *pure* strategies. We show, however, that the results are also valid for *mixed* strategies. A mixed strategy is defined for every controlled vertex x_i as a nonnegative vector $\bar{\lambda}_i = (\lambda_i^1, \ldots, \lambda_i^{n_i})$ with the l_1-norm equal to 1, i.e., $\sum_{j=1}^{n_i} \lambda_i^j = 1$. Combine all the constraints for each controlled variable x_i into a weighted sum $x_i \leq \sum_{j=1}^{n_i} \lambda_i^j (p_i^j(\mathbf{y}^*) + w_i^j)$, and denote the resulting system by $S(\lambda) = S(\bar{\lambda}_1, \ldots, \bar{\lambda}_n)$. Besides proving Theorem 1 for the *mixed* case, we also prove the following theorem that shows that every optimal strategy is discrete.

Theorem 3. *For any stable mixed strategy there is a pure/discrete no-worse stable strategy.* □

These important properties allows us to design a wide range of simplex-like and interior point algorithms for several problems discussed below.

3 Stability and Optimality

It follows that whenever a stable strategy is also optimal, the decision problem is in NP∩coNP. For NP, just guess a strategy of the controller and verify that the optimal solution of the resulting system is above the given threshold. For coNP, guess a strategy, verify that it is stable and hence optimal, and then confirm that the resulting system has an optimal value smaller than the bound.

We show that for several interesting restricted versions of the CLPP this property does indeed hold; see, e.g., [2]. Therefore, iterative improvement algorithms that start with an initial admissible strategy and repeatedly make attractive switches, will eventually find the global optimum. For the general case, however, we present several counterexamples, and do not yet know if the problem is in NP∩coNP. If negative coefficients are allowed in the polynomials we show that the problem is actually NP-hard.

In the following cases Theorem 2 does hold:

- Univariate Discounted Case, which is defined like the general case but with the further requirements that the right hand sides of the constraints are on the form: $w + \delta \cdot y$, for $0 < \delta < 1$ fixed, $w \in \mathbb{R}$ and y a variable.
- Multivariate Discounted case, where the right hand sides of the constraints have form: $\sum_i a_i \cdot v_i + w_i$, for v_i a variable, $w_i \in \mathbb{R}$ and a_i's positive constants that fulfill $\sum_i a_i < 1$.
- Univariate Non-Discounted Case, with right hand sides: $w + v$ for v a variable and $w \in \mathbb{R}$. Also one or more compulsory constraints on the form $t \leq 0$ can be introduced.

4 Algorithms

Switching Algorithms. Whenever stability implies optimality the following generic algorithm solves the CLPP: 1) start with any admissible strategy; 2) make attractive switches until the global optimum is found. Different switching schemes can be employed with this template algorithm and we look at *Single Random Switch*, *Single Greedy Switch*, *All Attractive Switches* and *Random Multiple Attractive Switches*.

Subexponential Algorithms. Given an instance S of CLPP, a *subproblem* F of S is obtained by fixing one optional constraint for some controlled variable and allowing other to vary. In combinatorial terms, F defines a *facet* in the space of pure strategies of the controller in S. This establishes the "constraints $\leftrightarrow$ facets" relation. Associating optional constraints with facets in the space of controller's pure strategies provides the intuitions necessary for applying different approaches from combinatorial linear programming to the CLPP. In particular, we adapted (see Algorithm 1) the randomization scheme due to Matoušek, Sharir, and Welzl (MSW) (more intuitive and easier to explain for than Kalai's randomization [1]). The function *opt* $S(\sigma)$ works as the target function being optimized. The SWITCH function takes a strategy σ and an optional constraint $x \leq e$ and returns the strategy obtained by changing the choice of σ for x to $x \leq e$. In combinatorial optimization terms, line 3 of the algorithm corresponds to selecting a facet in the space of positional strategies. Line 4 corresponds to recursive optimization on the rest of this space.

Algorithm 1: MSW-Style Algorithm for CLPP

CLPP-MSW(S, σ_0)

(1) **if** every controlled variable has only one optional constraint in S
(2) **return** σ_0
(3) choose a random optional constraint $x \leq e$ not selected by σ_0
(4) $\sigma^* \leftarrow$ CLPP-MSW$(S \setminus \{x \leq e\}, \sigma_0)$
(5) $\sigma_1 \leftarrow$ SWITCH$(\sigma^*, x \leq e)$
(6) **if** *opt* $S(\sigma_1) >$ *opt* $S(\sigma^*)$
(7) $S' \leftarrow S \setminus \{x \leq e' : e' \neq e\}$
(8) **return** CLPP-MSW(S', σ_1)
(9) **else return** σ^*

The number of iterations of this algorithm is $2^{O(\sqrt{n \log(m/\sqrt{n})} + \log m)}$, where n is the number of controlled variables and m is the number of optional constraints (similar to Kalai's). If m is polynomial in n, the bound is $2^{O(\sqrt{n \log n})}$ [1].

References

1. H. Björklund, O. Nilsson, O. Svensson, and S. Vorobyov. The controlled linear programming problem. Technical Report DIMACS-2004-41, DIMACS: Center for Discrete Mathematics and Theoretical Computer Science, Rutgers University, NJ, September 2004. `http://dimacs.rutgers.edu/TechnicalReports/`.
2. H. Björklund, S. Sandberg, and S. Vorobyov. A combinatorial strongly subexponential strategy improvement algorithm for mean payoff games. In J. Fiala, V. Koubek, and J. Kratochvil, editors, *29th International Symposium on Mathematical Foundations of Computer Science (MFCS'2004)*, volume 3153 of *Lecture Notes in Computer Science*, pages 673–685. Springer, August 2004. Preliminary full version: TR DIMACS-2004-05 `http://dimacs.rutgers.edu/TechnicalReports/`.

Programming Languages Capturing Complexity Classes (Extended Abstract)

Lars Kristiansen[1,2] and Paul J. Voda[3]

[1] Oslo University College, Faculty of Engineering
larskri@iu.hio.no WWW home page: http://www.iu.hio.no/~larskri
[2] Department of Mathematics, University of Oslo
[3] Institute of Informatics, Comenius University Bratislava
voda@fmph.uniba.sk WWW home page: http://www.fmph.uniba.sk/~voda

Introduction. We will generalize and refine some of the results published in [1] and [2]. In [1] we characterize the complexity classes LOGSPACE, P, PSPACE, LINSPACE, EXP and EXPSPACE by neat and natural fragments of the standard programming language C. In [2] we characterize the Ritchie hierarchy by fragments of an imperative programming language. (The classes in the Ritchie hierarchy add up to the set of Kalmár elementary functions.) In the present paper we introduce programming languages equipped with variables and terms of higher types. Fragments of these languages capture the complexity classes in two alternating space-time hierarchies.

The alternating space-time hierarchies. A *problem* is a subset of $\mathbb{N}$. For $i \in \mathbb{N}$, TIME 2_i^{POLY} (SPACE 2_i^{POLY}) is the set of problem decidable by a Turing machine working in time (space) $2_i^{p(|x|)}$ for some polynomial p; TIME 2_i^{LIN} (SPACE 2_i^{LIN}) is the set of problem decidable by a Turing machine working in time (space) $2_i^{k|x|}$ for some $k \in \mathbb{N}$. Let $\text{STPOL}^0 = \text{LOGSPACE}$, $\text{STPOL}^{2i+2} = \text{SPACE } 2_i^{\text{POLY}}$ and $\text{STPOL}^{2i+1} = \text{TIME } 2_i^{\text{POLY}}$. Let $\text{STLIN}^{2i} = \text{SPACE } 2_i^{\text{LIN}}$ and $\text{STLIN}^{2i+1} = \text{TIME } 2_{i+1}^{\text{LIN}}$. Then we have $\text{STPOL}^i \subseteq \text{STPOL}^{i+1}$ ($\text{STLIN}^i \subseteq \text{STLIN}^{i+1}$) for each $i \in \mathbb{N}$, and $\bigcup_{i\in\mathbb{N}} \text{STPOL}^i$ ($\bigcup_{i\in\mathbb{N}} \text{STLIN}^i$) is an alternating space-time hierarchy. (Note that we find familiar complexity like classes LOGSPACE, P, PSPACE, LINSPACE etc. near the bottoms of the two hierarchies.) It is well known, and quite obvious, that $\text{STPOL}^i \subset \text{STPOL}^{i+2}$ ($\text{STLIN}^i \subset \text{STLIN}^{i+2}$) holds for any $i \in \mathbb{N}$. Still, the problem $\text{STPOL}^n = \text{STPOL}^{n+1}$ ($\text{STLIN}^n = \text{STLIN}^{n+1}$) is open for any fixed $n \in \mathbb{N}$. In particular, the following problems are open: Does P equal LOGSPACE? Does P equal PSPACE? Does EXP equal LINSPACE? Does EXP equal EXPSPACE? Complexity theorists in general, and the authors in particular, suspect the answer to all these questions to be no, and it is a bit of a mystery that it should be so hard to prove that this indeed is the case. A further study of the two alternating space-time hierarchies might shed some light on these and similar pivotal questions of complexity theory (e.g. if we could prove for *some* $n \in \mathbb{N}$ that $\text{STLIN}^{2n} \neq \text{STLIN}^{2n+1}$ it follows that $\text{P} \neq \text{LOGSPACE}$ and $\text{EXP} \neq \text{LINSPACE}$).

Types, arrays and numbers. The types are defined the standard way, that is, $\mathbf{0}$ is a *type*, and $\sigma \to \tau$ is a type when σ and τ are types. We say a type σ has *level* n when $\mathrm{lv}(\sigma) = n$ where $\mathrm{lv}(\mathbf{0}) = 0$ and $\mathrm{lv}(\sigma \to \tau) = \max(\mathrm{lv}(\sigma) + 1, \mathrm{lv}(\tau))$. We define the *cardinality of type* σ *at base* b, in symbols $|\sigma|_b$, by recursion on the build-up of σ: $|\mathbf{0}|_b = b$ and $|\rho \to \tau|_b = |\tau|_b^{|\rho|_b}$.

The natural number a is a *number of type* σ *at base* b, in symbols $a : \sigma_b$, iff $a < |\sigma|_b$. Any number $a : (\sigma \to \tau)_b$ can be uniquely written in the form

$$v_0 \ + \ v_1|\tau|_b^1 \ + \cdots + \ v_k|\tau|_b^k$$

where $k = |\sigma|_b - 1$ and $v_j : \tau_b$ for $j \in \{0, \ldots, k\}$. We call $v_0, \ldots, v_k$ the *entries* in a, and for any $i : \sigma_b$, we denote the i'th entry in a by $a[i]_b$, i.e. $a[i]_b = v_i$. The notation $a[i_1, \ldots, i_n]_b$, where $n \geq 1$, abbreviates $((a[i_1]_b)[i_2]_b) \ldots)[i_n]_b$. It is natural to view the number $a : (\sigma \to \tau)_b$ as an *array* with index set σ_b where each position $a[i]$ in the array a (for $i \in \sigma_b$) is an element in the set τ_b. We might also view $a : (\sigma \to \tau)_b$ as a finite functional.

An imperative programming language. The language has an infinite supply of *program variables* $\mathtt{x}_0^\sigma, \mathtt{x}_1^\sigma, \mathtt{x}_2^\sigma, \ldots$ for any type σ. Any variable of type σ is a *term* of type σ; $t\mathtt{[X]}$ is an is an term of type τ if $\mathtt{X}$ is a variable of type σ and t is a term of type $\sigma \to \tau$. We use $\mathtt{a[i_1, \ldots, i_n]}$ to abbreviate $\mathtt{a[i_1][i_2]} \ldots \mathtt{[i_n]}$. (Thus a term is a single variable, or it has the form $\mathtt{a[i_1, \ldots, i_n]}$ where $\mathtt{a}, \mathtt{i_1}, \ldots, \mathtt{i_n}$ are variables.) The syntax of a program is given by

$\mathtt{t} \in$ Term	::=	term of type $\mathbf{0}$
$\mathtt{s} \in$ Statement	::=	$\mathtt{accept} \mid \mathtt{t+} \mid \mathtt{ifnt\,\{\,p\}} \mid \mathtt{for}_\sigma\,\mathtt{\{\,p\,\}}$ (for any type σ)
$\mathtt{p} \in$ Program	::=	$\mathtt{s} \mid \mathtt{s;\,p}$

We will now give an explanation of the programming language's semantics. (A formalized semantics can be found in [3].) Program variables of type σ store natural numbers in the set $\{0, 1, \ldots, |\sigma|_b - 1\}$. The input to a program is a single natural number x. The program *accepts* its input if it executes the primitive statement `accept` ; otherwise the program rejects its input. A program is executed in a particular base b, and it will either be executed in *type 0 mode* or in *type 1 mode*. In type 0 mode, the input x is stored in the variable $\mathtt{x}_0^{\mathbf{0}}$, and the base b is set to $\max(x, 1) + 1$. Type 1 mode is similar, but the input x is stored in binary notation in the variable $\mathtt{x}_0^{\mathbf{0}\to\mathbf{0}}$, and the base b is set to $|x| + 1$ where $|x|$ denotes the length of the binary representation of $|x|$. (So when the program starts $\mathtt{x}_0^{\mathbf{0}\to\mathbf{0}}\mathtt{[}i\mathtt{]}$ equals the i'th digit in the binary representation of x for $i < b$.) The only primitive instruction which is capable of modifying the variables, has either the form (i) $\mathtt{a[X_1, \ldots, X_n]+}$ where $\mathtt{a}, \mathtt{X_1}, \ldots, \mathtt{X_n}$ are variables such that $\mathtt{a[X_1, \ldots, X_n]}$ is a type $\mathbf{0}$ term, or the form (ii) $\mathtt{Y+}$ where $\mathtt{Y}$ is a type $\mathbf{0}$ variable. In case (i) the entry $\mathtt{a[X_1, \ldots, X_n]}$ i is increased by 1 modulo b, that is, the entry will be increased by 1, except in the case when the entry equals $b - 1$, then it is set to 0. In case (ii) $\mathtt{Y}$ is increased by 1 modulo b. The *loop* $\mathtt{for}_\sigma\,\mathtt{\{\,p\,\}}$ executes the program $\mathtt{p;\,p;} \ldots \mathtt{p}$ where $\mathtt{p}$ is repeated $|\sigma|_b$ times. The *conditional*

`ifnt{p}` executes the program `p` if the value (entry) `t` refers to is different from 0; otherwise the conditional does nothing. The semicolon composing two programs has the standard meaning.

The characterization of the hierarchies. A program decides the problem A when the program accepts the input x iff $x \in A$. A program has *data level* n when every variable occurring in a program instruction of the form t+, has level $\leq n$. A for$_\sigma$-loop has level n where $n = \mathrm{lv}(\sigma)$. A program has *loop level* n when every loop in program has level $\leq n$. A problem A belongs to the class $\mathcal{L}^{i,j}_{\text{TYPE } m}$ iff A can be decided in the type m mode by some for-program of loop level i and data level j.

Theorem 1. *For all* $i \in \mathbb{N}$ *we have (i)* TIME $2_i^{\text{POLY}} = \mathcal{L}^{i,i+1}_{\text{TYPE 1}}$, *(ii)* TIME $2_{i+1}^{\text{LIN}} = \mathcal{L}^{i,i+1}_{\text{TYPE 0}}$, *(iii)* SPACE $2_i^{\text{POLY}} = \mathcal{L}^{i+1,i+1}_{\text{TYPE 1}}$ *and (iv)* SPACE $2_i^{\text{LIN}} = \mathcal{L}^{i,i}_{\text{TYPE 0}}$. *Besides, we have (v)* LOGSPACE $= \mathcal{L}^{0,0}_{\text{TYPE 1}}$. *See also Table 1.*

Thus, we see that all the major deterministic complexity classes are captured by fragments of a pure and simple imperative programming language. This clarifies the relationship between these complexity classes and diminishes the gap between complexity theory and programming language theory.

Table 1. The table relates the equations in Theorem 1 to a fairly standard nomenclature.

Level of the for-program		**Input objects**	
Loop	Data	Type 1	Type 0
0	0	LOGSPACE	LINSPACE
0	1	P	EXP
1	1	PSPACE	EXPSPACE
1	2	POLYEXP	TIME 2_2^{LIN}
2	2	POLYEXPSPACE	SPACE 2_2^{LIN}
$\vdots$	$\vdots$	$\vdots$	$\vdots$
i	$i+1$	TIME 2_i^{POLY}	TIME 2_{i+1}^{LIN}
$i+1$	$i+1$	SPACE 2_i^{POLY}	SPACE 2_{i+1}^{LIN}
$\vdots$	$\vdots$	$\vdots$	$\vdots$

A functional programming language. We define the *terms* of *the (standard) typed* λ*-calculus*: a variable of type σ is a term of type σ; $\lambda x M$ is a term of type $\sigma \to \tau$ if σ is a variable of type σ and M is a term of type τ; (MN) is a term of

type τ if M is a term of type $\sigma \to \tau$ and N is a term of type σ; $\langle M, N\rangle$ is a term of type $\sigma \times \tau$ if M is a term of type σ and N is a term of type τ; $\mathbf{fst}M$ ($\mathbf{snd}M$) is a term of type σ (τ) if M is a term of type $\sigma \times \tau$. The reduction rules of the standard typed λ-calculus are $(\lambda x M)N \rhd M[x := N]$; $\mathbf{fst}\langle M, N\rangle \rhd M$; and $\mathbf{snd}\langle M, N\rangle \rhd N$. The calculus T^- is the standard typed λ-calculus extended with the constant $1 : \mathbf{0}$, and for each type σ the *recursor* R_σ of type $\sigma, \mathbf{0} \to \sigma \to \sigma, \mathbf{0} \to \sigma$. The calculus T is the calculus T^- extended with the constants $0 : \mathbf{0}$ (zero) and $s : \mathbf{0} \to \mathbf{0}$ (successor), the reduction rule $1 \rhd s0$, and for each type σ, the reduction rules $R_\sigma(P, Q, 0) \rhd P$ and $R_\sigma(P, Q, sN) \rhd Q(N, R_\sigma(P, Q, N))$. We use $\overline{n}$ to denote the *numeral* $s^n 0$ where $s^0 0 = 0$ and $s^{n+1}0 = (ss^n 0)$.

Note that the successor cannot occur in a T^--term! Further, note that the calculus T^- has no reduction rules in addition to those of the standard typed λ-calculus, and that e.g. the term $\mathrm{R}_\sigma(M, N, 1)$ is irreducible in the calculus T^- if M and N are irreducible.

It is well known that any closed T-term N of type $\mathbf{0}$ normalizes to a unique numeral. Thus, a closed T^--term M of type $\mathbf{0} \to \mathbf{0}$ can be viewed as a functional program, and we will simply call such a term *a program*. The program M is executed on the input n by normalizing the term $M\overline{n}$. The T^--program M decides the problem A when $M\overline{n}$ normalizes to $\overline{0}$ iff $n \in A$. The $\mathrm{rk}(M)$ of the T^--program M equals the least $n \in \mathbb{N}$ such that for any recursor R_σ occurring in M we have $\mathrm{lv}(\sigma) \leq n$. A problem A belongs to the class $\mathcal{F}^n$ iff A can be decided by a T^--program of rank $\leq n$.

Theorem 2. $\textsc{space}\ 2^{\textsc{lin}}_n = \mathcal{F}^{2n}$ *and* $\textsc{time}\ 2^{\textsc{lin}}_{n+1} = \mathcal{F}^{2n+1}$.

Th. 1 and Th. 2 yield an interesting comparison of the computational power of the fragments of our imperative and the fragments of our functional language: we have $\mathcal{F}^{2n+\epsilon} = \mathcal{L}^{n,n+\epsilon}_{\textsc{type}\ 1}$ where $\epsilon \in \{0, 1\}$. Moreover, our proofs of the theorems yield algorithms for transforming imperative programs into functional programs, and vice versa, such that the various complexity constraints are respected.

More details can be found in [3].

References

1. L. Kristiansen and P.J. Voda. *Complexity classes and fragments of C.* Information Processing Letters 88 (2003) 213-218.
2. L. Kristiansen and P.J. Voda. *The surprising power of restricted programs and Gödel's functionals.* In M. Baaz and J.A. Makowsky (Eds.): Computer Science Logic, LNCS 2803, pp. 345-358, Springer, 2003.
3. L. Kristiansen and P.J. Voda. *Complexity classes and higher types.* E-print (preprint) Dept. of Math. University of Oslo, Pure mathematics No. 16, May 2004. (`http://www.math.uio.no/eprint/pure_math/2004/pure_2004.html`)

Coin Games – Abstract

Rafał Somla
Uppsala University
rsomla@it.uu.se

Parity games – a two-player graph games with parity winning conditions – provide an elegant framework for verification of regular properties of reactive systems. Quite naturally, an accepting run of an alternating tree automaton can be viewed as a winning strategy in a parity game. It is also standard to reduce verification of a property expressed by a mu-calculus formula to the problem of deciding the winner in a parity game [1]. Good understanding of the parity winning conditions can help in developing better algorithms for model checking the general regular properties of systems.

