FM-Tools 2000

The 4th Workshop on Tools for System Design and Verification

Gerhard Schellhorn, Wolfgang Reif (eds.)
Universität Ulm

Abstract

There is a growing awareness that tool support is needed to develop high-assurance industry-size software/hardware systems. The aim of these workshops was to provide a forum of researchers interested in the use and development of tools which support the use of mathematical techniques for the specification, development, analysis and verification of systems. This workshop was the fourth in a series of biennial events devoted to this topic. The first three workshops took place in Kiel (1994), Bremen (1996) and Malente (1998), Germany. This workshop was held at the "Reisensburg", a beautiful castle near Ulm in South Germany.

The following technical report contains all accepted submissions chronologically ordered according to the workshop programme.

Nr. 2000-07
Ulmer Informatik Berichte
Mai 2000

Accepted Submissions

Submitted to the 4th Workshop on Tools for System Design and Verification, 10-13 July, 2000.

An open software architecture for the verification of industrial controllers

Heinz Treseler, Stefan Kowalewski

Process Control Laboratory, Chemical Engineering Department,
University of Dortmund, D-44221 Dortmund, Germany
Phone: ++49-231-755 5171, Fax: ++49-231-755 5129
E-Mail: {h.treseler, s.kowalewski}@ct.uni-dortmund.de

Abstract

An open tool architecture for the formal verification of logic controllers for chemical processes is presented. It supports a model-based verification approach which builds on models of the controller and of the plant. The architecture consists of a hierarchical modeling editor, translators for the input of given controllers and an interface to integrate available model-checkers. In the paper, the tool architecture is described and the necessity for front-ends to modeling paradigms used in process engineering is explained on the example of *Process Control Event Diagrams* (PCEDs).

1 Introduction

For formal verification of logic controllers in process industry, discrete models of the controller and often also of the basically continuous process dynamics are necessary, because the correct operation of such controlled processes depends on the interaction of the discrete actions of the controller and the continuous quantities of the plant (e.g., if a discrete controller starts the cooling of a reactor as soon as a certain temperature threshold is reached, it is necessary to know how fast the reaction temperature rises and how much heat can be carried away by the cooling). The resulting model of the controller or of the controlled process can be checked by a formal verification whether a given specification is fulfilled or not (e.g., the temperature of a reactor may be only within certain boundaries).

Up to now, formal verification was hardly applied to analyze logic controllers. For this the following reasons seem to be responsible:

Even for simple examples the necessary models can become very complex. Obviously, the descriptive and computational effort is the standard problem in the application of the formal verification. Furthermore, the industrial user is usually not familiar with the complex underlying theory developed in computer science by theorists for theorists. Therefore, well-known and available modeling paradigms from process engineering should be used to model plant and controller. For this several modeling front-ends have to be offered to the user. The model of the controller can be created from an already implemented controller or from an available specification.

This contribution describes the software environment VERDICT (*Ver*ification of *Di*screte Controllers for *Cont*inuous Processes) [KT97] which supports the application of formal verification techniques in an industrial environment by dealing with the ideas mentioned above.

The paper is organized as follows: In Section 2 the software architecture of VERDICT is described and in Section 3 the used modeling framework is introduced. Section 4 presents the available front ends to modeling paradigms known in process engineering. This front ends allow to generate discrete models automatically from already available descriptions of the plant and of the controller. Section 5 sketches the analysis by available model checkers. The paper finishes with a short conclusion.

2 Architecture

The VERDICT environment connects available tools for discrete event and real-time systems to a suit of user interfaces which allow the process engineer to apply model-based verification without having to get into the details of specifying modeling paradigms and verification theories. Instead, the process engineer shall be able to specify the problem using familiar representations from process or automation industry. Through this, it is possible to include available specifications without having additional modeling effort.

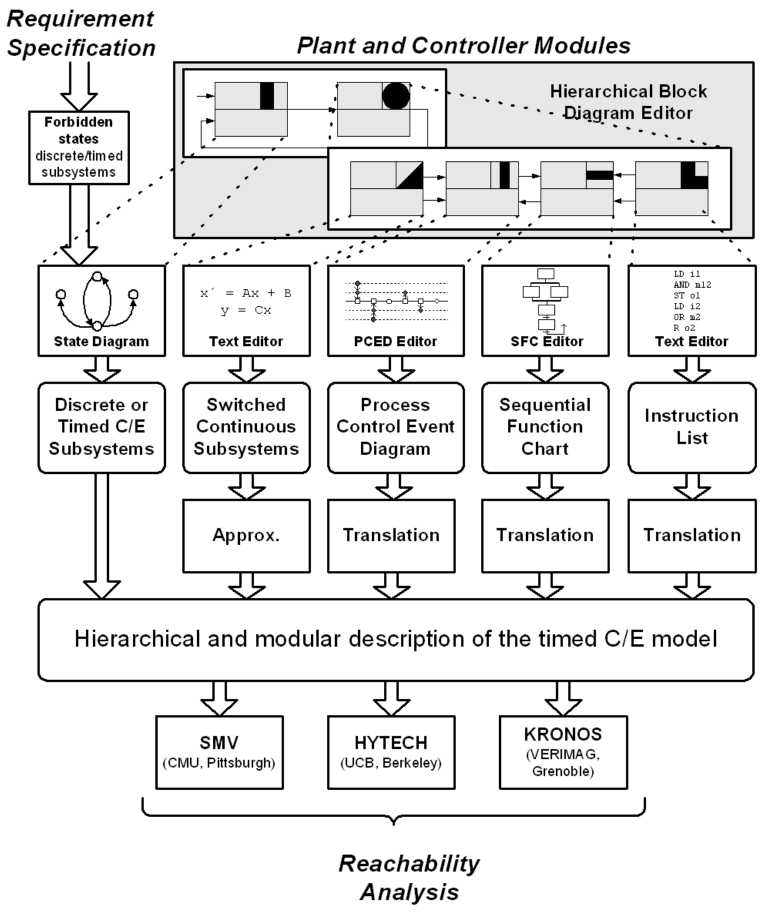

Requirement Specification
Plant and Controller Modules
Hierarchical Block Diagram Editor
Forbidden states discrete/timed subsystems
State Diagram
x´ = Ax + B
y = Cx
Text Editor
PCED Editor
SFC Editor
LD i1
AND m12
ST o1
LD i2
OR m2
R o2
Text Editor
Discrete or Timed C/E Subsystems
Switched Continuous Subsystems
Process Control Event Diagram
Sequential Function Chart
Instruction List
Approx.
Translation
Translation
Translation
Hierarchical and modular description of the timed C/E model
SMV (CMU, Pittsburgh)
HYTECH (UCB, Berkeley)
KRONOS (VERIMAG, Grenoble)
Reachability Analysis

Figure 1: Software architecture of VERDICT

In figure 1 the software architecture is illustrated. Due to its design, VERDICT is an open system, which consists of different components. Several components can be used alternatively to each other. For the modeling, different graphical editors are available and for the analysis, one tool can be chosen from different analysis tools. The main part of the user interface consist of a hierarchical block diagram editor in which the subsystems of plant elements and control devices and their interaction can be specified using so-called Condition/Event systems (CESs) [SK91] (see next section). Blocks can be timed discrete state CESs (TCESs) [EKKJ95], continuous systems or controller devices specified in different controller languages. For the analysis the continuous systems [SKE97] and the controller devices can be automatically replaced by TCESs. The overall result is a TCES representation of the whole controller or of the whole controlled plant. The internal dynamics of TCESs are specified using hierarchical state graphs in which the requirements can be indicated by forbidden states. For the specification of the continuous systems and the controller devices further front ends are available, e.g., a text editor to specify continuous systems as differential algebraic equation (DAE) or special graphical editors (see also section 4). TCES modules can be stored and managed by the library function of the block diagram editor. Therefore, TCES modules already specified can be reused and assembled to a new model later.

3 Modeling paradigm

TCESs as underlying modeling framework were chosen, because they meet certain requirements: In order to be able to model also complex technical processes, TCES support a modular modeling approach. Complex models can be created by interconnecting simple TCES submodules which represent parts of the process or of the controller. The individual TCES modules are finite state machines with quantitative timing information. The interactions between the TCES modules are represented as signals. Two different classes of signals can be used: *Condition* signals are piece-wise constant and their values correspond to discrete states. *Event* signals are point-wise nonzero and carry information about currently occurring state transitions. The condition signals are used to disable or enable transitions in other systems, whereas the event signals can enforce transitions in other systems. In contrast to the a-causal synchronization mechanisms of the automata models from computer science (e.g Times Automata (TA) [AD94] or Hybrid Automata (HA) [ACHH93]), the condition and event signals reflect cause and effect of the interactions between different parts of the process and the controller.

Quantitative timing information of the process and of the timers of the controller can be modeled by TCES. The timing in the TCES modules is modeled by clocks. Clocks are reset by an input signal or by a transition and their value rises with the gradient one. Other transitions can depend on the values of these clocks. This concept is closely related to that of TA.

4 Modeling front ends

To support the user in developing verifiable control code, editors and a translators which map the available control code to TCES are provided. At present the textual language *Instruction List* (IL) and the graphical language *Sequential Function Charts* (SFC) according to the IEC61131-3 norm and so-called *Process Control Event Diagrams* (PCED) are supported. Yang and Chung suggest using PCED [YC97], [YC98] during safety analyses of logic controlled processes. PCEDs are an abstract and qualitative model of the controller. The advantage of this representation is that connections between process variables and the control logic can be visualized in a very simple and descriptive manner. The PCED illustrates the interaction between nodes arranged on five different layers of a controlled process (from the top to the bottom layer: Operator, Human Interface Device (HID), Computer, Sensor/Actuator, Process). The nodes represent the components (e.g., sensors, actuators, control algorithm) involved in the system. An edge linking two nodes is either the propagation of a signal or the causal action or effect. The example PCED in figure 2 illustrates an emergency control. If a fault occurs, i.e., if the measured oil level falls below a certain threshold, an alarm is displayed and the output values *catalyst flow* and *cooling flow* are kept constant by holding the *catalyst valve* (CATV) and the *cooling valve* (COOV).

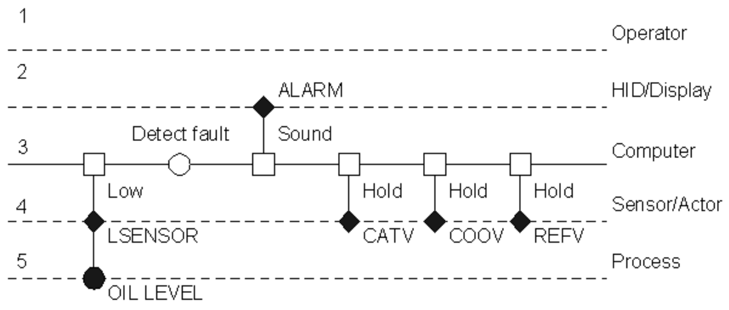


Figure 2: An example PCED

PCEDs can be read into VERDICT. VERDICT replaces the PCEDs by a TCES model. Every node of the PCED is modeled by an individual TCES module. The data flow between the nodes is represented by condition or event signals between the TCES modules modeling PCED nodes. The translation algorithm will be described more detailed in the full paper.

5 Analysis

The complete TCES model can be analyzed. The concept of VERDICT is not to develop a specific analysis algorithm but to provide an interface to already available tools.

To facilitate the analysis of the TCES model by existing analysis tools, the model has to be converted into the input format of the chosen analysis tool. The tools KRONOS [OY93] and HyTech [HHWT96] are able to analyze TA. To integrate these tools, we take advantage of the fact that the TCESs can be expressed by TAs. First, the underlying transition system of the TCES is mapped into the state transition structure of the TA. Then the relevant conditions and events of the TCESs are mapped into transition constraints of the TA. An algorithm for automatic translation of TCESs into TA is integrated in VERDICT.

Additionally, the discrete model-checker SMV was chosen. To use a discrete model-checker, the infinite space of all clock values of the TCES has to be approximated by a finite automaton, which combines infinitely many time points to a discrete state. The idea of the translation is based on so-called *region automata* [AD94], whose regions can be modeled by representative states [Yov98], so that it can be modeled by a finite state machine. Similar to the translator into TA, the translator maps the state transition structure of the TCESs into a finite state machine. This algorithm is also integrated in VERDICT.

6 Conclusion

A tool environment for formal verification of logic controllers for chemical plants was presented. The tool named VERDICT supports a model-based approach with interfaces to common specification paradigms for modeling the plant and the controller. Timed C/E systems as underlying modeling framework are modular and signal-oriented which helps to build complex models. Continuous models and controllers are replaced automatically by TCES models. This provides a systematic procedure to develop verifiable models of the controlled plants.

The main objective of VERDICT is to provide a consistent modeling environment and interfaces to different analysis tools. The integration of several tools, all based on different modeling paradigms, in a modeling environment based on TCES systems, is representing a major challenge in developing VERDICT. The application of VERDICT is described in [KEPS99] and [KES99].

References

[ACHH93] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, LNCS 1201, pages 209–229. Springer, 1993.

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[EKKJ95] S. Engell, S. Kowalewski, B.H. Krogh, and J.Preußig. Condition/Event systems: A powerful paradigm for timed and untimed discrete models of technical systems. In *EUROSIM*, pages 421–426, Vienna, Austria, 1995. Elsevier.

[HHWT96] T. Henzinger, P. Ho, and H. Wong-Toi. A user guide to HyTech. http://www-cad.eecs.berkeley.edu/ tah/HyTech/, UC Berkeley, USA, 1996.

[KEPS99] S. Kowalewski, S. Engell, J. Preußig, and O. Stursberg. Verification of logic controllers for continuous plants using timed condition/event-system models. *Automatica*, 35:508–518, 1999.

[KES99] S. Kowalewski, S. Engell, and O. Stursberg. Verification of logic controllers for continuous plants. In *Advances in Control (Highlights of ECC'99)*, pages 345–389. Springer, 1999.

[KT97] S. Kowalewski and H. Treseler. VERDICT - a tool for model-based verification of real-time logic process controllers. In *5th Int. Workshop on Parallel and Distributed Real-Time Systems (WPDRTS'97)*, Geneva, Switzerland, 1997.

[OY93] A. Olivero and S. Yovine. KRONOS: A tool for verifying real-time systems. userguide. http://www-verimag.imag.fr/TEMPORISE/kronos/, VERIMAG, Grenoble, France, 1993.

[SK91] R.S. Sreenivas and B.H. Krogh. On Condition/Event Systems with discrete state realizations. In *Discret Event Dynamic Systems: Theory and Applications 1*, pages 209–236, Boston, 1991. Kluwer Academic Publishers.

[SKE97] O. Stursberg, S. Kowalewski, and S. Engell. Generating timed discrete models of continuous systems. In *2nd MATHMOD*, pages 203–209, Vienna, Austria, 1997.

[YC97] S.H. Yang and P.W.H. Chung. Functional modelling for hazard identification. In *Workshop Notes of the 5th International Workshop on Advances in Fuctional Modelling of Computer Technical System*, Paris, France, 1997.

[YC98] S.H. Yang and P.W.H. Chung. Life cycle hazard analysis for computer controlled processes. *Computers and Chemical Engineering*, pages 483–490, 1998.

[Yov98] S. Yovine. Model-checking timed automata. In *Embedded Systems*, Lecture notes in computer science, 1998.

DisCo Toolset – The New Generation

Timo Aaltonen, Mika Katara, and Risto Pitkänen
Software Systems Laboratory
Tampere University of Technology
P.O. Box 553, FIN-33101 Tampere, Finland
{tta, clark, rike}@cs.tut.fi

1 Introduction

Formal methods are widely considered one possible solution to the so-called *software crisis* arising from the increasing complexity of systems. However, they are not easy to adopt in industrial use. They often require some mathematical knowledge and new ways of thinking. Some difficulties can be overcome by providing appropriate tool support for users.

When talking about formal method tools, people usually first think of verification. Formal proofs are usually complicated and long, and therefore theorem provers such as PVS [8] and model checkers such as Kronos [10] have been developed to assist in or completely automatize them. However, formal proof is not the only way to analyze a formal specification.

Simulation and animation have proved valuable aids to validation and testing of formal models. They require that the specification has an *operational interpretation*, i.e. that it can be executed in some way.

DisCo [4] is a formal specification method for reactive systems. It focuses on collective behaviour of objects and provides a simple refinement mechanism preserving safety properties. Tool support for animation of DisCo specifications has existed already in the beginning of the 1990's [9]. An improved version of the DisCo language containing support for real-time specification and a more flexible type system among other new features has been developed during the last few years. Due to technical limitations (e.g. poor portability) of the first tool generation, support for the new language was not added in the old implementation. Instead, a whole new toolset was designed and implemented. This paper describes the new toolset.

The structure of the rest of this paper is as follows. Section 2 introduces the DisCo method and describes example specification. In Section 3, the architecture of the new DisCo toolset and the individual tools are described. Section 4 contains some concluding remarks.

2 DisCo

2.1 Method

DisCo is a formal specification method for reactive systems. Specifications are written in a programming-language-like notation, whose formal semantics is given in terms of Temporal Logic of Actions [6]. The basis of DisCo is in the *joint action* theory [3], which allows describing systems at a high level of abstraction, at which implementation-level details are superfluous.

The basic building blocks of the specifications are *classes* and *actions*. The global state of the modelled system is formed by local the states of individual objects, relations, global clock Ω, and a global set of deadlines. The only way to change the state is to execute actions. Actions are atomic units of execution. They consist of *roles*, in which objects may participate (only the states of the participating objects may change in an action); *parameters*, which are values of basic types or records; a *guard*, which limits the possible participant and parameter combinations and a *body*, which is one parallel assignment clause. Assertions may be stated on the global state, but they do not limit the behaviour of the system, thus they need to be verified.

DisCo specifications are generic. The number of objects need not be fixed. Even an infinite number of objects is allowed.

The specifications are structured in behavioural units called *layers*, each of which encapsulates a different aspect of the system being specified. A layer describes how a specification is changed when the new aspect is modelled. This structuring mechanism is orthogonal to ordinary architectural structuring, in which the specifications are divided in architectural units reflecting the implementation-level units. Behavioural structuring—unlike architectural structuring—does not lead to specifying the interfaces between units, before knowing the collective behaviour of the total system.

DisCo specifications are written in layered manner:

the refinement mechanism is superposition, which only allows adding new variables to classes, strengthening the guards of existing actions, introducing new actions, adding new assignments to actions, and assigning to the new variables only.

2.2 Example Specification

In this subsection a very simple cash-point service system is specified in DisCo. The system consists of four kinds of entities: *accounts*, *tills*, *cash cards* and *customers*. Cards may be inserted and ejected to and from tills. Money may be withdrawn from accounts using tills.

The DisCo specification consists of four layers: `till`, `card`, `customer` and `complete`. Layers `card` and `customer` refine layer `till`, and layer `complete` refines the composition of `card` and `customer`.

Layer `till` (see below) defines the most abstract view to the system. Classes `Account` and `Till` are introduced. Assertion `balanceOK` states that balance of all accounts is always non-negative. Withdrawal is possible only from a till in our simple specification. Action `withdraw` has two roles: `acc` (of class `Account`) and `till` (of class `Till`) and one parameter: `amount` (of type **`integer`**). It may be executed if the withdrawn amount is positive and there exists enough money on the account. At this level of abstraction we do not specify anything about customers or cards. Deposit is also possible in the layer.

```
layer till is
  class Account is
    balance: integer;
  end;

  class Till is end;

  assert balanceOK is ∀ acc: Account :: acc.balance >= 0;

  action withdraw(acc:Account; till: Till; amount:integer) is
  when amount > 0 ∧ acc.balance >= amount do
    acc.balance := acc.balance − amount;
  end;

  action deposit(acc:Account; amount:integer) is
  when amount > 0 do
    acc.balance := acc.balance + amount;
  end;
end;
```

Layer `customer` (see below) refines specification `till` by adding aspects related to customers to the model. The layer specifies that withdrawals are only possible for customers from their own accounts. Ownership is specified by relation `CustAcc`.

```
layer customer is
  import till;

  class Customer is
    wallet: integer;
  end;

  relation CustAcc(Customer, Account) is 0..1:1;

  refined withdraw(cust:Customer; acc:Account;
                   till:Till; amount:integer)
  of withdraw(acc, till, amount) is
  when ... CustAcc(cust, acc) do
    ...
    cust.wallet := cust.wallet + amount;
  end;
end;
```

Layer `card` adds aspects of cash cards to the specification, and finally layer `complete` gathers all the refinements together. These layers are omitted here to save space.

3 Tools

3.1 Architecture

With the experience gained with the first generation of DisCo tools, we saw *portability*, *extensibility* and *usability* as the most important considerations when choosing implementation technologies and designing the general architecture of the new toolset. Portability was ensured by choosing ISO C++ and Java as the implementation languages. Extensibility was achieved by designing a general architecture (depicted in Figure 1) centered around a multi-purpose intermediate form and utilizing *hooks* that allow modifying and adding functionality.

The intermediate form produced by the DisCo compiler front end is an explicit and flat representation of a layered specification. It is utilized by the compiler itself and several back ends that produce input for different tools.

3.2 Compiler

DisCo Compiler plays a central role in the DisCo toolset. It works as a link between different tools and DisCo source code. Standard C++ was chosen as the implementation language for its good performance and portability. Object-oriented features and genericity of C++ are heavily utilized.

Functionality of the compiler is divided into two phases: *front-end* and several *back-ends.* The front-end produces an intermediate form of DisCo source. After successful translation into intermediate form, back-ends of the compiler may be used to produce input for

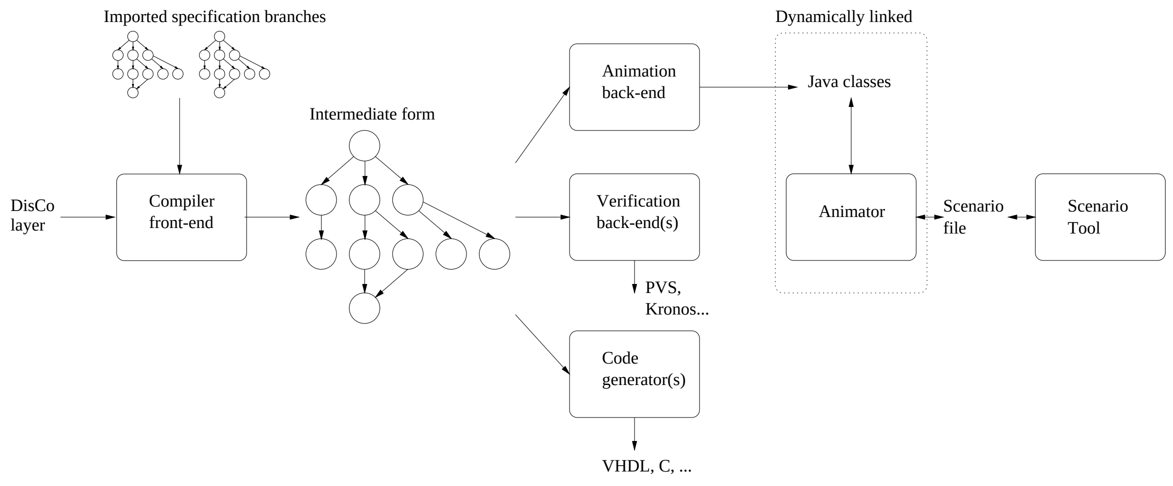


Figure 1: General architecture of the DisCo toolset.

different tools, like Animator or different verification tools.

The front-end of the compiler takes one DisCo layer and the intermediate forms of possibly imported specification branches as its inputs, and produces an intermediate form of the specification. The intermediate form corresponds one to one to the internal representation of the semantical tree (or DAG) of the specification being compiled. The intermediate forms of imported branches are merged into the tree representing the specification being compiled. Checking syntactic and semantic correctness of the code is carried out by the front-end. If an error occurs during the compilation, no intermediate form is produced.

Animation back-end of the compiler produces a *specification engine*, which is a Java package, for Animator. It offers an interface by which Animator can instantiate objects in the creation phase, and execute enabled actions in the execution phase. The engine notifies Animator about state changes and some other events. It also informs Animator about enabled actions. The assertions of the specification are guarded by the engine during execution.

Back-ends for producing input for several verification tools could be added to the toolset. We are also researching possibilities of producing VHDL or C code from an instantiation of a DisCo specification.

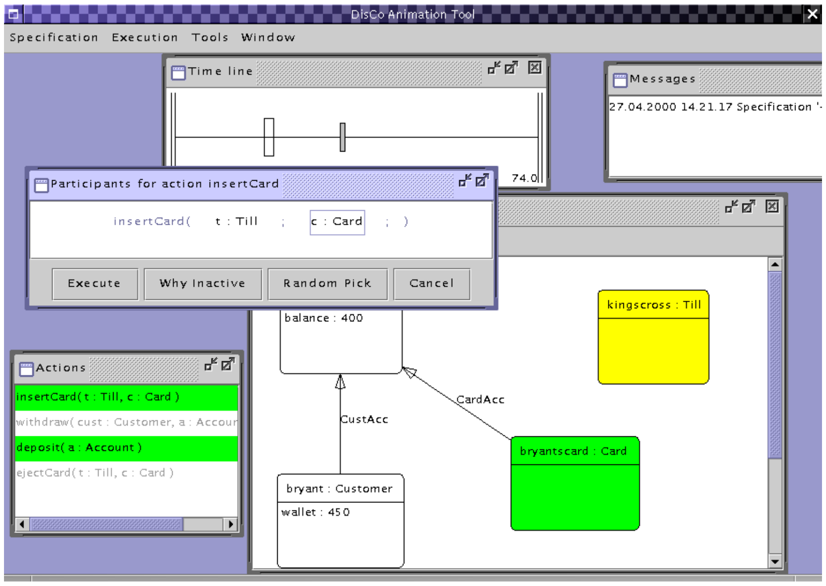
3.3 DisCo Animator

DisCo specifications can be executed and animated in Animator (Figure 2). Animation enables validation, testing and debugging of specifications and offers an enhanced means of communication for designers, application experts and customers. Animation of DisCo specifications is very visual: objects, their state and their relations are depicted as graphical objects, executed actions and state changes are visualized by animation sequences and real time is depicted by a scrolling timeline displaying current time and any set deadlines. Animator offers a graphical user interface for instantiating and executing specifications.

Java was chosen as the implementation language of Animator for easy portability of the user interface and to enable producing a WWW-based version running as an applet for demonstration purposes. Functionality of the tool is easily extensible by *hooks*—the user can supply Java code to be executed upon certain events.

Animation of a specification starts with instantiation. The user drags objects of classes she wants to instantiate onto the object view window and sets the initial values of variables using pop-ups that appear on the screen. Relations between objects are set by selecting a relation and then pointing at the objects one wants to add as relation pairs. Once the instantiation has been completed, animation may be begun. First the tool checks that the initial conditions and assertions of the specification hold for the instantiation. Then, action guards are evaluated, and enabled actions are indicated by a green highlight color. In Figure 2, the user has selected action *insertCard* of the example cash-point specification to be executed and is now picking objects to participate in its roles.

Execution traces can be rerun and saved as *scenario* files that can be processed by the Disco Scenario Tool.


DisCo Animation Tool
Specification Execution Tools Window
Time line
74.0
Messages
Participants for action insertCard
insertCard(t : Till ; c : Card ;)
Execute
Why Inactive
Random Pick
Cancel
kingscross : Till
balance : 400
Actions
insertCard(t : Till, c : Card)
withdraw(cust : Customer, a : Accour
deposit(a : Account)
ejectCard(t : Till, c : Card)
CustAcc
CardAcc
bryantscard : Card
bryant : Customer
wallet : 450

27.04.2000 14.21.17 Specification '-

Figure 2: DisCo Animator.

3.4 DisCo Scenario Tool

Message Sequence Charts (MSCs) are a widespread notation for describing inter-object communication. Their main strength is intuitive visual representation. Objects are depicted as vertical lines and messages sent to other objects as horizontal lines.

DisCo Scenario Tool (DST) is a tool for displaying execution scenarios as MSCs. Executed actions are interpreted as messages between participating objects. DST can be used to modify existing and create new MSCs which can be animated by Animator.

In Figure 3, an execution scenario of the example specification is illustrated as a MSC. In the figure, the larger window displays the MSC and the smaller window contains buttons corresponding to different instruments available for modifying the MSC.

3.5 Verification

In system design, validation and verification complement each other. The former answers the question whether we are designing the *right product* and the latter whether we are designing the *product right*. The DisCo toolset does not include any dedicated verification tool, instead a number of more general purpose verification tools can be used.

The state space of a DisCo specification is inherently infinite. Therefore, the most natural verification method is theorem proving. In [5], a mapping from a subset of the DisCo language into the logic of the PVS [8] theorem prover was described. There exists a prototype tool to support mechanical verification of invariant properties. However, the current mapping does not support verification of real-time properties.

For verification of specifications including real-time constraints, a mapping from instantiations of DisCo specifications into *timed automata* [2] was described in [1]. There are a number of model checkers that can be used to verify systems given as timed automata, for example Kronos [10] and Uppaal [7]. Currently, instantiations have to translated manually, but mechanical support could be added to the current DisCo tools in the way explained in Section 3.2.

Besides verification of real-time properties, the model checking approach enhances user-controlled mechanical theorem proving by finding counterexamples efficiently. Moreover, proposed invariants can be prechecked for specific instances before attempting to prove them for the generic specification.

4 Conclusions

The new generation DisCo toolset includes Compiler, Animator and Scenario Tool. Additional tools have been planned to assist in verification and code generation. The toolset has an extensible architecture centered around a multi-purpose intermediate

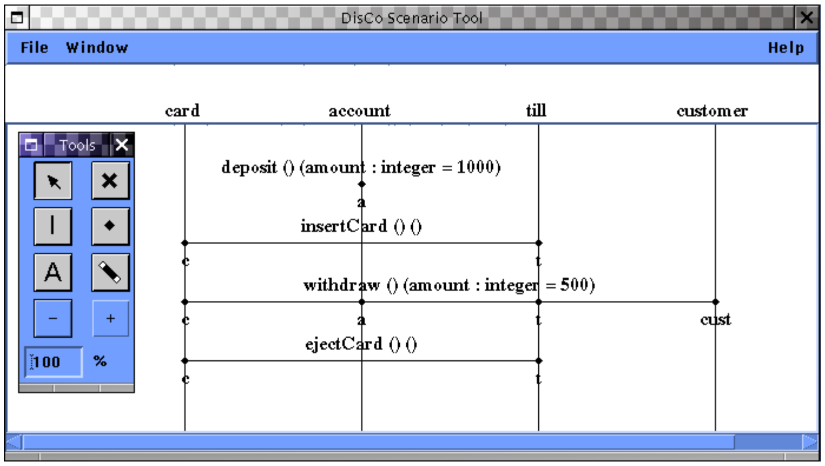


Figure 3: DisCo Scenario Tool.

form for DisCo specifications. Portability has been ensured by the use of standard C++ and Java as the implementation technologies. Usability has been a central factor in the design of the user interface.

The toolset is still under development, and at the present alpha testing stage consists of about 40,000 LOC C++ (Compiler) and 60,000 LOC Java (Animator and Scenario Tool). Altogether around ten people have contributed to the development of the toolset during a period of two and a half years.

References

[1] Timo Aaltonen, Mika Katara, and Risto Pitkänen. Verifying real-time joint action specifications using timed automata. In *Proceedings of the International Conference on Software: Theory and Practice*, Beijing, China, August 2000. To appear.

[2] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

[3] R. J. R. Back and R. Kurki-Suonio. Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, October 1988.

[4] H.-M. Järvinen, R. Kurki-Suonio, M. Sakkinen, and K. Systä. Object-oriented specification of reactive systems. In *Proceedings of the 12th International Conference on Software Engineering*, pages 63–71. IEEE Computer Society Press, 1990.

[5] Pertti Kellomäki. Verification of reactive systems using DisCo and PVS. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, number 1313 in Lecture Notes in Computer Science, pages 589–604. Springer–Verlag, 1997.

[6] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

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

[8] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, USA, June 1992. Springer-Verlag.

[9] Kari Systä. A graphical tool for specification of reactive systems. In *Proceedings of the Euromicro'91 Workshop on Real-Time Systems*, pages 12–19, Paris, France, June 1991. IEEE Computer Society Press.

[10] Sergio Yovine. Kronos: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1(1+2):123–133, 1997.

RAVEN: Real-Time Analyzing and Verification Environment[1]

Jürgen Ruf
Wilhelm-Schickard-Institute
University of Tübingen, Sand 13, 72076 Tübingen, Germany
ruf@informatik.uni-tuebingen.de
http://www-ti.informatik.uni-tuebingen/~ruf/raven.html

1 Introduction

Formal verification has become an important task in the design of systems. Techniques like symbolic model checking have reached industrial applicability. These techniques are well suited for fully synchronous systems modeled with qualitative time. If systems are embedded in a real-time environment and upper bounds for reaction times are important to guarantee a proper and save functionality, the verification of real-time properties is very important. We target at this application area with our tool **RAVEN**.

RAVEN is a real-time model checker extended by analysis algorithms. The system description is specified as a network of communicating parallel working real-time processes. Each process is a time extended finite state machine (I/O-interval structure [1,2]). The properties are specified in the quantitative temporal logic CCTL. The queries for the analysis capabilities cover minimum, maximum and maximal stability computations. **RAVEN** is able to generate counter examples and witnesses for CCTL formulas. Analysis results can be visualized by traces. All traces are graphically presented in an integrated wave form browser. Moreover, **RAVEN** offers additional checks. For instance, it can detect dead- and live locks and visualizes traces to these "locks" in its integrated wave form browser tool. It is also possible to generate random simulations of the composed system.

RAVEN uses MTBDDs for a symbolic representation of the systems [8]. This data structure results in a compact system representation and efficient verification algorithms. On some examples we have shown that this approach outperforms some state of the art tools for the verification of timed systems.

The next section gives an overview of the processing steps of the verification and the underlying architecture of the **RAVEN** system. In Section 3 we present the input language of **RAVEN**. This language is used to describe parallel working processes and to specify formal properties to prove and timing queries to analyze. Some general remarks about the implementation are given in Section 4. Afterwards we present some experimental results in Section 5.

2 Architecture

The main tasks of **RAVEN** after parsing the input file, is the construction of the MTBDDs for each process, the composition and synthesis of the MTBDD for the system transition relation. The resulting MTBDDs are then used for checking specifications and for answering timing queries. After the composition, **RAVEN** can be switched to an interactive mode allowing the user to manipulate his specifications and queries and to add new ones. The architecture of **RAVEN** is shown in figure Fig. 2.1.

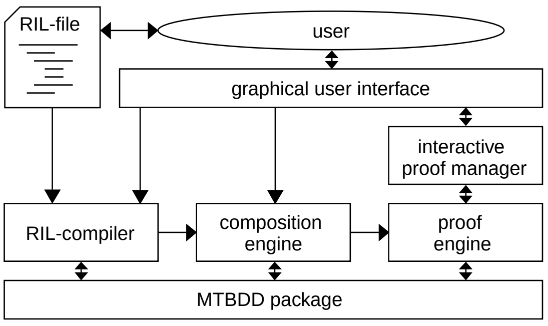
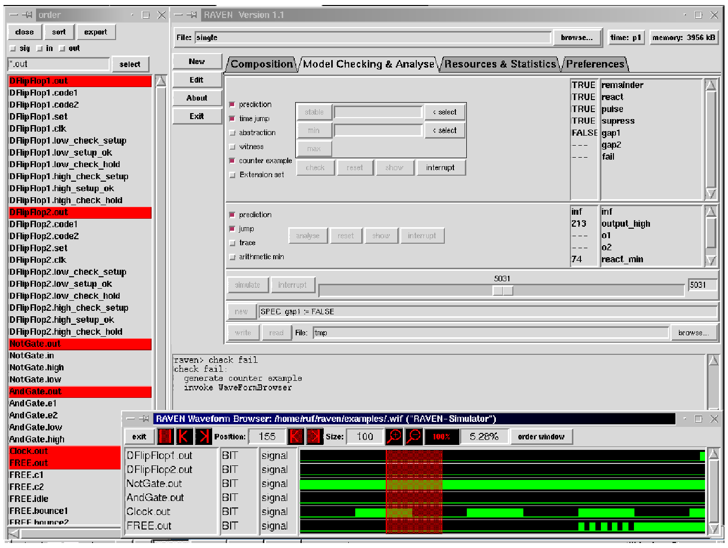


Fig. 2.1. RAVEN's architecture

After calling xraven, the graphical user interface appears. In this window the user specifies the input file and chooses some global options. Afterwards, the RIL-compiler (**RAVEN** input language, see Section 3) and the composition engine are started. After the composition is completed, **RAVEN** activates the window of the interactive proof manager. A screen shot showing the proof manager window, the wave form browser and the waveform order window is printed below. The proof manager window shows all specifications and their proof state. Also the analysis queries and their computed values are shown. New specifications or queries may be typed in this window or read in from an external file.

1.This work is sponsored by the German Research Grant (DFG-Project GRASP)



File: single   time: p1   memory: 3956 kB

*.out

DFlipFlop1.out
DFlipFlop1.code1
DFlipFlop1.code2
DFlipFlop1.set
DFlipFlop1.clk
DFlipFlop1.low_check_setup
DFlipFlop1.low_setup_ok
DFlipFlop1.low_check_hold
DFlipFlop1.high_check_setup
DFlipFlop1.high_setup_ok
DFlipFlop1.high_check_hold
DFlipFlop2.out
DFlipFlop2.code1
DFlipFlop2.code2
DFlipFlop2.set
DFlipFlop2.clk
DFlipFlop2.low_check_setup
DFlipFlop2.low_setup_ok
DFlipFlop2.low_check_hold
DFlipFlop2.high_check_setup
DFlipFlop2.high_setup_ok
DFlipFlop2.high_check_hold
NotGate.out
NotGate.in
NotGate.high
NotGate.low
AndGate.out
AndGate.e1
AndGate.e2
AndGate.low
AndGate.high
Clock.out
FREE.out
FREE.c1
FREE.c2
FREE.idle
FREE.bounce1
FREE.bounce2

TRUE	remainder
TRUE	react
TRUE	pulse
TRUE	supress
FALSE	gap1
---	gap2
---	fail

inf	inf
213	output_high
---	o1
---	o2
74	react_min

5031   5031

SPEC gap1 := FALSE

File: tmp

```
raven> check fail
check fail:
  generate counter example
  invoke WaveFormBrowser
```

RAVEN Waveform Browser: /home/ruf/raven/examples/.wif ("RAVEN-Simulator")

Position: 155   Size: 100   100%   5.28%

DFlipFlop1.out	BIT	signal
DFlipFlop2.out	BIT	signal
NotGate.out	BIT	signal
AndGate.out	BIT	signal
Clock.out	BIT	signal
FREE.out	BIT	signal

Fig. 2.2. Screenshot

3 The input format RIL

RIL (**RAVEN** input language) is a simple language for specifying networks of communicating time extended finite state machines (I/O-interval structure [2]). Each RIL module contains one I/O-interval structure. The structures are defined as state transition graphs. The transitions are labeled with time intervals and input restrictions. Inputs are functional connected to output variables of other modules. The following paragraph introduces the I/O-interval structures.

Structures are state-transition systems modeling HW- or SW-systems. The fundamental structures are Kripke structures (unit-delay structures, temporal structures) which may be derived from FSMs. Our basic models for real-time systems are interval structures, i.e., state transition systems with additional labelled transitions. We assume that each interval structure has exactly one clock for measuring time. The clock is reset to zero if a state is entered. A state may be left if the actual clock value corresponds to a delay time labelled at an outgoing transition. The state must be left if the maximal delay time of all outgoing transitions is reached. One clock tick is the lowest granularity for the time modeling. To expand interval structures by a possibility for communication, we have extended them to I/O-interval structures. These structures carry additional input labels on each transition. Such an input label is a Boolean formula over the inputs. We interpret this formulas as input conditions which have to hold during the corresponding transition times. For instance input-insensitive edges carry the formula *true* .

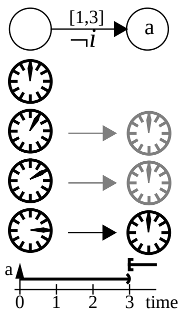


Fig. 3.1. Structure

The I/O-interval structure of the figure above may be expressed by the RIL-description shown on the left side.

```
MODULE structure
  SIGNAL a : BOOL
  INPUT i := other_module.output
  STATES
    s0 := {}
    s1 := {a}
  INIT s0
  TRANS |= s0 == !i:[1,3] ==> s1
END
```

```
MODULE sync
  SIGNAL
    a : BOOL
    b : BOOL
  INPUT i := om.output
  INIT a & b
  NEXT a' = i & b
       b' = i & a
END
```

Fig. 3.2. RIL description of a timed process and a synchronous unit-delay process

RAVEN allows to mix timed modules with full synchronous modules. The transition relations of these modules are preceded by the keyword **NEXT**. Then the transition relation is defined by a conjunctive connected sequence of boolean formulas. The usual way to specify these modules is by an transition function for each signal. The description on the right side shows an example of an synchronous module. All state changes take implicitly one unit time step

CCTL [1] is a temporal logic extending CTL [11] with quantitative bounded temporal operators. It is used to describe the real-time specifications. Two new temporal operators are introduced to ease the specification of timed properties. The syntax of CCTL is shown in (1); where $p \in P$ is an atomic proposition, $a \in \mathbb{N}$ and $b \in \mathbb{N} \cup \{\infty\}$ are time bounds. All interval operators can also be accompanied by a single time-bound only. In this case the lower bound is set to zero by default. If no interval is specified, the lower bound is implicitly set to zero and the upper bound is set to infinity. If the X-operator has no time bound, it is implicitly set to one. The semantics of CCTL is given in [2].

$$\varphi := \begin{cases} p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \mid \varphi \leftrightarrow \varphi \\ \mid \mathsf{EX}_{[a]}\varphi \mid \mathsf{EF}_{[a,b]}\varphi \mid \mathsf{EG}_{[a,b]}\varphi \mid \mathsf{E}(\varphi\ \mathsf{U}_{[a,b]}\varphi) \\ \mid \mathsf{E}(\varphi\ \mathsf{C}_{[a]}\varphi) \mid \mathsf{E}(\varphi\ \mathsf{S}_{[a]}\varphi) \\ \mid \mathsf{AX}_{[a]}\varphi \mid \mathsf{AF}_{[a,b]}\varphi \mid \mathsf{AG}_{[a,b]}\varphi \mid \mathsf{A}(\varphi\ \mathsf{U}_{[a,b]}\varphi) \\ \mid \mathsf{A}(\varphi\ \mathsf{C}_{[a]}\varphi) \mid \mathsf{A}(\varphi\ \mathsf{S}_{[a]}\varphi) \end{cases} \tag{1}$$

The semantics of CCTL is defined over runs of the given structure. A run is a sequence of configurations. A configuration is an association of an interval structure state with an clock value: $g \in S \times \mathbb{N}_0$. For the configurations of a run $r = (g_0, g_1, \ldots)$ holds either:

- that the system remains in its state: $g_i = (s_i, v_i)$ and $g_{i+1} = (s_i, v_i + 1)$
- that the system changes its state according to the transition relation and the corresponding delay times: $g_i = (s_i, v_i)$ and $g_{i+1} = (s_{i+1}, 0)$.

The semantics of the EX-operator is for instance defined through:

$$\mathfrak{I}, g_0 \models \mathsf{EX}_{[n]}\varphi \quad :\Leftrightarrow \quad \text{there exists a run } r = (g_0, \ldots) \text{ such that } \mathfrak{I}, g_0 \models \varphi \tag{2}$$

$\mathfrak{I}$ is an interval structure, g_0 is a configuration of $\mathfrak{I}$ and $\models$ is the model relation. **RAVEN** can automatically determine if $\mathfrak{I}, g_0 \models \varphi$ holds, i.e. if the given structure satisfies the given specification.

RAVEN also allows the computation of critical time delays of the given system, e.g., minimal reaction times of an embedded system or the maximal wait time of a work piece in a production automation system. For these tasks the current version of **RAVEN** supports three different algorithms:

- **MIN** requires two sets of configurations: the start and the destination configurations. Then this algorithm computes the minimal delay time which is necessary to reach a configuration of the destination starting in a configuration of the start set.
- **MAX** analogously computes the maximal delay time necessary to reach a configuration of the destination starting in a configuration of the start set.
- **STABLE** requires one set of configurations. This algorithm computes the length of the longest path which do not leave the given set.

The set of configurations are specified by CCTL formulas, e.g. if we are interested in the maximal delay time from the moment the input signal rises until the output becomes high, we may write this query as follows:

$$\mathsf{MAX}(\neg input \wedge \mathsf{EX}input, output) \tag{3}$$

4 Implementation

The main idea of the model checking algorithm is to identify (sub-)formulas of the CCTL specification with sets of configurations which satisfy them. Therefore, it is very important to have a compact representation with efficient manipulation of sets of configurations. We use an extension of characteristic functions to symbolically represent sets of configurations. The extended characteristic functions map interval structure states to the set of associated clock values. For instance, the configuration set A is represented by: $\Lambda_A : S \rightarrow \wp(\mathbb{N}_0)$ with: $\Lambda_A(s) := \{v \mid (s, v) \in A\}$.

We use MTBDDs [5,6] to represent extended characteristic functions. MTBDDs are extensions of ROBDDs [7] with more than two terminal values. In this application, the terminals carry the sets of delay times or clock values [8]. The MTBDDs support efficient operations for synthesis which we have extended to our representation.

A main operation for the model checking operations is the computation of the set of predecessor configurations of a given set. For this purpose we have adapted the relational product used in standard model checking for predecessor computation on sets of states represented by ROBDDs. Since in our application the timing information of each state is locally stored in the MTBDD leaves, we are able to apply an optimization which uses the this information to perform the predecessor computation as long as possible locally in the leaves. This technique is called time prediction [1].

Due to space limitations we do not present further details of the basic model checking algorithm, the optimizations, the composition algorithms [3], minimization heuristics [3] or the analysis algorithms [4].

5 Experimental results

As a first example we have examined the priority inheritance protocol [8] on systems with different numbers of processes. We have compared our algorithm to SMV (a CTL model checker for finite state machines [9]) and KRONOS (a TCTL model checker for timed automata [10]). The translation of interval structures to timed automata [12] is shown in Fig. 5.1. The clock has to be reset explicitly at each transition. The maximal state time has been formalized using a state invariant.

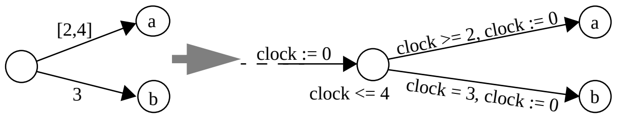


Fig. 5.1. Timed Automaton modeling an interval structure

Fig. 5.2 shows the experimental results for the three compared systems. The curve labeled with **RAVEN**+ denotes the runtime and memory requirement with time prediction. The model was not created by using composition but by an example specific construction algorithm.

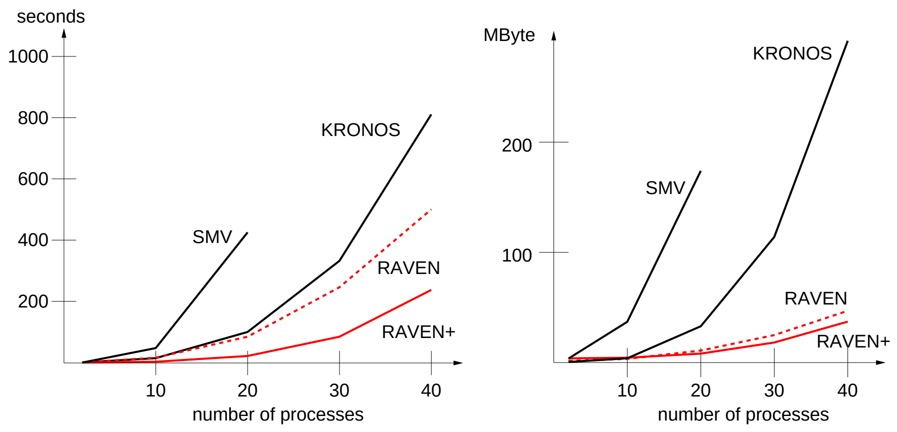


Fig. 5.2. Comparing Model Checking Results (Memory and Runtime)

The next example we examine is a system consisting of several communicating structures. An simple reader/writer system, where the writer accesses a shared memory. After writing, it signals the reader processes, that they can start their work on the memory. All reader act parallel. The run-times shown in Fig. 5.3 contain the composition an the checking time. The left part of the Fig. 5.3 shows another simple example composed by several toggling structures. A toggling structure consists of two states and two transitions connecting these states. The delay times on the transitions are in the range from 300 to 6000. All delay times are different.

The last table compares the analysis algorithms of **RAVEN** with the analysis algorithms of VERUS [13]. We compared only the min/max algorithms because VERUS do not support the stable query but on the other hand VERUS offers two queries mincount and maxcount which are actually not supported by **RAVEN**. We have chosen the same variable ordering for the signals.

The examined case studies are the single pulser [14] circuit enriched by timed gates (SP), a production cell [15] (PC) and the arbitration mechanism of the J1850 bus protocol [16]. The first two systems are widely used to compare formal methods. Details of the examined systems may be found in [2].

For the single pulser we computed the minimal and the maximal length of the output impulse. In the production cell we were interested in the minimal and the maximal time when the first work piece leaves the cell. In the J1850 example we checked the minimal and the maximal delay time when a node will leave the sending mode. The following table compares the runtimes of both tools. For the VERUS run-times we tried various options and choose the best results.

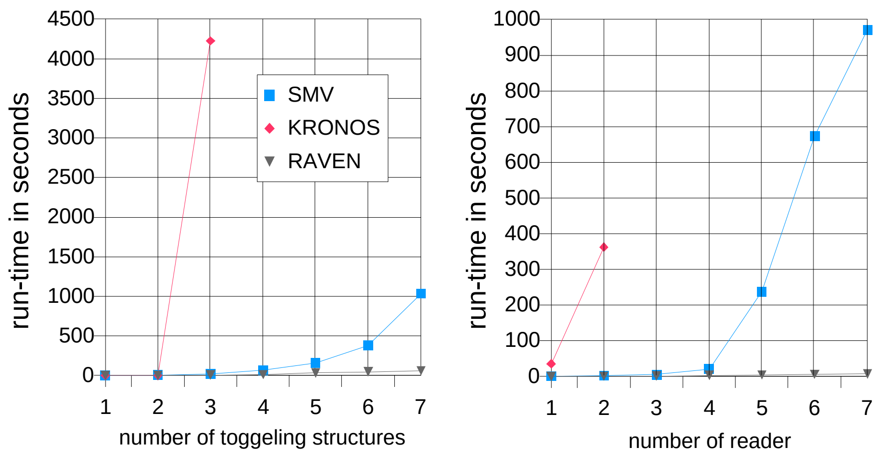


Fig. 5.3. Run-time and memory comparison of SMV, KRONOS and **RAVEN**

The run-times in the table with and without optimizations seems to show, that these techniques cause only a tiny speedup. But the run-times shown in the table contain besides the analysis times also the composition times of the structures. In all three examples the composition consumes the major part of the times (11.73 sec. for the single pulser, 2000 seconds for the production cell and 25 sec. for the J1850). If there will be more than two analysis (as computed in the examples), than the fraction of composition time to analysis time will shrink and the the optimizations will cause a larger speedup.