In this work we propose games with different winning criteria which are equivalent to parity games but, in our opinion, are simpler to understand and work with. The main idea is to introduce fees for entering certain positions in the game. One of the players has a collection of coins which are used to pay these fees. His goal is to spend as few coins as possible or, equivalently, to play as long as possible. The other player's goal is to force the first player to spend all his coins thus making him unable to continue the play.

The winning conditions are simple and are determined by the current configuration of the game only. The second player wins a play if the first player does not have enough coins to pay for the next move. If this is not the case then the play can continue forever and the first player wins. In a sense this can be interpreted as safety/reachability winning conditions: the first player wants to stay in the set of safe configurations, where he have enough coins to pay the fees; the second player wants to reach a configuration where the first player is blocked. A possibility of reducing parity winning conditions to safety ones was also reported in [2]. However, this is done at the expense of adding exponentially many new positions in the game, while our approach does not change the set of game positions.

In a coin game one can evaluate each position by fining the minimal collection of coins which enables the first player to win a game starting from that position. Having such evaluation, a winning strategy for the first player is to always choose a successor with minimal optimal cost. This resembles very much the way optimal strategies can be found in payoff-based games like games with finite game tree or simple stochastic games. Inspired by the methods used to solve the latter [3], we formulate local optimality equations and prove that the optimal costs satisfy them. We propose to solve the local optimality equations by simple iterative fixed point computation.

The resulting algorithm is equivalent to the small-progress measure method for solving parity games proposed in [4]. Our work shows that this method is an instance of a general principle of finding optimal values and strategies in a payoff-based game by so-called value iteration.

References

[1] Colin Stirling. Bisimulation, modal logic and model checking games. *L. J. of the IGPL*, 7(1), 1999.

[2] Julien Bernet, David Janin and Igor Walukiewicz. Permissive strategies: from parity games to safety games. *Theoretical Informatics and Applications (RAIRO)*, volume 36, pages 251–275, 2002.

[3] Anne Condon. On algorithms for simple stochastic games. In Jin-Yi Cai, editor, *Advances in Computational Complexity Theory*, volume 13 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 51–73. AMS, 1993.

[4] Marcin Jurdziński. Small progress measures for solving parity games. In *Proceedings of STACS*, volume 1770 of *LNCS*, pages 290–301, 2000.

Automated Black-Box Testing of Functional Correctness using Function Approximation

[Extended Abstract]

Karl Meinke
Department of Numerical Analysis and Computer Science
Royal Institute of Technology
100-44 Stockholm, Sweden
April 20, 2004
karlm@nada.kth.se

ABSTRACT

We consider black-box testing of functional correctness as a special case of a satisfiability or constraint solving problem. We introduce a general method for solving this problem based on function approximation. We then describe some practical results obtained for an automated testing algorithm using approximation by piecewise polynomial functions.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Algorithms, Theory, Verification

Keywords

approximation, black-box test, constraint solving, formal specification, functional test, satisfiability problem, test coverage

1. INTRODUCTION.

Black-box functional testing of a software system involves generating and executing a set of test cases (i.e. inputs) and judging each outcome (the oracle step) using only a set of functional requirements. This approach is usually justified on the grounds that structurally based ("glass-box") testing methods, such as exhaustive path exploration, have a superlinear time complexity with respect to the size of the system under test (SUT). Thus glass-box testing methods do not scale well to large systems.

ISSTA'04 July 11–14, 2004 Boston, Massachusetts, USA

For functional test automation, some kind of formal and machine recognisable requirements language is necessary. Without any loss of generality, we can assume that first-order predicate logic is an adequate formal requirement language. (See for example [6].)

We shall consider software systems whose functional correctness can be modeled by pre and postconditions *pre* and *post* which are first-order predicate formulas. A *successful black-box test* of a system S, with respect to a functional specification *pre* and *post* is an assignment α of values to the input variables $\overline{x}$ of S which satisfies the precondition *pre*, such that S terminates given α and the resulting assignment ω to the output variables $\overline{y}$ of S (together with α) satisfies the negated postcondition $\neg post$. Black-box functional testing of S is therefore the search for successful tests with respect to a functional specification $\{pre\}S\{post\}$ of S.

We will consider a particular algorithmic approach to this search problem. As we have seen, software testing involves solving the pair of constraints *pre* and $\neg post$. However, a naive application of classical constraint solving methods does not answer all the questions we are interested in. An important issue is the choice of a stopping criterion for any search based testing algorithm. Quite simply, how much testing is enough? This problem is well known in the testing community as the *coverage problem*. We will consider a measure of coverage, based on convergence of functional approximations, which reflects the underlying complexity of the SUT.

In the classical theory of function approximation, some general class F of functions is shown to be approximable by a much more limited and tractable subclass $A \subseteq F$ of functions. Well known classes of approximating functions include polynomials, trigonometric functions, sigmoidal functions, radial basis functions, wavelets, etc. We can apply function approximation theory to testing in the following way. After executing some finite number n of test cases on an SUT S, we will have constructed n test input assignments $\alpha_1, \ldots, \alpha_n$. Assuming termination in each case, we will have computed with S a corresponding series of n output assignments, $\omega_1, \ldots, \omega_n$. Suppose that we have still not found a successful test case and wish to continue testing. Instead of throwing away the negative results obtained so far, we could try to use them in some intelligent way to construct a new (and hopefully better) test case α_{n+1}. At

the very least, α_{n+1} should differ from $\alpha_1, \ldots, \alpha_n$. A more sophisticated approach than random choice of α_{n+1} is to combine $\alpha_1, \ldots, \alpha_n$ and $\omega_1, \ldots, \omega_n$ into some sort of approximate model M_n of the SUT S. From this approximate model, we may be able to construct a "best guess" of where a successful test case might be found, and choose α_{n+1} to be this estimate. Now even if α_{n+1} fails as a test, it can be used together with the output ω_{n+1} to produce a more refined approximate model M_{n+1}. By choosing α_1 at random and constructing each α_{n+1} from M_n, we obtain an iterative test case generation algorithm. In a sense, we approach the testing problem as a kind of learning problem about the underlying SUT S. One advantage of the approximation approach is that a stopping criterion can be formulated as a convergence criterion on approximation.

Our approach opens up a large family of interesting algorithms based on different kinds of approximations. We will illustrate the approach, and a simple case study, for a particular class of approximations, namely piecewise polynomial functions.

The approach to testing as a computational learning (CL) problem has been pursued by several authors, starting with pioneering work of Budd and Angluin [3] and Weyuker's early work (e.g. [13]). The CL approach seeks to infer a program from test results, and use the inferred program to establish test adequacy. It therefore has some similar principles to our approach. However, functional requirements are implicit in the CL approach, and convergence within an approximation space is not used to define adequacy. Nor is the automatic generation and optimisation of test cases considered.

Other approaches to learning-based testing include [2], which uses classical constraint solving algorithms and a glass box approach, but do not use approximation theory. Theoretical research on the complexity of testing measured by the complexity of learning using Vapnik-Chervonenkis (VC) dimension is [11] and [12], though this research does not yet seem to have led to the design of practical testing algorithms.

The coverage problem for black-box testing has been extensively analysed in work on mutation testing, starting with [5] and [7]. In practise, mutation testing is limited by theoretical problems, such as the recursive unsolvability of the equivalence problem for programs (see e.g. [9]). Mutation testing is concerned with quantitative adequacy measures of a specific test set, and not numerical or probabilistic results about coverage as such. Furthermore, it does not attempt to provide any feedback on how to improve an inadequate test set, as our approximation approach does.

The organisation of this paper is as follows. In Section 2 we formalise some basic definitions of black-box functional testing. In Section 3 we describe our algorithmic approach. In Section 4 we present a case study for testing a simple numerical algorithm. Finally, in Section 5 we draw some conclusions and discuss some issues for further research.

The pre-requisites of our paper are some familiarity with functional requirements specification using formal methods and logic. Technically, we make as few assumptions as possible. In particular, we introduce the concept of a functional specification formalised as a pair of pre and postconditions over a first-order language from first principles. A more detailed introduction can be found in [1] or [8].

1.1 References.

[1] J.W. de Bakker, *Mathematical Theory of Program Correctness*, Prentice-Hall, 1980.

[2] F. Bergadano, D. Gunetti, Testing by means of inductive program learning, *ACM Trans. Software Engineering and Methodology* 5 (2) 119-145, 1996.

[3] T.A. Budd, D. Angluin, Two notions of correctness and their relation to testing, *Acta Informatica* 18, 31-45, 1982.

[4] E.W. Cheney, *Approximation Theory*, American mathematical Society Chelsea Publishing, Providence, Rhode Island, 1966.

[5] R.A. DeMillo, R.J. Lipton, F.G. Sayward, Hints on test data selection: help for the practicing programmer, *IEEE Computer* 11(4), 34-41, 1978.

[6] H.B. Enderton, *A Mathematical Introduction to Logic*, Academic Press, 1972.

[7] R.G. Hamlet, Testing programs with the aid of a compiler, *IEEE Transactions on Software Engineering*, 3(4), 279-290, 1977.

[8] J. Loeckx and K. Sieber, *The Foundations of Program Verification*, John Wiley, Chichester, 1987.

[9] A.J. Offut, J. Pan, Automatically detecting equivalent mutants and infeasible paths, *The Journal of Software Testing, Verification and Reliability* 7 (3), 165-192, 1997.

[10] M.S. Phadke, Planning efficient software tests, *Crosstalk* 10 (10), 11-15, 1997.

[11] K. Romanik, Approximate testing and its relationship to learning, *Theoret. Comput. Sci.* 188, 79-99, 1997.

[12] K. Romanik and J.S. Vitter, Using Vapnik-Chervonenkis dimension to analyze the testing complexity of program segments, *Information and Computation* 128 (2), 87-108, 1996.

[13] E.J. Weyuker, Assessing test data adequacy through program inference, *ACM Trans. Program. Lang., Syst.*, 5 (4), 641-655, 1983.

Specifying Test Cases Using Observer Automata

Johan Blom, Anders Hessel, Bengt Jonsson, and Paul Pettersson

Department of Information Technology, Uppsala University, P.O. Box 337,
SE-751 05 Uppsala, Sweden. E-mail: {johan,hessel,bengt,paupet}@it.uu.se.

Abstract. We present a technique for specifying coverage criteria and a method for generating test suites for systems whose behaviours can be described as extended finite state machines (EFSM). To specify coverage criteria we use observer automata with *parameters*, which monitor and accept traces that cover a given test criterion of an EFSM. The flexibility of the technique is demonstrated by specifying a number of well-known coverage criteria based on control- and data-flow information using observer automata with parameters. We also develop a method for generating test cases from coverage criteria specified as observers. It is based on transforming a given observer automata into a bitvector analysis problem that can be efficiently implemented as an extension to an existing state-space exploration such as, e.g. SPIN or UPPAAL.

Introduction. Model based test case generation has in recent years been developed as a prominent technique in testing of reactive software systems. A model serves both the purpose of specifying how the system should respond to inputs from its environment, and of guiding the selection of test cases, e.g., using suitable coverage criteria.

Since different coverage criteria are suitable in different situations, and satisfy different constraints on fault detection capability, cost, information about where potential faults may be located, etc., it is highly desirable that a test generation tool is able to generate test suites in a flexible manner, for a wide variety of different coverage criteria. In other words, a test generation tool should accept a simple specification of a coverage criterion, given in a language that can easily specify a large set of coverage criteria, and be able to generate test suites accordingly.

We present a technique for specifying coverage criteria in a simple and flexible manner, and a method for generating test cases according to such coverage criteria. The technique fits well as an extension of a state-space exploration tool, such as, e.g., SPIN [2] or UPPAAL [5], which performs enumerative or symbolic state-space exploration. It can also be used to generate monitors that measure the coverage of a specific test suite by monitoring the test execution.

In our technique, a coverage criterion is given as a set of *coverage items*, each of which represents an interesting structural property of the EFSM which should be examined by a test suite. A coverage item can state that a particular state, edge, or similar, should be visited, it can be an explicit test purpose, etc. Each coverage item is specified by an *observer*, which observes the execution of a test

case, and reports acceptance when the test case has *covered* the coverage item that it specifies. For instance, a coverage item stating that a control state l of an EFSM model should be visited simply observes how the EFSM executes and reports acceptance when it enters l.

A typical coverage criterion is given as a (often rather large) set of coverage items. An important mechanism to facilitate specification of many coverage criteria is to allow *parameterization* of observers. In this way, one can specify a set of coverage items parameterized over, e.g., control states, data variables, edges, etc. of the EFSM model. Using this simple and general mechanism, we can specify most of the coverage criteria that have been used in the literature, and also tailor coverage to specific features of a particular SUT. For instance, if a particular interface is very error prone, we can specify a coverage criterion which requires all possible interleavings of interactions on that interface to be exhibited in a test suite.

A specification of a coverage criterion can be used for test suite generation using a state-space exploration tool. First, we superpose the coverage observers onto the EFSM, then we search for a test sequence or set of test sequences in which as many observers as possible report acceptance. For parameterized observers, we can record the achieved coverage by a (typically small) set of bitvectors, indexed by parameter values, which concisely represent the states of a large set of parameterized observers. The same machinery can also be used to monitor the achieved coverage of a certain test suite.

Releted work. Some approaches present more flexible techniques for specifying a variety of coverage criteria. Hong et al [4, 3] describe how flow-based coverage criteria can be expressed in temporal logic. A particular coverage item is expressed in CTL, and a model checker generates a trace which covers the coverage item. In our approach, we use observers instead of temporal logic, which avoids some of the limitations of temporal logic [6]. Friedman et al [1] specifies coverage by giving a set of projections of the state space (e.g., on individual state variables, components of control flow) that should be covered, possibly under some restrictions. Our approach generalizes this one, by allowing to define observers. Also, we can let one pass of a state-space exploration tool generate a test suite that covers a large set of coverage items, whereas the above approaches invoke a run of a model checker for each coverage item.

Observers. As a very simple example, the coverage item *"visit location l of the EFSM"* can be represented by an observer with one initial state, and one accepting location, named $loc(l)$, which is entered when the EFSM enters location l. The coverage criterion *"visit all locations of the EFSM"* can be represented by a parameterized observer with one initial state, and one parameterized accepting location, named $loc(\mathtt{L})$, where `L` is a parameter that ranges over locations in the EFSM. For each value l of `L`, the location $loc(l)$ is entered when the EFSM enters location l.

Conclusions. In previous works, we have implemented special cases of this test case generation technique, using UPPAAL, indicating that the approach is practical. We are currently working on a general implementation of the observer concept, and plan to apply it in a larger case study.

References

1. G. Friedman, A. Hartman, K. Nagin, and T. Shiran. Projected state machine coverage for software testing. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 134–143, 2002.
2. G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, SE-23(5):279–295, May 1997.
3. H.S. Hong, S.D. Cha, I. Lee, O. Sokolsky, and H. Ural. Data flow testing as model checking. In *ICSE'03: 25th Int. Conf. on Software Enginering*, pages 232–242, May 2003.
4. H.S. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage. In J.-P. Katoen and P. Stevens, editors, *Proc. TACAS '02,* 8^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 327–341. Springer Verlag, 2002.
5. K.G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Software Tools for Technology Transfer*, 1(1-2), 1997.
6. P. Wolper. Temporal logic can be more expressive. In *Proc.* 22^{nd} *Annual Symp. Foundations of Computer Science*, pages 340–348, 1981.

Modeling and Simulating an Industrial Robot

Johan Andersson[2], Pavel Krčál[1], Leonid Mokrushin[1], Christer Norström[2], Anders Wall[2], and Wang Yi[1]

[1] Dept. of Information Technology, Uppsala University, Sweden.
Email: {pavelk,leom,yi}@it.uu.se.
[2] Dept. of Computer Science and Engineering, Mälardalen University, Sweden. Email: {anders.wall,johan.x.andersson,christer.norstrom}@mdh.se.

Abstract We have modeled a robotic control system developed by ABB Robotics using probabilistic timed automata with asynchronous tasks. The TIMES tool has been extended to enable for simulation of such model. Results of our experiments are consistent with the ones achieved by simulating the actual system. This work provided us with several new insights on automated analysis of systems with real-time tasks.

1 Modeling

Automated analysis of industrial size real-time systems is an important research area. All attempts to model such systems in some formalism suitable for verification can give a lot of insight for development of modeling languages and verification tools.

Our aim is to use TIMES [AFM+02] tool to analyze a robotic control system implemented by ABB Robotics. A considerable amount of work has been spent on modeling and simulation of this system using ART-ML during an earlier case-study at the company [WAN+03]. The main objective of the case study was to investigate the temporal behavior of the control system and document it in a model, suitable for analyzing the impact on response-times and resource utilization caused by adding new features to the system. An ART-ML simulator was used to analyze the model, with respect to task response-times. The response-time distribution from the simulations was compared with measurements from a real executing robot controller, showing a significant resemblance.

We have used the ART-ML model as the specification. TIMES tool was developed for schedulability analysis of the set of tasks whose release patterns can be specified using timed automata. In contrast with ART-ML based tools, TIMES can also perform model-checking using UPPAAL [LPY97] back-end. Therefore, successful modeling of the system in TIMES would extend the possibilities of robotic control system verification.

In ART-ML tasks may have variable computation times by assigning a discrete probability distribution over different values of computation times to each subtask. However, computation times of tasks in TIMES are fixed. Therefore, we introduce probabilistic transitions in timed automata model and map each possible execution of a subtask to one task in TIMES.

Syntactically, we annotate all transitions outgoing from a location with a discrete distribution. Semantics is defined as a labeled transition system with probabilistic transitions (each finite run has its own probability). Our modeling language is then probabilistic timed automata extended with tasks, where tasks are executables parametrized with a computation time, a deadline, and a priority.

ART-ML allows for a rich control structure within a task body. Since there is no task structure in TIMES, branching, loops, and communication are handled by timed automata. To measure response times of robot tasks we wrap their corresponding parts in the timed automata model using a special task and precedence constraints.

2 Experiments

We have performed series of experiments with the model of the robotic controller using the TIMES tool extended with probabilistic transitions. If several transitions become enabled at a time (non-determinism), the probabilities are taken into account to make a random choice accordingly. Probabilities of transitions in parallel composition of automata are evaluated proportionally. All transitions that have not been annotated with probabilities are considered to be equally probable.

The TIMES tool has been also extended with the offline simulation option allowing to run a set of simulations in a batch mode using command line execution script. The worst case response times of each task in a single simulation are collected in a common log file and further used for analysis and visualization.

The ART-ML model of ABB Robotics control system consists of 4 tasks, namely A, B, C, and OH. Each task has a control structure and calls to its subtasks. Each subtask is parameterized with a set of pairs (C_i, P_i), where C_i is the computation time and P_i is the probability with which the execution of this subtask takes C_i time units. Naturally, all probabilities P_i of one subtask add up to 100%. Task A consists of 2 subtasks with choice of 3 and 4 probable computation times respectively, task B contains 3 subtasks with choice of 4 each, task C has only 1 subtask with choice of 3, and task OH has 4 subtasks with the following choices: 4, 4, 4, and 2. The results of experiments are shown on Figure 1. The plots represent task response times collected during 100 independent simulations.

3 Future Work

This case study has brought into focus the need of richer models for control structure of tasks. Quite often while modeling it is feasible to abstract a piece of executable code with a higher level descriptions of task control structure and probabilistic model of its execution. One direction of our future work is to introduce a task control structure in our modeling language. The tasks in TIMES will become jobs, higher level description of a subtask structure. One can see it

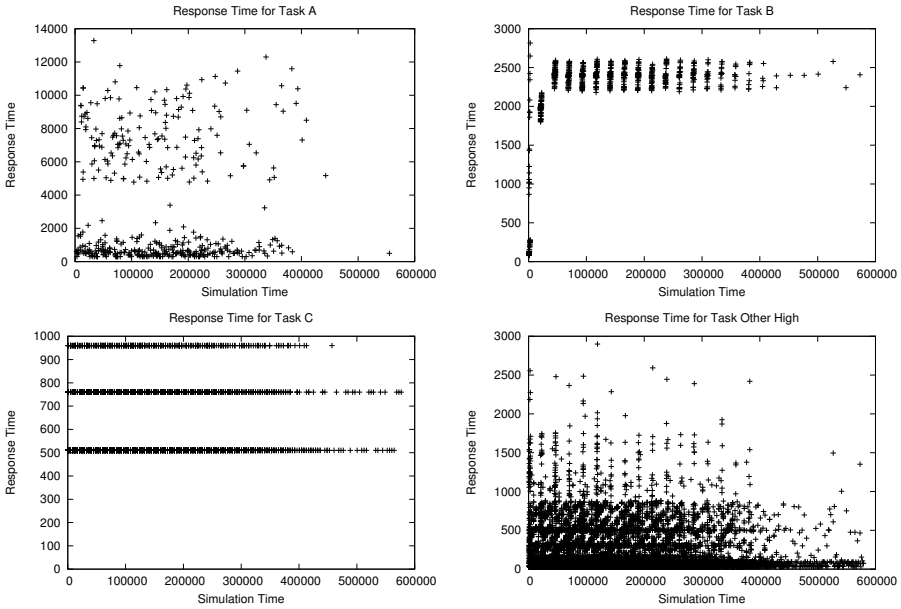


Figure1. Response times of the control system tasks.

as an extension of similar concept of responses and actions described in UML SPT profile [Obj02].