Table 1. Comparison **RAVEN** and VERUS

	SP	PC	J1850
VERUS	01:59.24	-[a]	-[b]
RAVEN	00:12.72	38:34	06:46
RAVEN +	00:12.42	34:04	01:35

a. VERUS error: „string table overflow“

b. VERUS memory consumption over 600MB

Bibliography

[1] J. Ruf and T. Kropf. Symbolic model checking for a discrete clocked temporal logic with intervals. In *CHARME 97*, Montreal, Canada, Oct. 1997. Chapman and Hall.

[2] J. Ruf and T. Kropf. Modeling and Checking Networks of Real-Time Systems. In *CHARME 99*, Bad Herrenalb, Germany. Springer Verlag, Septemper 1999.

[3] J. Ruf and T. Kropf. Using MTBDDs for composition and model checking of real-time systems. In *FMCAD 1998*, Palo Alto, CA. Springer Verlag 1998.

[4] J. Ruf and T. Kropf. Analyzing Real-Time Systems. In *DATE 2000*, Paris, France. IEEE Computer Society Press.

[5] E. Clarke, K. McMillian, X. Zhao, M. Fujita, and J.-Y. Yang. Spectral Transforms for large Boolean Functions with Application to Technologie Mapping. In *DAC 93*, Dallas, TX, June 1993.

[6] R. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic Decision Diagrams and Their Applications. In *ICCAD*, Santa Clara, CA, Nov. 1993. ACM/IEEE, IEEE CSP.

[7] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, August 1986.

[8] J. Ruf and T. Kropf. Using MTBDDs for discrete timed symbolic model checking. Multiple-Valued Logic – An International Journal, 1998. Gordon and Breach publisher.

[9] K.L. McMillan. Symbolic Model Checking. Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[10] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool {KRONOS}. In Hybrid Systems III, volume LNCS. Springer, 1996.

[11] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic. In ACM Symposium on Principles of Programming Languages (POPL), 1983.

[12] R. Alur and D. Dill. Automata for Modeling Real-Time Systems. In *International Colloquium on Automata, Languages and Programming, LNCS*, NY, 1990. Springer-Verlag.

[13] S. Campos, E. Clarke, and M. Minea. The verus tool: A quantitative approach to the formal verification of real-time systems. In CAV, LNCS. Springer Verlag, June 1997.

[14] S. Johnson, P. Miner, and A. Camilleri. Studies of the single pulser in various reasoning systems. In *International Conference on Theorem Provers in Circuit Design (TPCD)*, Bad Herrenalb, Germany, September 1994. Springer-Verlag, 1995.

[15] C. Lewerentz and T. Lindner, editors. *Formal Development of Reactive Systems - Case Study Production Cell*, number 891 in LNCS. Springer, 1995.

[16] SAE. J1850 class B data communication network interface. *The Engeneering Society For Advancing Mobility Land Sea Air and Space*, October 1995.

Do you trust your model checker?*

Wolfgang Reif[1] and Jürgen Ruf[2] and Gerhard Schellhorn[1] and Tobias Vollmer[1]

[1] Abt. Programmiermethodik, Universität Ulm, D-89069 Ulm, Germany
email:{reif,schellhorn,vollmer}@informatik.uni-ulm.de
[2] Wilhelm-Schickard-Institute, University of Tübingen, D-72076 Tübingen, Germany
email: ruf@informatik.uni-tuebingen.de

Abstract. In this paper we describe the formal specification and verification of the efficient algorithm for real-time model checking implemented in the model checker RAVEN. It was specified and proved using the KIV system. We demonstrate how to decompose the correctness proof into several independent subtasks and indicate the corresponding verification efforts. The formal verification revealed some errors, reduced the code size, and improved the efficiency of the implementation.

1 Introduction

Model checking is an important technique to detect errors or to prove their absence in safety critical soft- and hardware systems. Model checking automatically verifies properties of state based systems. For efficiency, it is usually implemented using highly optimized data structures and algorithms. On the other hand, when a property can be shown, the only result we usually get from a model checker, is a "yes". The absence of a comprehensible proof raises the question: can the model checker be trusted?

In this paper, we will answer this question for the case of the real-time model checker RAVEN [9]. RAVEN uses time-extended finite state machines (interval structures) to describe systems and a timed version of CTL (clocked CTL, CCTL) to describe their properties. Optimized algorithms based on extended characteristic functions are used to compute the extension sets in the model checker.

Our solution consists in the application of formal methods to ensure the correctness of formal methods: We apply the interactive specification and verification system KIV to formalize and prove the algorithms of RAVEN. To our knowledge, this is the first case study tackling formal verification of a state-of-the-art real-time model checker. This paper is the result of the cooperation of two groups, in the context of a research programme on formal methods for engineering applications*: the developer of RAVEN [10, 12] (second author), and the development group of KIV [8] (remaining authors).

* This work is supported by the DFG (Deutsche Forschungsgemeinschaft) under the priority programme "Integrating Software Specification Techniques for Engineering Applications"

The KIV case study described in this paper consists of four steps: first, we define a formal specification of the semantics of CCTL, the basic model checking algorithm and the optimizations. Second, we verify the correctness of the simple and the optimized algorithm. Third, we give an efficient implementation of the abstract algorithms based on bitvectors, and finally we prove the implementation correct. This implementation relies on a standard software package for extended characteristic functions [5] (based on multiterminal BDDs, MTBDDs [1, 3]). We formalized the interface to this package in KIV. The verification relies on the specification of this interface. Verifying [5] against the interface is an independent subtask which was not part of the case study.

Our case study shows that it is possible to give a modular specification, such that the correctness of the model checker can be split into several independent verification tasks. With the help of the correctness proofs we found some critical definition errors in the formal specification of the optimizations. After correcting them we proved that the basic algorithms and the optimizations using bitvectors are correctly implemented. Parts of the code were shrinked. One prediction function (see Sect. 5.3) worked too pessimistic and was optimized.

In Section 2, we will describe interval structures and the logic CCTL, which constitutes the basis of RAVEN. Section 3 discusses the used optimizations and the efficient implementation. Section 4 gives our approach to formalization and verification. An overview over the specifications and correctness proofs is presented in Section 5. Section 6 concludes the paper.

2 Real-Time Model Checking

Model checking is a well established method for the automatic verification of finite state systems. It checks if a given state transition system satisfies a given property specified as a propositional temporal logic formula.

The approach we will examine is developed for timed systems and timed specifications. It is presented in [10, 12]. In this section, we will explain the main ideas behind the model checking verification technique which are necessary for the remaining part of the paper. First we will present the formal model and the temporal logic. Afterwards we will introduce the representation with extended characteristic functions and the main model checking procedure.

2.1 Interval Structures

Interval structures are finite state transition systems. The transitions are labeled with intervals of natural numbers to represent delay times. The structures use the notion of clocks to represent time: every structure contains exactly one clock working in a discrete time domain. A transition is enabled if the actual clock value is within the interval of an outgoing transition. The successor state as well as the delay time is chosen indeterministically w.r.t. the transition relation and the labeled delay intervals. The clock is reset if a transition fires.

Definition 1. *An interval structure $\mathcal{J} = (P, S, S_0, T, L)$ is a tuple with a set of atomic propositions P, a set of states S, a set of initial states S_0, a function $T : S \times S \to \wp^{\omega}(\mathbb{N}_0)$ that connects states with labeled transitions and a state labeling function $L : S \to \wp(P)$.*

Every state of an interval structure must be left after the maximal state time: $MaxTime(s) := \max\{\ v \mid \exists s'.v \in T(s, s')\}$

Besides the states, we now also have to consider the currently elapsed time to determine the transition behavior of the system. Hence, the actual configuration of a system is given by an interval structure state $s \in S$ and the actual clock value $v \in \mathbb{N}_0$. The set of all configurations of an interval structure is given by: $G_{\mathcal{J}} = \{\ (s, v) \mid s \in S \wedge v \leq MaxTime(s)\}$

The semantics of interval structures is defined over runs. A run is a sequence of configurations $r = (r_0, r_1, \ldots)$ with $r_i = (s_i, v_i) \in G_{\mathcal{J}}$ and for all $i \geq 0$ holds either

- $r_{i+1} = (s_i, v_i + 1)$ and $v_i < MaxTime(s_i)$ or
- $r_{i+1} = (s_{i+1}, 0)$ and $v_i \in T(s_i, s_{i+1})$

In the following, we call (s_i, v_i) the local predecessor of $(s_i, v_i + 1)$, which corresponds to the first case of the definition. Similar, we call (s_i, v_i) a global predecessor of $(s_{i+1}, 0)$ if $v_i \in T(s_i, s_{i+1})$. Note that for a set of configurations, only the computation of global predecessors depends on the transition relation T.

2.2 CCTL

CCTL (Clocked Computation Tree Logic) is a propositional temporal logic using quantitative time bounds for expressing real time properties (e.g. bounded liveness). The following definition describes the syntax of CCTL formulas.

Definition 2. *Given a set of atomic propositions P. The set of CCTL formulas F_{CCTL} is defined to be the smallest set with*

- $P \subseteq F_{CCTL}$
- *if* $m \in \mathbb{N}_0, n \in \mathbb{N}_0 \cup \{\infty\}$, $m \leq n$ *and* $\varphi, \psi \in F_{CCTL}$ *then*
  $\neg\varphi$, $\varphi \wedge \psi$,
  $\mathsf{EX}_{[m]}\varphi$, $\mathsf{EF}_{[m,n]}\varphi$, $\mathsf{EG}_{[m,n]}\varphi$, $\mathsf{E}(\varphi\mathsf{U}_{[m,n]}\varphi)$, $\mathsf{E}(\varphi\mathsf{S}_{[m]}\varphi)$, $\mathsf{E}(\varphi\mathsf{C}_{[m]}\varphi)$
  $\mathsf{AX}_{[m]}\varphi$, $\mathsf{AF}_{[m,n]}\varphi$, $\mathsf{AG}_{[m,n]}\varphi$, $\mathsf{A}(\varphi\mathsf{U}_{[m,n]}\varphi)$, $\mathsf{A}(\varphi\mathsf{S}_{[m]}\varphi)$, $\mathsf{A}(\varphi\mathsf{C}_{[m]}\varphi) \in F_{CCTL}$
  The symbol ∞ *is defined through:* $\forall\ i \in \mathbb{N}_0 : i < \infty$.

All interval operators can also be accompanied by a single time-bound only. In this case the lower bound is set to zero by default. If no interval is specified, the lower bound is implicitly set to zero and the upper bound is set to infinity. If the X-operator has no time bound, it is implicitly set to one.

For this paper we will only define the semantics for the EF-operator, the semantics for the other operators may be found in [12]. The semantics of CCTL is given by a model relation ($\models$):

Definition 3. *Given the interval structure $\mathcal{J}= (P, S, S_0, T, L)$, a starting configuration $r_0 \in G_{\mathcal{J}}$ and the CCTL formula $\varphi \in F_{CCTL}$.*

$$\begin{aligned} \mathcal{J}, r_0 \models \mathsf{EF}_{[m,n]}\varphi :\Leftrightarrow\ & \textit{there exists a run } r = (r_0, \ldots) \\ & \textit{and an } i \in [m,n] \textit{ such that } \mathcal{J}, r_i \models \varphi \end{aligned}$$

A formula φ is valid in a model $\mathcal{J}$, iff $\mathcal{J}$, $(s,0) \models \varphi$ for all initial states $s \in S_0$. The defined interval operator may be expressed by operators only carrying an upper time bound, e.g. $\mathsf{EF}_{[m,n]}\varphi := \mathsf{EX}_{[m]}\mathsf{EF}_{[n-m]}\varphi$.

2.3 The basic model checking algorithm

The main idea of model checking algorithms is the following: building the syntax graph of the formula to check, computing bottom-up sets of configurations representing sub formulas (called extension sets), checking if the set of initial states is a subset of the extension set of the complete formula.

The computation of the extension sets is done by a function *ext*. The extension sets of atomic formulas (i.e. the leaves of the syntax tree) are given by the labeling function and the possible clock values in the appropriate states of the interval structure $\mathcal{J}$. Boolean connections can be computed by applying the corresponding set operations on the extension sets. Finally, the computations of the extension sets of temporal logic operators are defined recursively. We will present them using the EF-operator as an example:

$$\begin{aligned} &\text{ext}(\mathsf{EF}_{[0]}\varphi) := \text{ext}(\varphi) \\ &\text{ext}(\mathsf{EF}_{[\text{n}+1]}\varphi) := \text{ext}(\varphi) \cup \text{ext}(\mathsf{EX}(\mathsf{EF}_{[\text{n}]}\varphi)) \end{aligned}$$

The operator EX ("Next") states, that a certain formula is fulfilled after the next step. The time bound n for $\mathsf{EF}_{[n]}$ has to be finite. For $n = \infty$ we can show, that the extension sets reach a fixpoint, i.e. $\exists\ i.\ \textit{ext}(\mathsf{EF}_{[i]}) = \textit{ext}(\mathsf{EF}_{[i+1]})$. This means, that the recursion can be terminated if the $\textit{ext}(\mathsf{EF}_{[i]})$ will not change anymore. The realization of the recursive definition of the EF-operator leads to the algorithm shown in Fig. 1.

The other CCTL operators can be implemented with similar algorithms. The operators EX, ES, EC, which do not reach a fixpoint during computation can not be computed for $n = \infty$.

Two questions remain: how are the sets of configurations represented in order to achieve efficient computations and what how is the EX-operator computed?

A main advance in the field of model checking was made with the introduction of symbolic representations of state spaces [2]. Instead of an explicit enumeration of sets of states, they are represented by characteristic functions. For our time extended model checking algorithm, we use an extension of characteristic functions (ECF) which maps interval structure states to sets of associated clock values [10]. The function $\Lambda_A : S \to \wp(\mathbb{N})$ represents a set $A \subseteq G_{\mathcal{J}}$ of configurations:

```
**confset** EF(**confset** C, **natinfty** n, **transrel** T)
**begin**
   **confset** old := $\emptyset$, R := C;
   **while** n > 0 $\wedge$ old $\neq$ R **do**
      old := R;
      res := C $\cup$ EX(R,1,T);
      n := n − 1;
   **end**
   **return** R;
**end**
```

Fig. 1. The basic EF-algorithm

$\Lambda_A(s) := \{ v \mid (s,v) \in A \}$

In the following, we will use sets of configurations and the ECFs representing these sets synonymous, i.e. we will write A instead of Λ_A. Since we aim at verifying an implementation of a model checker we will furthermore use the notation and intuition of MTBDDs (multi-terminal BDDs [1, 3]), which are used to implement extended characteristic functions.

MTBDDs representing sets of configurations code the state space in the decision diagram and associate a set of natural numbers representing the clock values with each state (i.e. each leaf of the decision diagram). An example MTBDD representing the configuration set $\{(a,3),(a,4),(b,5),(ab,2)\}$ is depicted in the dashed box in Fig. 2. Set operations can be implemented by performing the appropriate operations on the leaves of the MTBDD. The transition relation is represented in a similar way. As shown in Fig. 2, two MTBDDs are cascaded, the first representing the state space of the model $\mathcal{J}$ and the second representing the sets of predecessor configurations for the individual states.

The second open question is, how can the extension set of an EX-operator be computed? If the extension set of its argument formula is known, the extension set of the EX-operator is given by all predecessor configurations.

The local predecessors of a set of configurations may be computed by a function *local-pre* as follows. Using MTBDDs, the computation may be reduced to the leaves of the MTBDD where each clock value contained is decremented.

Global predecessors only exist for configurations with a zero clock value. The computation is done by a function *global-pre*(C,T). For every state s' with $(s',0) \in C$, the set of predecessor configurations is looked up in the transition relation and the resulting configuration sets are combined to a new configuration set.

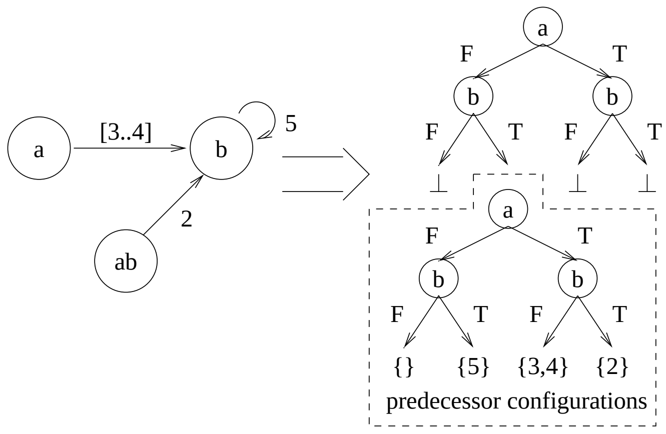


Fig. 2. Representation of the transition relation using two cascaded MTBDDs

3 The Central Idea of the RAVEN Model Checker

3.1 Time Prediction

If we analyze the predecessor computation we observe the following:

- The global predecessor computation is more expensive than the local predecessor computation since it takes the complete transition relation into account.
- The global predecessors often do not change between two predecessor computations.

We use a technique called time prediction to overcome the single step traversal and to avoid unnecessary global predecessor computations [10, 11]. The idea is to define a time prediction function that computes how many steps the global predecessors stay constant.

Again, we exemplarily discuss the EF-operator. Although the basic idea of time prediction can be applied to all operators, every temporal operator needs a separate prediction function.

The time prediction function *predict-EF* is computed locally, i.e. for each state separately by a function *local-pr-EF*. The minimum of the prediction times *mp* is the time span which can elapse without any change in the set of the global predecessors. Arguments of *local-pr-EF* are the sets of clock values $c \subseteq \mathbb{N}_0$ and $g \subseteq \mathbb{N}_0$ which contain the last interim result of the computation and the results of the last computation of global predecessors:

$$\text{predict-EF(C,G)} = \min_{s \in S}\text{local-pr-EF(C(s),G(s))}$$

$$\text{local-pr-EF}(c, g) := \begin{cases} \text{v} & \text{if v} = \min(\text{c,g} - 1) \wedge \text{v} > 0 \\ \infty & \text{otherwise} \end{cases}$$

The set operation $g - 1$ decrements all members of g by one.

After the prediction the fixpoint iteration of the temporal operators may be performed *mp* times. Analogous to the above, a function *apply-EF* is defined which performs the fixpoint iteration locally for every state by using the recursively defined function *local-EF*.

$$\text{apply-EF(C,G,mp)(s)} = \text{local-EF(C(s),G(s),mp)}$$

$$\text{local-EF(c,g,mp)} = \begin{cases} \text{c} & \text{if mp} = 0 \\ \text{local-EF(c,g,mp} - 1) \cup \text{local-EF(c,g,mp} - 1) - 1 \cup \text{g} \end{cases}$$

Putting together the above definitions and considerations, we obtain the optimized algorithm shown in Fig. 3.

```
confset EF'(confset C, natinfty n, transrel T)
begin
   confset old := ∅, R := C, G := ∅;
   natinfty p;
   while n > 0 ∧ old ≠ R do
      old := R;
      G := global-pre(R,T);
      p := predict-EF(R,G);
      if p > n then p := n;
      R := apply-EF(R,G,p);
      n := n − p;
   end
   return R;
end
```

Fig. 3. The optimized EF-algorithm

3.2 Time Jumps using Bitvectors

The local fixpoint iteration needs O(*mp*) set operations for execution. We define a technique called time jumping which replaces this iterative execution by an efficient implementation using either bitvectors or interval lists.

Using interval lists has the advantage of little memory consumption but lacks the efficiency of the bitvector based algorithms. Hence, we now will concentrate on the bitvector based implementation.

The implementation *local-EF#* for the EF-operator using bitvectors is shown in Fig. 7. Bitvectors are used to represent sets of natural numbers. A the number n is contained in a set, iff the nth bit of the bitvector representation is 1. The basic idea of the implementation of *local-EF#* is to traverse bitvectors: in the nth step of the algorithm the nth bit of the input and an internal state of the algorithm are used to compute the nth bit of the result. Hence, we only need a single specialized operation to compute *local-EF* instead of $O(mp)$ set operations when using the recursive definition.

4 Formal Specification and Verification Concept

Our case study consists of five parts:

1. specification of CCTL and its semantics. The left half of Fig. 4 illustrates this step.
2. specification of the basic model checking algorithm and verification, that the algorithm implements the semantics (cf. right half of Fig. 4).
3. specification of the optimized algorithms with time prediction for the temporal operators (e.g. *EF′* as defined in Fig. 3) and verification that they give the same results as the simple recursive variant (e.g. *EF* as given in Fig. 1). Figure 5 visualizes this step.
4. nonrecursive definition *local-EF′* of the local computations *local-EF* used in *EF′* and verification, that both definitions are equivalent (cf. upper half of Fig. 6).
5. implementation of *local-EF′* based on bitvectors and correctness proof for the implementation. The lower half of Fig. 6 shows this part.

These five parts will be discussed in the five subsections of the next section. All five parts were specified using the structured, algebraic specifications of KIV, which are similar to the standard algebraic specification language CASL [4]. To do the correctness proofs, KIV offers a concept of modules, which describe a refinement relation between specifications. KIV automatically generates proof obligations that assure the correctness of modules.

An important goal in the design of the case study was to structure specifications, such that each of the four verification steps could be expressed as the correctness of one module. This has two advantages: First, each module can be verified independently of all others, which means that the correctness problem is decomposed into several orthogonal subproblems. Second, a general theory (see [6]) of modular systems (consisting of specifications and modules) assures, that a correct model checking algorithm (that implements the $\models$ predicate) can be "plugged" together by repeatedly replacing abstract definitions with more concrete versions: Starting with the unoptimized algorithm *modelcheck*, first *EF* is replaced with *EF′* (and similar for the other temporal operators), then the call to *local-EF* in *EF′* is replaced with a call to *local-EF′*, and finally this call is replaced again with the bitvector implementation. The final algorithm is identical to the one used in RAVEN, except that it has an abstract interface of extended characteristic functions. Their implementation using actual BDD operations could be added using another module (which could be separately verified).

Developing a modular system of specifications and modules, such that "plugging the algorithm together" and separate verification of the steps described above became possible, was a major creative step in this case study. Two important design decisions were to use higher-order operations on ECFs (*apply* and *reduce*, see Sect. 5.3) to have an abstract interface to BDDs, and to use the intermediate nonrecursive definition *local-EF'* (see Sect. 5.4).

Technically, modular systems are developed in KIV using a graphical representation, called development graphs. Such a graph contains rectangular boxes for specifications and rhombic boxes for modules. Arrows represent the structure of specifications and implementation relations. The following section will show for each of the five steps some relevant part of the development graph and sketch the contents of the specifications. Putting all parts together, i.e. merging the development graphs of figures 4, 5 and 6 gives the full modular specification and implementation of the model checker. Full details on all specifications, modules and proofs can be found in [13].

5 Verification of Correctness

5.1 Specification of CCTL Semantics

The structure of the algebraic specification for CCTL and its semantics is shown in the left half of Fig. 4. The main predicate specified in the top-level specification is $\mathcal{J}$, $(s,v) \models \varphi$ (φ holds in configuration (s,v) over model $\mathcal{J} = (T,L)$[1]). Two typical axioms of this specification are

$$\begin{aligned}&\mathcal{J},\ (\mathrm{s,v}) \models \mathsf{EF}_{[\mathrm{n}]}\ \varphi\\ &\leftrightarrow \exists\ \mathrm{r.\ run(r,T)} \wedge \mathrm{first(r)} = (\mathrm{s,v}) \wedge \mathcal{J},\ \mathrm{r}_i \models \varphi\end{aligned}$$

$$(\mathrm{T,L}),\ (\mathrm{s,v}) \models \mathrm{p} \leftrightarrow \mathrm{p} \in \mathrm{L(s)}$$

The definition is based on a specification of the data type of CCTL formulas, the specification *transrel* of the transition relation of an interval structure and (indirectly) on the specification *confsets* of configuration sets (in algebraic terms, the top-level specification is an enrichment of *transrel* and *confsets*). The transition relation is defined as a function $T: \mathit{state} \to (\mathit{state} \to \mathit{set}(\mathit{nat}))$. $T(s')(s)$ gives the possible delay times for a transition from state s to state s'. Configuration sets are specified as functions $C: \mathit{state} \to \mathit{set}(\mathit{nat})$. A configuration set C contains a configuration (s,v), iff $v \in C(s)$. This representation corresponds to extended characteristic functions.

Both configuration sets and the transition relation are specified as actualizations of a generic datatype of extended characteristic functions $\mathit{ecf}: \mathit{state} \to \mathit{elem}$: the parameter type *elem* is instantiated by *set*(*nat*) and *confset* respectively. The use of generic ECFs allows us to be fully abstract in our correctness analysis of the model checking algorithms, while it is still possible to implement ECFs with MTBDDs (with *elem* being the type of the BDD leaves) and to verify this implementation separately.

[1] The set P of atomic propositions, and the set S of states are carrier sets in our algebraic specification. Therefore they need not be explicitly mentioned in the definition of a model

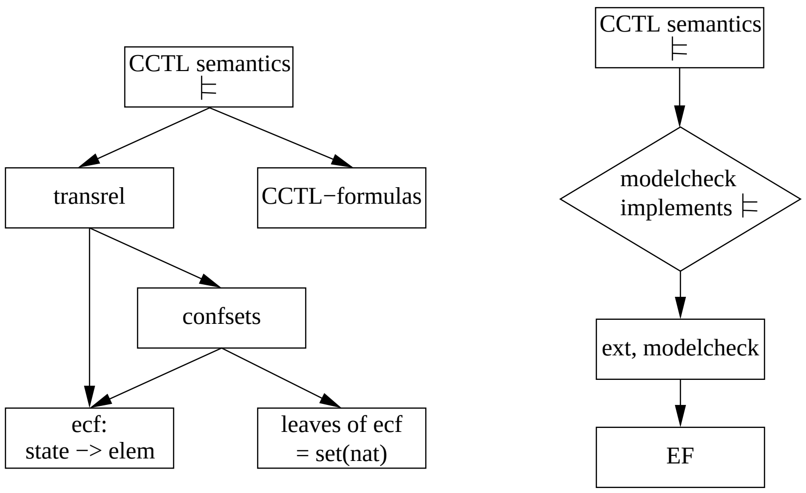


Fig. 4. Specification of CCTL Semantics

5.2 Correctness of Simple Model Checking

The right half of Fig. 4 shows the part of the KIV development graph that deals with the correctness proof of the simple model checking algorithm with respect to the semantics of CCTL (the structure of the specification of the CCTL semantics above the module has now been omitted).

Specification of Simple Model Checking The specification shown below the module in Fig. 4 contains the simple model check algorithm *modelcheck*. It is specified by the following axioms:

$$\text{modelcheck}(\mathcal{J},\ \text{c},\ \varphi) \leftrightarrow \text{c} \in \text{ext}(\varphi,\ \mathcal{J}),$$

$$\text{ext}(\mathsf{EF}_{[\text{n}]}\ \varphi,(\text{T},\text{L})) = \text{EF}(\text{ext}(\varphi),\text{n},\ \text{T}),$$

$$\text{ext}(\neg\ \varphi,\mathcal{J}) = \text{G}_{\mathcal{J}} \setminus \text{ext}(\varphi,\mathcal{J}),$$

$$(\text{s},\text{v}) \in \text{ext}(\text{p},(\text{T},\text{L})) \leftrightarrow \text{p} \in \text{L}(\text{s}) \wedge (\text{s},\text{v}) \in \text{G}_{(\text{T},\text{L})},$$

$$\ldots$$

Again, the specification is based on configuration sets and the transition relation (not shown in the figure). It is also based on a subspecification which defines a tail-recursive function *EF*, which computes the extension set of the temporal operator $\mathsf{EF}_{[n]}$. The specification also contains similar functions for the other temporal operators, but like in the previous sections, we will now concentrate on the implementation of *EF*. We have preferred the tail-recursive version over the program given in Fig. 1 for two reasons: First, the specification remains independent of an implementation language. Second, proofs using the tail-recursive function are smaller compared to proofs using while-loops and invariants.

To define the computation of local and global predecessors, two generic higher-order operations *apply* and *reduce* on ECFs are used:

$$\text{local-pre(C)} = \text{apply(C, } -1)$$

$$\begin{aligned}&\text{global-pre(C,T)}\\ &= \text{reduce(T, } (\lambda\ \text{C}_0\text{,s}'.\ \text{if } 0 \in \text{C(s}') \text{ then } \text{C}_0 \text{ else } \emptyset), \cup, \emptyset)\end{aligned}$$

apply(ecf, f) applies function f on each leaf of *ecf*. *reduce*(ecf, f, f', a) applies the function f on each leaf of *ecf* and combines the results using function f', starting with the value a. The function -1 used in the definition of *local-pre* decrements each number contained in a leaf of an ECF and drops zeros. The λ expression contained in the definition of *global-pre* looks up the predecessors of all states that contain a zero clock value.

Proof of Correctness The KIV module shown in the development graph of Fig. 4 automatically generates proof obligations. Their verification guarantees, that the simple model checking algorithm *modelcheck* satisfies the axioms of the predicate $\models$, i.e. that *modelcheck* implements the predicate $\models$. Proving the proof obligations is straightforward using theorems that assure the existence of fixpoints for the operators EF, EG, etc. and takes only a few hours of verification time.

5.3 Time Jumps and Time Prediction

Figure 5 shows the part of the development graph that is relevant for the verification of the optimization step that introduces time prediction and time jumps.

Specification of Time Prediction Most parts of the specification of the optimized version *EF′* of the computation (cf. Fig. 3) can be adopted from the unoptimized algorithm. The functions required to specify time prediction and time jumps, *predict-EF* and *apply-EF* are defined using the functions *apply* and *reduce*:

$$\text{apply-EF(ecf)} = \text{apply(ecf,local-EF)}$$

$$\text{predict-EF(ecf)} = \text{reduce(ecf,local-pr-EF,min,}\infty)$$

Thus, the functions are reduced to functions *local-EF* and *local-pr-EF* which work on the leaves of the ECFs. The specification of the latter functions follows directly the definition in Sect. 3.

Proof of Correctness To prove correctness of the module, it must be shown, that the axioms of the simple algorithm are satisfied by the optimized algorithm. The main theorem needed in the proof is:

$$\begin{aligned}&\text{n} < \text{predict-EF(C,global-pre(C,T))}\\ \rightarrow\ &\text{global-pre(C,T)}\\ &= \text{global-pre(apply-EF(C,global-pre(C,T),n),T)}\end{aligned} \tag{1}$$

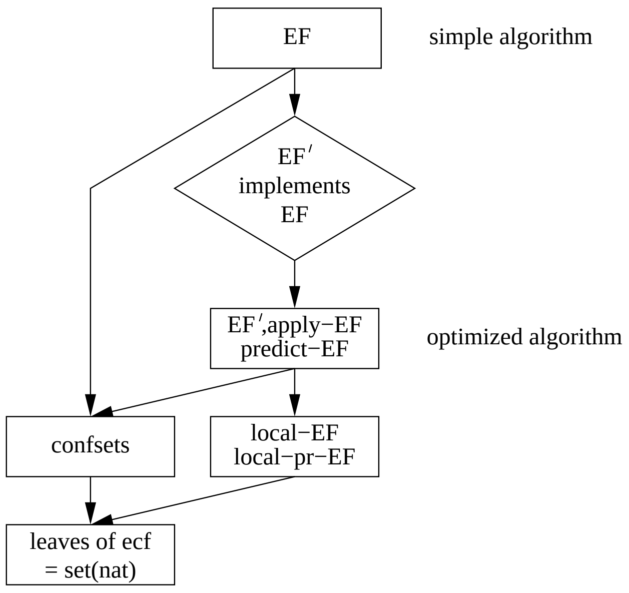


Fig. 5. Specification of time-prediction

It expresses the central idea of the optimization step, that the global predecessors do not change during the computation of as many steps as the time prediction permits. Since *predict-EF* yields the minimum of the results of the predicted values for all leaves, the proof can be done by reducing the theorem to the leaves of the ECF considered and proving the analogous theorem for each single leaf:

$$\begin{aligned} &\text{p} = \text{local-pr-EF(C(s),G(s))} \wedge (\text{n} < \text{p} \vee \text{p} = \infty) \\ &\rightarrow (0 \in \text{local-EF(C(s),G(s),n)} \leftrightarrow 0 \in \text{C(s)}) \end{aligned} \tag{2}$$

Here, the term $0 \in$ *local-EF*() $\leftrightarrow$ …states that the global predecessors of the leaf considered remain unchanged. For reasons explained in the next section, we assume property (2) as an axiom here and postpone its proof until then.

The proof obligation for the operator EF is proved by induction over the number of steps the operator computes. Two important theorems are needed in the induction step of the proof.
The first,

$$\text{EF}'(\text{C,n} + 1,\text{T}) = \text{EF}'(\text{EF}'(\text{C,n,T}),1,\text{T}) \tag{3}$$

ensures, that a single step can be split off from the computation of EF'. The proof is conducted by induction over n. Expanding the definition of a single recursion "computes" p steps of the operator using function *apply-EF*. Since only one step has to be split off, a similar decomposition lemma is also needed for *apply-EF*. This lemma can be shown by proving the following, analogous lemma for the leaves of the ECF:

$$\text{local-EF}(c,g,n+1) = \text{local-EF}(\text{local-EF}(c,g,n),g,1) \tag{4}$$

The second important theorem is required, because the recursion schemes of the simple and of the optimized version are slightly different. While the simple recursion $EF(C,n+1,T) = C \cup EF(C,n,T)$ always adds its argument C to the interim result, the optimized version calls *apply-EF*(R,p) with the interim result R to compute p steps. Hence, in any step of the algorithm, the result of the last major step is added to the configuration set. Since the operator considered increases monotonous, both recursion schemes produce the same results:

$$\begin{array}{l} \quad \text{R} = \text{EF}'(\text{C,n,T}) \\ \rightarrow \text{R} \cup \text{EX(R,1,T)} = \text{C} \cup \text{EX(R,1,T)} \end{array} \tag{5}$$

Results of Verification. During verification, we found erroneous time prediction functions for two operators, *local-pr-EU* (for strong-until) and *local-pr-ES* (for the successor operator). In some cases, the original definition of *local-pr-EU*,

$$\text{local-pr-EU}(c_1,c_2,g) = \begin{cases} n & \text{if } n = \min(c_1,g+1) \wedge [0,n] \subseteq c_2 \\ \infty & \text{otherwise} \end{cases}$$

yields too high values. Therefore the algorithm sometimes forgets to recompute the global predecessors, which leads to incorrect results. The corrected version of *local-pr-EU* (corrections **bold**) is

$$\text{local-pr-EU}(c_1,c_2,g) = \begin{cases} n \;\; \text{if} \;\; \neg\, \mathbf{0} \in \mathbf{c_1} \wedge n = \min(c_1,g+1) \\ \qquad\quad \wedge\, [0, n - \mathbf{1}] \subseteq c_2 \\ \infty \;\text{otherwise} \end{cases}$$

Inspection of both implementations of RAVEN showed, that the bitvector based version behaves correctly. The implementation using interval lists contained the error and was subsequently corrected.

The time prediction *local-pr-ES* not only contained a case where too high results were computed, but also a too pessimistic case. Since the definition of *local-pr-ES* was rather complex (8 lines), we corrected it by reduction to the EX operator. This led to a much more compact definition using only 3 lines.

The verification also showed a critical point in the computation of $\mathsf{EX}_{[n]}\ \varphi$ (next operator), that was not detected during design and informal analysis of the algorithm. Normally, the EX operator, which does just computations of predecessors, never reaches a fixpoint. Nevertheless, cycles (i.e. $\mathsf{EX}_{[m]}\ \varphi = \mathsf{EX}_{[m+p]}\ \varphi$ may occur in the computation. To stop the computation as early as possible, the simple version of EX (which is similar to the algorithm in Fig. 1) performs a fixpoint test after every step of the computation by comparing $\mathsf{EX}_{[m]}\ \varphi$ to $\mathsf{EX}_{[m+1]}\ \varphi$. The optimized version contains this fixpoint test, too. But since the optimized version computes p steps at a time, $\mathsf{EX}_{[m]}\ \varphi$ is compared to $\mathsf{EX}_{[m+p]}\ \varphi$. Therefore, if the computation of $\mathsf{EX}_{[n]}\ \varphi$ with $n > m + p$ contained such a cycle of length p, the computation would stop too early.

However, we could show, that this situation can never occur, because the time-prediction function permits either an infinite number of steps or only one step at a time if a cyclic computation takes place:

$$\begin{array}{l} \text{apply-EX(C,G,n)} = \mathrm{C} \land \mathrm{p} = \text{predict-EX(C,G)} \land \mathrm{n} \le \mathrm{p} \\ \to \mathrm{p} = 1 \lor \mathrm{p} = \infty \end{array} \tag{6}$$

Verification Effort. All seven temporal operators implemented in RAVEN were proven correct. We found, that the proofs of all operators share a common pattern. This pattern consists of theorems (1), (2), (3), (4), (5) and some auxiliary theorems. Additionally, for operators, that do not reach a fixpoint during computation, a theorem like (6) was needed. Due to this pattern, the verification time required decreased from one week for the first operator to about 2 days. Although the complexity of the operators increased, the correctness proofs all have about the same length, because the growing experience helped to compensate the extra complexity with higher automation.

5.4 Nonrecursive Representation of Time Jumps

A look at the implementation of time jumps (cf. Fig. 7) and time prediction shows, that the computations of these programs do not fit to the recursive definition of *local-EF* very well. Therefore, even simple proofs using the recursive definition are technically very complex. Therefore we decided to introduce a nonrecursive function *local-EF'*, which describes the results of the operator EF:

$$\begin{array}{ll} & \text{local-EF}'(\mathrm{c,g,n}) \\ = & \{\mathrm{v} \mid \exists \mathrm{v}_0 \in \mathrm{c}\ \cap\ [\mathrm{v}, \ldots, \mathrm{v} + \mathrm{n}]\} \\ & \cup\ \{\mathrm{v} \mid \exists \mathrm{v}_0 \in \mathrm{g}\ \cap\ [\mathrm{v}, \ldots, \mathrm{v} + \mathrm{n} - 1]\ \land\ \mathrm{n} \neq 0\} \end{array} \tag{7}$$

Since we want to use this function instead of the recursive function *local-EF*, we first have to prove the equivalence of both representations. Again, this is done using a KIV module. The corresponding part of the development graph is depicted in the upper half of Fig. 6. The proof obligations generated for the module ensure, that the nonrecursive function *local-EF'* satisfies the axioms of *local-EF*.

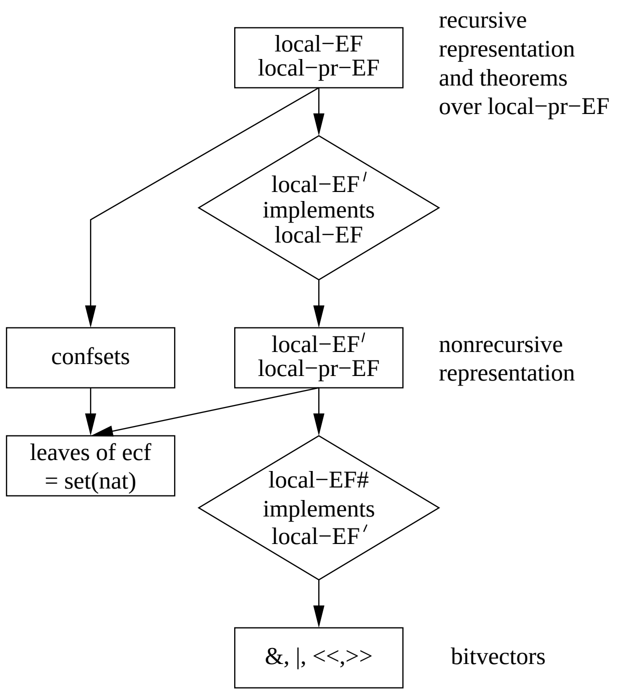

local−EF
local−pr−EF
recursive representation and theorems over local−pr−EF
local−EF′ implements local−EF
confsets
local−EF′
local−pr−EF
nonrecursive representation
leaves of ecf = set(nat)
local−EF# implements local−EF′
&, |, <<,>>
bitvectors

Fig. 6. Specification of explicit representation of time jumps

An additional advantage of this approach is, that we can use the nonrecursive function *local-EF′* to prove the theorems which use *local-EF* and *local-pr-EF*. To do this, we added these theorems as axioms in the specification which contains *local-EF* and used them as assumption in the previous section.

Now, we get these theorems as additional proof obligations for the nonrecursive definition *local-EF′* in the module.

The proofs of these theorems do not require any new ideas – they are typical for proofs over sets of natural numbers. Therefore, a discussion is omitted. The time required to do the correctness proofs was about two days per operator, including the proofs of the theorems assumed in the previous verification step.

5.5 Implementation using Bitvectors

The previous sections were concerned with deriving an efficient model checking algorithm on abstract data types. This section considers the efficient implementation of the optimized algorithm. RAVEN offers two representations. One that represents sets of natural numbers with bitvectors and one that uses interval lists. Since some of the algorithms used in the latter representation were already verified in an earlier case study with KIV [7] (unrelated to this project), and bitvectors are used as the default in RAVEN, we decided to verify this version.

The implementation verified was derived directly from the RAVEN source-code by omitting code concerned with memory allocation and partitioning of bitvectors into words.

Again, the verification was done using a KIV module, which is shown in the lower half of Fig. 6. In contrast to the previous sections, real programs are used in the module. The programs *local-EF#* and *local-pr-EF#* depicted in Fig. 7 implement the functionality of *local-EF'* and *local-pr-EF* using bitvectors.

Bitvectors are defined to be strings of binary digits. In addition, the basic operations & (binary and), | (binary or), $\ll$ (logical shift left), $\gg$ (logical shift right), a length function # and a bit-selection function denoted by array-like subscripts are defined.

```
local-EF#(c, g, n; var r)
var m = 0, pos = #(c | g) + 1,
   state = 0 in
   r := 0;
   if n = ∞ then m := #(c | g) + 1
   else m := n;
   while pos ≠ 0 do
      pos := pos − 1;
      if g[pos] = 1 then state := m;
      if c[pos] = 1 then state := m +1;
      if state ≥ 1 then
        r := (1 ≪ pos) | r;
        state := state − 1;
      end
   end
end
```

```
local-pr-EF#(c, g; var n)
begin
   if c = 0 ∧ g = 0 then n := ∞
   else if c[0] = 1 then n := ∞
   else var pos = 0 in
        n := 0;
        while n = 0 do
           if c[pos] = 1 then n := pos;
           else if g[pos] = 1 then
             n := pos + 1;
           pos := pos + 1;
        end
   end
end
```

Fig. 7. Implementation of time-jump function *local-EF*

As proof obligations of the module it must be shown, that the implementation programs terminate and satisfy the axioms of *local-EF'* and *local-pr-EF* (for a general introduction to the theory of program modules see [6]; verification techniques for proof obligations in KIV are discussed in [7]).

To stay as close as possible to the implementation of RAVEN we decided to consider bitvectors without leading zeros only. This restriction is formalized as a predicate r. Additional proof obligations are generated by the KIV system to ensure that the programs terminate and keep the restriction invariant.

The correctness proofs for *local-EF#* and *local-pr-EF#* both require invariants for the while-loops. The one for *local-pr-EF#* was easy to obtain, since the current state of computation depends on few factors only. The invariant for *local-EF#* is shown in Fig. 8. It consists of two major parts. INV_1 states, that the postcondition is satisfied for the computations made so far. The main difficulty of the proof was to construct INV_2. It describes the variable *state*, which represents the “memory” of the algorithm. The construction of the invariants took

$$
\begin{array}{l}
\mathrm{INV_{EF}} \equiv \mathrm{INV_1} \wedge \mathrm{INV_2} \\[1em]
\mathrm{INV_1} \equiv \forall\, \mathrm{n_1}.\ \mathrm{pos} \leq \mathrm{n_1} \\
\qquad\qquad \rightarrow (\quad \mathrm{r[n_1]} = 1 \\
\qquad\qquad\qquad \leftrightarrow \quad (\exists\, \mathrm{i.\ c[i]} = 1 \wedge \neg\, \mathrm{i} < \mathrm{n_1} \wedge \neg\, \mathrm{n_1} + \mathrm{n} < \mathrm{i}) \\
\qquad\qquad\qquad\qquad \vee\ (\exists\, \mathrm{i.\ g[i]} = 1 \wedge \neg\, \mathrm{i} < \mathrm{n_1} \wedge \neg\, \mathrm{n_1} + \mathrm{n} - 1 < \mathrm{i}) \wedge \mathrm{n} \neq 0) \\[1em]
\mathrm{INV_2} \equiv \quad \mathrm{r(r)} \wedge \mathrm{state} \leq \mathrm{n} \\
\qquad\quad \wedge\ (\quad \mathrm{state} = 0 \vee \mathrm{c[pos + n - state]} = 1 \\
\qquad\qquad\quad \vee\ \mathrm{state} < \mathrm{n} \wedge \mathrm{g[pos + n - state + 1]} = 1) \\
\qquad\quad \wedge\ (\forall\, \mathrm{i.\ i} < \mathrm{n - state} \rightarrow \mathrm{c[pos + i]} = 0) \\
\qquad\quad \wedge\ (\mathrm{state} < \mathrm{n} \rightarrow (\forall\, \mathrm{i.\ i} < \mathrm{n - state} + 1 \rightarrow \mathrm{g[pos + i]} = 0))
\end{array}
$$

Fig. 8. Invariant for while-loop of *local-EF#*-procedure

several iterations and days, depending on the complexity of the operator and the number of “memory” variables. The size of the invariants ranges between 11 and 25 lines. Once the correct invariant was found, the proofs were large, but easy and automatic.

Summarizing, the effort taken to prove the correctness of the bitvector based implementation of RAVEN was about 2 weeks. On average, it took three iterations to find the correct invariant. A beneficial side effect of the verification was the discovery of inefficient and redundant code. The implementation of *local-EU* could be shortened from 73 to 18 lines of code.

6 Conclusion

In this paper we investigated the correctness of an optimized real-time model checking algorithm. We demonstrated that it is possible to develop the efficient implementation from the specification of the semantics in a a series of refinement steps. Each step could be verified independently of the others.

During the verification, several errors were discovered in the time prediction functions, which constitute the heart of the optimization. Also, some inefficient code could be eliminated. We were able to define a common scheme for the correctness proofs of the different temporal operators, which reduced the verification effort significantly.

The time required to formalize and verify the model checking algorithm was about 4 months. Compared to the total time that was needed to develop and implement the optimizations, the extra effort was modest.

With the verification of the kernel algorithm we now feel justified to answer the question posed in the title of this paper with "yes", assuming the standard BDD package works correctly. Since model checkers are often used in safety critical applications, we hope our results encourage further research in their correctness.

References

1. R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic Decision Diagrams and Their Applications. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 188–191, Santa Clara, California, November 1993. ACM/IEEE, IEEE Computer Society Press.
2. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–33, Washington, D.C., June 1990. IEEE Computer Society Press.
3. E. Clarke, K.L. McMillian, X. Zhao, M. Fujita, and J.C.-Y. Yang. Spectral Transforms for large Boolean Functions with Application to Technologie Mapping. In *ACM/IEEE Design Automation Conference (DAC)*, pages 54–60, Dallas, TX, June 1993.
4. CoFI: The Common Framework Initiative. Casl — the CoFI algebraic specification language tentative design: Language summary, 1997. http://www.brics.dk/Projects/CoFI.
5. D. Long. Long-package sun release 4.1 overview of c-library functions, 1993.
6. W. Reif. Correctness of Generic Modules. In Nerode and Taitslin, editors, *Symposium on Logical Foundations of Computer Science*, LNCS 620, Berlin, 1992. Logic at Tver, Tver, Russia, Springer.
7. W. Reif, G. Schellhorn, and K. Stenzel. Interactive Correctness Proofs for Software Modules Using KIV. In *COMPASS'95 – Tenth Annual Conference on Computer Assurance*, Gaithersburg (MD), USA, 1995. IEEE press.
8. W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications.* Kluwer Academic Publishers, Dordrecht, 1998.
9. J. Ruf. RAVEN: Real-time analyzing and verification environment. Technical Report WSI 2000-3, University of Tübingen, Wilhelm-Schickard-Institute, January 2000.
10. Jürgen Ruf and Thomas Kropf. Symbolic Model Checking for a Discrete Clocked Temporal Logic with Intervals. In E. Cerny and D.K. Probst, editors, *Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 146–166, Montreal, Canada, October 1997. IFIP WG 10.5, Chapman and Hall.
11. Jürgen Ruf and Thomas Kropf. Using MTBDDs for Composition and Model Checking of Real-Time Systems. In *FMCAD 1998.* Springer, November 1998.
12. Jürgen Ruf and Thomas Kropf. Modeling and Checking Networks of Communicating Real-Time Systems. In *Correct Hardware Design and Verification Methods (CHARME 99)*, pages 265–279. IFIP WG 10.5, Springer, September 1999.
13. T. Vollmer. Korrekte Modellprüfung von Realzeitsystemen. Diplomarbeit, Fakultät für Informatik, Universität Ulm, Germany, 2000. (in German).

Modelling Realtime in VSE-II

Werner Stephan   Georg Rock   Andreas Nonnengart
Deutsches Forschungszentrum für
Künstliche Intelligenz (DFKI GmbH)
Email: stephan,rock,nonnengart@dfki.de

1 Introduction

In this paper we present a technique for modelling realtime in a temporal logic close to TLA [3] which is implemented in the VSE system [2, 4, 5]. The approach is based on a global clock architecture with a discrete time scale.

We start by illustrating our technique by a real world example. The emergency closing system (ECS) is part of the control system of a storm surge barrier that physically consists of several huge gates to isolate the North Sea from the Eastern Scheldt. The ECS keeps track of the changes of the water levels and closes the gates if the water level gets dangerous.

In the second part of the paper we show how the insights gained by this case study can be turned into a general method for modelling real time systems that can be applied to a wide range of scenarios. This concerns the general structure of the model, the way to express real time properties, and assumptions about the system and the environment. We close by indicating how real time properties like the ones occurring in our example can be treated as an independent specification by incorporating techniques from hybrid systems [1].

2 The ECS Model

The requirements for the ECS are given by verbal descriptions and an informal, mainly graphical design[1] of the system. The most important requirements mentioned in the document deal with the behaviour of the system in time. As discussed more detailed in section 3.1 the formal system specification uses natural numbers for the physical entities, like seconds, meters, and milliamperes mentioned in the document.

[1]While the verbal description is not very precise and therefore not sufficient in itself the graphical description contains many details of a particular solution, actually it is already close to a technical realization.

2.1 Overall Structure of the Model

The formal model consists of three components: the *system*, the *environment*, and a component containing a global *clock* (see Figure 1). The system takes as inputs values from sensors measuring various water levels and values from switches that are set by an operator. Basically it computes two output signals, one for closing the barrier and one for opening it again. The design is *fail safe* in the sense that the first signal going down means *close*.

Both, the environment and the clock are separated from the system to allow for a refinement of the abstract specification to the actual system. The environment comprises changes of the water levels as well as changes of the switches.

2.2 Assumptions

While in this case there are no complex assumptions about the behaviour of the environment itself it is assumed that the system immediately (i.e. without any delay) reacts to changes of the input variables caused by the environment or the clock and needs no time to compute an output. Obviously the concurrent execution of the three components has to be restricted appropriately to model these assumptions. The clock has to be blocked until the system has finished its computation[2] In such situations *fairness* which forces a step to be executed sometimes in the future is not enough.