Our next goal is to verify the model by schedulability analysis and model-checking. To achieve this, we have to apply some further abstractions to the original model preserving the properties we want to verify. TIMES tool cannot verify probabilistic systems. Therefore, we will have to either abstract from probabilities by means of fairness constraints or we will have to extend the verification procedure to be able to handle probabilistic models.

References

[AFM+02] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Times – a tool for modelling and implementation of embedded systems. In *Proceedings of TACAS'02*, volume 2280 of *LNCS*, pages 460–464. Springer–Verlag, 2002.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[Obj02] Object Management Group. *UML™ Profile for Schedulability, Performance, and Time Specification*, March 2002.

[WAN+03] A. Wall, J. Andersson, J. Neander, C. Norström, and M. Lembke. Introducing Temporal Analyzability Late in the Lifecycle of Complex Real-Time Systems. In *Proc. International Conference on Real-Time Embedded Computing Systems and Applications*, 2003.

A simulator approach to data race detection

Karl Marklund * Björn Victor †

Department of Information Technology
Uppsala University
Box 337, SE-751 05 Sweden

September 6, 2004

Abstract

Correct parallel programming is intrinsically difficult. A data race occurs when two or more concurrently executing processes access the same shared-memory location in an unsynchronized way, and at least one of the processes modifies the contents of the location. On the other hand, adding incorrect synchronization may cause the program to deadlock. The occurrence of a data race in a particular execution depends on small timing variations. Running the program with the same input over and over again may not be enough to detect the error and for non-trivial programs, formal proofs and static methods are not applicable.

Unlike static analysis, dynamic analysis detects the race conditions exhibited by a particular execution of a program. Typically, probes are inserted into the code and data race detection is performed on-the-fly during a monitored execution. To make the analysis independent of programming language, the probes are inserted directly into the executable binary by a technique called binary rewriting. Features such as data in code and code in data makes binary rewriting hard. The intrusion caused by the probes may alter the timing and change the set of execution schedules likely to occur.

We propose a simulator based approach for detecting data races in shared memory parallel programs. Compared to other dynamic state-of-art tools such as RecPlay by Ronsse and De Bosschere, we do not use binary rewriting techniques to collect information during the analysis. Instead, the analysis is performed as part of a simulated execution of the unmodified executable, hence the method does not induce any probe effects. As a result, in contrast to non-zero intrusion methods, there is no possibility for a bug that disappears or alters its behavior when one attempts to probe or isolate it.

Using the vector clock method found in the RecPlay tool, a limited prototype data race detector have been developed as a plug-in module for the full-system, instruction-level simulator SIMICS. The prototype requires a special form of threaded programs that are simulated without an operating system. Each thread is executed on a separate CPU inside the simulator and each time a memory access occurs a data race detection routine is called.

To synchronize, the threads uses simple spin-locks which can easily be detected by pattern matching on the stream of executed instructions. Within a thread, all instructions between two consecutive locking operations is called a segment. The synchronization limits the possible interleavings of segments and this is recorded by giving each segment a vector time stamp. All memory locations read and written by a segment is kept in a load set and a store set. Two segments can execute in parallel only if their vector timestamps are unordered. By examining the load and store sets of all parallel segments, a data race can be found.

We plan to extend our analysis to real programs, running on top of a real operating system. By utilizing the data structures used by the operating system itself, thread and process information can be extracted. The goal is to build a functional tool, capable of finding data races in arbitrary programs, written in any programming language, possibly lacking source code, running on top of the Linux operating system.

*karl.marklund@it.uu.se
†bjorn.victor@it.uu.se

Encoding the Applied π-calculus in the π-calculus * †

Anders Bloch abloch@gmail.com
Morten V. Frederiksen mortenf@gmail.com
Bjørn Haagensen bh@cs.auc.dk

Abstract

In the π-calculus [SW01, Mil99] the only kind of messages that can be communicated are atomic names and this simplicity is appealing from a theoretical point of view. But for many purposes one often finds it necessary or convenient to be able to represent, for instance, data types such as pairs or even functions and object-like structures. In order to be able to directly represent such data types a plethora of extensions of the π-calculus has emerged. These are often aimed at capturing common features and properties of programming languages.

Appπ [AF01], proposed by Abadi and Fournet, is a generalization of the π-calculus which aims at capturing several of these extensions. More specifically, messages are allowed to be terms defined through a signature. Along with the signature a set of equations is defined, hence obtaining a presentation. The syntax and semantics of Appπ is parameterized by this presentation. Thus allowing for the development of a theory for which is valid for all calculi obtained by specifying a concrete presentation. Consider how pairs can be represented in Appπ. The signature includes a binary function symbol pair and two unary function symbols fst and snd. The set of equations includes

$$\mathsf{fst}(\mathsf{pair}(x, y)) = x$$
$$\mathsf{snd}(\mathsf{pair}(x, y)) = y.$$

The main contribution of our work is an encoding of Appπ in the π-calculus. Such an encoding is appealing for several reasone. For instance automated analysis of Appπ-processes could be performed by first encoding a process and then use one of the many automated tools that exist for the π-calculus. e.g. the Mobility Workbench [VM94] developed by Victor and Moller.

The same arguments apply for the substantial amount of theoretical tools that have been developed for the π-calculus. It should be possible to utilise some of these theoretical results in order to obtain a better understanding of Appπ. In fact, due to the generality of Appπ this arguments apply to any calculi expressible in Appπ.

*Full report is available online at http://www.cs.aau.dk/~bh/files/dat6project.ps.gz

†Aalborg University, Department of Computer Science Fredrik Bajers Vej 7E, DK-9220 Aalborg Øst

Although the π-calculus is well-known to be Turing-complete, the existence of such an encoding is not immediate, and indeed the encoding is non-trivial. The main difficulty lies in the encoding of the above mentionede equational theory which underlies the syntax and semantics of Appπ. Our encoding is inspired by a recent proposal for an encoding of the spi-calculus in the π-calculus [BPV03], but differs considerably in the way that terms are encoded. We rely on well-known results for transforming an equational theory into a terminating and confluent rewrite system. In fact this transformation can be done by using the Knuth-Bendix procedure [KB70]. Moreover the encoding requires the rules of the rewrite system to be left linear.

The general idea is that all terms are forced to reduce to their irreducible form before they can be accessed by the remaining parts of the system. This is done by initially encoding the terms in paralell with the encoding of the rewrite system which reduces the terms until they become irreducible. When on irreducible form the term becomes an object-like process with a number of predefined methods, including methods for comparing syntactically against another term. Since the rewrite system is terminating and confluent checking for equality then becomes a matter of checking for syntactic identity. The encoding also imposes a bottom-up evaluation strategy of terms. The choice of evaluation strategy is inessential for confluent rewrite systems.

As is often the case for this type of encodings, the representation of an encoded process will in general have to do a number of transitions in order to match a single transition of the original process. Therefore care needs to be taken when relating operational steps of an encoded process with that of the original process. In particular the encoding of an Appπ-term M requires a number of transitions where M itself can be rewritten in a single step. We have established that various subparts of the encoding are correct. This has been done to the extent that we have been able to prove operational soundness for rewriting of terms. I.e. if a term M is rewritten to its irreducible form N, then the encoding of M reduces to the encoding of N. A crusial property which is necessary is that the encoding of a term M is equivalent to the encoding of the irreducible form of N. We have made some modifications to encoding we first came up with, which we believe are sufficient in order to obtain such a result. Assuming this result holds we have proved soundess with respect to barbed equivalence.

References

[AF01] Martín Abadi and Cédric Fournet. Mobile Values, New Names, and Secure Communication. *ACM SIGPLAN Notices*, 36(3):104–115, March 2001.

[BPV03] Michael Baldamus, Joachim Parrow, and Björn Victor. Spi Calculus Translated to π-calculus Preserving May-Testing. Technical report, Department of Information Technology, Uppsala University, Sweden, December 2003.

[KB70] D. E. Knuth and P. B. Bendix, editors. *Simple word problems in universal algebra*, pages 263–297. Pergamon Press, 1970.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.

[SW01] Davide Sangiorgi and David Walker. *The π-calculus - A Theory of Mobile Processes.* Cambridge University Press, 2001.

[VM94] Björn Victor and Faron Moller. The Mobility Workbench — A tool for the π-calculus. In David Dill, editor, *Computer Aided Verification (Proc. of CAV'94)*, volume 818 of *LNCS*, pages 428–440. Springer, 1994.

A Timed Mobile Calculus *

Jing CHEN
Laboratoire d'Informatique Fondamentale d'Orléans
Université d'Orléans
Rue Léonard de Vinci, B.P. 6759
45067 ORLEANS Cedex 2, France
E-mail: chen@lifo.univ-orleans.fr

Extended Abstract

In the research on concurrency, real time has been studied intensively since time is a crucial factor in many protocols and controlling systems. Systems such as wireless/wired controllers, mobile equipments and protocols may not work correctly without considering time: the result of a computation finished after the deadline may be useless or even lead to fatal error.

Process algebra is a formal setup which uses algebraic techniques to model and analyze abstract programs, especially concurrent systems. A mobile calculus is a process algebraic calculus which can model mobile computation. Among the currently well-known are: the π calculus [1], the Join calculus [2], and the Mobile ambients [3]. Although these languages can model mobility, they cannot specify *when* a 'move' takes place – how it is related to time, in a satisfactory manner. On the other hand, the real-time calculi such as Timed CCS [4], Timed CSP [5] and etc. are not for mobile systems, and the current timed extensions for π calculus as in [6] and [7] have no new theory of bisimulations with respect to time. We propose to use π calculus as the basis for the timed calculi presented in this paper, because the primitives for channels in π calculus are simple yet powerful; channels can be created and passed, for example from server to clients. The scoping rules of π calculus guarantee that the environment of a protocol (the 'attacker') cannot access a channel that is not explicitly allowed.

Following the convention of standard π calculus, assuming that an infinite set of *names*, ranged over by x, y, a, b etc., will be used for communication channels. By introducing the new prefix $\epsilon(d)$ and an extended action prefix $\alpha@t$ (following the technique used in [4]), the π calculus can be extended to be a timed calculus, by the grammar:

$$\begin{array}{lcl} E & ::= & \mathbf{0} \mid \alpha@t.E \mid \epsilon(e).E \mid \nu x E \mid [x=y]E \mid E|E \mid E+E \mid T(\tilde{v}) \\ T & ::= & X \mid (\tilde{x})E \mid \mathbf{fix}XT \\ \alpha & ::= & \tau \mid \overline{x}y \mid x(y) \end{array}$$

where, as usual, $\mathbf{0}$ is the inactive process which is capable of doing nothing (but can have time delay). $\alpha@t.E(\alpha \neq \tau)$ can either wait for any time, or perform a discrete action while storing the total time elapsed before the action is executed. τ is modeled as an urgent action, thus $\tau@t.E$ cannot wait any more, which is the basis of the *Maximal Progress assumption*[4, 5]. $\epsilon(e).E$ must wait for at least e units of time and then it behaves as E. $[x=y]E$ has the same behavior as E when x is equal to y, otherwise it will become deadlocked. The x in $\nu x E$ is a local name which can not be accessed directly from outside. '+' is the standard nondeterministic choice operator, and '|' is the parallel composition operator. $T(\tilde{v})$ is an instantiation of abstraction T using parameters $\tilde{v}$. For example, Given $T = (x,y)\overline{x}y@t.E$, we have $T(a,b) = \bar{a}b@t.E\{a/x, b/y\}$.

*Supported by National Science Foundation of China, and by **le_STUDIUM**, http://lestudium.cnrs-orleans.fr/, Orléans

Similar to the Timed CCS [4], we have the same semantics rules such as $\alpha@t.E \xrightarrow{\alpha} E\{0/t\}$ and $\alpha'@t.E \xrightarrow{\epsilon(d)} \alpha'@t.E\{t+d/t\}$ $(\alpha' \neq \tau)$. Therefore the bound clock variable t can be used to record the waiting time of $\alpha'@t.E$. For example: $\alpha'@t.E(t) \xrightarrow{\epsilon(2.5)} \alpha'@t.E(t+2.5) \xrightarrow{\alpha'} E(2.5)$. And the rules concerning time delay prefix $\epsilon(d)$ are as follows:

$$\text{Delay}_0: \frac{}{\epsilon(e+d).E \xrightarrow{\epsilon(d)} \epsilon(e).E} \qquad \text{Delay}_1: \frac{E \xrightarrow{\epsilon(d)} E'}{\epsilon(e).E \xrightarrow{\epsilon(e+d)} E'}$$

$$\text{Delay}_2: \frac{E \xrightarrow{\eta} E'}{\epsilon(0).E \xrightarrow{\eta} E'} \qquad \text{Idle}_0: \frac{}{\mathbf{0} \xrightarrow{\epsilon(d)} \mathbf{0}}$$

All the operational rules of π calculus are kept intact in this extended calculus, and we need to add one more rule concerning the *Match*: $[x=y]E \xrightarrow{\epsilon(d)} [x=y]E$ provided $x \neq y$. This rule intends to guarantee that a deadlocked process should not stop time elapsing, and the process is therefore bisimilar to $\mathbf{0}$. To keep *Time Determinacy*, we need the rule that $E+F \xrightarrow{\epsilon(d)} E'+F'$ only if $E \xrightarrow{\epsilon(d)} E'$ and $F \xrightarrow{\epsilon(d)} F'$. The time delay behavior of $E|F$ is similar but a side condition must be provided that no internal communication should happen within d units of time, details can be found in [8] (similar to [4], a *Time Sort* for each process should be used).

Strong Ground Bisimulation $\dot{\sim}$ is defined as usual as in [1], where the time delay action can be consistently treated as a free action (an action without bound name). As in standard π [1], the relation is not preserved by *Input* prefix due to the problem of name substitutions, and it can be augmented to a congruence called *Strong Bisimulation* $(\sim)$. All the algebraic laws for Strong bisimulation over π calculus hold in this timed case, and the new calculus keeps the real time properties such as *Maximal Progress, Time Determinacy, Time Continuity* and *Time Persistency* [8, 4]. Weak Bisimulations have also been studied [8].

The work of axiomatization of such a calculus is now in progress, and we have first axiomatized the strong bisimulation over the recursion-free fragment of this calculus [8]. This work mainly relies on *Symbolic Bisimulation*, since the *Match* construction $[x=y]E$ can not be handled without (symbolic) reasoning over names [1].

Following the notations in [9, 10], let's first introduce **Conditions**, ranged over by C, D, which are finite sets of equality or inequality tests on names. $n(C)$ is the set of names appearing in C. A substitution σ satisfies C, written $\sigma \models C$, if $\sigma(x)=\sigma(y)$ for any $x=y \in C$, and $\sigma(x) \neq \sigma(y)$ for any $x \neq y \in C$. We write $C \Rightarrow D$ to mean that $\sigma \models C$ implies $\sigma \models D$ for any substitution σ. A condition C is *consistent* if there are no $x, y \in \mathcal{N}$ such that $C \Rightarrow x=y$ and $C \Rightarrow x \neq y$. C is *maximally consistent* on $V \subset \mathcal{N}$ if for any $x, y \in V$, either $C \Rightarrow x=y$ or $C \Rightarrow x \neq y$. C' is a *maximally consistent extension* of C on V, written $C' \in MCE_V(C)$, if $C \subseteq C'$ and C' is *maximally consistent* on V.

The symbolic equivalence between actions needs to be defined:

Definition 1 $\eta =^C \eta'$ *if*

If $\eta \equiv \tau$ *or* $\epsilon(d)$, *then* $\eta = \eta'$
If $\eta \equiv \overline{a}x$, *then* $\eta' \equiv \overline{b}x$ *and* $C \Rightarrow a=b \wedge x=y$
If $\eta \equiv \overline{a}(x)$, *then* $\eta' \equiv \overline{b}(x)$ *and* $C \Rightarrow a=b$
If $\eta \equiv a(x)$, *then* $\eta' \equiv b(x)$ *and* $C \Rightarrow a=b$

Similar to the work in [9], to use symbolic technique, we turn to a more abstract form of transitions - *Symbolic Transition*. For example, we have $\alpha.E \xrightarrow{true,\alpha} E$ which models the unconditional behavior, and for *Match*, we have $[x=y]E \xrightarrow{x \neq y, \epsilon(d)} [x=y]E$ which models the deadlock case for $[x=y]E$. We need another new and important symbolic transition rule:

$$\frac{E \xrightarrow{M,\epsilon(d)} E'}{\nu x E \xrightarrow{M',\epsilon(d)} \nu x E'} \quad \begin{array}{l} \nexists\ x=y \in M, \\ M'=M \setminus \{x \neq y \mid x \neq y \in M\} \end{array}$$

which makes a distinction between delay transitions generated by the deadlock case and those generated by the normal idle cases. Note that if $\exists\, x = y \in M$, the transition must be implied from an idle case (by transition induction), and then by νx, this symbolic transition will be restricted or cut, because x will become a distinct bound name (therefore not equal to any other names) in νxE. This symbolic semantics is very fundamental in proving the corresponding relationship between *Symbolic Transiton* and *Concrete Transition*, and in proving the completeness of the axiom system.

So far, we can adopt the definition of *Symbolic Bisimulation* in [9] as follows:

Definition 2 (Symbolic Bisimulation) *A condition indexed family of symmetric relations* $\mathcal{S} = \{S^C\}$ *is a symbolic bisimulation, if* $(E, F) \in S^C$ *implies:*

Whenever $E \xrightarrow{M,\eta} E'$ with $bn(\eta) \cap fn(E, F, C) = \emptyset$, then for each $C' \in MCE_{fn(E,F)}(C \cup M)$ there is a $F \xrightarrow{N,\eta'} F'$ such that $C' \Rightarrow N, \eta =^{C'} \eta'$, and $(E', F') \in S^{C''}$, where

$$C'' = \begin{cases} C' \cup \{x \neq y | y \in fn(\eta.E', \eta'.F')\} & \text{if } \eta \equiv \overline{a}(x) \\ C' & \text{otherwise} \end{cases}$$

If $(E, F) \in S^C \in \mathcal{S}$ where $\mathcal{S}$ is a symbolic bisimulation, then $E \sim^C F$. And we have proved that $E \sim^C F$ **iff** for any $\sigma \models C$, $E\sigma \dot{\sim} F\sigma$. Therefore easy to see that $E \sim^{true} F$ **iff** $E \sim F$.

An axiom system can then be presented, which consists of a set of inference rules together with some standard equations. The judgements of the inference system are of the form $C \triangleright P = Q$, where C is a *Condition*, and P, Q are process terms. The soundness and completeness of the axiomatization can then be proved, thus we have $C \triangleright E = F$ if and only $E \sim^C F$.

References

[1] R.Milner,J.Parrow,and D.Walker. A calculus of mobile processes, part I,II. *Information and Computation*, **100**:1-77,1992.

[2] C.Fournet and etc. A Calculus of Mobile Agents. In *Proc. of the 7th International Conference on Concurrency Theory (CONCUR'96)*, LNCS **1119**:514-529, Springer-Verlag, 1996.

[3] L.Cardelli and A.D.Gordon. Mobile ambients. *Theoretical Computer Science*, **240**(1):177–213, Elsevier, 2000.

[4] Y.Wang. *A Calculus of Real Time Systems. PhD thesis*, Chalmers University of Technology, Göteborg, Sweden,1991.

[5] S.Schneider. *Concurrent and Real-time Systems: The CSP Approach*. Wiley, 2000.

[6] M.Berger, K.Honda. The two-phase commit protocol in an extended π-calculus. *Electronic Notes in Theoretical Computer Science*, **39**:No.1, Elsevier Science, 2000.

[7] P.Degano, J.Loddo, C.Priami. Mobile Processes with Local Clocks. In: *Proceedings of Workshop on Analysis and Verification of Multiple-Agent Languages*, Stockholm 1996, LNCS **1192**:296-319, (M. Dam Ed.), Springer-Verlag.

[8] J.Chen. Study of Real-time Value-passing and Real-time mobile systems. PhD Thesis in Institute of Software. Chinese Academy of Sciences, 2003. Extended abstract can be found at: http://www.univ-orleans.fr/SCIENCES/LIFO/Members/chen/paper/Ex_PhD.ps

[9] H.Lin.Symbolic bisimulation and proof systems for the π-calculus. Technical Report 7/94, University of Sussex, 1994.

[10] H. Lin. A Complete Inference Systems for Weak Bisimulation Equivalences in the π-Calculus. *Information and Computation*. 180(1), 1-29 (2003). Academic Press. An extended abstract of this paper appeared in *TAPSOFT'95*, Aarhus, Denmark, May 1995. LNCS 915:187-201.