Technically the scheduling among the three components is realized by shared variables acting as guards (indicated by circles in Figure 1).

Unless there was a tick of the clock the variable *time* has the same value. Hence the environment having changed the input of the system and the system having computed certain outputs the clock should tick, because otherwise we would have two different situations with the same time stamp.

2.3 Properties

Even if the components are synchronized appropriately with respect to the given assumptions about their behaviour in time there are certain intermediate states without a meaningful interpretation. For example, immediately after the environment has changed some input values the output values of the system might not have the values requested by the requirements specification.

To overcome this problem we distinguish between internal variables and variables visible to the outside. The environment, the system, and the clock change only internal variables. Whenever the clock ticks the corresponding visible variables are updated with the current values of the internal ones. The observable variables remain the same in the intermediate states mentioned above.

The synchronization among the components as well as the choice of observable variables implicitly formalize our way of looking at the system and are

[2]Note that also in cases where there are certain (restricted) reaction and computation times a similar, slightly more complicated scheduling regime is necessary to rule out behaviours where reaction or computation takes to much time.

therefore relevant for the notion of correctness[3].

The formal requirements specification only refers to the visible variables and has no access to the variables local to the system and those used for the synchronisation mechanism. It uses a variable *time* that gives the current time in each state.

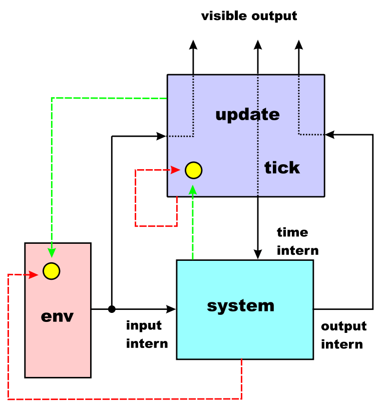


Figure 1: The complete scenario

Due to lack of space we only mention the specification of two properties the ECS system has to satisfy.

1. $\square(\neg(\texttt{OPEN} = \texttt{T} \wedge \texttt{CLOSE} = \texttt{T}))$

2. $\square((\mathit{time} = t_0 \wedge \mathit{Change\_Sensor\_Sig}) \Longrightarrow$
   $\square((t_0 < \mathit{time} \leq t_0 + d + 1) \Longrightarrow \texttt{Close} = \texttt{T}))$

Property 1 says that the OPEN and the CLOSE signal are never true simultaneously. Property 2 says that if the waterlevels get dangerous (expressed by the formula *Change_Sensor_Sig*) at time t_0 then the system reacts by setting the CLOSE signal to true for at least d time units beginning at time $t_0 + 1$.

The proofs of these properties are all done locally to the system component where the proof of property 2 needs assumptions about the environment.

3 General Features

The main general features of our approach are

- the use of discrete values for the time scale (and other physical entities),
- the treatment of assumptions by restricting the concurrent execution of the clock, the environment, and the system (wich are modelled as separate components), and

[3]Perhaps this could be compared to a test bed.

- the hiding of non-meaningful states by distiguishing between internal and external (visible) variables and by using an update function as part of the clock component.

3.1 Expressing Realtime Properties

We have used a global *discrete* time scale and a *clock* that increases the value of a flexible variable *time* (of type *Nat*) by one upon each tick. The technique is applicable for many scenarios where the granularity of the time scale can be fixed afterwards.

Figure 2 informally shows an admissible behaviour of a system (different from the ECS) that reacts to an input signal ($signal_1$). A corresponding description

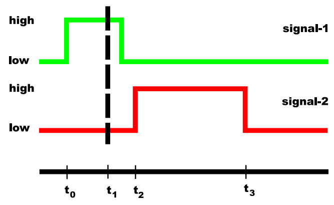


Figure 2: A possible behaviour of a real time system

of a safety requirement could be the following: If $signal_1$ is high for at least $\frac{1}{500}$ sec and $signal_2$ is low, then after at most $\frac{1}{100}$ sec $signal_2$ will be high for at least $\frac{1}{70}$ sec. If for the time points t_0, t_1, t_2 and t_3 shown in the diagram, we have the conditions $t_1 - t_0 \geq \frac{1}{500}$ sec, $t_2 - t_0 \leq \frac{1}{100}$ sec and $t_3 - t_2 \geq \frac{1}{70}$ sec then this particular behaviour fulfils the requirement.

In the formal model we use a discrete time scale and replace concrete durations by constants representing arbitrary but fixed number of time steps. The above mentioned requirement then becomes:

$$\begin{aligned}
&\Box\,\forall\, t_0.(((t_0 = time\ \wedge signal_2 = low \wedge \\
&\qquad \Box(t_0 \leq time \leq t_0 + d_1 \rightarrow signal_1 = high)) \\
&\rightarrow \\
&\exists\, t_2.((t_2 - t_0) \leq d_2\ \wedge \Box((t_2 \leq time \leq t_2 + d_3) \rightarrow signal_2 = high)))
\end{aligned}$$

Having successfully proved the safety requirements from the assumptions discussed below we are free to choose concrete values for a time step. If all durations are given as rational numbers the only constraint is that a single step has to be small enough to have *all* durations as multiples of this value.

3.2 Assumptions

In each scenario there are particular assumptions about the the timing behaviour of the system and the environment. For example in the case of programmable controlers the time for a cycle determines the (maximal) delay until the system notices a change and also the (maximal) time the system needs to respond.

Assumptions of this kind are modelled by restricting the parallel execution of the three components appropriately relying on fairness assumptions for the component. This technique can be viewed as a refinement of fairness in that the restriction is as liberal as possible and does not assume a fixed scheduling regime.

If in our example the system needs a time d_4 to notice a change and also a certain time d_5 to compute output values then, in order to prove the requirements from above, one has to assume that $d_4 \leq d_1$ and $(d_4 + d_5) \leq d_2$.

3.3 Relation to Hybrid Systems

Having hidden non-meaningful states, we are free to formulate timing requirements for the visible variables. But although our requirements are more general than the informal diagrams we have shown above one would like to specify the complete (realtime) behaviour in a comprehensive way without relying on a system specification like that of the ECS.

Following techniques that have been used for hybrid systems this can be done by introducing *abstract states* that correspond to sets of states (over the visible variables). Using a kind of automaton in the sense of [1] one can then specify the complete behaviour of a system to be designed. The specification we have discussed before would then become a refinement of this more abstract description.

References

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[2] D. Hutter, H. Mantel, G. Rock, W. Stephan, A. Wolpers, M. Balser, W. Reif, G. Schellhorn, and K. Stenzel. VSE: Controlling the Complexity in Formal Software Development. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Proceedings Current Trends in Applied Formal Methods, FM-Trends 98*, Boppard, Germany, 1999. Springer-Verlag, LNCS 1641.

[3] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3), 1994.

[4] G. Rock, W. Stephan, and A. Wolpers. Tool support for the compositional development of distributed systems. In *Tagungsband 7. GI/ITG-Fachgespräch Formale Beschreibungstechniken für verteilte Systeme*, number 315 in GMD Studien. GMD, 1997.

[5] G. Rock, W. Stephan, and A. Wolpers. Modular reasoning about structured TLA specifications. In R. Berghammer and Y. Lakhnech, editors, *Tool Support for System Specification, Development and Verification*, Advances in Computing Science, pages 217–229. Springer, WienNewYork, 1999.

Strategies for the Verification of Object-oriented Programs

J. Meyer and A. Poetzsch-Heffter

May 2, 2000

1 Introduction

Verification of typical object-oriented programs is more complex than verification of imperative programs with non-recursive procedures. It has to cope with dynamic binding of methods, recursive method invocations, subtyping, abstract types, and aliasing through object references. The use of these features and the combination of data and procedures into classes lead to a different programming style which also influences verification techniques. The goal of our research is to overcome the resulting challenges by interactive verification techniques that can be formulated as proof strategies similar to the classical approach for tactical theorem proving.

In this extended abstract, we analyze typical verification steps within proofs of object-oriented programs. We focus on the differences to verification of imperative programs with non-recursive procedures. In particular, we investigate which information about programs is needed in order to support the verification steps by strategies in a tactical theorem prover. The rest of this abstract is structured as follows. Section 2 sketches the used verification framework. In particular, it describes a simple overall technique to prove that an OO-program satisfies its specification. Section 3 analyses the steps of this simple proof technique w.r.t. formalizing them as strategies. Section 4 concludes by sketching the realization of such strategies within tactical theorem provers.

2 Verification of Object-oriented Programs

The results of the following investigation hold for most existing object-oriented programming languages. However, to have a concrete setting, we consider the kernel of Java as a representative for this language class (cf. [PHM99]). Similarly, we refer here only to a specific programming logic, namely a Hoare-style partial correctness logic, although the results of the investigation apply as well to more powerful programming logics (e.g. dynamic logic or temporal logic).

Program verification can be done under different perspectives. The classical perspective assumes a complete program with exactly one main procedure. The goal is to verify that the main procedure has a certain property. This perspective does not reflect the reuse of program modules, as it underlies modular programming in general and object-oriented programming in particular. Consequently, we assume a perspective where the goal is to verify interface specifications of a set[1] of types ST. As the implementation of these types may use other, already verified types STU, we assume that the used types are equipped with interface specifications that can be applied during verification of ST. This means that we have to relate types and specifications in such a way that the theorem proving environment can access *the* specification of a type. In particular, the verification of ST can only use properties about STU that are formulated in STU's interface specifications.

A user-defined type in Java is either declared by a class or a so-called interface. To keep things simple here, we assume that the specification of a type T consists of the specifications of T's methods (cf. [PH97] for more elaborated specification constructs). The specification of a method consists of a set of pre-/postpairs. From a theoretical point of view, it would be sufficient to consider only one pre-/postpair. In practical situations, this would lead to large pre- and

[1]As types can and usually do recursively depend on each other, we have to deal with sets of types instead of single types.

postconditions. A more structured approach using several pairs turns out to be easier to handle where each pair deals with a different aspect of the method behavior: E.g. one pair to describe the returned value of the method; one pair to specify the modifications to the object store; one pair to capture the invariant properties to the method etc.

Proof Task. Program verification techniques can be used for different purposes; e.g. to prove that a program never throws a NullPointerException or to derive the needed properties of an auxiliary method. Here, we only consider the following standard proof task: Given a set of specified and implemented types ST that use a set of specified types STU; prove that the implementation of ST satisfies its specification assuming that the types STU behave as specified. Before we present a simple technique to structure the proof task we have to summarize an important aspect of OO-program verification.

Dynamic Binding and Virtual Methods. In OO-programs, method invocation are dynamically bound, i.e. it is statically in general not known which method implementation is executed at an invocation site. Even worse for verification, several different implementations can be invoked from an invocation site during the execution of a program. There are essentially two techniques to handle dynamic method binding in a programming logic. If we know the whole program, we can figure out which methods are possibly called at the site using type and subtype information. Then, we show that the needed property at the site is guaranteed by all method implementations possibly bound to the site. The disadvantages of this technique are that we usually cannot access the implementation of the whole program and that we have to show behavioral subtyping properties (cf. [LW94]) for each invocation site again.

That is why we use so-called *virtual methods* to capture the behavior of all subtype methods; the properties of the virtual method m of type T are then used to verify an invocation site of m with *static* type T. If T is an interface type and m is declared in T, T:m denotes the virtual method that abstracts the common behavior of all implementations of m in subtypes of T. To put it the other way round, all subtype methods have to satisfy the specification of T:m. If T is a class type, T:m is again used to denote the virtual method capturing the implementation of m in T and in all subtypes of T. By T@m we denote the implementation of m in T. Notice that this implementation can be overridden in subtypes of T so that the properties of T:m are in general more abstract than those of T@m (for details of this technique we refer to[PHM99]). The specification of a method refers to the properties of T:m.

A Simple Proof Technique. In general, the verification of a set of types ST can be divided into the following steps where the specification of the used types STU serve as axioms. In the first step, the properties of virtual methods are reduced to the required properties of given implementations and and to proof obligations for types in ST that are subtypes of types in STU (for brevity, the latter aspect will be neglected in the following). In a second step, the method implementations are verified w.r.t. the required properties. To capture recursion, the properties of the virtual methods may be assumed in this step. In a third step, these assumptions have to be eliminated.

3 Automating Program Proofs for OO-Programs

In this section, we explain some of the verification steps outlined above in more detail and discuss further aspects relevant to the verification of OO-programs. The described proof techniques serve as examples for strategies which are needed for the proof of OO-programs. Program proofs are constructed using the rules and axioms of the underlying Hoare logic. Rules and axioms are provided as operations (cf. tactics in the LCF approach [GMW79]) with forward and backward direction use. Using an operation leads to an extension at the root (forward proof) or at a leaf (backward proof). Operations can be combined to form strategies (cf. tacticals in LCF).

3.1 Verifying behavioral Subtyping

As we consider object-oriented programs, we are directly confronted with subtyping. This means that specified properties of a virtual method T:m have to be shown for the methods S:m in subtypes S of T. As an example we consider the following program fragment:

```
class T {
 int m() { ... }
}
```

```
class S1 extends T {
}

class S2 extends T {
 int m() { ... }
}
```

$$\begin{tikzcd} & \begin{array}{c}\texttt{T:m()}\\ \texttt{T@m()}\end{array} & \\ \texttt{S1:m()} \arrow[ur] & & \begin{array}{c}\texttt{S2:m()}\\ \texttt{S2@m()}\end{array} \arrow[ul] \end{tikzcd}$$

The program consists of three classes where S1 and S2 are subtypes of class T. T has a virtual method m, therefore subtypes S1 and S2 have virtual methods S1:m resp. S2:m. T:m is implemented in class T. Class S1 inherits the implementation T@m for S1:m and S2 reimplements T@m with S2@m. Suppose now you want to show the program proof obligation $\vdash \{\ \mathbf{P}\ \}$ T:m() $\{\ \mathbf{Q}\ \}$[2], which arises from the program specification. The proof of the given goal can be divided into two phases. 1) Take proof obligations for virtual methods in supertypes back to virtual methods in subtypes. 2) Show that the implementations of virtual methods in subtypes have the properties of the methods in supertypes. In the following we sketch the first phase, which is mainly performed by the operations of the following rules (where τ denotes *typeof(this)*):

subtype-rule:

$$\frac{\mathrm{S} \preceq \mathrm{T} \qquad \mathcal{A} \vdash \{\ \tau \preceq S \wedge \mathbf{P}\ \}\ \mathrm{S{:}m()}\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \vdash \{\ \tau \preceq S \wedge \mathbf{P}\ \}\ \mathrm{T{:}m()}\ \{\ \mathbf{Q}\ \}}$$

class-rule:

$$\frac{\mathcal{A} \vdash \{\ \tau = \mathrm{T} \wedge \mathbf{P}\ \}\mathit{impl}(\mathrm{T,m()})\{\ \mathbf{Q}\ \} \qquad \mathcal{A} \vdash \{\ \tau \prec \mathrm{T} \wedge \mathbf{P}\ \}\ \mathrm{T{:}m()}\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \vdash \{\ \tau \preceq \mathrm{T} \wedge \mathbf{P}\ \}\ \mathrm{T{:}m()}\ \{\ \mathbf{Q}\ \}}$$

disjunct-rule:

$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}_1\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}_1\ \} \qquad \mathcal{A} \vdash \{\ \mathbf{P}_2\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}_2\ \}}{\mathcal{A} \vdash \{\ \mathbf{P}_1 \vee \mathbf{P}_2\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}_1 \vee \mathbf{Q}_2\ \}}$$

The following algorithm builds a proof tree for the behavioral subtyping proof in backward direction. It uses the subtype-, class-, strength-, and disjunct-rule of the underlying Hoare logic until only program proof obligations for method implementations are left:

Input: A proof goal g
Result: The algorithm constructs a proof tree with g as root and open leafs (slots) for the proofs of method implementations.

```
$\mathbb{G} \leftarrow \{g\}$
**while** $\mathbb{G} \neq \{\ \}$ **do**
  select any $g$ from $\mathbb{G}$
  **if** $g$ matches $\mathcal{X} \vdash \{\ \tau \preceq \mathrm{T} \wedge \mathbf{A}\ \}$ T@m() $\{\ \mathbf{B}\ \}$ **then**
    /* show specification for all subtypes of T and T */
    use class-rule backward for $g$
    $\mathbb{G} \leftarrow \mathbb{G} \cup \{\mathcal{X} \vdash \{\ \tau = \mathrm{T} \wedge \mathbf{A}\ \}$ T@m() $\{\ \mathbf{B}\ \}\}$
    $\mathbb{G} \leftarrow \mathbb{G} \cup \{\mathcal{X} \vdash \{\ \tau \prec \mathrm{T} \wedge \mathbf{A}\ \}$ T:m() $\{\ \mathbf{B}\ \}\}$
  **else if** $g$ matches $\mathcal{X} \vdash \{\ \tau \prec \mathrm{T} \wedge \mathbf{A}\ \}$ T:m() $\{\ \mathbf{B}\ \}$ **then**
    /* show specification for all subtypes of T */
    **if** T has no subtypes **then**
      /* as the precondition evaluates to *false*, $g$ can be derived from the
      false-axiom $\vdash \{$ FALSE $\}$ comp $\{$ FALSE $\}$ (not shown here) */
    **else**
      let $S_{1 \leq i \leq n}$ be all direct subtypes of T /*thus $\tau \not\preceq T \Rightarrow \bigvee_{i=1...n} \tau \preceq S_i$*/
      use strength-rule backward to reduce $g$ to $\mathcal{X} \vdash \{\ \bigvee_{i=1...n} \tau \preceq S_i \wedge \mathbf{A}\ \}$ T:m() $\{\ \mathbf{B}\ \}$
      **for** $j = n \ldots 2$ **do**
        use disjunct-rule backward on $g$
        /* split $\bigvee_{i=1...j} \tau \preceq S_i$ up to $\bigvee_{i=1...j-1} \tau \preceq S_i \vee \tau \preceq S_j$*/
        g$\leftarrow \mathcal{X} \vdash \{\ \bigvee_{i=1...j-1} \tau \preceq S_i \wedge \mathbf{A}\ \}$ T:m() $\{\ \mathbf{B}\ \}$
        $\mathbb{G} \leftarrow \mathbb{G} \cup \{\mathcal{X} \vdash \{\ \tau \preceq S_j \wedge \mathbf{A}\ \}$ T:m() $\{\ \mathbf{B}\ \}\}$
      **end for**
      $\mathbb{G} \leftarrow \mathbb{G} \cup \{g\}$
```

[2]We use the sequent notation of Hoare logic triples where an optional set of method assumptions is noted before the turnstyle, **P** and **Q** are pre,- resp. postcondition and the middle part denotes a program part.

```
    end if
  else if g matches $\mathcal{X} \vdash \{ \tau \preceq \mathrm{S} \wedge \mathbf{A} \}$ T:m() $\{ \mathbf{B} \}$ then
    /* show specification for subtype S (which is direct subtype of T) */
    use subtype-rule backward on $g$
    $\mathbb{G} \leftarrow \mathbb{G} \cup \{\mathcal{X} \vdash \{ \tau = \mathrm{S} \wedge \mathbf{A} \}$ S:m() $\{ \mathbf{B} \}\}$
  else if $g$ matches $\_ \vdash \{ \tau = \mathrm{T} \wedge \_ \}$ T@m() $\{ \_ \}$ then
    /* implementations are left open in proof tree and not considered further. A strategy to prove
    method implementations could be attached here */
    ...
  end if
  $\mathbb{G} \leftarrow \mathbb{G} \setminus \{g\}$ /* remove $g$ from $\mathbb{G}$ */
end while
```

Running the algorithm with the goal $\vdash \{ \mathbf{P} \}$ T:m() $\{ \mathbf{Q} \}$ as input leads to a proof tree with three open slots: 1) $\vdash \{ \tau = T \wedge \mathbf{P} \}$ T@m() $\{ \mathbf{Q} \}$ 2) $\vdash \{ \tau = S1 \wedge \mathbf{P} \}$ T@m() $\{ \mathbf{Q} \}$ and 3) $\vdash \{ \tau = S2 \wedge \mathbf{P} \}$ S2@m() $\{ \mathbf{Q} \}$ which represent proof obligations for the different implementations of the virtual method T:m. Furthermore it can be seen that 1) and 2) only differ in the type of *this*. If the method implementations T@m and S2@m do not depend on the type of *this* it would be sufficient to prove the triple $\vdash \{ \tau \preceq T \wedge \mathbf{P} \}$ T@m() $\{ \mathbf{Q} \}$, because 1) and 2) are directly derivable from it. This could be done by a strategy too. The strategy coded within the algorithm shows that properties of a virtual method can be automatically reduced to properties of implementations. The proof construction, which is performed automatically, is guided by the structure of the subtype relation of the underlying program.

3.2 Verification of Method Implementations

The strategy described in the preceding section leads to proof obligations for method implementations. Thus at this point it is useful to have strategy support for this task. The proof of method implementations is ultimately based on the proof its bodys statement sequence. To automate this one can use techniques similar to weak precondition generation with extensions for method invocations. For all statements except method invocations, where an appropriate method specification has to be used, a proof can be constructed automatically. For all simple statements like assignment-, field-read- and field-write-, cast-, and empty-statement exists a forward operation which instantiates the appropriate axiom and thus constructs a forward proof fragment for that statement. For composed statements like block-, if-, sequence- and loop-statements there exist forward proof operations which construct a proof tree for that statement from proofs for its components. Combining these operations in an appropriate algorithm results in a strategy which performs a weak precondition generation proof and constructs a proof tree for the given statement sequence. As we consider object-oriented programs, the difference of this algorithm to described techniques in the literature [Gri81] is that we have to heavily care about method invocations.[3] If the strategy reaches an invocation statement, interaction with the user is needed. The user obtains the information at which method invocation (e.g. of method S:n) the strategy has stopped and which postcondition $\mathbf{Q}$ is generated for that invocation statement up to that point. He now can 1) select one of the given method specifications for that method and interactively adapt it to the local needs or 2) provide a precondition $\mathbf{P}$ and continue the strategy leaving the slot $\vdash \{ \mathbf{P} \}$ S:n() $\{ \mathbf{Q} \}$ open in the current proof and care later about it. The following algorithm sketch summarizes the technique described above:

Algorithm *wp*
Input: A statement *stmt* and a postcondition $\mathbf{Q}$.
Result: A proof tree for $\vdash \{ \mathbf{P} \}$ stmt $\{ \mathbf{Q} \}$ with a generated precondition $\mathbf{P}$ and postcondition $\mathbf{Q}$

```
if stmt matches ; then
```

[3]Remember that it is in general not possible to generate a sufficiently weak precondition if the underlying programming language allows recursion, dynamic binding and aliasing.

```
    return use inst_empty(stmt,Q)                                  /* empty statement */
  else if stmt matches return _ ; then
    return use inst_field-read(stmt,Q)                             /* field-read statement */
  else if stmt matches stmt;stmts then
    g1 ← wp(stmts,Q), g2 ← wp(stmt,precondition(g1))               /* statement-list */
    return seq_forward(g1,g2)
  else if stmt matches if(expr) stmt1 else stmt2 then
    g1 ← wp(stmt1,Q), g2 ← wp(stmt2,Q)                             /* if-statement */
    f ← (precondition(g1) ∧ expr) ∨ precondition(g2) ∧ ¬expr)
    g1 ← use strength_forward(g1, f ∧ expr), g2 ← use strength_forward(g2, f ∧ ¬expr)
    return if_forward(g1,g2)
  else if stmt matches _ = _ ( _,. . . ,_) then
    user interaction at this point results in a proof fragment for the     /* method invocation */
    method invocation
  else if . . . then
    /* similar for while-, field-write-, assign statement, cast-, and return-statement */
    /* while-loops could additionally enforce interaction e.g. for loop-invariants */
  end if
```

The above sketched algorithm shows how traditional weak precondition generation can be enhanced to be useful in the area of object-oriented programs. User interaction and thus the proof for a method invocation statement can be considered as a strategy of its own. The overall algorithm divides the given program part *stmt* recursively into program fragments and constructs a proof with a weak precondition in forward direction while unrolling the recursion. Therefore in this example a program proof for *stmt* and postcondition Q with a computed precondition is constructed automatically guided by the program structure.

3.3 Handling Recursion

In the preceding section we did not care about the handling of recursion and assumptions. As OO-programs make direct or indirect (e.g. via dynamic dispatch) of recursion, an important task which can be supported by automatically working strategies is the elimination of assumptions, which are used to unroll recursion. In the following we demonstrate an assumption elimination strategy for method implementations to give an impression of this technique. Suppose you want to prove A_i[4], $1 \leq i \leq n$. 1. Prove $B_j := \bigcup_{i=1\ldots n} A_i \vdash body(A_j)$, $j = 1 \ldots n$. 2. Derive $C_j := \bigcup_{i=1\ldots n, i\neq j} A_i \vdash A_j$ from B_j, for $j = 1 \ldots n$ using the implementation rule. 3. Eliminate step by step all assumptions in $C_{1\ldots n}$, for example use $C_1 = \bigcup_{i=2\ldots n} A_i \vdash A_1$ to eliminate A_i in $C_2, \ldots, C_n$ etc. Strategies like this can be optimized with a preceding syntactical program analysis. One can for example reduce the assumptions for the proof of each $body(A_i)$ if one previously computes the methods which are called by its implementation.

implementation-rule:

$$\frac{\begin{array}{l}\mathcal{A}\,,\ \{\mathbf{P}\}\ \mathrm{T@m}(\mathrm{T}_{p_1}\ \mathrm{p}_1,\ldots,\mathrm{T}_{p_n}\ \mathrm{p}_n)\ \{\mathbf{Q}\}\ \vdash \\ \{\mathbf{P} \wedge \mathrm{this} \neq \mathit{null} \wedge \bigwedge_i(\mathrm{v}_i = \mathit{init}(\mathrm{TV}_i))\}\ \mathrm{BODY}(\mathrm{T@m}(\mathrm{T}_{p_1}\ \mathrm{p}_1,\ldots,\mathrm{T}_{p_n}\ \mathrm{p}_n))\ \{\mathbf{Q}\}\end{array}}{\mathcal{A} \vdash \{\mathbf{P}\}\ \ \mathrm{T@m}(\mathrm{T}_{p_1}\ \mathrm{p}_1,\ldots,\mathrm{T}_{p_n}\ \mathrm{p}_n)\ \ \{\mathbf{Q}\}}$$

3.4 Combining Syntactic Checks and Verification

If full information about the program structure and the results of the analysis of static program properties like variable binding, type analysis, and invocation call graph is available, it is possible to combine that information. An example for this is to prove the property that a method does not change the object store. A method does not change the object store, if its implementation does not change the object store. This means especially that invoked methods do not change

[4]Where A_i abbreviates $\vdash \{\ \mathbf{P}\ \}\ \mathrm{T@m}\ \{\ \mathbf{Q}\ \}_i$.

the object store. A conservative approximation of this is that a method has no writing attribute access (conservative, because an implementation could undo an object store modification). The absence of writable attribute access is a static program property which can be computed by a static program analysis, e.g. during a syntactical program analysis. This information can be used to automatically compute program proofs for the invariance of the object store, because the proofs for methods with this property are simple and can be automated. Notice that the syntactic checks can be as complex as any static program analysis.

As object-oriented programs usually contain lots of methods with special syntactic properties (think of the `get`-methods of `get-set` pairs), such techniques combining static analysis and tactical theorem proving can reduce the amount of interactive proof work a lot. To exploit such benefits, the verification environment has to be sufficiently powerful to perform static program analyses.

3.5 Proof Guidance as Strategy

The proof that a specified object-oriented program fulfills its specification is ultimately based on the proof of several different properties, whose proofs can depend on each other. Because manual proof work and automized proof work are mixed, there may be proof states, where several completed and uncompleted proofs can exist simultaneously. To provide an overall proof guidance, a main strategy can be formulated which examines the current proof state, combines existing proof fragments, and continues the overall proofs by delegating proof obligations to substrategies. Thus these main strategy is a strategy, which is based on the state of all current proofs.

4 System Requirements

Within this extended abstract we presented overall techniques for proving OO-programs. From the demonstrated strategies the following properties of a tactical theorem prover can be derived: 1. Tactical program provers have to provide possibilities to formulate strategies as sketched within the examples. 2. It must be possible to manage proof obligations and proof parts. Furthermore it must be possible to inspect the current proof state and proofs. 3. Proof construction can be guided by several syntactical program properties, e.g. by the program structure, by the subtype relation, or by the call graph. Strategies and operations must have access to this information, which is needed during the whole proof process. 4. Interaction in strategies is needed to overcome problems of proof complexity. The described techniques are part of a research project, which currently led to the development of an interactive proof tool (cf. Jive[MPH00]) for a subset of Java. Jive currently supports the above described techniques.

References

[GMW79] M. Gordon, A. Milner, and C. Wadsworth. Edinburgh lcf. 1979.

[Gri81] David Gries. *The Science of Programming.* Springer-Verlag, 1981.

[LW94] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6), 1994.

[MPH00] J. Meyer and A. Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, editors, *TACAS00, Tools ans Algorithms for the Construction and Analysis of Software*, volume 276 of *Lecture Notes in Computer Science*, pages 63–77, 2000.

[PH97] A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, January 1997.

[PHM99] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In D. Swierstra, editor, *ESOP '99*, LNCS 1576. Springer-Verlag, 1999.

A Practical Extension Mechanism for Decision Procedures

Alessandro Armando   Silvio Ranise

DIST – Università di Genova, Viale Causa 13 – 16145 Genova, Italia
{armando, silvio}@dist.unige.it

1 Introduction

The lack of automated support is probably the main obstacle to the application of formal methods in the industrial setting. As witnessed by the success of model checking techniques, formal methods are readily employed in industry as soon as automated reasoning tools providing a sufficiently high level of automation are available. On the other hand, the only way to meet the requirements posed by many industrial applications is to combine the expressiveness of general purpose provers with the efficiency of specialized reasoners (such as, e.g., decision procedures and unification algorithms). Unfortunately this turns out to be a surprisingly difficult task. The main problem is that only a tiny portion of the proof obligations arising in many practical applications falls exactly into the domain the specialized reasoners are designed to solve.

To illustrate let us assume that a decision procedure for Presburger Arithmetic is available and consider the problem of proving the formula:

$$l \leq \mathsf{min}(a) \wedge \mathbf{0} < k \Rightarrow l < \mathsf{max}(a) + k \tag{1}$$

where l and k are constants denoting arbitrary integers, a is a constant denoting an arbitrary list of integers, max (min) is a unary function symbol denoting a function which returns the maximum (minimum, resp.) element of the list of integers given as input, and the remaining symbols (namely $\mathbf{0}$, $+$, and $<$) have their usual arithmetic interpretation. The key point here is that the decision procedure for Presburger Arithmetic is only aware of the interpretation of the arithmetics symbols (i.e. $\mathbf{0}$, $+$, and $<$) and treats all the terms whose top-most function symbol is non-arithmetic as uninterpreted.[1] Under such assumptions, the decision procedure can not possibly establish the validity of (1).

Boyer & Moore recognized this difficulty when they integrated a decision procedure for the quantifier-free fragment of Presburger Arithmetics into their prover [4]. To cope with the problem, they proposed an extension mechanism for the decision procedure, called *augmentation*, as well as an elaborated integration schema between the extended decision procedure and the rewrite engine which improved dramatically the performance of their system both in execution time and in reduced user interaction. Augmentation aims at making the initial decision procedure aware of properties of function symbols the procedure is otherwise not aware of through the use of a set of available lemmas. Going back to our example, if the following lemma is available

$$\mathsf{min}(X) \leq \mathsf{max}(X) \tag{2}$$

then augmentation inspects the internal state of the decision procedure, instantiates (2) with the substitution $\{a/X\}$, and finally extends the internal state of the decision procedure with the resulting instance. The new state is easily found unsatisfiable by the decision procedure. The situation is complicated if conditional lemmas are allowed: when trying to establish the conditions of the lemmas the prover can be recursively invoked and special devices must be put in place to ensure termination.

In [1] we introduced an extended form of contextual rewriting, called *Constraint Contextual Rewriting* or CCR for short, which generalizes Boyer & Moore's integration schema by providing an abstract and concise

[1]Here and in the rest of the paper we assume that the decision procedures can handle uninterpreted function symbols. As already noted in [8] this is a trivial extension to achieve.

specification of the interplay between rewriting and the decision procedure. In [3] we refined CCR and proved its termination. CCR, as well as some of its instances resulting from the incorporation of decision procedures for Presburger Arithmetic, the theory of ground equality, and their combination have been implemented in the theorem prover *Rewrite and Decision procedure Laboratory* (**RDL**).[2]

The extension mechanism used in the previous accounts of CCR already generalizes Boyer & Moore's augmentation in that it is independent from the theory decided by the decision procedure. In this paper we go a step further and propose an extension mechanism which allows for the 'on-the-fly' generation of lemmas. This is an important improvement since—if suitably coupled with a lemma speculation facility—it relieves the user from the burden of providing lemmas in many situations thereby resulting in increased automation. Furthermore, in order to show its practical usability, we illustrate how our schema can lift a decision procedure for the quantifier-free fragment of Presburger Arithmetic to tackle non-linear problems of significant difficulty, based on affinization techniques [6].

The plan of the paper is as follows. In Section 2 we present our extension mechanism. In Section 3 we propose an instance of the extension mechanism that enables a decision procedure for Presburger Arithmetic to tackle non-linear problems. Finally, in Section 4 we discuss our extension schema and draw some conclusion.

Formal Preliminaries. By Σ, Π (possibly subscripted) we denote finite sets of function and predicate symbols (with their arity), respectively. A *signature* is a pair of the form (Σ, Π). A Σ-*term* is a term built out of the symbols in Σ and variables in the usual way. A (Σ, Π)-*atom* is either an expression $q(t_1, \ldots, t_n)$ where $q \in \Pi$ and t_i $(i = 1, \ldots, n)$ is a Σ-term or one of the propositional constants $\mathtt{true}$ and $\mathtt{false}$ denoting truth and falsity respectively. A (Σ, Π)-*literal* is either a (Σ, Π)-atom, $p(t_1, \ldots, t_n)$, or a negated (Σ, Π)-atom, $\neg p(t_1, \ldots, t_n)$. A (Σ, Π)-*formula* is built in the obvious way using the standard logical connectives (i.e. $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$). A (Σ, Π)-*clause* is a disjunction of literals which we indicate as a finite set of (Σ, Π)-literals. We will write $s \neq t$, $s \not< t$, ... in place of $\neg(s = t)$, $\neg(s < t)$, ... (resp.). If a is an atom, then $\overline{a}$ abbreviates $\neg a$ and $\overline{\neg a}$ stands for a. If Q and C are finite sets of literals, then $\overline{Q}$ abbreviates $\{\overline{q} : q \in Q\}$, $\bigwedge Q$ stands for any conjunction of all the literals in Q, $Q \to C$ stands for the formula $\bigwedge Q \Rightarrow \bigwedge C$, and $Q \to c$ abbreviates $Q \to \{c\}$. Let ϕ and ψ be (Σ, Π)-formulae, Γ and Δ sets of (Σ, Π)-formulae. ϕ is a *logical consequence* of Γ iff $\Gamma \models \phi$, where $\models$ denotes entailment in classical logic. $\Gamma, \Delta \models \alpha$ abbreviates $\Gamma \cup \Delta \models \alpha$. A (Σ, Π)-*theory* is a set of (Σ, Π)-formulae closed under logical consequence. If T is a theory, then $\Gamma \models_T \phi$ abbreviates $T, \Gamma \models \phi$ and we say that ϕ is T-*entailed* by Γ. ϕ is T-*satisfiable* iff there exists a model of $T \cup \{\phi\}$, and T-*unsatisfiable* otherwise. ϕ is T-*valid* iff ϕ is a logical consequence of T or, equivalently, iff $\phi \in T$; ϕ and ψ are T-*equivalent* iff $(\phi \Leftrightarrow \psi)$ is T-valid. We consider two theories T_c and T_j of signature (Σ_c, Π_c) and (Σ_j, Π_j) respectively s.t. $\Sigma_c \subseteq \Sigma_j$, $\Pi_c \subseteq \Pi_j$, and $T_c \subseteq T_j$.

2 The Extension Schema

We assume that a *decision procedure* is an incremental and state-based procedure whose states (called *constraint stores*) are finite sets of ground (Σ_j, Π_c)-literals represented in some internal form[3] and whose interface functionalities are *cs-simp* and *cs-unsat*. *cs-simp*(P, C) computes and returns the constraint store C′ resulting from the addition of a finite set P of (Σ_j, Π_c)-literals to the constraint store C and is such that P, $\|$C$\|$ $\models_{T_c} \bigwedge \|$C′$\|$ holds.[4] *cs-unsat*(C) is a boolean valued function characterizing a sub-set of the T_c-unsatisfiable states whose T_c-unsatisfiability can be checked by means of a computationally inexpensive check. More precisely, *cs-unsat* is s.t. *cs-unsat*(C) implies the T_c-unsatisfiability of C. The function *cs-extend* extends *cs-simp* in the sense that if *cs-extend*(P, C) = C′ then P, C $\models_{T_j} \bigwedge \|$C′$\|$. In particular, the composition of *cs-extend* with *cs-unsat* forms a proof procedure for T_j, i.e. *cs-unsat*(*cs-extend*(P, C)) implies the T_j-unsatisfiability of $\bigwedge$(P $\cup$ $\|$C$\|$). The key functionality in *cs-extend* is *genlemma*(C), which speculates formulae T_j-entailed by the constraint store C. This is so in order to cope with the situations in

[2]The system distribution can be accessed via the Constraint Contextual Rewriting Home Page, `http://www.mrg.dist.unige.it/ccr/`. The system is based on CCR and simplifies clauses in a quantifier-free first-order logic with equality.

[3]If C is a constraint store, then $\|$C$\|$ denotes the set of literals represented by C.

[4]More than this is usually required but here, for the lack of space, we focus on soundness requirements only.

which C is T_j-unsatisfiable but not T_c-unsatisfiable. More precisely, ***genlemma***(C) returns a constraint store C' and a list of formulae `lemmas` of the form $Q \rightarrow \{c_1, ..., c_n\}$ s.t. $\bigwedge ||C'||$ is T_j-equivalent to $\bigwedge ||C||$ and the formulae in `lemmas` are T_j-entailed by $||C||$.

Our extension mechanism is given in Figure 1. Given a set of (Σ_j, Π_c)-literals P and a constraint store C,

```
     fun cs-extend(P: set of $(\Sigma_j,\ \Pi_c)$-literals,
                  C: constraint store): constraint store
     begin
1:     $C_1$ := cs-simp(P, C);
2:     if cs-unsat($C_1$)
3:     then return $C_1$
       else begin
4:        $\langle C_2$, lemmas$\rangle$ := genlemma($C_1$);
          while ((lemmas $\neq$ []) and not cs-unsat($C_2$))
          do begin
5:           [Q$\rightarrow$ {$c_1$, ..., $c_n$}|rest] := select(lemmas);
6:           entailed := true; {q} $\cup$ Q$'$ := Q;
7:           while (Q$\neq \emptyset$ and entailed)
             do begin
8:              entailed := cs-unsat(cs-extend({$\overline{q}$}, $C_2$));
9:              Q := Q$'$; {q} $\cup$ Q$'$ := Q;
             end
10:          if entailed
11:          then $C_2$ := cs-simp({$c_1$, ..., $c_n$}, $C_2$);
12:          lemmas := rest;
          end
13:       return $C_2$
       end
     end
```

Figure 1: The Extension Mechanism

we add the literals in P to C (line 1) and we check whether the resulting constraint store C_1 is T_c-unsatisfiable (line 2). If it is, then we return C_1 (line 3). Otherwise, ***genlemma*** is invoked (line 4) in the attempt to speculate new lemmas. If no lemma is speculated, then ***genlemma*** returns the empty list (i.e. `[]`), the test of the while-loop is not satisfied, and we return C_1 (line 13). Otherwise, ***genlemma*** returns a non-empty list and we enter the outermost while-loop. (Notice that $\bigwedge ||C_2||$ is T_c-equivalent to $\bigwedge ||C_1||$ and therefore ***cs-unsat***(C_2) is necessarily true.) Then, we heuristically choose a lemma from `lemmas` (line 5)[5] and we add its conclusions to the constraint store (line 11). However, before adding the conclusions, we must relieve the conditions of the lemma under consideration. We do this by recursively calling ***cs-extend*** on the negation of each condition and then by checking the resulting constraint store for T_c-unsatisfiability. This is done by the innermost while-loop at line 7: we consider each literal in Q (the conditions of the lemma) and we check whether it is entailed by the constraint store or not (line 8). If we succeed in relieving the current hypothesis, we set the variable `entailed` to true and we consider the remaining hypotheses (line 9). (Notice that `entailed` is just a flag variable saying whether all the hypotheses of the selected lemma have been relieved or not, line 10). Otherwise, the current hypothesis can not be relieved and we exit the inner-most loop. If either the hypotheses of a lemma have not been relieved or its conclusions do not enable ***cs-unsat*** to detect the T_c-unsatisfiability of the constraint store, then we consider the remaining lemmas (line 12). Termination and soundness of our extension schema are formally stated and proved in the full version of this paper [2].

[5] ***select*** takes a list L and returns a list whose head is an element E of L and the tail is L with E removed.

3 Extending a Decision Procedure for Presburger Arithmetic

We now show that our extension mechanism allows us to extend a decision procedure for the quantifier-free fragment of Presburger arithmetic over the rationals (T_c) to tackle non-linear problems in the quantifier-free fragment of arithmetics over the integers (T_j). From here on, we assume that *cs-simp* implements an incremental version of the Fourier-Motzkin elimination method (see, e.g., [5]). The constraint store keeps a set of linear inequalities in some internal form and *cs-simp*(P, C) extends C with arithmetic information stored in the literals in P and closes the resulting constraint store w.r.t. the operation of "cross-multiplying and adding" the inequalities in it. (As an example, consider the two inequalities $\mathbf{3}a + b \leq \mathbf{0}$ and $-\mathbf{2}a + \mathbf{3}c \leq \mathbf{0}$, then the result of "cross-multiplying and adding" them is the inequality $\mathbf{2}b + \mathbf{9}c \leq \mathbf{0}$.) If the resulting constraint store is T_c-unsatisfiable, then the process will eventually add a trivially T_c-unsatisfiable inequality such as, e.g., $\mathbf{0} \leq -\mathbf{1}$. *cs-unsat* simply checks whether a trivially T_c-unsatisfiable inequality is in the input constraint store.

In the following, we describe an instance of *genlemma* exploiting a lemma speculation mechanism which allows for the 'on-the-fly' generation of lemmas. To illustrate, let us consider the problem of *cs-extend*ing the constraint store $C = \{a \leq -\mathbf{1}, b \leq \mathbf{0}, -c \leq \mathbf{0}\}$ with the atom $-a * c - b * c \leq -\mathbf{1}$. The application of *cs-simp* yields the constraint store $C' = \{a \leq -\mathbf{1}, b \leq \mathbf{0}, -c \leq \mathbf{0}, \ -a * c - b * c \leq -\mathbf{1}\}$. We factor out the two occurrences of c in the left hand side of $-a * c - b * c \leq -\mathbf{1}$ (obtaining $(-a - b) * c \leq -\mathbf{1}$) and by noticing that this fact is an *hyperbolic inequality*, i.e. an inequality of the form $s * t \leq k$ (where s and t are terms and k is a numeric constant). By resorting to its geometrical interpretation it is easy to verify that $s * t \leq -\mathbf{1}$ is T_j-equivalent to $(s \geq \mathbf{1} \wedge t \leq -\mathbf{1}) \vee (s \leq -\mathbf{1} \wedge t \geq \mathbf{1})$. Since the two disjuncts represent disjoint areas, the following four lemmas are T_j-entailed by the constraint store: $\{s \geq \mathbf{1}\} \to \{t \leq -\mathbf{1}\}$, $\{t \leq -\mathbf{1}\} \to \{s \geq \mathbf{1}\}$, $\{s \leq -\mathbf{1}\} \to \{t \geq \mathbf{1}\}$, and $\{t \geq \mathbf{1}\} \to \{s \leq -\mathbf{1}\}$. After instantiating the third lemma above with the substitution $\{(a + b)/s, c/t\}$, *cs-extend* is recursively invoked to determine whether the condition $(a + b) \leq -\mathbf{1}$ is T_j-entailed by the constraint store. Since this test succeeds, then the instantiated conclusion of the lemma, namely $c \leq -\mathbf{1}$, is added to the constraint store via the invocation of *cs-simp* and this yields a T_c-unsatisfiable constraint store (since the trivially unsatisfiable inequality $\mathbf{0} \leq -\mathbf{1}$ is obtained by cross-multiplying and adding $c \leq -\mathbf{1}$ with $-c \leq \mathbf{0} \in C'$).

In the general case, it turns out (see [6] for details) that any inequality of the form $s * t \leq k$ is T_j-equivalent to a formula of the form

$$(s \geq \mathbf{1} \wedge t \geq \mathbf{1} \wedge \bigwedge_{j=1}^{m} c_j) \vee (s \leq -\mathbf{1} \wedge t \leq -\mathbf{1} \wedge \bigwedge_{j=1}^{n} d_j) \tag{3}$$

where c_j and d_j are inequalities that are linear in s and t. Furthermore, notice that the conjunct $s \geq \mathbf{1}$ ($t \geq \mathbf{1}$) is mutually exclusive with $s \leq -\mathbf{1}$ ($t \leq -\mathbf{1}$, resp.). This fact allows us to "compile" (3) into the following lemmas:

$$\begin{array}{ll} \{s \geq \mathbf{1}\} \to \{c_1, ..., c_m\} & \{s \leq -\mathbf{1}\} \to \{d_1, ..., d_n\} \\ \{t \geq \mathbf{1}\} \to \{c_1, ..., c_m\} & \{t \leq -\mathbf{1}\} \to \{d_1, ..., d_n\} \end{array} \tag{4}$$

Similar reductions are possible for *elliptical inequalities*, i.e. inequalities of the form $a * s * s + b * t * t \leq k$, where s and t are terms and a, b, and k are constants.

The above ideas can be implemented as shown in Figure 2. *choose_ineq* selects, from the constraint store, an inequality (not marked with the `used` tag, see below) which can be transformed into a special form (e.g. for hyperbolic inequalities, it checks whether a factor occurs in all the multiplicands, except for a constant term). If no such inequality exists in the constraint store, the special element `fail` is returned. *factorize* takes as input an inequality and returns a data structure representing the factorization of the input inequality (e.g. for hyperbolic inequalities, a 4-tuple consisting of two factors, a constant term, and the predicate symbol is returned). *affinize* transforms the factorization of a non-linear inequality into a boolean combination of linear inequalities (such as, e.g., (3)). *compile* returns a list of facts of the form $\{c\} \to \{c_1, ..., c_n\}$ (such as, e.g., formula (4)). *mark_ineq_in_cs* leaves the constraint store untouched if `ineq` has the special value `fail` (and *genlemma* returns the empty list as the second element of the pair), otherwise it returns a new constraint store where `ineq` is marked with the `used` tag in order to avoid reconsidering the same inequality.

```
fun genlemma(C: constraint store)
begin
    ineq := choose_ineq(C);
    cl-list := compile(affinize(factorize(ineq)));
    C' := mark_ineq_in_cs(ineq, C);
    return ⟨C', cl-list⟩
end
```

Figure 2: The Implementation of *genlemma* for Lemma Speculation

4 Discussion

We have presented an extension mechanism which enables decision procedures to tackle problems falling outside the scope they have been originally designed for, thereby considerably enhancing their usefulness in practical applications. As shown in the extended version of this paper [2], the mechanism is both sound and terminating. We have presented an instance of the mechanism that enables a decision procedure for the quantifier-free fragment of Presburger Arithmetic to tackle non-linear problems of significant difficulty. We have coded three versions of our extension schema, i.e. augmentation, lemma speculation, and a combination of the two within **RDL**. Computer experiments carried out using our prototype implementation confirm the validity of the proposed approach. For example, our system is able to decide formula $\mathsf{ms}(\mathsf{c}) + \mathsf{ms}(\mathsf{a})^2 + \mathsf{ms}(\mathsf{b})^2 < \mathsf{ms}(\mathsf{c}) + \mathsf{ms}(\mathsf{b})^2 + 2 * \mathsf{ms}(\mathsf{a})^2 * \mathsf{ms}(\mathsf{b}) + \mathsf{ms}(\mathsf{a})^4$ (where ms is an interpreted function symbol, s.t. $0 < \mathsf{ms}(\mathsf{E})$ holds for any E), by using a combination of augmentation and lemma speculation on top of a decision procedure for Presburger Arithmetic. Notice how such a formula falls well outside the theory of Presburger Arithmetic (more experimental results are reported in [2]). Finally, we have also implemented a version of our extension schema that lifts a decision procedure for the theory of ground equality (already available in **RDL**) to a procedure for the quantifier-free theory of LISP list structure, based on the ideas stated in [7]. An interesting direction of research is to devise suitable requirements on *genlemma* s.t. *cs-extend* is a decision procedure for T_j. This would allow us to obtain the completeness result of [7] for the theory of LISP list structure as a special case of our extension schema. This is part of our ongoing work.

References

[1] A. Armando and S. Ranise. Constraint Contextual Rewriting. In *Proc. of the 2nd Intl. Workshop on First Order Theorem Proving (FTP'98), Vienna (Austria)*, pages 65–75, 1998.

[2] A. Armando and S. Ranise. A Practical Extension Mechanism for Decision Procedures. Technical report, DIST-Università degli Studi di Genova, 2000. Available at http://www.mrg.dist.unige.it/~silvio/docs/ext-decproc.ps.gz.

[3] A. Armando and S. Ranise. Termination of Constraint Contextual Rewriting. In *Proc. of the 3rd Intl. Workshop on Frontiers of Combining Systems (FroCos'2000)*, March 2000.

[4] R.S. Boyer and J S. Moore. Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic. *Machine Intelligence*, 11:83–124, 1988.

[5] J.-L. Lassez and M.J. Maher. On Fourier's Algorithm's for Linear Arithmetic Constraints. *J. of Automated Reasoning*, 9:373–379, 1992.

[6] V. Maslov and W. Pugh. Simplifying Polynomial Constraints Over Integers to Make Dependence Analysis More Precise. Technical Report CS-TR-3109.1, Dept. of Computer Science, University of Maryland, 1994.

[7] G. Nelson and D. Oppen. Fast Decision Procedures Based on Congruence Closure. *J. of the ACM*, 27(2):356–364, April 1980.

[8] R.E. Shostak. Deciding Combination of Theories. *Journal of the ACM*, 31(1):1–12, 1984.

Modeling Sequences within the RelView System

Rudolf Berghammer and Thorsten Hoffmann

Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität Kiel, Olshausenstraße 40, D-24098 Kiel

1 Introduction

Since many years, axiomatic relational algebra (cf. [8]) has been used very successfully for formal problem specification and program derivation. Relations are well suited for modeling and reasoning about many discrete structures and computations on them. This holds in particular for those graph algorithms which manipulate sets of arcs resp. vertices since sets of arcs and relations are essentially the same and there are many simple and elegant ways to model sets of vertices by specific relations like vectors, partial identities, and injective embedding mappings. For details see [8]. Here one finds also examples for relation-algebraic derivations of graph algorithms; further derivations can be found, e.g., in [4, 7, 5].