Axioms and Algorithms for True Concurrency Equivalences

Michael Kanellos, Mila Majster-Cederbaum

August 26, 2004

Topic

Event Structures and True Concurrency Equivalences

Whenever a specification should distinguish between parallel and sequential execution of actions, the semantics of the specification language that is used for the specification has to take this distinction into account. For process algebras, semantics that support this distinction are called *true concurrency semantics*. A crucial part of the semantics is to define when two specifications are equivalent. Therefore, a true concurrency semantics needs to be equipped with a *true concurrency equivalence notion* that distinguishes specifications also by their degree of parallelism. An important true concurrency semantics for process algebras are event structures. In [GLGo], [Fe] and [GW] we find the definitions of many true concurrency equivalences for event structures (and therefore for process algebra terms via the semantical meaning function). Furthermore, in these papers it has been shown that there are three true concurrency equivalences that are particularly well-suited because they are preserved under all relevant operators and under action refinement: *pomset trace equivalence*, *causal testing equivalence* and *history preserving bisimulation equivalence*. As the names already suggest, pomset trace equivalence is a trace-style linear-time equivalence, causal testing equivalence is a testing equivalence and history preserving bisimulation equivalence is a branching-time equivalence. All three equivalences are strictly finer than their interleaving counterpart (but causal testing equivalence and interleaving bisimulation equivalence are incomparable). For the hierarchy of the true concurrency equivalences see [Fe] and [GW].

Axiomatizing True Concurrency Equivalences

Two properties of equivalences for processes are in particular interesting.
The first is if there is a finite axiomatization that characterizes the equivalence. If so, this can be used in theorem proving.
The other is if the equivalence can be characterized by some suitable logic, i. e. P is eqivalent to Q iff they satisfy the same set of formulas. We are dealing

with the first property.
For some interleaving equivalences complete axiomatizations were given. See for example [HM] or [Gl1]. These describe equivalences algebraically and yield a set of transformation rules that allow transforming process definitions into each other if and only if they are equivalent. We intend to axiomatize true concurrency equivalences. We start with a process algebra containing prefix, choice, parallel composition, variables and a recursion operator. As event structures are much more expressive than this process algebra, we first define syntactically the set of event structures and mappings between event structures that occur as semantics of these process algebra terms. This set of event structures is called the set of *relevant* event structures.
We give a complete axiomatization for pomset trace equivalence for finite relevant event structures.
We also give a set of axioms for history preserving equivalence, *weak history preserving bisimulation equivalence* (for the definition see [GLGo], [Fe]), causal testing equivalence and *weak causal testing equivalence* (for the definition see [GW]).
We currently investigate the completeness of these sets of axioms (this is work in progress) for finite relevant event structures.
We are looking for complete axiomatizations for the general case, which contains the relevant infinite event structures (this is work in progress).

Algorithmic Treatment of Finite Relevant Event Structures

We investigate efficient algorithms to decide equivalence for finite relevant event structures. We give a polynomial algorithm for deciding pomset trace equivalence that makes use of the complete axiomatization for the finite case.

Algorithmic Treatment of Infinite Relevant Event Structures

Process algebra terms that contain at least one recursion operator always yield relevant infinite event structures as meaning. Therefore the question for checking the equivalence of relevant infinite event structures arises. We give a semi-decision criterion for the equivalence of relevant infinite event structures. For all three equivalences (history preserving bisimulation, causal testing, pomset trace equivalence) we can give a proof for the axiom that yields this criterion.
The proof makes use of the facts that in our setting causal testing equivalence is preserved under action refinement and that pomset trace equivalence and history preserving bisimulation equivalence are congruences for action refinement.

References

[GLGo] *Rob van Glabbeek, Ursula Goltz: Refinement of Actions ond Equivalence Notions for Concurrent Systems; Acta Informatica 37 (2001); p.229 - 327.*

[Fe] *Harald Fecher: A Completed Hierarchy of True Concurrent Equivalences; Information Processing Letters 89 (2004).*

[GW] *Ursula Goltz, Heike Wehrheim: Causal Testing; Hildesheimer Informatik-Berichte 5/96; can be obtained at University of Hildesheim, Germany.*

[HM] *Matthew Hennessy, Robin Milner: Algebraic Laws for nondeterminism and Concurrency; Journal of the ACM 32 (1985); p.137 - 161.*

[Gl1] *R.-J. van Glabbeek; A complete axiomatization for branching bisimulation congruence of finite-state behaviours; LNCS 711 (1993), Springer-Verlag; p. 473-484.*

Slicing CSP-OZ Specifications*

Ingo Brückner
Department für Informatik, Universität Oldenburg
26111 Oldenburg, Germany
Fax: +49-441-798-2965
ingo.brueckner@informatik.uni-oldenburg.de

6th September 2004

1 Introduction

The combination of the two well known formal specification techniques CSP [Hoa78, Hoa85] for specification of behavioural aspects of systems and Object-Z (OZ) [Smi00, Spi92, WD96] for specification of data aspects of systems into the specification language CSP-OZ [Fis97] has already been subject of intense research and is currently further extended by an integration with Duration Calculus (DC) [HHF+94, HZ97, ZHR91] for specification of real time aspects of systems. This leads to the formalism of CSP-OZ-DC [HO02] which gives rich possibilities to specify any aspect of complex systems with a suitable formalism in an overall coherent way.

An important challenge, especially, when trying to automatically or semi-automatically analyse such system specifications is their inherent complexity which quickly goes beyond the scope of current analysis techniques such as model checking – in spite of the amazing progress that has recently been done in this research area.

In order to tackle this problem on a different level we propose the application of a technique that is already very long and well known in the area of program analysis, namely applying a kind of 'program slicing' [Wei81, Tip95, HDZ00] to CSP-OZ-DC specifications. The basic idea is to reduce a given specification by eliminating some of its components in such a way that its semantics remains unchanged w.r.t. a given property under consideration (the slicing criterion).

This reduction is based on a preceding analysis of the specification that is very similar to well known program analysis techniques [ASU97, Muc00, NNH99]. More specifically, it includes the construction of a specification dependence graph which comprises – similar to a program dependence graph [HRB90] – all relevant kinds of dependences that are present in the specification such as control or data dependences between specification elements.

Our current work is a first step towards the goal of slicing CSP-OZ-DC specifications which is yet restricted to an application to its untimed predecessor CSP-OZ and accordingly untimed properties that are expressed in an untimed DC variant called State/Event Duration Calculus. From this starting point we develop a slicing algorithm that we show to be correct in the initially claimed sense, i.e. from the given property's point of view the resulting reduced specification exhibits the same behaviour as the original specification does.

2 Specifying with CSP-OZ

We will introduce CSP-OZ by specifying an air condition system as an example. The system consists of several components which communicate with each other. In the example specification we will focus on the actual air condition component. A fragment of its CSP-OZ specification is depicted in figure 1.

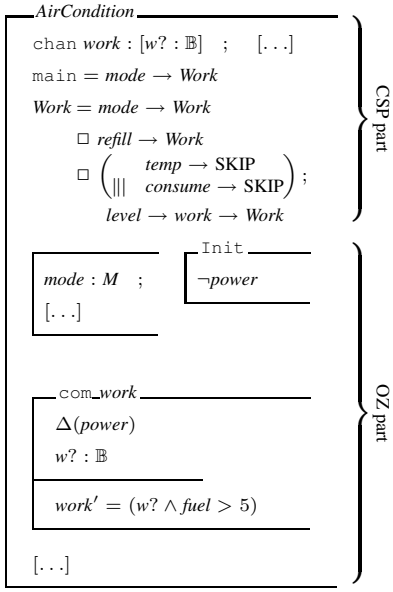


Figure 1: Air condition CSP-OZ specification fragment

The CSP part of the specification defines the air condition's behaviour. This is done by introducing a set of channels that define the air condition's means of communicating with

*This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See www.avacs.org for more information.

its surrounding components. How this communication looks like is defined in several CSP processes that represent different communications the controller can engage in. These may be communications together with or independent of the surrounding components.

The system's state space, its initial configuration and operations on the state space are then defined in the specification's Object-Z part by so called schemas. Each schema consists of two parts: The upper part may contain a list of variables that are defined in the lower part and a list of channels that are used in the lower part for incoming or outgoing communications. The lower part can contain preconditions which determine wether the associated operation is enabled or not and it can contain equations that describe the effect that the associated operation has on the state space. Schemas can be connected to communications in the CSP part but they can also be independent from CSP communications, i.e. the associated operations are then possible at any time.

3 Constructing the Specification Dependence Graph

Following the usual approaches in program slicing [HRB90], a necessary precondition is to analyse the given specification w.r.t. dependences that it contains. The underlying idea is to be afterwards able to backtrack any specification element that might directly or indirectly be relevant for components of the property which serves as the slicing criterion.

Control Flow Graph The first step in constructing the specification dependence graph is to determine the specification's control flow graph which is defined inductively over the structure of the specification's CSP part. Nodes of the control flow graph represent single communications, CSP operators or process calls while edges of the control flow graph represent the possible control flow between the nodes they connect.

Control Dependences Based on the analysis of control dependences between separate specification elements, groups of control flow graph nodes are accumulated into basic blocks, i.e. groups of nodes which are not control dependent on each other. Accordingly, control flow graph edges inside such basic blocks are removed at this point and edges between basic blocks are now regarded as representing control dependences in the sense that the originating node controls the target node.

Data Dependences The last step in constructing the specification dependence graph consists in analysing the specification w.r.t. data dependences between control flow graph nodes, i.e. dependences due to the definition of a variable in one node and the subsequent reference of the same variable in a different node. Two basic kinds of data dependences are considered: First, direct data dependence which occurs between two nodes that are directly connected by a path in the control flow graph from the first to the second node without an intermediate node with an intervening definition. Second, interference data dependence is considered which occurs between two nodes that are located in different branches of a parallel interleaving CSP operator. Data dependence edges can connect different nodes inside the same basic blocks as well as nodes that are located in different basic blocks.

Specification Dependence Graph The specification dependence graph finally comprises all nodes that were introduced during the control flow graph construction and all edges determined during the control dependence analysis and the data dependence analysis.

4 Slicing

When the previously described preparations have been performed, the actual slicing of the specification is easily done: Initially, a set of marked nodes in the specification dependence graph is determined by analysing which node has direct influence on the property that constitutes the slicing criterion. Direct influence is present if a node's associated schema name appears directly in the property or if a node's associated schema contains a definition of a variable which appears in the property.

After this initialisation, the set of marked nodes is repeatedly augmented with yet unmarked nodes that are the origin of edges which lead to already marked nodes. This backtracking is performed until a fixpoint is reached and the set of marked nodes can not be increased any more by application of this rule. At this point only some further nodes are added to the set of marked nodes. These further nodes have certain types and are located in the same basic block as already marked nodes. They are so called structure nodes which are needed to maintain the wellformedness of the specification slice.

This slice is finally derived from the original specification based on the set of marked nodes in the specification dependence graph: All specification elements that correspond to unmarked nodes can safely be removed from the specification. A last step consists in removing all variables from the specification's state space that are not used inside any schema any more.

In order to formalise this slicing algorithm, as it is informally described here, we first define a formal seman-

tics of CSP-OZ specifications based on a variant of labeled Kripke structures (LKS) [CCO+04]. This is done separately for the specification's CSP and for its OZ part which are then combined by parallel composition in order to give the semantics of the full specification.

Further, we need to formalise the properties which serve as slicing criteria. To this end we use State/Event Duration Calculus (SE-DC), a discrete and untimed variant of DC. With SE-DC formulas we can express properties of CSP-OZ specifications as well as properties of the underlying LKS semantics.

Finally, the formalisation of the actual slicing corresponds directly to the informal description given here: Similar to the way the specification slice is derived from the original specification by removing certain specification elements, the specification slice's LKS is related to the original specification's LKS in that certain states and transitions are missing that correspond to the removed specification elements.

5 Slicing Correctness

The presented slicing approach can be regarded as correct if the following holds: The original specification fulfils a property if and only if also the specification slice w.r.t. to this property fulfils this property.

In the correctness proof, which at this point is still work in progress, we plan to apply two steps: First, a definition of stuttering equivalence [Lam83] of LKSs w.r.t. a set of atomic propositions and events is given and it is shown that the presented slicing algorithm guarantees stuttering equivalence between the original specification's LKS and the specification slice's LKS w.r.t. to a set of atomic propositions and events derived from the property that served as a slicing criterion. In a second step it is shown that this kind of stuttering equivalence between two LKSs is equivalent to the notion of correctness that we stated above.

6 Conclusion

This contribution proposes the application of the well known technique of program slicing in a different area, namely in the specification of behavioural and data aspects of complex systems with CSP-OZ. It is shown which adaptations are needed when transferring the existing approaches of program analysis to the new application field of specification analysis.

Furthermore the correctness of the proposed algorithm is shown based on a formally defined semantics of the specification language and an appropriate logic for expressing specification properties.

Future steps will include the extension of the approach to timed specifications (CSP-OZ-DC), and tool support for automatically performing the slicing of such specifications. The latter will definitely be needed when dealing with larger case studies that are foreseen in the project AVACS [AVA04] which forms the overall context of this work.

References

[ASU97] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley, 1997.

[AVA04] AVACS. Transregional collaborative research center 14 avacs: Automatic verifi cation and analysis of complex systems. 2004.

[CCO+04] Sagar Chaki, Edmund M. Clarke, Joel Ouaknine, Natasha Sharygina, and Nishant Sinha. State-event-based software model-checking. In *Integrated Formal Methods: 4th International Conference, IFM 2004, Canterbury, UK*, pages 128–147. Springer, April 2004.

[Fis97] C. Fischer. Csp-oz: A combination of object-z and csp. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)*, volume 2, pages 423–438. Chapman & Hall, 1997.

[HDZ00] John Hatcliff, Matthew B. Dwyer, and Hongjun Zheng. Slicing software for model construction. *Higher-Order and Symbolic Computation*, 13(4):315–353, 2000.

[HHF+94] J. He, C.A.R. Hoare, M. Fr¨anzle, M. M¨uller-Olm, E.-R. Olderog, M. Schenke, M.R. Hansen, A.P. Ravn, and H. Rischel. Provably correct systems. In *Formal Techniques in Real-Time and Fault Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 288–335. Springer, 1994.

[HO02] J. Hoenicke and E.-R. Olderog. Csp-oz-dc: A combination of specifi cation techniques for processes, data and time. *Nordic Journal of Computing*, 9(4):301–334, 2002.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *CACM*, 21:666–677, 1978.

[Hoa85] C.A.R. Hoare. *Communicating sequential processes*. Prentice Hall, 1985.

[HRB90] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[HZ97] M.R. Hansen and C. Zhou. Duration calculus: Logical foundations. *Formal Aspects of Computing*, 9:283–330, 1997.

[Lam83] L. Lamport. What good is temporal logic. *Information Processing*, 83:657–668, 1983.

[Muc00] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 2000.

[NNH99] Hanne Riis Nielson, Flemming Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[Smi00] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publisher, 2000.

[Spi92] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall International Series in Computer Science, 2nd Edition, 1992.

[Tip95] Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[WD96] J. Woodcock and J. Davies. *Using Z – Specification, Refinement, and Proof*. Prentice-Hall, 1996.

[Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[ZHR91] C. Zhou, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.

Generic implementations of process calculi in Isabelle

Jesper Bengtson

7th September 2004

Abstract

Many process calculi have several aspects in common, such as parallel composition, name passing and notions of bisimilarity. Our goal is to create a generic library for these calculi in the automatic theorem prover Isabelle, to verify extensions of existing calculi as well as completely new ones.

1 Introduction

There are several tools available to perform verification on process calculi e.g, *The Concurrency Workbench* [2] for CCS and *The Mobility Workbench* [8] for the π-calculus. These can perform preset tasks such as model-checking or bisimulation-checking of agents. They are fast and automatic as long as the agents they are working on have a finite state space.
Using theorem provers, like Isabelle [6], it is possible to model agents with infinite state space. Moreover, it is possible to perform proofs on the actual calculi themselves. It must be stressed that these provers are not automatic – Isabelle needs to be "hand held" to a large degree and the two methods would complement each other.
One of the main goals of the project is to make a generic library in Isabelle to reason with different varieties of process calculi – different calculi often have aspects in common, such as parallel composition, name passing and notions of bisimilarity. With a generic library in Isabelle, new operators or even entirely new calculi could be added and verified reusing the old proofs.

2 Modelling CCS in Isabelle

The first step has been to model CCS in Isabelle. The operational semantics (except for relabelling) has been added as have strong and weak bisimilarity and equivalence of agents. Isabelle has been used to prove that strong bisimilarity is a congruence.

2.1 Defining the operational semantics

The operational semantics of CCS has been implemented in Isabelle using an inductively defined set. The set consists of tuples of the form (P, α, P') where P is an agent, α is an action, and P' is an α-derivative of P. Every rule in the operational semantics is used as an inductive rule for the set with the base case of the semantics representing the base case of the set definition. The operational rules with their corresponding set rules can be found in table 1.

2.2 Weakening the transitions

The operational semantics described above is used when defining strong bisimilarity. In order to describe weak bisimilarity and equivalence the transitions must allow for hidden τ-transitions, also called silent actions. The transitions for weak bisimilarity and equivalence are $\stackrel{\hat{\alpha}}{\Longrightarrow}$ and $\stackrel{\alpha}{\Longrightarrow}$ respectively where $\stackrel{\alpha}{\Longrightarrow}$ specifies at least the τ-actions occurring in α and $\stackrel{\hat{\alpha}}{\Longrightarrow}$ specifies nothing about the τ-actions occurring in α [4]. The operational semantics for the two transitions can be found in table 2. They are modelled in Isabelle using inductively defined sets – just as the regular transition system.

2.3 Bisimulation and congruences

Two processes are said to be bisimilar if they can mimic each others actions indefinitely. The bisimulation relations are co-inductive and are extensively

Table 1: The operational semantics for CCS and the construction rules for the inductively defined set. The symmetric versions (of the rules marked with †) have been elided.

	Operational semantics	Isabelle representation
[**Act**]	$\frac{}{\alpha.P \xrightarrow{\alpha} P}$	$(\alpha.P, \alpha, P) \in actsSet$
[**Sum**]	$\frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'}$ †	$(P, \alpha, P') \in actsSet \Longrightarrow (P + Q, \alpha, P') \in actsSet$
[**Com**]	$\frac{P \xrightarrow{\alpha} P'}{P\|Q \xrightarrow{\alpha} P'\|Q}$ †	$(P, \alpha, P') \in actsSet \Longrightarrow (P\|Q, \alpha, P'\|Q) \in actsSet$
[**Com$_3$**]	$\frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\bar{\alpha}} Q'}{P\|Q \xrightarrow{\tau} P'\|Q'}$ †	$[(P, \alpha, P') \in actsSet; (Q, \bar{\alpha}, Q') \in actsSet] \Longrightarrow$ $(P\|Q, \tau, P'\|Q') \in actsSet$
[**Res**]	$\frac{P \xrightarrow{\alpha} P'}{(\nu\beta)P \xrightarrow{\alpha} (\nu\beta)P'}$ $\alpha \notin \{\beta, \bar{\beta}\}$	$[(P, \alpha, P') \in actsSet; \alpha \notin \{\beta, \bar{\beta}\}] \Longrightarrow$ $(\nu(\beta)P, \alpha, \nu(\beta)P') \in actsSet$

Table 2: The operational rules for $\stackrel{\alpha}{\Longrightarrow}$ and $\stackrel{\hat{\alpha}}{\Longrightarrow}$.

$$\stackrel{\alpha}{\Longrightarrow} \equiv \frac{P \xrightarrow{\alpha} P'}{P \stackrel{\alpha}{\Longrightarrow} P'} \quad \frac{P \stackrel{\alpha}{\Longrightarrow} P' \quad P' \stackrel{\tau}{\Longrightarrow} P''}{P \stackrel{\alpha}{\Longrightarrow} P''} \quad \frac{P \stackrel{\tau}{\Longrightarrow} P' \quad P' \stackrel{\alpha}{\Longrightarrow} P''}{P \stackrel{\alpha}{\Longrightarrow} P''}$$

$$\stackrel{\hat{\alpha}}{\Longrightarrow} \equiv \frac{}{P \stackrel{\hat{\alpha}}{\Longrightarrow} P} \; \alpha = \tau \quad \frac{P \stackrel{\alpha}{\Longrightarrow} P'}{P \stackrel{\hat{\alpha}}{\Longrightarrow} P'}$$

covered in [4]. Isabelle supports both inductively and co-inductively defined sets so creating co-inductively defined sets containing the tuples of all bisimilar agents comes very natural. The way they are represented can be found in table 3.

The proofs that have been done so far are that strong bisimulation is a congruence – that is an equivalence relation and substitutive under all combinators. We are also one transitivity proof short of proving that equivalence is a congruence. Since bisimulation is a co-inductive definition the proofs have been made by reasoning about the greatest fixed point – adding all possible derivatives of a combinator to the bisimulation relation should preserve the greatest fixed point. Weak bisimulation is not a congruence as the **Sum** rule breaks this premise.