Sets are not the only datatype used by graph algorithms. Also sequences are very important since many descriptions and algorithmic solutions of fundamental graph problems require different types of them. E.g., to calculate a path means to compute a sequence of vertices and to enumerate the strongly connected components means to compute a sequence of sets of vertices. We claim that in many cases relations can also be used for modeling sequences in a way which is well suited for calculations and formal program derivations. To prove our point we present in this paper a simple relation-algebraic model for sequences via binary direct sums which especially works for the relational manipulation and prototyping system RelView (cf. [3, 1]) and show a typical application.

2 Relation-algebraic Preliminaries

In this section we collect some basic concepts of relational algebra; for more details see [8] for example. We denote the set (or type) of all (binary) relations with domain X and range Y by $[X \leftrightarrow Y]$ and write $R : X \leftrightarrow Y$ instead of $R \in [X \leftrightarrow Y]$. If the sets X and Y are finite and of cardinality m resp. n, then we may consider R as a Boolean matrix with m rows and n columns. Since this Boolean matrix interpretation is well suited for many purposes and also used within the RelView system, in the following we often use matrix terminology and matrix notation. The latter means that we write R_{xy} instead of $(x, y) \in R$.

We assume the reader to be familiar with the basic operations on relations, viz. R^{T} (transposition), $\overline{R}$ (negation), $R \cup S$ (join), $R \cap S$ (meet), RS (composition), $R \subseteq S$ (inclusion), and the special relations O (empty relation), L (universal relation), and I (identity relation). The theoretical framework for all the well-known algebraic properties of relations is that of a relational algebra. As constants and operations of this abstract algebraic structure we have those of the set-theoretic relations; its axioms are those of a complete Boolean lattice for negation, join, meet, ordering, empty and universal relation, the axioms of a monoid for composition and identity relation, the so-called *Schröder equivalences*

$$QR \subseteq S \iff Q^{\mathsf{T}}\,\overline{S} \subseteq \overline{R} \iff \overline{S}\,R^{\mathsf{T}} \subseteq \overline{Q}\,, \tag{1}$$

and the so-called *Tarski rule*

$$R \neq \mathsf{O} \iff \mathsf{L}R\mathsf{L} = \mathsf{L}\,. \tag{2}$$

An immediate consequence of (2) is $\mathsf{O} \neq \mathsf{L}$ which in turn implies that domain and range of each relation are non-empty. This agrees exactly with the use of relations in practice.

A relation R is said to be *reflexive* if $\mathsf{I} \subseteq R$, *transitive* if $RR \subseteq R$, and *symmetric* if $R \subseteq R^{\mathsf{T}}$. By an *equivalence relation* we mean a reflexive, transitive, and symmetric relation. For such a relation domain and range coincide, i.e., it is *homogeneous*, and thus the product RR is defined. In the Boolean matrix model of relations a homogeneous relation is quadratic. An arbitrary (also called *heterogeneous*) relation R is said to be *univalent* if $R^{\mathsf{T}}R \subseteq \mathsf{I}$ and *total* if $R\mathsf{L} = \mathsf{L}$. As usual, an univalent and total relation is said to be a (total) *mapping*. A relation R is called *injective* if R^{T} is univalent and *surjective* if R^{T} is total. An injective and surjective relation is said to be *bijective*. Another class of heterogeneous relations are *vectors*, which are defined by $v = v\mathsf{L}$ and can be used to describe subsets of a given set. If $[X \leftrightarrow Y]$ is the type of v, then $v = v\mathsf{L}$ means that whatever set Z, universal relation $\mathsf{L} : Y \leftrightarrow Z$, and element x from X we choose either $(v\mathsf{L})_{xz}$ holds for all elements z of Z or for none element z of Z. Consequently, for a vector the range is irrelevant. In the following we only consider vectors $v : X \leftrightarrow \mathbf{1}$ with a specific singleton set $\mathbf{1} = \{\perp\}$ as range and omit the second subscript, i.e., write v_x instead of $v_{x\perp}$. Then v can be considered as a Boolean column vector and describes the subset $\{x \in X : v_x\}$ of X. A vector is said to be a *point* if it is injective and surjective. For $v : X \leftrightarrow \mathbf{1}$ these properties mean that it describes a singleton set, i.e., an element of X if we identify a singleton set with its only element. In the Boolean matrix model a point is a Boolean column vector in which exactly one component is true.

The operators on relations we will present now, are introduced in terms of the basic operations and in most cases only partially defined. Let R be a homogeneous relation. The least reflexive and transitive relation containing R is its *reflexive-transitive closure* $R^* = \bigcup_{i \geq 0} R^i$. Since we only deal with set-theoretic relations, the so-called point axiom of [8] holds. It says that for $R \neq \mathsf{O}$ there exist points p, q with $pq^\mathsf{T} \subseteq R$. As a consequence, for each vector $v \neq \mathsf{O}$ there exists a point p fulfilling $p \subseteq v$. The choice of such a point is fundamental for relational programming since it corresponds to the choice of an element from a non-empty set. For a non-empty vector v an axiomatization of the *choice* $\mathsf{point}(v)$ is given by

$$\mathsf{point}(v) \subseteq v \qquad\qquad \mathsf{point}(v) \text{ is a point}. \tag{3}$$

The symmetric quotient $\mathsf{syq}(R,S)$ of relations R and S is defined by

$$\mathsf{syq}(R,S) = \overline{R^\mathsf{T}\overline{S}} \cap \overline{\overline{R}^\mathsf{T} S}\,. \tag{4}$$

The right-hand side of (4) shows that $\mathsf{syq}(R,S)$ is only defined if R and S have the same domain. Then the domain of $\mathsf{syq}(R,S)$ is the range of R and the range of $\mathsf{syq}(R,S)$ is the range of S. Many properties of the symmetric quotient can be found in [8]. In the following lemma we collect some further properties.

Lemma 2.1. Let relations R, S, p, and v be given. Then we have:

(i) If p is a point and $p^\mathsf{T}p = \mathsf{I}$, then $\mathsf{syq}(Rp, Rp) = \mathsf{I}$.
(ii) If p is a point, then $\mathsf{syq}(R,S)p = \mathsf{syq}(R, Sp)$.
(iii) If v is a vector, then $\mathsf{syq}(R,v)$ is a vector.
(iv) If R is an equivalence relation, then $\mathsf{syq}(S,R)R = \mathsf{syq}(S,R)$.
(v) If v is a non-empty vector and $R\mathsf{L} \subseteq \overline{v}$, then $\mathsf{syq}(R,v) = \mathsf{O}$.

Symmetric quotients are closely related to powersets. If we translate (4) in predicate logic notation using the set-theoretic definitions of the basic operations, then the result is

$$\mathsf{syq}(R,S)_{xy} \iff \forall z : R_{zx} \leftrightarrow S_{zy}\,. \tag{5}$$

Now we consider this equivalence for the special case of R being a membership relation $\in\, : X \leftrightarrow 2^X$ and S being a vector $v : X \leftrightarrow \mathbf{1}$. Then $\mathsf{syq}(\in, v)$ is of type $[2^X \leftrightarrow \mathbf{1}]$ and for each set Y from 2^X we have $\mathsf{syq}(\in, v)_Y$ if and only if $\forall z : z \in Y \leftrightarrow v_z$ holds. The latter shows that $\mathsf{syq}(\in, v)$ describes exactly the same set than v but as an element of 2^X instead of a subset of X as v does.

3 A Simple Relation-algebraic Model for Sequences

Our modeling of sequences within RelView is based upon a relation-algebraic characterization of binary direct sums. Within the relational framework it is natural to do the latter by means of the natural injective embedding mappings. This leads to the following relation-algebraic specification: A pair $\imath_1$ and $\imath_2$ of relations is called a *binary direct sum* if

$$\imath_1\imath_1^\mathsf{T} = \mathsf{I} \qquad \imath_2\imath_2^\mathsf{T} = \mathsf{I} \qquad \imath_1^\mathsf{T}\imath_1 \cup \imath_2^\mathsf{T}\imath_2 = \mathsf{I} \qquad \imath_1\imath_2^\mathsf{T} = \mathsf{O}\,. \tag{6}$$

Given sets X_1 and X_2 it is easy to verify that the injective embedding mappings from these sets to the set-theoretic direct sum $X_1{+}X_2$ are a model of (6). Furthermore, by purely relation-algebraic reasoning it can be shown that the binary direct sum is uniquely characterized up to isomorphism by (6).

Based upon a binary direct sum $X_1 + X_2$ with the injective embedding mappings $\imath_1 : X_1 \leftrightarrow X_1{+}X_2$ and $\imath_2 : X_2 \leftrightarrow X_1{+}X_2$, now we define for relations $R : X_1 \leftrightarrow Y$ and $S : X_2 \leftrightarrow Y$ their *relational sum* $R + S : X_1{+}X_2 \leftrightarrow Y$ by

$$R + S = {\imath_1}^{\mathsf{T}} R \cup {\imath_2}^{\mathsf{T}} S\,. \tag{7}$$

This construction behaves like the relation R for all elements from $X_1{+}X_2$ which come from X_1 and like the relation S for all elements from $X_1{+}X_2$ which come from X_2.

Having defined the relational sum, now we are in a position to model non-empty (and finite) sequences of sets and elements in a very simple way. In Section 2 we have already shown how a subset (resp. an element) of a set X can be modeled by a vector (resp. a point) $v : X \leftrightarrow \mathbf{1}$. Therefore, it suffices to model sequences $\mathbf{v} = (v_i)_{1 \leq i \leq n}$ of vectors of type $[X \leftrightarrow \mathbf{1}]$ with relation-algebraic means. As we are interested in algorithms, especially executable with the RelView system, in the following we restrict us to relations $R : X \leftrightarrow Y$ with finite and enumerated carrier sets X and Y. Hence, we are allowed to consider R as a Boolean $m \times n$ matrix with m being the cardinality of X and n those of Y. In this Boolean matrix interpretation, which is also the standard way of RelView to depict a relation on its screen, R models the sequence $\mathbf{v} = (v_i)_{1 \leq i \leq n}$ of vectors, where $v_i : X \leftrightarrow \mathbf{1}$ corresponds to its i^{th} column ($1 \leq i \leq n$). Guided by this model, we define

$$R \,@\, S = (R^{\mathsf{T}} + S^{\mathsf{T}})^{\mathsf{T}}\,. \tag{8}$$

However, it must be pointed out that this concatenation of relations models the usual concatenation of sequences only within RelView because of the specific definition of the relational sum operation of the system which forms the Boolean matrix of $R + S$ by putting the Boolean matrices of R and S one upon another. Independent of the system, for a correct modeling of sequences via relation-algebraic binary direct sums we have to demand that the direct sum $X{+}Y$ of two disjoint and finite sets X and Y the elements of which are enumerated as $x_1, \ldots, x_m$ resp. $y_1, \ldots, y_n$ is enumerated as $x_1, \ldots, x_m, y_1, \ldots, y_n$.

The following lemma states some facts of the concatenation operation of (8) which are used in the next section.

Lemma 3.1. Let relations R, S, and T be given. Then we have:

(i) If $\mathsf{syq}(R,R) = \mathsf{I}$, $\mathsf{syq}(S,S) = \mathsf{I}$, and $\mathsf{syq}(R,S) = \mathsf{O}$, then $\mathsf{syq}(R \,@\, S, R \,@\, S) = \mathsf{I}$.

(ii) $\mathsf{syq}(R, S \,@\, T) = \mathsf{syq}(R,S) \;@\; \mathsf{syq}(R,T)$.

(iii) $(R \,@\, S)\mathsf{L} = R\mathsf{L} \cup S\mathsf{L}$.

This lemma does not fall out of the blue but relation-algebraically formalizes rather clear properties of sequences. We demonstrate this by means of (i): Using matrix terminology, from (5) one immediately obtains that $\mathsf{syq}(R,R) = \mathsf{I}$ holds if and only if the columns of R are pair-wise distinct and $\mathsf{syq}(R,S) = \mathsf{O}$ holds if and only if R and S have no column in common. Changing to sequence terminology, hence Lemma 3.1.i says: If the elements of a sequence $\mathbf{v}$ as well as of a sequence $\mathbf{w}$ are pair-wise distinct and $\mathbf{v}$ and $\mathbf{w}$ have no element in common, then also the elements of their concatenation are pair-wise distinct.

4 An Application: Computing Equivalence Classes

We assume $R : X \leftrightarrow X$ to be an equivalence relation on a finite set X. Our goal is to combine relational algebra and the Dijkstra-Gries program development method (see [6]) to derive formally a RelView program which enumerates the set $\mathcal{C}$ of all equivalence classes of R column-wise as a relation $C : X \leftrightarrow \mathcal{C}$.

First we have to specify the problem by relation-algebraic pre- and postconditions. As we assume the relation R to be the input of the program we want to derive, the precondition $\mathit{pre}(R)$ is obvious:

$$\mathit{pre}(R) \;\stackrel{\wedge}{=}\; \mathsf{I} \subseteq R \,\wedge\, RR \subseteq R \,\wedge\, R = R^{\mathsf{T}}\,.$$

The relation C is the program's output. Using matrix terminology, therefore we have to specify that its columns are pair-wise distinct and describe all equivalence classes of R. In Section 3 we have already shown how to formalize the first of these properties with relation-algebraic means. This leads to $\mathsf{syq}(C,C) = \mathsf{I}$ as first part of the postcondition $\mathit{post}(R,C)$. To formalize also the second of the above properties within relational algebra, we modify the relation-algebraic specification of the strongly connected components given in [2] and get, with $\in\, : X \leftrightarrow 2^X$ as membership relation, $\mathsf{syq}(\in, R)\mathsf{L} : 2^X \leftrightarrow \mathbf{1}$ as vector for describing the equivalence classes of R as elements of 2^X. This leads to $\mathsf{syq}(\in, R)\mathsf{L} = \mathsf{syq}(\in, C)\mathsf{L}$ as second part of $\mathit{post}(R, C)$. In words it says that each column of C describes an equivalence class of R and, conversely, each equivalence class of R is described by a column of C. Altogether, we have the postcondition

$$\mathit{post}(R, C) \;\overset{\wedge}{=}\; \mathsf{syq}(C, C) = \mathsf{I} \wedge \mathsf{syq}(\in, R)\mathsf{L} = \mathsf{syq}(\in, C)\mathsf{L}\,.$$

Our next goal is to calculate a loop invariant and a guard from $\mathit{post}(R, C)$. Here we follow the most common approach of generalizing a postcondition by introducing new variables. In the present case it seems to be a good idea to compute the equivalence classes of R, i.e., the sequence of vectors modeled by C, one after the other and to use a vector for describing the elements yet to be checked. If we use $v : X \leftrightarrow \mathbf{1}$ for the latter purpose, then we arrive at

$$\mathit{inv}(R, C, v) \;\overset{\wedge}{=}\; \mathsf{syq}(C, C) = \mathsf{I} \wedge \mathsf{syq}(\in, R)\,\overline{v}\; = \mathsf{syq}(\in, C)\mathsf{L} \wedge Rv \cap C\mathsf{L} = \mathsf{O}$$

as loop invariant $\mathit{inv}(R, C, v)$. In words its second equation says that the columns of C describe the equivalence classes of the already checked elements (which again are described by $\overline{v}$) and its third equation says that no element of an equivalence class which is still to be computed occurs in an equivalence class which has already been computed.

It is obvious that $\mathit{inv}(R, C, v)$ implies $\mathit{post}(R, C)$ when v is empty. This leads to $v \neq \mathsf{O}$ as guard of the loop. Hence, it remains to develop an initialization of C and v which establishes $\mathit{inv}(R, C, v)$ and a body of the loop which maintains $\mathit{inv}(R, C, v)$ and ensures termination.

Let's start with the initialization. Here it seems to be a good idea to choose arbitrarily an equivalence class, i.e., to initialize C with the vector $R\mathsf{point}(\mathsf{L})$, where $\mathsf{L} : X \leftrightarrow \mathbf{1}$. Consequently, we have to initialize v with $\overline{C}$, i.e., with $\overline{R\mathsf{point}(\mathsf{L})}$. This initialization establishes the loop invariant. In the subsequent proof of this fact we abbreviate the choice $\mathsf{point}(\mathsf{L})$ as p. The first equation $\mathsf{syq}(Rp, Rp) = \mathsf{I}$ of $\mathit{inv}(R, Rp, \overline{Rp})$ is an immediate consequence of Lemma 2.1.i since p is a point due to (3) and from its type $[X \leftrightarrow \mathbf{1}]$ we get $p^\mathsf{T} p = \mathsf{I}$. A proof of the second equation of $\mathit{inv}(R, Rp, \overline{Rp})$ is

$$\mathsf{syq}(\in, R)\,\overline{\overline{Rp}}\; = \mathsf{syq}(\in, R)Rp = \mathsf{syq}(\in, R)p = \mathsf{syq}(\in, Rp) = \mathsf{syq}(\in, Rp)\mathsf{L}\,,$$

where we successively have applied lattice theory, Lemma 2.1.iv in combination with the precondition, Lemma 2.1.ii, and Lemma 2.1.iii. For a proof of its third equation we use that it is equivalent to $R\,\overline{Rp} \subseteq \overline{Rp}$ and prove this inclusion. Symmetry and transitivity of R show $R^\mathsf{T} R \subseteq R$ which in turn implies $R^\mathsf{T} Rp \subseteq Rp$. Now, we apply (1) and are done.

To complete the derivation, we have to work out a loop body and to verify that it maintains $\mathit{inv}(R, C, v)$ and ensures termination. Since C finally shall enumerate the equivalence classes of R and v describes those elements of X during the computation of C the classes of which still have to be added to C, it seems to be promising to explore the effect of the assignments $C := C \mathbin{@} R\mathsf{point}(v)$ and $v := v \cap \overline{R\mathsf{point}(v)}$ which add a new class to C and change v accordingly. In RelView syntax, hence the final program is

```
classes(R)
  DECL conc(R,S) = (R^ + S^)^;
       C, v
  BEG  C = R*point(Ln1(R));  v = -C;
       WHILE -empty(v) DO
         C = conc(C,R*point(v));  v = v & -(R*point(v)) OD
       RETURN C
  END.
```

To prove that its loop body maintains the loop invariant, we assume $v \neq \mathsf{O}$ and $\mathit{inv}(R, C, v)$. Furthermore, we abbreviate $\mathsf{point}(v)$ as p. To prove the first equation of $\mathit{inv}(R, C \mathbin{@} Rp, v \cap \overline{Rp})$ we apply Lemma 3.1.i and must only verify its three premises. Equation $\mathsf{syq}(C, C) = \mathsf{I}$ is part of the supposed invariant $\mathit{inv}(R, C, v)$. A proof of $\mathsf{syq}(Rp, Rp) = \mathsf{I}$ follows from Lemma 2.1.i since p is again a point with $p^\mathsf{T} p = \mathsf{I}$. Finally, to prove $\mathsf{syq}(C, Rp) = \mathsf{O}$ we use $Rv \cap C\mathsf{L} = \mathsf{O}$ since this part of $\mathit{inv}(R, C, v)$ in combination with $p \subseteq v$ shows $C\mathsf{L} \subseteq \overline{Rp}$ which in turn implies the desired result due to the vector property of Rp, the inequation $\mathsf{O} \neq Rp$ (which follows from surjectivity of p and reflexivity of R), and Lemma 2.1.v. Here is the proof of the second equation of $\mathit{inv}(R, C \mathbin{@} Rp, v \cap \overline{Rp})$:

$$\begin{aligned}
\mathsf{syq}(\in,R)\,\overline{v \cap \overline{Rp}} &= \mathsf{syq}(\in,R)\,\overline{v} \cup \mathsf{syq}(\in,R)Rp \\
&= \mathsf{syq}(\in,R)\,\overline{v} \cup \mathsf{syq}(\in,R)p\mathsf{L} && \mathit{pre}(R),\ \text{Lemma 2.1.iv},\ p = p\mathsf{L} \\
&= \mathsf{syq}(\in,R)\,\overline{v} \cup \mathsf{syq}(\in,Rp)\mathsf{L} && p \text{ point, Lemma 2.1.ii} \\
&= \mathsf{syq}(\in,C)\mathsf{L} \cup \mathsf{syq}(\in,Rp)\mathsf{L} && \mathit{inv}(R,C,v) \\
&= \mathsf{syq}(\in, C \mathbin{@} Rp)\mathsf{L} && \text{Lemma 3.1.iii, Lemma 3.1.ii}\,.
\end{aligned}$$

Finally, a proof of the third equation of $\mathit{inv}(R, C \mathbin{@} Rp, v \cap \overline{Rp})$ is given by

$$\begin{aligned}
R(v \cap \overline{Rp}) \cap (C \mathbin{@} Rp)\mathsf{L} &= R(v \cap \overline{Rp}) \cap (C\mathsf{L} \cup Rp\mathsf{L}) && \text{Lemma 3.1.iii} \\
&\subseteq Rv \cap R\,\overline{Rp} \cap (C\mathsf{L} \cup Rp) && p = p\mathsf{L} \\
&= R\,\overline{Rp} \cap ((Rv \cap C\mathsf{L}) \cup (Rv \cap Rp)) \\
&= R\,\overline{Rp} \cap Rp && \mathit{inv}(R,C,v),\ p \subseteq v \\
&= \mathsf{O} && R^{\mathsf{T}}Rp \subseteq Rp,\ (1)\,.
\end{aligned}$$

Since X is finite and $Rp \neq \mathsf{O}$, the above program obviously terminates if $Rp \cap v \neq \mathsf{O}$. We prove this inequation by contradiction: Assume $Rp \subseteq \overline{v}$. Then we get $R^{\mathsf{T}}v \subseteq \overline{p}$ due to (1). Now $\mathit{pre}(R)$ implies $v \subseteq Rv = R^{\mathsf{T}}v \subseteq \overline{p}$, i.e., $p \subseteq \overline{v}$, and combining this with $p \subseteq v$ yields the contradiction $p = \mathsf{O}$.

5 Conclusion

We have presented a relation-algebraic model of sequences. It contrasts with all existing work in this domain since it is specifically tailored to the relational manipulation and prototyping system RelView without touching the system's present state. Afterwards, we have combined relational algebra with the Dijkstra-Gries program development method to derive formally a RelView program for the column-wise enumeration of equivalence classes.

Experiments have shown that, using higher-order constructs like the membership relation $\in\, : X \leftrightarrow 2^X$, our approach often allows to specify a problem involving sequences as an executable RelView-function which then may be used for protyping. See [2, 4, 3] for many examples. We have found it also very attractive to use such functions for producing good examples in teaching.

Strictly speaking, we only have presented a RelView-model of non-empty sequences with a concatenation operation. But it is easy to refine it to a model of stacks with the additional well-known operations first and rest using the relation-algebraic operators inj for the injective mapping generated by a vector and init for the initial element of a carrier set. Due to lack of space we have to renounce details. We have applied this refinement to solve many further problems involving sequences with RelView, e.g., graph-theoretic ones (strongly connected resp. biconnected components, vertex bases, Eulerian cycles), Petri-net problems (reachable resp. live markings of CE-nets), and problems on ordered sets (cut resp. ideal completion). To give an impression for running times, for the well-known five dining philosophers CE-net computing all reachable markings takes one second on a Sun Ultra V workstation.

References

1. Behnke R., Berghammer R., Meyer E., Schneider P.: RelView – A system for calculating with relations and relational programming. In: Astesiano E. (ed.): Proc. FASE '98, LNCS 1382, Springer, 318-321 (1998)
2. Berghammer R., Gritzner T., Schmidt G.: Prototyping relational specifications using higher-order objects. In: Heering, J. et al. (eds.): Proc. HOA '93, LNCS 816, Springer, 56-75 (1994)
3. Berghammer R., Karger B. von, Ulke C.: Relation-algebraic analysis of Petri nets with RelView. In: Margaria T., Steffen B. (eds.): Proc. TACAS '96, LNCS 1055, Springer, 49-69 (1996)
4. Berghammer R., Karger B. von: Algorithms from relational specifications. In: Brink C. et al. (eds.): Relational methods in Computer Science. Advances in Computing Sci., Springer, 131-149 (1997)
5. Berghammer R., Karger B. von, Wolf A.: Relation-algebraic derivation of spanning tree algorithms. In: Jeuring J. (ed.): Proc. MPC '98, LNCS 1422, Springer, 23-43 (1998)
6. Gries D.: The science of computer programming. Springer (1981)
7. Karger B. von, Berghammer R.: Computing kernels in directed bichromatic graphs. Inform. Proc. Let. 62, 5-11 (1997)
8. Schmidt G., Ströhlein T.: Relations and graphs. Discrete Mathematics for Computer Scientists, EATCS Monographs on Theoret. Comput. Sci., Springer (1993)

Automated Testing with RT-Tester - Theoretical Issues Driven by Practical Needs

Markus Dahlweid[1] Oliver Meyer[1] and Jan Peleska[1,2]

[1] TZI, Universität Bremen, P.O. Box 330440, D-28334 Bremen,
e-mail: {dahlweid,emm,jp}@tzi.de
[2] Verified Systems International GmbH, Bremen

Abstract. The RT-Tester tool has been developed by Verified Systems International in cooperation with the TZI at Bremen University in order to support automated testing for reactive real-time systems. In this article, we give an overview on theoretical issues concerning variants of timed transition systems and their interpretation in hard real-time which have been stimulated by the practical requirements of automated test generation, execution and on-the-fly evaluation. We sketch future developments with the objective to include testing of hybrid systems into our theoretical framework and in the tool architecture of RT-Tester.

1 Introduction

1.1 Motivation: Formal Methods and Testing

This article discusses issues about automated testing of reactive real-time systems. The authors consider formal verification and testing as complementary activities that are both part of the quality assurance process. Ideally, the product-related quality assurance tasks would be split between formal verification and testing as follows:

- Logical correctness properties of requirements specifications, design specifications and code should be formally verified.
- The proper integration of software, firmware and hardware should be tested.
- The reliable operation of controllers should be tested by *built-in test equipment* which monitors operations and performs on-the-fly checks of compliance with the specified behaviour.
- Completeness properties of requirements specifications which cannot be deduced from other reference specifications should be validated by a combination of formal verification and simulation, that is, testing on symbolic specification level.