2.4 Future work

The goal of this project is to make a generic library for reasoning about process calculi in Isabelle. What most notably is lacking at the moment is name passing. The next step will be to extend the library to include the π-calculus [5], a calculus for mobile processes which supports the passing of names across links. Representing the π-calculus becomes problematic as α-equivalence has to be taken into account. Some work has already been done in this area by Röckl and Hirschkoff [7] and collaboration with them is underway. We are currently

Table 3: The co-inductive definition of strong bisimulation $\sim$, weak bisimulation $\approx$ and equivalence $=$.

$$\begin{array}{rcl} \sim & \equiv & [\forall \alpha Q'.Q \xrightarrow{\alpha} Q' \longrightarrow (\exists P'.P \xrightarrow{\alpha} P' P' \sim Q'); \\ & & \forall \alpha P' P \xrightarrow{\alpha} P' \longrightarrow (\exists Q'.Q \xrightarrow{\alpha} Q' P' \sim Q')] \Longrightarrow P \sim Q \\ \\ \approx & \equiv & [\forall \alpha Q'.Q \xrightarrow{\alpha} Q' \longrightarrow (\exists P'.P \stackrel{\hat{\alpha}}{\Longrightarrow} P' P' \approx Q'); \\ & & \forall \alpha P' P \xrightarrow{\alpha} P' \longrightarrow (\exists Q'.Q \stackrel{\hat{\alpha}}{\Longrightarrow} Q' P' \approx Q')] \Longrightarrow P \approx Q \\ \\ = & \equiv & [\forall \alpha Q'.Q \xrightarrow{\alpha} Q' \longrightarrow (\exists P'.P \stackrel{\alpha}{\Longrightarrow} P' P' \approx Q'); \\ & & \forall \alpha P' P \xrightarrow{\alpha} P' \longrightarrow (\exists Q'.Q \stackrel{\alpha}{\Longrightarrow} Q' P' \approx Q')] \Longrightarrow P = Q \end{array}$$

looking at FM-sets by Gabbay [3] which looks promising. Care must also be taken not to make the library to specialised. A few attempts have already been made to add operators from CSP [1] to the existing implementation with good results.

References

[1] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.

[2] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench: a semantics-based tool for the verification of concurrent systems. *ACM Trans. Program. Lang. Syst.*, 15(1):36–72, 1993.

[3] M. J. Gabbay. The π-calculus in FM. In Fairouz Kamareddine, editor, *Thirty-five years of Automath*. Kluwer, 2003.

[4] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., 1989.

[5] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i/ii. *Inf. Comput.*, 100(1):1–77, 1992.

[6] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, 2002.

[7] C. Röckl and D. Hirschkoff. A fully adequate shallow embedding of the π-calculus in Isabelle/HOL with mechanized syntax analysis. *Journal of Functional Programming*, 2003.

[8] Björn Victor and Faron Moller. The Mobility Workbench — a tool for the π-calculus. In David Dill, editor, *CAV'94: Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 428–440. Springer-Verlag, 1994.

Inference of Timed Transition Systems

Olga Grinchtein [1] Bengt Jonsson [2] Martin Leucker [3]

IT Department
Uppsala University
Uppsala, Sweden

Abstract

We extend Angluin's algorithm for on-line learning of regular languages to the setting of *timed transition systems*. More specifically, we describe a procedure for inferring systems that can be described by *event-recording automata* by asking a sequence of membership queries (does the system accept a given timed word?) and equivalence queries (is a hypothesized description equivalent to the correct one?). In the inferred description, states are identified by sequences of symbols together with timing information. The number of membership queries is polynomially in the region graph and in the biggest constant of the automaton to learn.

[1] Email: Olga.Grinchtein@it.uu.se
[2] Email: Bengt.Jonsson@it.uu.se
[3] Email: Martin.Leucker@it.uu.se
This author is supported by the European Research Training Network "Games".

Comparative Linear-Time Semantics on Action-Labelled Continuous-Time Markov Chains

Verena Wolf[†], Christel Baier[‡]

[†] *University of Mannheim, D-68131 Mannheim, Germany, vwolf@pi2.informatik.uni-mannheim.de*

[‡] *University of Bonn, Römerstraße 164, D-53117 Bonn, Germany, baier@informatik.uni-bonn.de*

Extended Abstract: Comparative linear-time semantics like (maximal/finite) trace and readiness/failure equivalence have been widely considered for labelled transition systems (for an overview see [vG90]) and also in the (discrete-time) probabilistic setting [DTH92, Seg95]. We extend these semantics to the stochastic setting by considering them for *action-labelled continuous-time Markov chains* (aCTMCs), the underlying model of stochastic process algebras [BG96, Hil96, NGR92]. Similar to probabilistic processes, transitions occur with a certain probability. Additionally, each state has an associated *residence time*[1] given by an exponentially distributed random variable. ACTMCs can also been seen as an action-labelled extension of continuous-time Markov chains, a standard model used in performance analysis [Ste95].

We present several equivalence relations on aCTMCs that have, to the best of our knowledge, not yet been defined and analysed in this way. Although in many cases equivalences based on the linear-time structure provide the best abstraction for verification purposes, most work on stochastic extensions of labelled transition systems is restricted to equivalences based on the branching-time structure of a process (an overview is given in [CBW03]). We extend the *testing scenario* of [vG90, SV03] to the stochastic setting to motivate our semantics by considering the following experiment:

Intuitively, two aCTMCs are considered as "equal" if they can not be distinguished by observing the sequence of performed actions (*trace*) in infinitely many runs. Assume that an aCTMC M is modelled as a black box (*stochastic trace machine*, illustrated in Fig. 1) with two information displays and one button such that

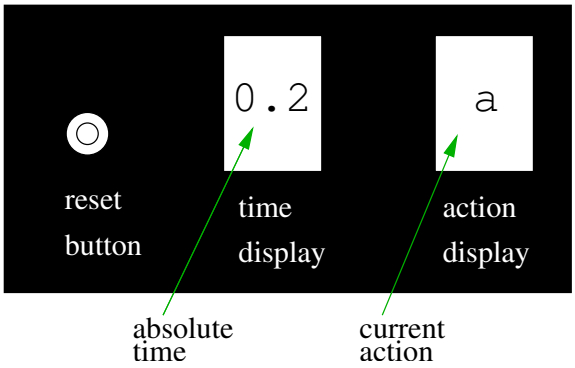


Figure 1: Stochastic Trace Machine.

[1]The residence or sojourn time of a state is the time the process spends in this state.

- the first display shows the action currently performed by M,
- the second display shows the absolute time since the beginning of M's current run,
- the button resets the process.

The information on the displays (*observation*) is recorded by an external observer and at an arbitrary time instant she decides to push the reset button to observe another run. After infinitely many runs the probability of each possible observation can be calculated: Let O be the set of all possible observations and $Pr^M(o)$ the probability of observing $o \in O$ during a run of M. For aCTMCs M, M' define

$$M \sim M' \iff \forall o \in O : \; Pr^M(o) = Pr^{M'}(o).$$

The testing scenario described above motivates several equivalence relations on aCTMCs by considering slight variants of the information given by the time display:

1. Assume the observer records the values on the time display when an action is carried out. The set of observation $O = (Act \times \mathbb{R}_{\geq 0})^*$ and $Pr^M(a_0, t_0, a_1, t_1, \ldots, a_n, t_n)$ is the probability to observe the action sequence $a_0\ a_1 \ldots a_n$ where a_0 is performed in $[0, t_0]$ and a_i is performed within $x \leq t_i$ time units after the occurrence of a_{i-1}, $1 \leq i \leq n$ [2].

2. Suppose that when the reset button is pushed the time display shows the time instant where the last action of the trace was carried out by the process and no further information is given by the time display. Then $O = (Act)^* \times \mathbb{R}_{\geq 0}$ and $Pr^M(a_0, a_1, \ldots, a_n, t)$ is the probability that the trace $a_0, a_1, \ldots, a_n$ is performed within t time units.

3. Consider slight variants[3] of $Pr^M(o)$ for the cases 1./2.: (i) In case 1 the observer records the information on the time display at some arbitrary time instant between the occurrence of two successive actions. So we add an additional condition on the duration x between the occurrence of two successive actions a_{i-1}, a_i (beside the condition $x \leq t_i$) by requiring that no further action is performed within the remaining $t_i - x$ time units, i.e. the process remains in the state reached via a_i at least $t_i - x$ time units. (ii) Consider a similar variant of 2. The time display shows the absolute time when the reset button is pushed. Let $Pr^M(a_0, a_1, \ldots, a_n, t)$ be the probability of performing $a_0, a_1, \ldots, a_n$ within $x \leq t$ time units and remaining at least $t - x$ time units in the state reached via a_n.

4. Consider the stochastic extension of *failure and readiness equivalence* [DTH92, vG90].

Relationships between semantics that identify processes with behaviour "equal" on a certain level of abstraction turned out to be helpful for verification purposes. Hence, we examine if one equivalence relation makes (strictly) more identifications (is "finer") on aCTMCs than another. Our contribution consists of the following three results:

[2] The probability of performing a_i exactly t_i time units after a_{i-1} equals zero, so the cumulative probability is considered similar to the operators in the continuous stochastic logic describing transient properties [BHHK00]

[3] This definition of $Pr^M(o)$ is similar to the definition of transition probabilities widely used in performance analysis [Ste95].

- Case 1. is finer than case 2.,
- Case 1. is equal to case 3.(i) and 2. is equal to 3.(ii).
- Failure and readiness equivalence coincide in the stochastic setting (as in the discrete-time case but opposed to the nonprobabilistic case), but finer than case 1. (2. resp.).

References

[BG96] M. Bernardo and R. Gorrieri. Extended markovian process algebra. In *Proc. CONCUR 1996*, number 1119 in LNCS, pages 315–330. Springer, 1996.

[BHHK00] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. Model cheching continuous-time markov chains by transient analysis. In *Proc. CAV 2000*, volume 1855 of *LNCS*, pages 358–372, 2000.

[CBW03] Joost-Pieter Katoen Christel Baier, Holger Hermanns and Verena Wolf. Comparative branching time semantics for markov chains. In *Proc. CONCUR 2003*, number 2761 in LNCS, pages 492–507. Springer, 2003.

[DTH92] L. Tian D. T. Huynh. On some equivalence relations for probabilistic processes. *Fundamenta Informaticae*, 17(3):211–234, 1992.

[Hil96] J. Hillston. *A compositional approach to performance modelling.* Cambridge University Press, 1996.

[NGR92] U. Herzog N. Gotz and M. Rettelbach. Tipp – a language for timed processes and performance evaluation. report Technical Report 4/92, IMMD VII, University of Erlangen-Nurnberg, 1992.

[Seg95] R. Segala. Modeling and verification of randomized distributed real-time systems. report PhD thesis, Technical Report MIT/LCS/TR-676, Massachusetts Institute of Technology, 1995.

[Ste95] W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains.* Princeton University Press, 1995.

[SV03] M.I.A. Stoelinga and F.W. Vaandrager. A testing scenario for probabilistic automata. In *Proc. ICALP 2003*, volume 2719 of *LNCS*, pages 407–418. Springer, 2003. Also, Technical Rapport NIII-R0307, Nijmegen Institute of Computing and Information Sciences, University of Nijmegen, 2003.

[vG90] R. J. van Glabbeek. The linear time-branching time spectrum. In *Proceedings of the Theories of Concurrency: Unification and Extension*, pages 278–297. Springer-Verlag, 1990.

A Robust Interpretation of Duration Calculus

Martin Fränzle* and Michael R. Hansen**

Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, Bldg. 322
DK-2800 Kgs. Lyngby, Denmark

As embedded systems become more and more complex, early availability of unambiguous specification of their intended behaviour has become an important factor for quality and timely delivery. Consequently, the quest for automatic analysis methods for specifications arises. This quest becomes even more pronounced if specifications are to be formal, because formal specifications are often found to be particularly hard to write and maintain, such that decision procedures for entailment between specifications, satisfiability of specifications, etc., may be extremely helpful in their design process. The price to be paid for such procedures is, however, a firmly constrained expressiveness of the specification formalisms: one has to sacrifice all elements that could give rise to undecidability.

However, the logically motivated notions of entailment between specifications, satisfiability of specifications, etc., have often been criticized from an engineering standpoint, as their validity or invalidity may well depend on the *exact* values of certain constants (e.g., the exact length of a steering rod relative to the exact distance of two joints), while any technical realization of these constants can only be approximate. In system design, the role of any decision problem prone to changing its truth value under arbitrarily small variations of constants may be considered questionable. Based on this insight, research has in recent years addressed more "robust" notions of property satisfaction, where a property is considered to be "robustly (in-)valid" iff it does not change its validity under small variation of constants and/or values of variables [GHJ97,Pur98,Frä99,AB01,Frä01,Rat02a,Rat02b]. The ultimate hope is that, besides being more relevant to engineering problems, such robust notions enhance decidability as, e.g., existence of non-computable reals cannot influence their validity.

With respect to embedded system design, such robust properties have by now mainly been investigated in the automata-based modeling context. Starting with Gupta's, Henzinger's, and Jagadeesan's [GHJ97] as well as Puri's [Pur98] investigation of timed automata, the idea has been to exploit topological properties of systems in order to obtain robust answers. Asarin and Bouajjani [AB01] have applied this approach to reach set computation of, a.o., hybrid automata and Turing machines. Fränzle introduced a variant thereof in [Frä99] by applying the concept to decision problems about hybrid automata instead of reach-set computation, e.g. invariance of a first-order property over hybrid states [Frä99]

* Email: mf@imm.dtu.dk; Phone: +45-4525 7512; Fax: +45-4593 0074.
** Email: mrh@imm.dtu.dk; Phone: +45-4525 3727; Fax: +45-4593 0074.

or progress [Frä01], thereby obtaining decision procedures that succeed in all robust cases, even such which are undecidable wrt. non-robust notions of property satisfaction.

Independently, constraint solving technology for numerical constraints over the real numbers was developed that has perfectly corresponding properties: one can solve otherwise undecidable constraints (containing functions over the real numbers other than polynomials) , provided they are robust, in the sense that their solvability does not change under small perturbations of the constants the constraints contain [Rat02a,Rat02b,Rat02c]. Even in cases where constraints are decidable, robust constraints can generally be solved much more efficiently.

In this talk, we unite above two lines of research by addressing logical models of embedded systems. We provide a robust interpretation of a very expressive metric-time temporal logic, Duration Calculus (DC) [ZHR91,ZH04], and show its equivalence to a multi-valued interpretation that uses the real numbers as semantic domain and assigns Lipschitz-continuous interpretations to all operators of DC.

We define a formula ϕ to be *robustly valid* iff not only ϕ itself is valid (in the classical sense of DC), but there additionally is a real number $\delta > 0$ such that all formulae ϕ' that differ from ϕ only in a change of constants of up to $\pm\delta$ are valid (again in the classical sense of DC). The fundamental idea of the *multi-valued semantics*, on the other hand, is to assign to each (sub-)formula ϕ and each valuation σ of its free variables a real number $\mathcal{MV}[\![\phi]\!](\sigma)$, where

- the sign of $\mathcal{MV}[\![\phi]\!](\sigma)$ signifies whether ϕ is satisfied by σ or not, and
- the magnitude of $\mathcal{MV}[\![\phi]\!](\sigma)$ corresponds to the maximum permissible change of the constants occurring in ϕ that keeps ϕ's satisfaction properties over σ unchanged.

I.e., if $|\mathcal{MV}[\![\phi]\!](\sigma)| = c$ then all formulae ϕ' that differ from ϕ only in a change of constants of up to $\pm c$ correspond to ϕ wrt. σ in so far as either both ϕ and ϕ' are satisfied by σ, or both are not. Intuitively, $|\mathcal{MV}[\![\phi]\!](\sigma)|$ is the *robustness margin* of ϕ wrt. σ, such that the multi-valued semantics and the notion of robust validity are in direct correspondence.

An analysis of $\mathcal{MV}[\![\phi]\!]$ reveals that the multi-valued semantics assigns Lipschitz-continuous interpretations to all operators of DC. This Lipschitz-continuity provides a handle for a plethora of approximability results. I.e., unlike the classical two-valued interpretation, truth values in the multi-valued interpretation can be approximated in various ways, some of which are algorithmic. In particular, there is a discrete-time approximation of the multi-valued dense-time interpretation of a formula. As interesting subsets of DC are decidable over discrete time [Han94], yet undecidable over dense time [ZHS93,ZH04,Frä04], this can provide novel approaches towards tool support for dense-time DC.

References

[AB01] E. Asarin and A. Bouajjani. Perturbed turing machines and hybrid systems. In *Proceedings of the Sixteenth Annual IEEE Symposium on Logic in*

Computer Science (LICS 2001). IEEE, 2001.

[Frä99] M. Fränzle. Analysis of hybrid systems: An ounce of realism can save an infinity of states. In J. Flum and M. Rodríguez-Artalejo, eds., *Computer Science Logic (CSL'99)*, volume 1683 of *Lecture Notes in Computer Science*, pages 126–140. Springer-Verlag, 1999.

[Frä01] ———. What will be eventually true of polynomial hybrid automata. In N. Kobayashi and B. C. Pierce, eds., *Theoretical Aspects of Computer Software (TACS 2001)*, volume 2215 of *Lecture Notes in Computer Science*, pages 340–359. Springer-Verlag, 2001.

[Frä04] ———. Model-checking dense-time duration calculus. *Formal Aspects of Computing*, 16(2):121–139, 2004.

[GHJ97] V. Gupta, T. A. Henzinger, and R. Jagadeesan. Robust timed automata. In O. Maler, ed., *Proceedings of the First International Workshop on Hybrid and Real-Time Systems (HART 97)*, volume 1201 of *Lecture Notes in Computer Science*, pages 331–345. Springer-Verlag, 1997.

[Han94] M. R. Hansen. Model-checking discrete duration calculus. *Formal Aspects of Computing*, 6(6A):826–845, 1994.

[Pur98] A. Puri. Dynamical properties of timed automata. In A. P. Ravn and H. Rischel, eds., *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'98)*, volume 1486 of *Lecture Notes in Computer Science*, pages 210–227. Springer-Verlag, 1998.

[Rat02a] S. Ratschan. Continuous first-order constraint satisfaction. In J. Calmet, B. Benhamou, O. Caprotti, L. Henocque, and V. Sorge, eds., *Artificial Intelligence, Automated Reasoning, and Symbolic Computation*, number 2385 in LNCS, pages 181–195. Springer, 2002.

[Rat02b] ———. Quantified constraints under perturbations. *Journal of Symbolic Computation*, 33(4):493–505, 2002.

[Rat02c] ———. Search heuristics for box decomposition methods. *Journal of Global Optimization*, 24(1):51–60, 2002.

[ZH04] Zhou Chaochen and M. R. Hansen. *Duration Calculus — A Formal Approach to Real-Time Systems*. EATCS monographs on theoretical computer science. Springer-Verlag, 2004.

[ZHR91] Zhou Chaochen, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.

[ZHS93] Zhou Chaochen, M. R. Hansen, and P. Sestoft. Decidability and undecidability results for duration calculus. In P. Enjalbert, A. Finkel, and K. W. Wagner, eds., *Symposium on Theoretical Aspects of Computer Science (STACS 93)*, volume 665 of *Lecture Notes in Computer Science*, pages 58–68. Springer-Verlag, 1993.

Implementing stronger encapsulation for Java-like languages

Ville Leppänen and Sami Mäkelä
University of Turku and TUCS, Finland

Abstract

Encapsulation in object-oriented languages is understood to mean hiding the internal representation and forcing the user to access the maintained data through the public interface. The problem is that via the public interface, one can import and export references to parts of the internal representation. In current OO languages, this problem is not tackled.

One can seek the solution from two directions: (a) Avoiding to provide a reference to an internal part, or (b) making sure that through additional references to the internal representation, one cannot break invariant conditions. Consider a binary search tree and the very natural operation of giving out access to the left subtree. Following (a) would mean that either a copy of the whole left subtree is made, or one cannot have a reference to the left subtree. Neither of the solutions seems acceptable.

We follow the other path, and consider sharing references safely to be desirable. Sharing is a fundamental programming concept in object oriented software design, and one should not try to avoid sharing but rather to control the possible effects caused by sharing. Previously Kniesel and Theisen [1] have considered this problem and studied extending Java with transitive read-only restriction to references. Their solution is static in the sense that compiler can check the correct usage of transitive read-only references at compile time.

We have implemented a much more general extension of protective references and implemented it with respect to the forthcoming Java 5.0. In our extension, one specifies objects to have named abstract properties (following [2]). Methods are specified to have an effect to some of those abstract properties. There is also a way to express whether a method exposes a reference to some internal part. The type system is then extended with expression `immu(<regexpr>) X`, where a regular expression characterizes those abstract properties that are protected from the transitive closure (reached via reference of the base type X).

In our implementation, the immu-expressions can be applied to any type expression, even to type parameters of generic classes. Basically each immu-expression corresponds implicitly to a certain kind of supertype of the base type. Thus, the checking of object integrity, when properly using protective references, reduces to type checking that can be done at compile time.

We have implemented such type type checking mechanism of the modified Java 5.0. With our constructions, it is possible write such a class representing binary search tree that it is safe to give out references to e.g. the left subtree. By safety we mean that via such references

one cannot break the order invariant of the whole binary search tree. Our implementation essentially provides one implementation for the ideas of [3].

References

[1] G. Kniesel and K. Theisen. JAC – Access right based encapsulation for Java. *Software Practice and Experience*, 31(6), pages 555 – 576, 2001.

[2] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*, MIT Press, Cambridge, MA, 1986

[3] H. Hakonen, V. Leppänen, T. Salakoski. Object Integrity while Allowing Aliasing *The 16th IFIP World Computer Congress International Conference on Software: Theory and Practice, ICS'2000*, pages 91 – 96, 2000.

Object Calculus and Self-Application

EXTENDED ABSTRACT

Neil Ghani* ng13@mcs.le.ac.uk
Johan Glimming† glimming@kth.se

September 6, 2004

In this paper we compare two interpretations of Abadi and Cardelli's first-order object calculus [1] (i.e. a typed operational semantics of objects with method update). The two interpretations are split-method interpretation [2, 4] and self-application encoding [6, 3]. The former has well-known support for subtyping, whereas in the second interpretation depth/width subtyping has known problems. However, as we will demonstrate in this paper, the two interpretations are related and a self-application semantics gives us a recursion principle that can be used for programming with objects. To this end, we give wrapper classes for algebraic datatypes, and instantiate the recursion principle for these wrappers.

This paper is based on work reported previously in [5]. However, we also compare self-application to split method interpretation, and give more (Haskell) examples of Freyd's direcursion applied to object types.

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An interpretation of objects and object types. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 396–409, 1996.

[3] Kathleen Fisher, Furio Honsell, and John C. Mitchell. A lambda calculus of objects and method specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.

*Department of Mathematics and Computer Science, University of Leicester, UK
†Department of Numerical Analysis and Computer Science, Stockholm University, Sweden

[4] Full Abstraction for First-Order Objects with Recursive Types and Subtyping. Ramesh viswanathan. In *Proceedings of the Thirteenth Annual IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 1998.

[5] Johan Glimming and Neil Ghani. Difunctorial semantics of object calculus. In *Electronic Notes in Theoretical Computer Science*, volume 3. Elsevier, 2004. To appear.

[6] Samuel N. Kamin. Inheritance in smalltalk-80: a denotational definition. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 80–87. ACM Press, 1988.

A Hoare Logic for Distributed Objects with Asynchronous Method Calls

Johan Dovland, Einar Broch Johnsen, and Olaf Owe

Department of Informatics, University of Oslo
email: {johand,einarj,olaf}@ifi.uio.no

Abstract

Distributed applications such as critical infrastructure are naturally modeled by concurrent objects. However, current object oriented programming languages do not capture the distributed setting equally well, resulting in low-level and error-prone programs. Reasoning about concurrent programs in these languages is at best very challenging.

Intuitive high-level programming constructs are needed to unite object orientation and distribution in a natural way. The Creol language attempts to address this issue. In this paper, we show how a proof system for Creol programs may be derived from Hoare rules for standard sequential programming constructs, capturing the highly non-deterministic behavior of distributed concurrent objects

Object oriented languages may be critisized for their approach to distribution and concurrency. In particular, it is very cumbersome to capture typical distributed communicating systems, such as peer-to-peer file sharing, at a high level of abstraction.

Furthermore, the current model of concurrency found in languages such as Java makes reasoning about code very challenging [1]. Intuitive high-level programming constructs are needed to unite object orientation and distribution in a natural way [2–4, 12]. The Creol language [6, 8] proposes programming constructs for concurrent objects based on *processor release points* and a notion of *asynchronous method calls*. A concurrent object has a dedicated processor for local computations. This allows Creol programs to be highly non-deterministic, reflecting the non-determinism of the distributed system, and very flexible with regard to the order in which computations are preformed. Still, Creol programs are fairly intuitive and easy to understand, avoiding explicit scheduling and low-level signaling mechanisms in the code.

Processor release points are used to influence the implicit internal control flow in the objects. This reduces time spent waiting for replies to method calls in a distributed environment and allows objects to dynamically change between active and reactive behavior (client and server roles). The Creol language has a formal operational semantics defined in rewriting logic [9], in which method calls are reflected by pairs of asynchronous messages [5, 7], allowing message overtaking. The result extends programming with so-called future variables [12]; the caller may continue its computation until the return value of the call is explicitly needed in the code. If the reply has not arrived at this point, the object must wait — unless it can proceed with other pending computations. For this purpose, the future variable programming style is extended in Creol by introducing interleaved method evaluations in objects. This is done by defining potential processor release points by means of inner guards in method bodies.

In this paper, we show how a proof system for the Creol language may be derived from Hoare rules for well-known sequential programming constructs by means of an encoding of

processor release points and asynchronous method calls into the sequential setting. For reasoning purposes, objects are extended with local variables reflecting the non-deterministic interleaving of object activity. The derived rules are sufficient for partial correctness reasoning. The approach extends to parallelism by the representation of finite communication histories as local program variables, and a rule expressing that different local communication histories are compatible [10,11]. The representation of communication in history variables results in a compositional proof system: for reasoning purposes it is sufficient to consider one class at a time. The approach is object oriented in the sense that local program variables are encapsulated and only available for local reasoning. The local history of observable communication provides an abstract view of an object's internal state. External and compositional reasoning is based on the observable behavior of objects. The proof rules establish a set of possible local histories in a bottom-up manner from program code. This set can be used to verify that the program code correctly implements a specification of observable object behavior, following a specification style and refinement relation based on observable behavior [5,7].

References

[1] E. Ábrahám-Mumm, F. S. de Boer, W.-P. de Roever, and M. Steffen. Verification for Java's reentrant multithreading concept. In *International Conference on Foundations of Software Science and Computation Structures (FOSSACS'02)*, volume 2303 of *Lecture Notes in Computer Science*, pages 5–20. Springer-Verlag, Apr. 2002.

[2] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C♯. In B. Magnusson, editor, *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002)*, volume 2374 of *Lecture Notes in Computer Science*, pages 415–440, Berlin, June 2002. Springer-Verlag.

[3] D. Caromel and Y. Roudier. Reactive programming in Eiffel//. In J.-P. Briot, J. M. Geib, and A. Yonezawa, editors, *Proceedings of the Conference on Object-Based Parallel and Distributed Computation*, volume 1107 of *Lecture Notes in Computer Science*, pages 125–147. Springer-Verlag, Berlin, 1996.

[4] G. Cugola and C. Ghezzi. CJava: Introducing concurrent objects in Java. In M. E. Orlowska and R. Zicari, editors, *4th International Conference on Object Oriented Information Systems (OOIS'97)*, pages 504–514, London, 1997. Springer-Verlag.

[5] E. B. Johnsen and O. Owe. A compositional formalism for object viewpoints. In B. Jacobs and A. Rensink, editors, *Proc. 5th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'02)*, pages 45–60. Kluwer Academic Publishers, Mar. 2002.

[6] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. In *Proc. 2nd IEEE International Conference on Software Engineering and Formal Methods (SEFM'04)*. IEEE Computer Society Press, Sept. 2004. To appear.

[7] E. B. Johnsen and O. Owe. Object-oriented specification and open distributed systems. In O. Owe, S. Krogdahl, and T. Lyche, editors, *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, volume 2635 of *Lecture Notes in Computer Science*, pages 137–164. Springer-Verlag, 2004.

[8] E. B. Johnsen, O. Owe, and E. W. Axelsen. A run-time environment for concurrent objects with asynchronous methods calls. In *Proc. 5th International Workshop on Rewriting Logic and its Applications (WRLA'04)*, Electronic Notes in Theoretical Computer Science. Elsevier, Mar. 2004. To appear.

[9] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[10] N. Soundararajan. Axiomatic semantics of communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 6(4):647–662, Oct. 1984.

[11] N. Soundararajan. A proof technique for parallel programs. *Theoretical Computer Science*, 31(1-2):13–29, May 1984.

[12] A. Yonezawa. *ABCL: An Object-Oriented Concurrent System.* Series in Computer Systems. The MIT Press, 1990.

Optimal Scheduling and Priced Timed Automata

Kim G. Larsen

Aalborg University, Denmark

Abstract

Recently timed automata technology has been successfully applied to the modelling and solution of several real time scheduling problem. Compared with traditional approaches – such as LP and MILP – the use of an automata based approach makes it much simpler to model complex logical constraints. Also the solution method – based on reachability analysis and model checking – seem very different (and efficiency-wise incomparable) to traditional approaches.

In order to address optimal scheduling problems the basic timed automata model has been extended with one additional continuous variable – cost – which may grow with different rates in different locations.

In the talk we shall introduce investigate the resulting model Priced Timed Automata. We shall review the first positive results for this model, namely computability of optimal reachability as well as datastructures allowing for efficient computability. Also, we present ongoing work on optimal infinite schedules as well as optimal dynamic scheduling using Priced Timed Automata with controllable as well as uncontrollable transitions. To address the ever-present problem of state-explosion a notion of refinement between Priced Timed Automata is introduced allowing (optimal) scheduling problems too complicated for analysis to be replaced with simpler and more abstract ones.

Abstraction Based Analysis and Arbiter Synthesis: Radar Memory Interface Card Case Study Revisited

Juhan Ernits

Institute of Cybernetics
Akadeemia tee 21, 12618 Tallinn, Estonia

Abstract

The work focuses on the analysis of an example of synchronous systems containing FIFO buffers, registers and memory interconnected by several private and shared busses. The example used in this work is based on a Terma radar system memory interface case study from the IST AMETIST project [1].

More general application domain of this work is targeted at abstraction and analysis methods that enable to extract scheduling problems from an attributed component models. An overview of the component model of the memory interface board is given in Fig. 1. The purpose of the board is to perform computations on the streams A and B using previous values in the streams and previous values of computations on the streams. The requirement for correct operation of the system is that none of the buffers nor registers should neither be starved nor overflowed.

The solution described in this paper differs from the solution described in [2] in the following aspects:

- The solution presented in [2] involves using SMV [3]; the current solution uses Uppaal [4] as the model checker;
- In [2] the schedule for the example is reached using a parameterized model as model checking the full system was infeasible due to state space explosion. In the current approach, the Uppaal model of the component system contains abstractions that enable the schedule to be synthesized for the whole system.

The current solution builds on Uppaal as an explicit state model checker and exploits bit-state hashing as the model checker provided memory consumption reduction technique for finding schedules. Clocks are used for bounding the depth of the search. The main characteristics of the system modelled in terms of Uppaal automata are the following:

- The state of the buffers and registers is represented by integers.
- To model synchronicity, all variables representing buffers and registers are updated on one transition discarding a great deal of interleavings that are irrelevant in this context. There is a separate variable representing the behaviour of the arbiter that controls which register can access memory at a time.
- The granularity of time ticks is aligned with the duration of a transfer on the shared bus. The buffers and registers are updated by relevant multiples of bytes at the beginning of each such abstract tick.

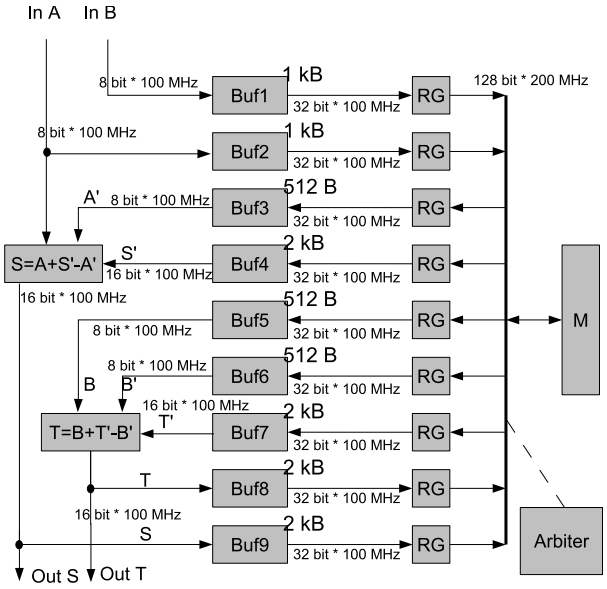


Fig. 1. Radar memory interface board

– The schedule is found using a $E\Diamond$ query and relevant parameters to the Uppaal backend *verifyta*.

The existence of a solution to this problem under further restrictions on resources yields optimisation as a by-product of formal analysis. The contribution of the work is a way to partition the problem into independent subproblems that enable to abstract away details that are irrelevant with respect to the concurrency issue while still guaranteeing correct behaviour in the concrete system.

References

1. Behrmann, G., Bernicot, S., Hune, T., Larsen, K.G., Lecamp, S., Skou, A.: Case study 2: Memory interface for radar system. Deliverable to the IST AMETIST project (2002)
2. Weiss, G.: Optimal Scheduler for a Memory Card. Research report, Weizmann (2002)
3. McMillan, K.L.: The SMV language (1999) Cadence Berkeley Labs.
4. Pettersson, P., Larsen., K.G.: UPPAAL2k. Bulletin of the European Association for Theoretical Computer Science **70** (2000) 40–44

Timed Traces and Strand Spaces

Robin Sharp Michael R. Hansen
Informatics and Mathematical Modelling
Technical University of Denmark
DK-2800 Lyngby, Denmark
{robin,mrh}@imm.dtu.dk

September 2004

Many security protocols only work correctly to provide the degree of security expected by their users if certain temporal constraints on their operation are fulfilled. A typical requirement is that the exchange of messages involved in the protocol must be completed within a given time. If this does not happen, the protocol exchange is considered invalid, and the secure service which the protocol is intended to implement cannot be made available to the user. To analyse and reason about this type of requirement, we need to use timed models which can describe both trusted, concurrently executing agents attempting to execute the protocol, and untrusted intruders trying actively to disturb the execution of the protocol by deleting, modifying or inserting messages.

Paulson [5] has shown that an approach to the analysis of security protocols based on inductively defined traces is effective when the goals to be shown are not time dependent. In [6], Pilegaard et al. reported an attempt at combining Paulson's approach with Interval Logics in order to be able to analyse temporal properties of security protocols. Interval logics with the concept of durations have been shown (see for example [2]) to be a framework which is well-suited to reasoning about the temporal properties of systems with concurrent activities, shared resources and various scheduling disciplines. In [6], these ideas were extended to deal with concurrently executing agents which construct messages, transmit them via a shared network and check the received messages in accordance with given protocols, in the presence of hostile intruders, thus making it possible to analyse active attacks and availability questions.

In this work, we consider how this approach can be based on the formalism of Strand Spaces introduced by Thayer, Herzog and Guttmann [8] instead of traces. In relation to traces, Strand Spaces only consider causal sequences of events in the behaviour of the protocol participants, thus reducing the number of possible event sequences and making it possible to introduce an algebra of events which are permitted according to the protocol. Our intention here is to show how this powerful formalism can be combined with temporal reasoning using an interval logic.

As an example, an Alice-and-Bob specification of the Otway-Rees protocol for authenticated key exchange [4] is shown in Figure 1. This protocol enables two parties, A and B, to convince one another that they know a shared secret, the integer valued key K_{AB}, by exchange of information with a trusted server, S. We use the notation that $A, B, \ldots$ are agents, $N_A, N_B, \ldots$ are nonces, K_{ij} is a key known only to agents i and j, and $\{M\}_K$ represents message M encrypted with key K.

1. $A \to B: \quad M, A, B, \{N_A, M, A, B\}_{K_{AS}}$
2. $B \to S: \quad M, A, B, \{N_A, M, A, B\}_{K_{AS}} \{N_B, M, A, B\}_{K_{BS}}$
3. $S \to B: \quad M, \{N_A, K_{AB}\}_{K_{AS}} \{N_B, K_{AB}\}_{K_{BS}}$
4. $B \to A: \quad M, \{N_A, K_{AB}\}_{K_{AS}}$

Figure 1: The Otway-Rees authentication protocol.

Strand Spaces, introduced by Thayer Fábrega, Herzog and Guttman [8], offer a framework for describing protocol behaviour and proving the correctness of protcols with respect to security specifications. A *strand* is a sequence of events, which can be performed by a legitimate participant in a protocol or by an intruder (in Thayer Fábrega et al.'s notation: a *penetrator*). A *strand space* is a set of strands together with a graph structure describing causal interactions. A portion of a strand space which describes a particular (desired or undesired) execution of a protocol is known as a *bundle*. An example of a bundle showing the desired behaviour of the Otway-Rees protocol is shown in Figure 2. The nodes of the graphs denote events, and the arrows denote causal relationships, with single arrows indicating exchange of messages, and double arrows causal relationships within the strands for the individual participants A, B and S.

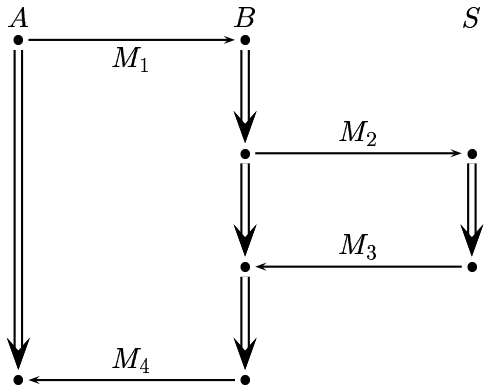


Figure 2: A bundle illustrating the desired behaviour of the Otway-Rees protocol.

In the strand space approach, it is assumed that an intruder posesses a certain set of encryption keys, K_P, and can perform the certain atomic actions. An intruder strand consists of a sequence of such actions. Figure 3 shows a bundle with intruder strands, which illustrates an *undesired* behaviour of the Otway-Rees protocol. The intruder, here denoted P, receives message M_2 which is intended for S, duplicates it and sends it twice to S. The first time, S chooses a new secret key K_{AB} and sends it in M_3 to B as intended, but B's subsequent message M_4 to A gets lost. The second time, S again chooses a new secret key, K'_{AB}, for A and B and attempts to send it to B in M'_3, but it is stolen by P and passed directly to A in message M'_4. The result is that A and B have different views of which secret key to use.

This undesired scenario can be avoided in a timed framework like, for example, that described in [6], by taking the freshness of the nonce created by A seriously. A similar analysis is useful for other protocols whose correctness assumes freshness of nonces, such as the well-known Needham-Schroeder secret key authentication protcols originally presented in [3] but subsequently shown incorrect due to a missing freshness property. In the presentation we

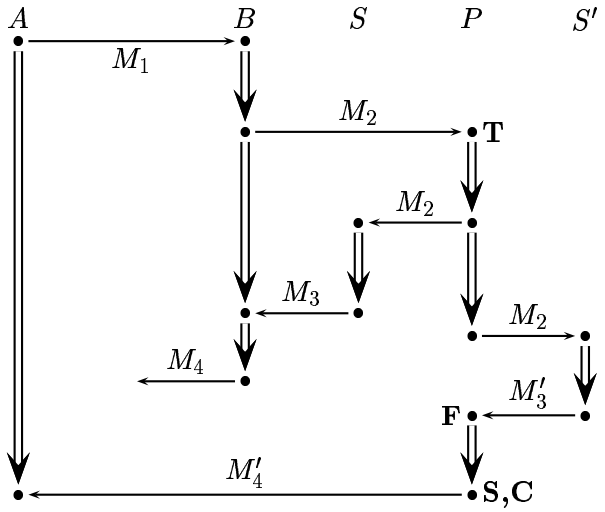


Figure 3: A bundle illustrating an undesired behaviour of the Otway-Rees protocol.

shall establish a connection between that timed framework and strand spaces, in order to provide a useful framework for the analysis of temporal properties of security protocols, and supplements the Strand Spaces approach.

References

[1] C. Meadows. A cost-based framework for analysis of denial-of-service in networks. *Journal of Computer Security*, 9(1/2):143–164, Mar. 2001.

[2] S. Mørk, J. C. Godskesen, M. R. Hansen, and R. Sharp. A timed semantics for SDL. In R. Gotzhein and J. Bredereke, editors, *Formal Description Techniques IX: Theory, application and tools*, pages 295–309. Chapman & Hall, 1996.

[3] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[4] D. Otway and O. Rees. Efficient and timely mutual authentication. *Operating Systems Review*, 21(1):8–10, 1987.

[5] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[6] H. Pilegaard, M. R. Hansen, and R. Sharp. An approach to analyzing availability properties of security protocols. *Nordic Journal of Computing*, 10(4):337–373, 2003.

[7] T. M. Rasmussen. Automated proof support for interval logics. In *LPAR 2001*, volume 2250 of *Lecture Notes in Computer Science*, pages 317–326. Springer-Verlag, 2001.

[8] F. J. Thayer Fábrega, J. C. Herzog and J. D. Guttman Strand Spaces: Proving security protocols correct. *Journal of Computer Security*, 7:191–230, 1999.

[9] C.-F. Yu and V. D. Gligor. A formal specification and verification method for the prevention of denial of service. In *1988 IEEE Symposium on Security and Privacy, Oakland, California*, pages 187–202. IEEE, IEEE Computer Society Press, Apr. 1988.

[10] Zhou Chaochen and M. R. Hansen. *Duration Calculus: A Formal Approach to Real-Time Systems*, EATCS: Monographs in Theoretical Computer Science, Springer, 2004.

Intelligent Sensory Using Compact Data Structures and Bayesian Networks

Jens A. Hansen
Affiliation: CISS, Aalborg University
{alsted@cs.auc.dk}

6th September 2004

Abstract

By using Bayesian networks we give a system description of a behavior and welfare sensory system intended for pig production. We use our knowledge of system to build a dynamical model that results in a simpler update function on the probabilities of the system states. The proposed update function is simpler compared to task of updating all possible potentials in the state space. In order to implement the model, we consider two compact data structures. Both structures are dependent on how the state space evolves and we give our considerations on why we expect the data structures to work.

1 Introduction

Modern pig production is moving towards ever growing production batches using the same number of farmers. This gives the farmer less time to monitor the pigs health and growth status. For several years automation in pig production has been developed and deployed. The current control systems act as climate controllers and is as such unaware of the welfare and behavior of the pigs. A climate control systems uses temperature and humidity to measure the climate and uses air fans and air intake valves as actuators. Furthermore, heating and water spraying mechanisms can also be deployed as actuators in pig barns.

As stated earlier the farmer is still very much involved in the production of the pigs since he has to monitor and control all non-climatic influences in the pig production cycle. This is also called "Human in the (control) loop", and is presenting a problem as the farmer gains more livestock to administer.

To aid the farmer, the notion of a welfare loop is developed. This consists of a sensory system that can produce information on the current behavior and welfare of the pigs. Along with the sensory system a welfare controller is needed such that there can be actuated accordingly to the results of the sensory system. The objective is that the farmer can leave some or all of the welfare control, to the welfare controller and sensory system. This enables the farmer to concentrate of the higher order pig production cycle.

We are concerned about the development of intelligent sensor networks systems such that the welfare loop can developed and closed. The primary goal is to develop an estimate of the pigs behavior given a set of sensory inputs. In the remainder of this abstract we present an outline for representing the system and expanding the system model with the required influence of past events. We then give pointers to which efficient and compact data structures we plan to use in an implementation and why we believe these representations to be feasible.

2 Representing the Welfare Loop

A basic climate and welfare control loop is presented in Figure 1, and represents all components of the system. Each box in the figure denotes a state change. The system uses several variables which represent state configurations. Time is used in a discrete manner and represented with n.

The box denoted A, has two inputs, namely U_n the current actuation and X_n the current state of the barn climate and pig welfare. As an output of this is X_{n+1} the climate state and pig welfare for the next time step. The current state of the pigs welfare and climate is monitored, this process is represented with the box C that has the output variable Y_n, which is the current set of sensor measurements. The sensor

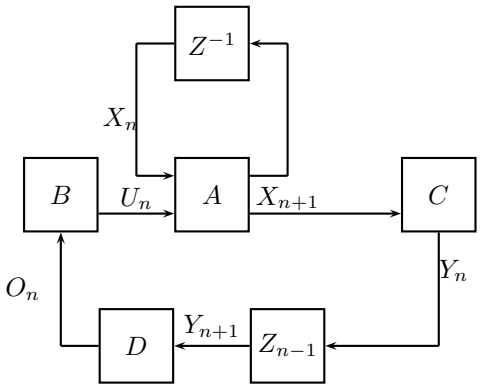


Figure 1: System Blok Diagram

measurements is fed into the controller, represented as box D. The controller starts a new actuation cycle based on the sensor measurements and outputs a set of actuator signals denoted as O_n. The final step in the loop is the actual actuation represented by the box B and the output is the new actuation levels.

Given that we annotate this system with probabilistic dependencies we get the following expressions:

$$\begin{aligned} &P_A(X_n|X_{n-1}, U_n) \\ &P_B(U_n|O_n) \\ &P_C(Y_n|X_n) \\ &P_D(O_n|Y_n) \end{aligned} \tag{1}$$

What can been seen from the equations in Listing (1), is that we have given a description very similar to Bayesian network [1]. Listing (1) is an expression for how we see sets of discrete random variables being related, e.g. could X be $X = (Temperature, Humidity, PigStress)$. Specifying Bayesian networks is a delicate task, since it incorporates specific domain knowledge, e.g. how does the temperature and humidity affect the stress level of the pig? We can formulate this as e.g. $P(PigStress|Temp, Hum)$ or just the simultaneous probability distribution $P(PigStress, Temp, Hum)$. It is most certainly possible to construct Bayesian Networks for climate/welfare loop model, but the correctness of the model has to determined with aid of domain experts.

We still need to account for the dynamic parts of the model, which basically gives us a Partially observable Markov Decision Process [1]. This can gives us several problems of intractability if we have to consider all past state configurations, this is also shown in [1].

3 Towards Embedded Platforms

We have given a description of the welfare loop, even though this description may become troublesome to use, due to growth of the state space. This problem is rather relevant since we consider an embedded and possibly distributed, target platform.

Current techniques used in Bayesian networks for updating probabilities, such as Junction Trees [5], are efficient but can be rather memory and processor consuming. Other classes of algorithms such as anytime algorithms partially solves some of problems, but at the cost of precision. The main issue here, is that we intend to use our knowledge of the system model to build a specific probability updating function for the dynamics of the system.

The simultaneous probability distribution $P(X_n, U_n, O_n, Y_n|\ldots)$ where we say $\ldots$ represent all prior state configurations, describes the probability of all states given the past state configurations. If we can compute this for each time step, we can estimate the current state of the welfare and climate.

First it should be noted that U and O are known i.e. we can say that we have evidence on these variables. Because of this we can state them as vectors where all places are zero except one. With this information at hand we can start manipulating $P(X_n, U_n, O_n, Y_n|\ldots)$ such that we can see which calculations we must perform for each time step. We denote the entire state space as a matrix $\underline{A}$, this represents the distribution $P(X_n, U_n, O_n, Y_n)$ and time step $n = 0$. To calculate the next time step we say that the current probability of all states is the vector P, such that we can calculate P^{n+1} as $P^{n+1} = \underline{A}^{n+1} \cdot P^n$. The last thing needed is to show how we intend to calculate $\underline{A}^{n+1}$, this is shown as Equation (2).

$$\begin{aligned} \underline{A}^{n+1} = \sum_{u_n \in U} P(X_{n+1}|X_n, U_i) \cdot \\ P(U_n|X_n, O_n) \cdot P(O_i|y_i) \cdot P(Y_i|X_i) \end{aligned} \tag{2}$$

This calculation may look daunting, however recall

that several of the variables are know i.e. we have evidence on them. With the marginalization of U the expression is rather small compared to what we started with.

With the system update function described as a vector/matrix multiplication we need to consider how to perform the matrix/vector multiplication efficiently. For this we consider two well known methods for representing matrices efficiently. The methods we consider here are Probabilistic Decision Graphs [4] and Multi Terminal Binary Decision Diagrams [6]. Both of these methods rely on the matrices represented are sparse. It is difficult to be precise about how success full thee two methods will be, but we offer the following thoughts about the system state space.

The largest part of the state space consists of the climate variables and the pig welfare variables. To consider the later of these first. We know that pigs have a group behavior, that is, as a group the pigs normally engage in the same activities at the same time. This could be e.g. feeding activities or sleeping. Even when we consider outliers from this observation, the pigs state space matrix should become reasonably sparse. The only problem is that we expect the pigs to change behavior abruptly, if exposed to sudden stress. We can describe the climate system as a set of differential equations, it is in fact this description the climate controller uses. From this set of equations we can calculate the climates maximum rate of change, which gives us firm ideas about how fast we can move around in the state space. We do not expect abrupt changes in the climate state, so within the maximum change rate of the climate system the rest of the climate stae space should be sparse.

The state space considerations can not be seen as anything but estimates. As with all other BDD like heuristics, test and tuning on real world data is essential.

4 Results

Bayesian Network models has been developed and are currently in the state of being revised through interaction with domain experts from Skov A/S [3]. Most of the connecting probability theory is in place and is ready to be implemented. The Bayesian models we have developed are made with the decision support tool Hugin [2]. We use Hugin to build the starting probabilities of $\underline{A}$, which gives us a practical way for specifying and doing small tests to obtain better correctness of the probabilities used.

Our next goal is to implement, test and evaluate the calculations and data structures described earlier, before we can say anything conclusive about the validity of the ideas presented.

5 Future Work

We have not touched upon how we actually intent do perform decision making based on the probabilities for each time step. Our goal is to include some form of utility support in the structure as well, even though it is the job of the control system to make any decision. The proposed system could also be used to do fault detection and location, exactly how this should be incorporated must be considered further.

There are several other issues that need to be resolved before we can start doing any real world testing of the proposed system. More sophisticated sensor systems are needed, these could be build from vision or sound inputs. We are in the process of developing such systems, but their feasibility are still untested. More over a general architecture for the sensor network is to be developed. We have to consider how we can integrate smoothly into already existing systems and find suitable hardware platforms to deploy the system on.

References

[1] *Bayesian networks and decision graphs*, Springer Verlag, 2001.

[2] *Hugin - decision support tool* *http://www.hugin.com*, 2004.

[3] *Skov A/S* *http://www.skov.dk/*, 2004.

[4] Marius Bozga and Oded Maler, *On the representation of probabilities over structured domains*, Computer Aided Verification, 1999, pp. 261–273.

[5] Steffen L. Lauritzen Finn V. Jensen and Kristian G. Olesen, *Bayesian updating in causal probabilistic networks by local computations*, Computational Statistics Quarterly (1990).

[6] J.C.-Y. Yang M. Fujita, P.C. McGeer, *Multi-terminal binary decision diagrams: An efficient data structure for matrix representation*, Formal Methods in System Design (1997).

Universal Semi-local Election Protocol Using Forward Links[1]

(abstract)

Dobiesław Wróblewski

Institute of Computer Science of PAS,
ul. Ordona 21, 01-237 Warsaw
{wrobldob@ipipan.waw.pl}

Introduction

We consider finite connected undirected graphs as a model for computer networks. The nodes of such a graph represent the computers, while the edges stand for the communication links. The nodes may have unique labels called identities (ordered or not). Otherwise the network/graph is called anonymous.

In anonymous networks certain problems have no solution if we use local methods, the example of such a problem is election, i.e. the process of establishing a 'centre' (or a 'leader') node in a network.

In this paper we present a non-global model of computations that makes possible the solution to the election problem in an anonymous network. We also prove that the model has the computational power of global approach while maintaining advantages of the local methodology (e.g. possibility of distribution of the computation). We call the model 'semi-local'.

The problem and related results

In [1], D. Angluin presented a model of local computations, in which the processors with the same number of communication ports are regarded identical. The computation starts with setting some uniform initial state for all the processors in the network and after that, the processors execute the same protocol, whose behaviour depends on their states and the number of neighbours (i.e. used communication ports). The author has investigated limitations of such a model and found a connection between the notion of covering (known from topological graph theory) and the limitations of the model defined. Besides, the author distinguished a class of networks called 'centered', which is of our particular interest.

The 'centered' network has exactly one processor, which is called a 'centre' or a 'leader' and the process of transforming a network to 'centred'. The election problem is particularly interesting due to the following facts:

[1] This work is supported by resources of KBN during the years 2004-2005 as a research project.

– the power of local computations in networks with a distinguished leader is the same as the power of global protocols,
– the election task is conceptually simpler than its equivalent problems (e.g. enumeration or finding a spanning tree),
– no universal solution to the problem exists for the general case of anonymous graph of unknown structure (using local computations).

Several specific solutions to the election problem already exist. The interested reader is referred to [2] and [10].

One group of protocols uses randomisation - as in [3]. Another group bases on the assumption that the network's nodes have unique and linearly ordered identities.

All the solutions for anonymous networks assume specific structure of the network and include protocols for complete graphs and trees [1], graphs called mosaics [6] and covering-minimal graphs with a priori known size [7, 8].

Generally, all solutions (that do not use randomization) use some global knowledge about the graph (e.g. unique comparable identities, size, specific structure). Thus, they can be seen as transferring one kind of global knowledge to another by local means. We are interested in a purely local solution that uses no global knowledge at all.

The authors of [4] characterize the families of graphs, in which the distributed election is possible. This work seems to end any further discussion and close all possibilities of further research. According to the authors some global knowledge is essential. However, the question arises whether it is possible to modify the local model so that the solution to election is possible and the main advantages of local methodology are preserved.

Our contribution

The local computations have three main properties: (1) they change only labeling, not the graph itself, the labels are changed (2) only in locality regions (connected subgraphs) of fixed size[2], and (3) only using information stored in the regions.

Breaking some of the above postulates leads to a nontrivial extension of the power of computation [11, 12], yet the main advantages of local computations are lost easily. In this paper we come up with an extension which seems to be a good compromise between advantages of locality and the power of computation.

In our model we preserve (1) and (3), but we give up the requirement for a locality region to be a subgraph of fixed size. The rationale follows below.

While considering the distributed computations, we generally do not object to the approximation of global knowledge being gathered by local means – by merging and spreading the knowledge stored in the locality regions (on the contrary – this is the goal or side-effect of many distributed algorithms). On the other hand, the locality regions themselves are fixed during the run of any protocol. Once used and recognised, the locality regions remain separated after each step and their structure stays unknown to the regions nearby. We might notice that at the end of each step the knowledge about the structure of underlying locality region is cleared and ignored

[2] by 'size' we mean number of nodes, diameter or radius of the subgraph,

while other knowledge is preserved and used in the next steps. The reasons of this duality are not clear.

Typical locality regions are balls of radius 1 (a node with its neighbours). Our intuition is that in addition to performing a step of some protocol, a node draws forward links between neighbours, thus merges its surrounding balls into one. The process still conforms to the intuition of a local computation – the algorithm starts with balls of radius 1 as local units and the merging steps act locally, too.

Using this methodology we present a semi-local election algorithm and prove its properties, presented in the form of the following propositions:

Lemma 1. The number of steps of our algorithm is bounded by n^3, where n is the number of nodes in the graph.

Theorem 1. The algorithm is correct, i.e. it solves the election problem.

We also get a more general result:

Theorem 2. every (global) relabelling transformation (or algorithm in general) can be simulated by means of semi-local computations.

By global relabelling transformations we mean the ones that are not required to satisfy postulates (2) or (3).

References

1. Angluin, D., Local and Global Properties in Networks of Processors, Proceedings of the 12th Symposium on Theory of Computing (1980) 82-93
2. Barbosa, V. C., An Introduction to Distributed Algorithms, The MIT Press, 1996
3. Dembinski, P., Randomized Enumeration, ICS PAS Reports 848 (1998)
4. Godard, E., Métivier, Y., A characterization of families of graphs in which election is possible (ext. abstract). In Proc. Of Foundations of Software Science and Computation Structures, FOSSACS'02 (2002), Nielsen, M. and Engberg, U., Eds., no. 2303 in LNCS, Springer-Verlag, pp. 159-171
5. Litovsky, I., Métivier, Y., Zielonka, W., The Power and Limitations of Local Computations in Graphs and Networks, Lecture Notes in Computer Science 657 (1993) pp. 333-345
6. Mazurkiewicz, A., Distributed Disassembly of Mosaics, Information Processing Letters 46 (1993) 173-178
7. Mazurkiewicz, A., Distributed Enumeration, Information Processing Letters 61 (1997) 233-239
8. Mazurkiewicz, A., Locally Computable Enumerations, Lecture Notes in Computer Science 1279 (1997) pp. 51-66
9. Schmidt, G., Ströhlein, T., Relations and Graphs, Springer-Verlag 1993
10. Tel, G., Introduction to Distributed Algorithms, Cambridge University Press, 1994
11. Wróblewski, D., Universal election protocol, ICS PAS reports, nr 967, 2003
12. Wróblewski, D., Universal Election Algorithm using an auxiliary graph, ICS PAS Reports 968, 2003

Towards programming logics for low-level languages (Extended abstract)

Ando Saabas[1], Gilles Barthe[2], Lilian Burdy[2], and Tamara Rezk[2]

[1] Institute of Cybernetics, Tallinn University of Technology
[2] INRIA Sophia-Antipolis, Project EVEREST

Introduction

Program verification techniques provide a means to guarantee that source code programs are correct with respect to a formal specification, and have been used extensively for showing functional or behavioral properties of programs. Furthermore, program verification environments, based on automated or interactive theorem provers, have been developed to aid the developer in verifying the program correctness.

However, the benefits of source code verification are not immediately exploitable in the context of mobile code, in which the code consumer downloads a compiled program originating from an untrusted code producer.

The goal of this work is to bring the benefits of program verification techniques to the code consumer, by developing a proof-carrying code (PCC) architecture [1] in which:

- the code producer writes a program A, and annotates it with specification S, and then builds a proof P that A abides by S using an interactive verification environment, and finally compiles the program, its specification and its proof, thereby obtaining a compiled program $\overline{A}$, a compiled specification $\overline{S}$, and a compiled proof $\overline{P}$;
- the code consumer retrieves $\overline{A}$, $\overline{S}$, and $\overline{P}$, and then verifies, with the help of $\overline{P}$, that $\overline{A}$ obeys $\overline{S}$, and finally, if the verification is successful, executes $\overline{A}$.

In order to establish the feasibility of such approach, we consider a simple imperative language and its corresponding assembly language and address the following points:

1. we develop a programming logic, and a weakest precondition calculus for the assembly language, and we prove that they enjoy the same properties as their counterparts for the source language;
2. we relate Hoare specifications of the source language to specifications of the assembly language, and we show that compilation preserves provability w.r.t. Hoare logic and weakest precondition. Moreover, we show the stronger property that the relation "P proves that A satisfies S" is preserved by compilation.

These results, while not unexpected, lay the foundations for a PCC architecture that brings the benefits of source code verification to the code consumer, and form the theoretical basis of a larger project that aims at secure dynamic component update for Java-based operating systems.

For the purpose of this paper, we adopt a simplification w.r.t. Java, namely local variables are treated in a similar fashion at the source and assembly level. We can show, that compilation and the weakest precondition calculi commute up to syntactic equivalence and that both the compilation of proofs and specifications is identity.

In the rest of this extended abstract, we will concentrate more thoroughly on describing the weakest precondition calculus for a low level language.

Related work Our work is most closely related to the work of G. Necula and P. Lee on proof-carrying code, see e.g. [1], but intends to go beyond the verification of safety properties by relying on functional specifications and interactive verification at the source code level. Also closely related to our work is the work of X. Rival [2] that uses abstract interpretation techniques to infer invariants at the source level and compile these invariants for the target level.

Furthermore, there have been a number of studies of programming logics for low-level languages. In particular, C. Quigley [3] has recently proposed a Hoare-like logic for a fragment of the JVM and N. Benton [5] has recently proposed a typed logic for a stack-based assembly language. However, their work does not consider weakest precondition calculi, nor the relationship between programming logics at the source code language with programming logics at the assembly language.

The weakest precondition calculus for assembly language

The simple assembly language used in this study includes basic operations to manipulate the stack, as well as conditional and unconditional jumps. The set of instructions is defined as follows:

$$
\begin{array}{rl}
i ::= & \mathsf{prim}\ op \text{ primitive binary operation} \\
\mid & \mathsf{push}\ n \text{ push } n \text{ on stack} \\
\mid & \mathsf{load}\ x \text{ load value of } x \text{ on stack} \\
\mid & \mathsf{store}\ x \text{ store top of stack in } x \\
\mid & \mathsf{if}\ l \text{ conditional jump to program point } l \\
\mid & \mathsf{goto}\ l \text{ unconditional jump to program point } l \\
\mid & \mathsf{return} \text{ end of program}
\end{array}
$$

where *op* is a primitive arithmetic operation $+, -, \times, \ldots$, comparison operation $<, \leq, =, \ldots$ or logical operation (all considered as arithmetic operations), and x ranges over a set $\mathcal{X}$ of variable names, n ranges over $\mathbb{Z}$ and l ranges over $\mathbb{N}$.

The assertion language used for annotating assembly programs is a standard first order language, with the notable difference being in arithmetic expressions, where two special variables *stack* and *top* are introduced for reasoning about the stack. The variable *top* represents the size of the stack in the current state, while the variable *stack* can be thought of as an array used for the abstract representation of the operand stack. Thus we can refer to the topmost element of the stack via $stack(top)$, to the second element in the stack via $stack(top - 1)$ etc. The set of arithmetic expressions is defined inductively as follows:

$$
\begin{array}{rl}
se & ::= top \mid se - 1 \\
a & ::= n \mid x \mid a_1\ op\ a_2 \mid top \mid stack(se)
\end{array}
$$

The assertion language is defined inductively as follows:

$$P, Q ::= a_1\ bop\ a_2 \mid \texttt{true} \mid \texttt{false} \mid P \vee Q \mid P \Rightarrow Q \mid P \wedge Q \mid \neg P \mid \exists x.P \mid \forall x.P$$

where *bop* is a comparison operation.

The calculus. The main obstacle in defining the weakest precondition calculus for the assembly language is the unstructuredness of code due to jumps. Here, it is addressed by dividing the code into a set of blocks (a block being a sequence of assembly instructions), where each block ends with a jump instruction, and the rest of the block (a sub-block) contains no jump instructions. As jump instructions we consider both `if` l and `goto` l, but also instructions directly followed by an instruction with label l (i.e. the target of an `if` l or `goto` l). For simplicity, we consider the latter case as if there was an extra `goto` instruction leading to that label in the end of the block.

This way, a program can be defined as a sequence of blocks, with a predecessor relation defining an order on them. In case there are circular references between blocks, *loop triples* of the program can be calculated, each triple constituting of a *loop conditional*, a *loop body* and a *loop predecessor* (a block leading to the loop conditional). An invariant has to be associated with each loop conditional (see section Compilation). This context information will be used when calculating the weakest precondition of a program.

The calculus is defined on two levels, firstly for sub-blocks and secondly for full programs (the latter considered as a sequence of blocks).

In the following, we will use b' to denote the sub-block of block b. $P[a_1 \leftarrow a_2]$ will denote the substitution of all occurrences of a_1 in P by a_2. The weakest precondition wp_s of sub-block b' with respect to postcondition φ is defined as follows.

$$
\begin{aligned}
wp_s(b'_1; b'_2, \varphi) &= wp_s(b'_1; wp_s(b'_2, \varphi)) \\
wp_s(\texttt{push}\ n, \varphi) &= \varphi[stack(top) \leftarrow n, top - 1 \leftarrow top] \\
wp_s(\texttt{prim}\ op, \varphi) &= \varphi[stack(top) \leftarrow stack(top)\ op\ stack(top - 1), top \leftarrow top - 1] \\
wp_s(\texttt{load}\ x, \varphi) &= \varphi[stack(top) \leftarrow x, top - 1 \leftarrow top] \\
wp_s(\texttt{store}\ x, \varphi) &= \varphi[x \leftarrow stack(top), top \leftarrow top - 1].
\end{aligned}
$$

Using the above definition of wp_s, we can now define the calculus for blocks. Taking *post* as the postcondition of the program and b_i the i-th block of the program we calculate the weakest precondition φ_i of the block b_i in the following way.

If the last instruction of b_i is

- `return`, then
  $\varphi_i = wp_s(b'_i, post)$
- `goto l`, then
  $\varphi_i = wp_s(b'_i, \varphi_{i,l})$
- `if l`, then
  $\varphi_i = wp_s(b'_i, stack(top) \neq 0 \Rightarrow \varphi_{i,i+1}[top \leftarrow top - 1] \wedge stack(top) = 0 \Rightarrow \varphi_{i,l}[top \leftarrow top - 1]))$

where

1. $\varphi_{i,l} = I$, if b_l is a loop conditional and b_i is the loop body
2. $\varphi_{i,l} = I \wedge\ (I \Rightarrow \varphi_l)$, if b_l is a loop conditional and b_i is its loop predecessor.
3. $\varphi_{i,l} = \varphi_l$ otherwise.

Looking at the above cases: the first case implies calculating the weakest precondition of the loop body. In the second case, the weakest precondition of the whole loop is constructed – the invariant together with the proof obligations. It is analogous to the weakest precondition of a loop in a higher level language: for a `while` loop with body c, conditional b and post-condition p, the proof obligations are $I \wedge \neg b \Rightarrow p$ and $I \wedge b \Rightarrow wp(c, I)$.

The weakest precondition for a sequence of blocks is defined by

$$wp(b_1; b_2, post) = wp(b_1; wp(b_2, \varphi)).$$

Compilation It is trivial to show that compilation preserves provability of Hoare triples, under the assumption that compilation preserves operational semantics. Likewise, it is easy to see that weakest precondition calculi coincide up to logical equivalence. Unfortunately, logical equivalence is too weak if we intend to use proofs produced at source code level for checking specifications at the level of compiled programs. When considering a non-optimizing compiler, we can show that compilation and the weakest precondition calculi commute up to syntactic equivalence and that both the compilation of proofs and specifications is identity, meaning that the pre- and post-conditions and invariants can be copied verbatim to the assembly code.

Conclusions

We have developed a programming logic and a weakest precondition calculus for a simple assembly language and shown that under mild assumptions compilation preserves weakest precondition calculi and provability of Hoare-triples.

These results lay the foundation for a PCC architecture that brings the benefits of source code verification to the code consumer. We have also began setting up this architecture to Java, using JML as the specification language for Java and the programming environment Jack [4] for computing weakest precondition calculi. The work includes defining extensions for class files that accommodate compiled JML annotations. In parallel, we have begun investigating the use of such framework for secure dynamic updates in the context of trusted personal devices that feature Java Virtual Machine.

References

[1] George C. Necula. Proof-Carrying Code. In 24th Symposium on Principles of Programming Languages, pages 106-119, 1997.

[2] Xavier Rival. Symbolic transfer function-based approaches to certified compilation. In 31th Symposium on Principles of Programming Languages, pages 1-13, 2004.

[3] Claire L. Quigley. A Programming Logic for Java Bytecode Programs. In 16th Int. Conf. on Theorem Proving in Higher Order Logics, pages 41-54, 2003.

[4] Lilian Burdy, Antoine Requet, Jean Louis Lanet. Java Applet Correctness: A Developer-Oriented Approach. In International Symposium of Formal Methods Europe, pages 422-439, 2003.

[5] Nick Benton. A typed logic for stacks and jumps. Draft, 2004.

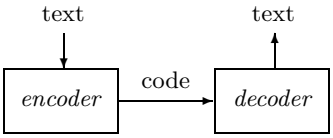
A Program Inverter LRinv and its Structure (Abstract)

Masahiko Kawabe[1⋆] and Robert Glück[2]

[1] Waseda University, Graduate School of Science and Engineering
Tokyo 169-8555, Japan, kawabe@futamura.info.waseda.ac.jp
[2] University of Copenhagen, DIKU, Dept. of Computer Science
DK-2100 Copenhagen, Denmark, glueck@acm.org

Keywords: programming languages, program transformation, program inversion, context-free grammars, functional programming.

The purpose of this paper is to describe a method for automatic program inversion which we developed [2, 3] for a first-order functional language and to show several inverse programs produced by our implementation. There are many computational problems that are inverse to each other, even though we often do not consider them from this perspective. Perhaps the most common example of two programs that are inverse to each other is the encoding and decoding of data:



Two inverse programs

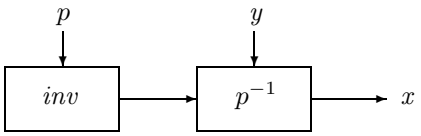
Usually, both of the encoder and the decoder are written by hand with the familiar problem of ensuring their correctness and maintaining their consistency. Why write two programs, if it suffices to write one program by hand and to derive the inverse program automatically? Beside saving time and effort, the inverse program is correct by construction and it is easy to produce a new inverse program if there are changes in the original program.

One of the main challenges of program inversion is to derive a *deterministic inverse program* from an injective source program. In [3], we observed that this problem is similar to the construction of a *deterministic parser*, if we view the inverse program as a context-free grammar where the traces of the program constitute its language. Determining which branch to follow in a nondeterministic inverse program is like determining which production rule to apply when constructing a deterministic parser. Our method makes use of this correspondence.

A *program inverter* is a program transformer similar to a translator, but instead of producing a target program that is functionally equivalent to the source program, a program inverter produces a program that is inverse to the source

⋆ Part of this research is supported by JSPS.

program. Inversion is a fundamental concept in Mathematics, Science, and Engineering, but has found little attention in Computer Science even though many algorithmic problems are inverse to each other, such as printing and parsing, compilation and decompilation, and encoding and decoding data.



Program inverter: generating an inverse program

In this paper, we describe the construction principles for automatic program inverters which we identified in our previous works [2, 3]. They include the design of an 'atomic language', local inversion by backward reading a program, and the elimination of nondeterminism, if possible.

There are different choices when designing a program inverter, which also means they can differ in their inversion power. The most promising approach which we identified for the elimination of nondeterminism is to view a program as a context-free grammar and to apply to it methods known from parsing theory. In [2] we used left-factoring, in [3] methods from LR(0) parsing, and in our recent work LR(k) parsing [5]. These methods all follow the same construction principles, but differ in the power of the eliminating nondeterminism. Our current implementation[1] is based on LR(0) parsing. We will use this implementation for the examples in this paper. The examples belong to different application area, including encoding and decoding, printing and parsing, and data conversion.

This full paper introduces the main concepts for program inversion in more detail, presents three construction principles for building a program inverter, shows inverse programs produced by our implementation, and discusses related work which started in the late seventies in the works of Dijkstra [1] and Gries [4].

References

1. E. W. Dijkstra. Program inversion. In F. L. Bauer, M. Broy (eds.), *Program Construction: International Summer School*, LNCS 69, 54–57. Springer-Verlag, 1978.
2. R. Glück, M. Kawabe. A program inverter for a functional language with equality and constructors. In A. Ohori (ed.), *Programming Languages and Systems. Proceedings*, LNCS 2895, 246–264. Springer-Verlag, 2003.
3. R. Glück, M. Kawabe. Derivation of deterministic inverse programs based on LR parsing. In Y. Kameyama, P. J. Stuckey (eds.), *Functional and Logic Programming. Proceedings*, LNCS 2998, 291–306. Springer-Verlag, 2004.
4. D. Gries. *The Science of Programming*, chapter 21 Inverting Programs, 265–274. Texts and Monographs in Computer Science. Springer-Verlag, 1981.
5. M. Kawabe, R. Glück. Experiments in program inversion based on LR parsing methods. submitted, 2004.

[1] Home page `http://www.ispt.waseda.ac.jp/inv/`

Extensions of Event Based B for Development of Grid Systems

Pontus Boström and Marina Waldén

Åbo Akademi University, department of Computer Science
Turku Centre for Computer Science (TUCS)
Lemminkäisenkatu 14 A, 20520 Turku, Finland
E-mail:{Pontus.Bostrom, Marina.Walden}@abo.fi

1 Introduction

Computational grids have become widespread in organizations for handling their vast amount of available information and need for computational resources. These grid systems are often complex and special care need to be taken in order to ensure their correctness. We propose an extension of Event based B [2], in order to enable their formal implementation. The extensions provide new constructs to take into account the client-server architecture of grid systems, as well as important features like communication and synchronisation.

2 A language for distributed systems

Organizations need the ability to efficiently utilise existing hardware and be able to effectively share information with each other. Computational grids have become a popular approach to enable organizations to handle the vast amount of available information. Grid computing is a distributed computing paradigm that differ from traditional distributed computing in that it is aimed toward large scale systems that even span organizational boundaries. Recently standards such as Open Grid Services Infrastructure (OGSI) [5] and toolkits such as Globus toolkit [4] for implementing those standards have been developed in order to simplify the construction of complex grid systems.

The standards above do, however, not help in designing and implementing *correct* grid systems. The use of formal methods are needed in order to ensure their correctness. The Action System formalism [3] is a formal method that is well suited for developing large distributed systems, since it supports stepwise development. However, it lacks tool support. The B Method [1] is a formal method originally developed for construction of

sequential programs and has good tool support. The B Method can be combined with Action Systems in order to formally reason about distributed systems as in the related methods B Action Systems [6] and Event based B [2]. B Action Systems models Action Systems in ordinary B, while the later Event based B also extends traditional B with some new constructs. With generic formal languages it is, however, possible to construct abstract models and specifications that are not implementable or very difficult to implement efficiently. The problem becomes especially apparent when developing distributed systems with complicated synchronization and communication patterns. Therefore, we propose new extensions to Event based B in order to be able to implement grid systems and verify their correctness in a convenient way.

The language extensions we propose for Event based B are targeted towards Grid systems using the Globus toolkit [4] middleware. Grid systems usually have a client-server architecture. This means that there is a client that initiates communication with the server, which only responds to the clients request. A client may access several servers simultaneously. Therefore, the language has to support client-server architectures with multiple concurrent accesses by the same client to several servers. The main communication mechanism of the grid middleware is remote procedure calls. However, the grid middleware also supports asynchronous notifications sent from a server to a client. Hence, the language should support both these communication primitives.

In grid systems a server is usually referred to as a grid service, since it provides services to other grid components. In order to meet the language requirements above we propose to extend Event based B with two types of machines, a *grid service machine* modelling abstract grid service features and a *grid refinement machine* for refining grid service machines and for introducing grid features to ordinary Event based B models. A grid service machine is a template of which the clients can obtain instances. The client can control several instances of the same grid service machine as a master can control several identical worker nodes. A grid service machine contains specifications of remote procedures, actions and notifications. The grid refinement machine on the other hand has clauses for refined remote procedures and actions, as well as statements for handling notifications. The clients and servers, i.e. grid services, use remote procedure calls and notifications to communicate and synchronize with each other. A client makes a requests to a grid service with a remote procedure call. When the request has been carried out a notification is sent back to the client. While the client waits for a notification from a grid service, it cannot make a new remote procedure call to the same grid service.

The development of the grid system, shown in Figure 1 starts with an initial specification in Event B, *D0,* that is refined in a number of steps, *D1.* The specification is then split up into a client, *D2*, and a number of grid services, *E0.* The grid services can in turn be independently refined further, *E1*, and reference new grid services, *F0.* Throughout the development of the system the grid constructs are translated to ordinary B machines for

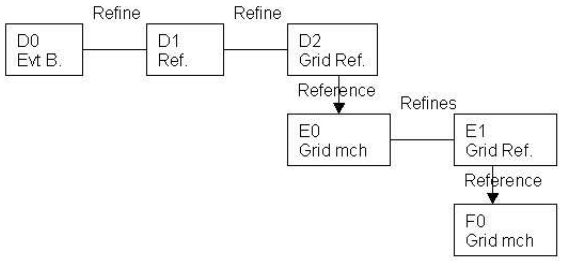


Figure 1: *The structure of the grid system development*

verification purposes.

3 Conclusions

We have proposed an extension of Event based B to facilitate the development of correct grid systems. Grid systems are large distributed systems and standard development tools cannot guarantee their correct implementation. Our extension can provide a convenient formal development process for such distributed systems. The systems will by construction have an architecture that is implementable. Furthermore, specifications of grid systems constructed in this language will be clear to understand, since the systems are modeled in terms of grid primitives with a precise meaning. We believe that this approach to adapt B to the Globus Toolkit middleware can also be useful for other types of middleware for distributed systems.

References

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

[2] Abrial, J. -R. and Mussat, L. 2001. (accessed 06.09.2004) *Event B Reference Manual*. `http://www.atelierb.societe.com/ressources/evt2b/eventb_reference_manual.pdf`

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

[4] The Globus Alliance. 2003. (accessed 06.09.2004) Globus Toolkit. `http://www.globus.org/`

[5] Graham, S., Kesselman, C., Maguire, T., Sandholm, T., Snelling, D. and Vanderbilt, P. 2003. (accessed 06.09.2004) Open Grid Services Infrastructure. `http://www.ggf.org/documents/final.htm`

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

Sequent Calculi for Temporal Logics of Common Knowledge and Belief

Jūratė Sakalauskaitė
Institute of Mathematics and Informatics
Akademijos 4,2600 Vilnius, Lithuania
email: jurates@ktl.mii.lt

In this paper we consider two temporal logics: a temporal logic of common knowledge and a temporal logic of common belief. These logics involve the discrete time linear temporal logic operators "next" and until"". In addition, they contain an indexed set of unary modal operators that allow us to represent the information possessed by a group of agents. In the temporal logic of common knowledge these operators satisfy analogues of the axioms of the modal system $S5$ ([2]), which is widely recognized as a logic of idealized knowledge [4]. In the temporal logic of common belief these operators satisfy analogues of the axioms of the modal system $KD45$, which is widely recognized as a logic of idealized belief. In the first logic there is also an operator for common knowledge of the group of all agents. In the second logic there is an operator of common belief of the group of all agents. An interpretation of these logics is based on the sequences of states of the distributed systems.

For these logics we present sequent calculi with a restricted cut rule. Thus, we get proof systems where proof-search becomes decidable. This calculus lays the basis for automatic theorem proving for these logics. We prove soundness and completeness theorems for these logics.

We mention some related works. In [6] a Hilbert-style axiomatic system for the temporal logic of common knowledge is presented and completeness of the system is proved. The temporal logic of knowledge without common knowledge operator is considered in [7], [3]. In [7] a tableau based decision procedure is presented for the considered logic. In [3] a resolution-based proof system is presented which is shown to be correct. The logic of common knowledge without temporal operators is considered in [1], where complete Tait-style sequent calculus with restricted cut rule for the logic is presented. Our work uses ideas from [1] and [6].

Multi-logics like temporal logics of common knowledge and belief have a number of applications both to computer science and artificial intelligence (see e.g. [4]). For example, in distributed systems theory, they are used for the specification and verification of knowledge-based protocols [5].

References

[1] L. Alberucci. The modal μ-calculus and the logic of common knowledge, Ph.D. thesis, Institut für Informatik und angewandte Mathematik, Universität Bern, 2002.

[2] B. Chellas. Modal Logic: An introduction. Cambridge University Press: Cambridge, England, 1980.

[3] C. Dixon, M. Fisher, M. Wooldridge. Resolution for temporal logics of knowledge, J. Logic Computat., 8(3): 345-372, 1998.

[4] R. Fagin, J.Y. Halpern, Y. Moses, and M.Y. Vardi. Resoning about Knowledge. Mit Press, 1995.

[5] J.Y. Halpern. Using reasoning about knowledge to analyze distributed systems. Annual Review of Computer Science, 2:37-68, 1987.

[6] J.Y. Halpern, R. van der Meyden, M.Y. Vardi. Complete axiomatizations for reasoning about knowledge and time. Submitted for publication, 1999.

[7] M. Wooldridge, C. Dixon, M. Fisher . A tableau-based proof method for temporal logic of knowledge and belief, Journal of applied non-classical logics 8(3): 225-258, 1998.

Recent technical reports from the Department of Information Technology

2004-022 Pascal Van Hentenryck, Pierre Flener, Justin Pearson, and Magnus Ågren: *Compositional Derivation of Symmetries for Constraint Satisfaction*

2004-023 Jarmo Rantakokko: *Interactive Learning of Algorithms*

2004-024 Mathias Spjuth, Martin Karlsson, and Erik Hagersten: *Low Power and Conflict Tolerant Cache Design*

2004-025 David Lundberg: *Feasibility Study of WLAN Technology for the Uppsala - Stockholm Commuter Train*

2004-026 David Lundberg: *Ad Hoc Protocol Evaluation and Experiences of Real World Ad Hoc Networking*

2004-027 Thorild Selén: *Reorganization in the Skewed-Associative TLB*

2004-028 Lars-Henrik Eriksson: *Using Formal Methods in a Retrospective Safety Case*

2004-029 Krister Åhlander and Hans Munthe-Kaas: *On Applications of the Generalized Fourier Transform in Numerical Linear Algebra*

2004-030 Wendy Kress and Per Lötstedt: *Time Step Restrictions using Semi-Implicit Methods for the Incompressible Navier-Stokes Equations*

2004-031 Malin Ljungberg: *Curvilinear Coordinates in a PDE Solver Framework; Analysis*

2004-032 Malin Ljungberg and Kurt Otto: *Curvilinear Coordinates in a PDE Solver Framework; Validation*

2004-033 Parosh Abdulla, Johann Deneux, and Pritha Mahata: *Closed, Open and Robust Timed Networks*

2004-034 Parosh Abdulla, Pritha Mahata, and Richard Mayr: *Decidability of Zenoness, Token Liveness and Boundedness of Dense-Timed Petri Nets*

2004-035 Emad Abd-Elrady and Torsten Söderström: *Bias Analysis in Least Squares Estimation of Periodic Signals Using Nonlinear ODE's*

2004-036 Stefan Johansson: *High Order Finite Difference Operators with the Summation by Parts Property Based on DRP Schemes*

2004-037 Torsten Söderström and Mei Hong: *Identification of Dynamic Errors-in-Variables Systems with Periodic Data*

2004-038 Martin Nilsson: *Different Methods that Reduce Cost in Monostatic RCS Computations for MOM Accelerated by MLFMA*

2004-039 Jing Gong and Jan Nordström: *A Stable Hybrid Method for Hyperbolic Problems*

2004-040 Magnus Svärd and Jan Nordström: *On the Order of Accuracy for Difference Approximations of Initial-Boundary Value Problems*

2004-041 Paul Pettersson and Wang Yi (Eds.): *Proceedings of the 16th Nordic Workshop on Programming Theory*

UPPSALA
UNIVERSITET

October 2004
ISSN 1404-3203
http://www.it.uu.se/