For today's reactive real-time systems — at least when they perform safety-critical or mission-critical control tasks — a high degree of automation is required. Otherwise it would be infeasible to achieve the necessary degree of test coverage and to perform regression testing on new product revisions within acceptable time/cost margins. As a consequence, testing has to be based on formal specifications which can be interpreted by computers in an automatic way. To substantiate the claim that the test process is *trustworthy* in the sense that it uncovers deviations of the product behaviour from its specification, testing has to be related to formal verification. As a consequence, automated testing has close links to formal methods: It is regarded by the authors as a variant of model checking, where the implementation model is not completely known.

1.2 Overview

In this article, we will introduce the RT-Tester test automation tool (Section 2). In section 3, three important theoretical problems related to automated test generation and test evaluation will be presented.

2 The RT-Tester Tool

The RT-Tester tool is a generic system for automated hardware-in-the-loop tests and software integration tests which can be instantiated for different types of hardware interfaces and operating systems. It performs automatic test generation, test execution and test evaluation by compiling formal test specifications and interpreting them in real-time. The functional components of RT-Tester can be structured as shown in Figure 1; the *System Under Test (SUT)* denotes the object to be tested.

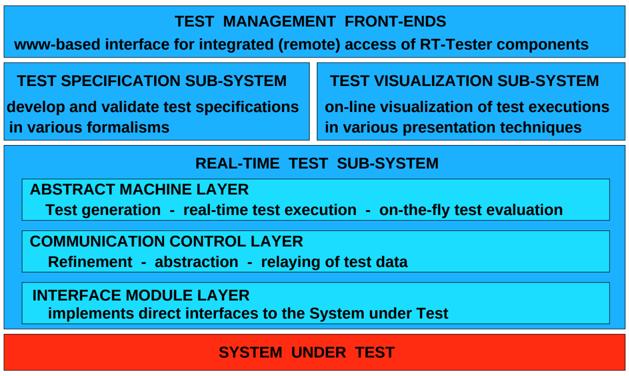

TEST MANAGEMENT FRONT-ENDS
www-based interface for integrated (remote) access of RT-Tester components
TEST SPECIFICATION SUB-SYSTEM
develop and validate test specifications in various formalisms
TEST VISUALIZATION SUB-SYSTEM
on-line visualization of test executions in various presentation techniques
REAL-TIME TEST SUB-SYSTEM
ABSTRACT MACHINE LAYER
Test generation - real-time test execution - on-the-fly test evaluation
COMMUNICATION CONTROL LAYER
Refinement - abstraction - relaying of test data
INTERFACE MODULE LAYER
implements direct interfaces to the System under Test
SYSTEM UNDER TEST

Fig. 1. Functional structure of RT-Tester.

The core component of RT-Tester is the *Real-Time Test Sub-System*. It is responsible for on-the-fly test generation, execution of tests in real time and on-the-fly test evaluation (Figure 2). The RT-Tester *test engine* generates test executions from various types of specifications, drives the execution in real-time, monitors the target system response and accommodates the test execution according to SUT response and test coverage strategy. Deviations of SUT behaviour from the specification can be either detected on-the-fly or by *a posteriori* analysis of the test execution logs maintained by the test engine. The test specifications describe inputs to be exercised on the SUT and expected SUT response in an abstract way without having to refer to concrete data. These specifications are interpreted by processes in the *abstract machine layer (AML)* in real-time, thereby creating timed sequences of abstract input events and monitoring abstract output events.

To facilitate interfacing to special purpose hardware, the RT-Tester configuration typically makes use of *interface modules (IFM)* acting as adapting devices between the test engine and the SUT. Interface modules receive data from the test engine on a standard interface and pass it to the associated special-purpose hardware interfaces used by the SUT. Conversely, interface modules collect outputs from the SUT interfaces and pass them on to the test engine. The task of the *communication control layer (CCL)* is to relay events between abstract machines and/or interface modules. Together with the AML, the CCL resides in the test engine. The mapping from abstract events to concrete data that may be passed to the SUT interface as well as the abstraction of raw SUT output data to abstract events that may be consumed by the abstract machines is performed in the IFM and/or CCL using specialised processes in combination with a general purpose RT-Tester *event mapping library*.

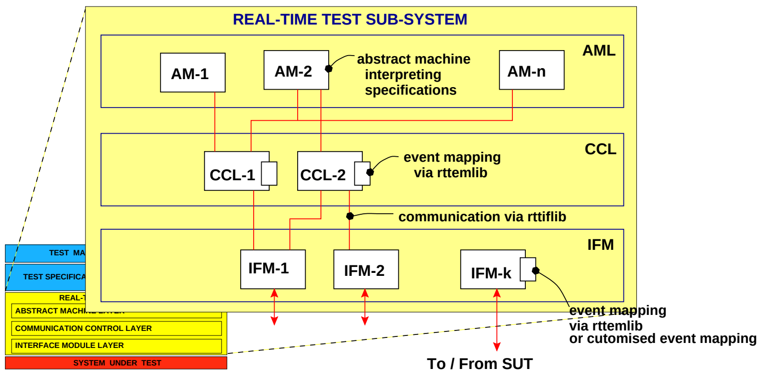

REAL-TIME TEST SUB-SYSTEM
AML
AM-1
AM-2
abstract machine interpreting specifications
AM-n
CCL
CCL-1
CCL-2
event mapping via rttemlib
communication via rttiflib
IFM
IFM-1
IFM-2
IFM-k
event mapping via rttemlib or cutomised event mapping
To / From SUT
TEST MA
TEST SPECIFICA
REAL-T
ABSTRACT MACHINE LAYER
COMMUNICATION CONTROL LAYER
INTERFACE MODULE LAYER
SYSTEM UNDER TEST

Fig. 2. Generic RT-Tester configuration for hardware-in-the-loop tests.

In addition to the real-time test sub-system, the other RT-Tester components shown in Figure 1 support test management (test activation, monitoring, documentation and configuration management), test visualisation (on-the-fly animation of timed input/output traces with different graphical views), and test specification in different formalisms. Specifications are compiled into a combination of binary transition graph structures and executable code. The compiled specifications are executed by real-time interpreters which generate inputs to the SUT and perform on-the-fly checks of SUT responses by means of transition graph interpretation in combination with execution of pre-compiled code. At present, Timed CSP (TCSP) [9], Moby Timed Automata [2], and simple variants of hybrid extensions of TCSP [1] are supported.

Since 1994, RT-Tester has been used for embedded systems testing in the field of space application, avionics, railway control systems and telecommunications. A detailed tool description can be found in [10]. The theoretical foundations related to untimed testing have been published in [4–6], the results related to real-time testing are currently prepared for publication by the authors of this article.

3 Test Automation: Three Fundamental Problems

3.1 Structural Decomposition of TCSP Specifications

In order to use TCSP specifications for automated testing it is necessary to have a means of executing them in real-time. The execution is indispensable for the generation of inputs to the SUT in response to preceding SUT outputs, as well as for the on-the-fly evaluation of SUT outputs.

Timed execution of programs is generally based on the employment of timers, which can be simply realized in an operating system by using counters and a real-time clock. Therefore, the main idea for the execution of TCSP is the decomposition of arbitrary TCSP specifications into a number of simple and independent timer processes, which can be directly realised by operating system timers, and a CSP part referencing the untimed operators only, while synchronising with the timers by means of auxiliary events. Theorem 1 describes the decomposition properties, where $\mathcal{L}_1$ denotes the subset of TCSP which is able to represent networks of sequential timed processes, i. e. a language which sets up a hierarchy of high-level (concurrency-) and low-level (sequential-) operators.

Theorem 1. *There exists a transformation $\Omega : \mathcal{L}_1 \to TCSP$ mapping each process P defined in syntax $\mathcal{L}_1$ to a process $\Omega(P) = (\Omega_U(P) \underset{\Omega_E(P)}{\|} \Omega_T(P)) \setminus \Omega_E(P)$ such that*

1. *$\Omega(P)$ is semantically equivalent to P in the timed failures semantics of TCSP.*
2. *$\Omega_U(P)$ refers to untimed CSP operators only.*
3. *$\Omega_E(P)$ is a set of auxiliary timer events $set.s$, $ela.s$ where the indices s denote unique timer identifiers of independent interleaved timer processes in $\Omega_T(P)$.*
4. *Each subprocess $Timer_s$ of $\Omega_T(P)$ is defined according to the pattern $Timer = \mu\, T \bullet (set?t \to ((WAIT\ t;\ ela\ \to T) \,\square\, T))$. Then, a concrete timer for index s is a renaming: $Timer_s = Timer\ [[set \leftarrow set.s,\, ela \leftarrow ela.s]]$* $\square$

For processes with single timeouts the conversion is quite straightforward, e. g. the untimed part of a process

$$P = (a \to \mathrm{STOP}) \overset{t}{\rhd} (b \to \mathrm{SKIP})$$

corresponds to

$$\begin{aligned} \Omega_U(P) = set.1!t \to\ & (a \to \mathrm{STOP}) \\ & \square \\ & (ela.1 \to b \to \mathrm{SKIP}) \end{aligned}$$

The transformation becomes much more complicated for nested timeouts, combinations of timeouts and choices and for time-recursive processes. A simple example of the latter is the process

$$P = Q \overset{t}{\rhd} (R \overset{s}{\rhd} P)$$

where it is possible that no event occurs for an unconstrained period of time, but the behaviour of the process switches constantly between Q and R. By finding the right combinations of active timeouts and generating new process references for these, the decomposition transformation is guaranteed to terminate in such cases.

In order to cope with arbitrary choice and timeout combinations, the transformation Ω_U uses a Head Normal Form for sequential TCSP processes. The occurrences of the currently relevant Internal Choices ($\sqcap$), External Choices ($\square$) and Timeouts ($\rhd$) in the Parse Tree for every sequential TCSP process can be rearranged in the mentioned order. This makes it possible to transform occurrences of Internal Choices correctly, whereas naive realisations lead to semantically wrong solutions. The combination of External Choices and Timeouts introduces a context dependability which is solved by always collecting all relevant subprocesses and merging their Head Normal Forms.

The synchronisation mechanisms and timer process implementations for applying decomposed TCSP specifications for real-time testing are already integrated in the RT-Tester tool.

3.2 Normalised Timed Transition Graph Encodings

In order to provide executability of TCSP specifications in hard real-time, they are pre-compiled by RT-Tester into transition graph representations. The pre-compilation process induces the problem of state space explosions, in some cases even for seemingly simple and practically relevant specifications. Just as in model checking, this problem may be tackled using a combined *location/variable representation* of the specification, where the full state space is partitioned into locations represented as nodes in a graph and a variable space providing additional state information. In a location/variable representation, additional guards must be introduced for transitions: The decision whether a transition may be taken not only depends on the present location, but on the present valuation of the variables as well. Furthermore, transition labels must be augmented by variable assignments.

In the "classical" generation of transition graphs from CSP specifications, as defined in [8, 7], each transition leading from any state is labelled with exactly one event. The size of the transition graph representing the process $P = a?x \to P$ therefore depends on the definition of the channel a. The channel can communicate an arbitrary finite number of values, and the resulting transition graph has the same number of edges leaving one node as shown on the left hand side of Figure 3. A location/variable representation of this process is much closer to the original CSP specification, where there is just one edge leaving the node. The edge contains the label of the event and an assignment of the possible values to the local variable x.

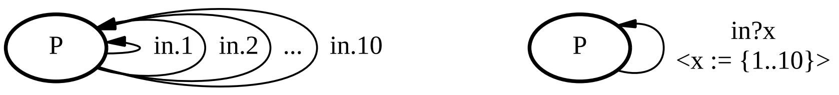


Fig. 3. Different types of transition graphs.

The location/variable representation is more compact than the conventional graphs, since the size of the graph no longer depends on the number of events in the alphabet. More complex examples may be given, where the classical representation would lead to an infeasible number of graph nodes, while the location/variable representation can still be handled.

The first step of the algorithm developed in order to produce location/variable transition graphs is based on the operational semantics of CSP. There are a number of rules defined for each CSP operator to produce an unnormalised graph with events, conditions and assignments attached to the edges of the graph. The next step is a normalization process, where the τ events are eliminated and the graph is determinised:

- Merge all nodes, which can be reached from a given node using only τ transitions, into a new state.
- Add the assignments of the used edges to the the new state.
- Combine all conflicting assignments.
- Each non-τ transition of any of the merged nodes is added to the new state.
- Transitions labeled with events of the same name are combined. The after-state of these edges are determined by guarding conditions connected to the edges.

The resulting transition graphs are much smaller than normal CSP graphs, but introduce one serious complication: The expressions of the assignments have to be evaluated on the traversal of the graph structure during the test execution. Therefore it can be useful to combine the conventional graph generation algorithms with the specified algorithm. This is possible, since the condition-event-assignment representation is an augmentation of the conventional event notation for edges.

3.3 Semantic Framework: Henzinger's Hybrid Automata

From a practical point of view, it is a natural requirement that

- different test specification formalisms should be applicable in parallel and contribute to the same test case,
- both time-discrete and time-continuous SUT interfaces should be stimulated and monitored by the test system.

RT-Tester provides solutions for both requirements: In one test case execution, different Abstract Machines may process compiled test specifications written in Timed CSP, Moby Timed Automata and hybrid extensions thereof. They communicate using standardised abstract channel interfaces for discrete data and shared memory areas dedicated to time-continuous float-values which may be communicated to/from the SUT using D/A and A/D converters, respectively.

It should be noted, however, that this heuristic requires to verify that the combination of formalisms used for test specification and test execution is really consistent with the SUT requirements defined in possibly yet another specification formalism. To this end, our present approach is to embed all formalisms which may be used in the RT-Tester tool into Henzinger's Hybrid Automata [3] which have sufficient expressive power to allow for such an encoding. As a consequence, the combined use of different formalisms may be regarded as a more convenient notation, where it would be tedious to express every test specification item as a Hybrid Automaton.

4 Conclusion

In this article, we have introduced the RT-Tester tool and listed important theoretical problems which have to be solved in order to provide practical and implementable solutions for the task of automated reactive systems testing. In the full version of this article, the authors will give more detailed presentations of these research problems and sketch several other questions which arose from practical considerations but led to new theoretical investigations.

References

1. P. Amthor: *Structural Decomposition of Hybrid Systems. Test Automation for Hybrid Reactive Systems.* Dissertation, Bremen University, FB3, October 1999.
2. Dierks, H.: PLC-Automata: A New Class of Implementable Real-Time Automata. In M. Bertran and T. Rus, editors, Transformation-Based Reactive Systems Development (ARTS'97), volume 1231 of Lecture Notes in Computer Science, pages 111-125. Springer-Verlag, 1997.
3. Thomas A. Henzinger. The Theory of Hybrid Automata. In *11th Annual IEEE Symposium on Logic in Computer Science (LICS 1996)*, pages 278–292, 1996.
4. J. Peleska: Test Automation for Safety-Critical Systems: Industrial Application and Future Developments. In M.-C. Gaudel and J. Woodcock (Eds.): FME '96: Industrial Benefit and Advances in Formal Methods. LNCS 1051, Springer-Verlag, Berlin Heidelberg New York (1996) 39-59.
5. J. Peleska: Formal Methods and the Development of Dependable Systems. Habilitationsschrift, Bericht Nr. 9612, Dezember 1996, Institut für Informatik und praktische Mathematik, Christian-Albrechts-Universität Kiel (1997).
6. J. Peleska and M. Siegel: Test Automation of Safety-Critical Reactive Systems. *South African Computer Jounal* (1997)19:53-77.
7. A. W. Roscoe: Model-Checking CSP. In *A Classical Mind, Essays in Honour of CAR Hoare.* Prentice-Hall International, Englewood Cliffs NJ (1994), 353-378.
8. A.W. Roscoe: The Theory and Practice of Concurrency. Prentice Hall, 1998
9. S. Schneider. An Operational Semantics for Timed CSP. *Information and Computation*, 116:193–213, 1995.
10. Verified Systems International GmbH: *RT-Tester 3.1 – User Manual.* Bremen (2000).

Automatic test generation for Java-Card applets

Lydie Du Bousquet*, Hugues MARTIN**

* IRISA,
Campus de Beaulieu – 35042 Rennes Cedex
ldubousq@irisa.fr

** Gemplus Research Lab., Gemplus,
Parc d'Activités de Gémenos – B.P.100 – 13881 Gémenos Cedex – France
hugues.martin@gemplus.com

1. Introduction

The specific domain of smart cards is close to the domain of embedded devices [VV99]. The applications require to be strongly validated before being deployed on thousands or millions of eventually personalized copies. The Open Card platforms have introduced the possibility to change and/or to update applications embedded during the full life cycle. This possibility of loading and executing new applications during the card life has raised new problems for embedded application validation, as developers and testers do not have necessary cards when they develop new applications. However, the quality and security requirements are still the same as for traditional smart cards. Developers and testers need to use methods and techniques that are compliant with a high security level and that fulfil industrial software development constraints such as time to market.

In this article, we propose a solution for Java Card application validation. We take benefits of object oriented software engineering, and we adapt them to the Java Card world. In order to perform this, we take into account the fact that the only permanent elements of the card are the Java Card Virtual Machine (JCVM) and the Operating System (OS), and that most of security requirements are based on it. In order to optimize the application validation process, we integrate the JCVM and OS validity as test hypotheses. Our main goal is then to verify the application conformity to the specification, and to verify the applet integration with other loaded applets within the card.

This article is structured in the following manner. Section two presents our process of Java Card application validation. Section three describes an experimentation of our process on a case study. Finally, section four is devoted to the conclusions and actual works.

2. Java applet validation

To perform applet validation, we use a conformance testing approach [Tre99] and proceed in four steps. We first detail the specification of the applet. Then, we express some test purposes and automatically derive abstract test cases from the specification and the test purposes. Finally, the abstract test cases are transformed into executable test cases.

2.1. Validation framework

Testing is an operational way to check the correctness of a system implementation by means of experimenting it. It increases confidence in the quality of a system, especially when proof is not possible or too expensive.

In our solution, we use a conformance testing approach, which a black-box testing approach. The key points are that there is a specification and a system implementation exhibiting behaviors. The specification is a prescription of what the system should do. These specification suppose to be detailed enough, that is to say it should describe all the expected behaviors. Moreover, it is supposed to be correct. The goal of conformance testing is to check whether the implemented system satisfies this specification.

For the present work, we express the specification with a UML model [UML99]. The specification is automatically translated into a Labeled Transition System (LTS) thanks to the UMLAUT tool [JLP98]. Then we use TGV[FJJ96] to automatically produce test cases from this LTS and from test purposes produced by hand.

2.2. An approach for modeling

There exists more than one process to develop UML models such as in [BJR99]. However, in most of them, the test workflow is based on human responsibility concerning test goal elicitation. Our goal is to help human tester to avoid the risk to forget test purposes or to define redundant test cases, by applying systematic or automatic test generation process based on application models.

UML provides different diagrams and notations to describe different point of view of one application. A recurrent question is to identify which diagrams should be used. Validation process provides one answer to this question, using the test generation process to drive the UML specification step. From the testing point of view, we consider that objects are entities that encapsulate data, states and operations. We use class diagrams to specify data and operations, statechart diagrams to represent states and their evolutions, and constraints on state transitions to represent data abstraction evolutions.

2.3. Test purposes

A precise definition of a test purpose can be founded in [FJJ96]. Informally, a test purpose can be defined as a set of behaviors that should be observable on the implementation. Considering smart card validation, the number of test purposes can be very large compared to the application size, depending on the abstraction level of the model. In case of Java Card application, an important part of security features is provided by the Java Card platform. As we suppose the platform validity, some tests purposes dedicated to specific smart card constraints can be removed.

One test strategy for conformance testing of Java Card applications consists in testing each function for every normal use and every possible misuse. Test purposes can be generated starting from common UML practice; every use case and every associated sequence diagrams will lead to test purposes. Then, every previously generated test purpose is analyzed in order to identify possible pertinent misuses, which become new test purposes associated with expected error codes. For the resulting test cases, these misuses include wrong parameters in operation call, unexpected execution context, wrong order in the operation call...

It is important to remark that as UML models are an abstraction of the application, test purposes integrate *de facto* hypotheses made to build this abstraction [Mar99].

3. Case study

3.1. Description

The case study considered consists in a purse applet. This applet has been developed in the PACAP project, partially funded by MENRT 98B0251. It provides most common functions such as debit, credit, and balance. It also provides administrative functions in order to manage usability constraints. In its full version, it contains nine operations intended to the cardholder and forty operations intended to the applet manager. However, for the purpose of this article, we provide only the model of the purse credit function.

3.2. UML model

In order to credit the purse, the user has to perform two actions: an initialization (*appInitCredit*, see figure 1), which aims at configuring a secure communication, and an "acknowledgement" (*appCredit* operation). For security reasons, the *appInitCredit* operation succeeds only if the user is authenticated. The key, computed for the secure messaging, has a limited lifetime. It is invalidated by the application if another operation than *appCredit* is performed just after an *appInitCredit* operation, or after an *appCredit* operation. Moreover, a credit can be performed only if the resulting balance value is not higher than a predefined value stored in the *MaxBalance* variable (*MB* in the following figures).

In order to represent this behavior, we use a UML model composed of three objets, which implement the secure messaging protocol (USM), the user authentication (UB) and the balance (BAL).

Figure 1 presents the statechart that corresponds to the behavior associated to the secure messaging protocol.

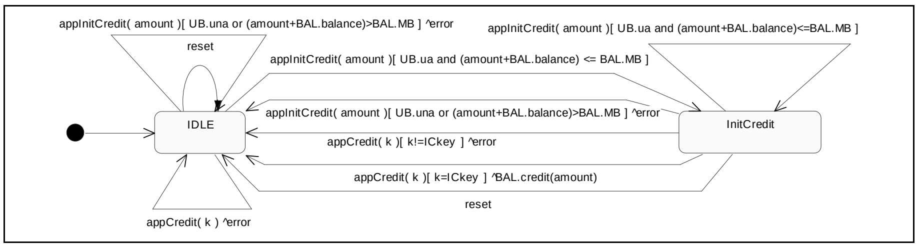

appInitCredit(amount)[UB.una or (amount+BAL.balance)>BAL.MB] ^error
reset
appInitCredit(amount)[UB.ua and (amount+BAL.balance)<=BAL.MB]
appInitCredit(amount)[UB.ua and (amount+BAL.balance) <= BAL.MB]
IDLE
appInitCredit(amount)[UB.una or (amount+BAL.balance)>BAL.MB] ^error
InitCredit
appCredit(k)[k!=ICkey] ^error
appCredit(k)[k=ICkey] ^BAL.credit(amount)
reset
appCredit(k) ^error

Figure 1. Limited Purse behavior (USM)

A credit can be done only if the user has been previously authenticated (condition *UB.ua* that is true if the state-machine of UB is in state *ua* - user authenticated) and if the resulting amount is still less or equal to the maximum amount possible in the purse. The secure messaging is simply represented by a test on the key value *k*.

The user authentication (Figure 2) is performed by another applet within the card. As the security policy applied to the user authentication is in charge of this other applet, we do not consider the fact that after three failed authentication, the authentication applet is blocked.

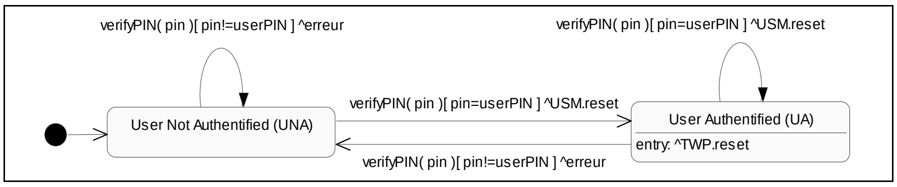

verifyPIN(pin)[pin!=userPIN] ^erreur
verifyPIN(pin)[pin=userPIN] ^USM.reset
verifyPIN(pin)[pin=userPIN] ^USM.reset
User Not Authentified (UNA)
User Authentified (UA)
entry: ^TWP.reset
verifyPIN(pin)[pin!=userPIN] ^erreur

Figure 2. Purse authentication behavior (UB)

Finally, the balance value (Figure 3) is constrained between zero and the maximum value *MaxBalance* (MB).

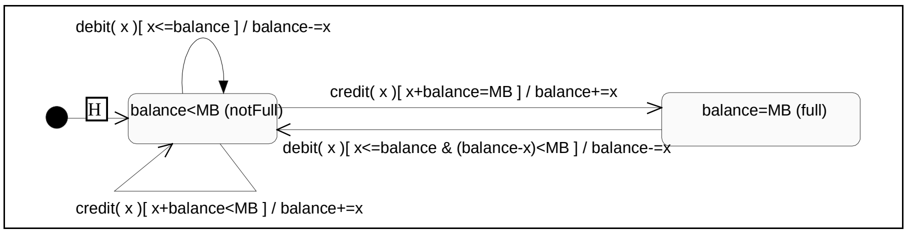

debit(x)[x<=balance] / balance-=x
credit(x)[x+balance=MB] / balance+=x
H
balance<MB (notFull)
balance=MB (full)
debit(x)[x<=balance & (balance-x)<MB] / balance-=x
credit(x)[x+balance<MB] / balance+=x

Figure 3. Balance behavior (BAL)

3.3. Informal test purposes

To test the credit function, four operations are involved: *initVerifyPIN()*, *verifyPIN(userPin)*, *appInitCredit(amount)* and *appCredit(key)*. A subset of the abstract test purposes corresponding to a correct use of the credit function can be a set of sequences respecting this order:

1. The user authenticates himself by calling the *verifyPIN(PIN)* operation one or several times with his PIN.
2. The user initializes the credit operation by calling the *appInitCredit(amount)* operation one or several times with an amount that respects the balance value constraints.
3. The user credits the purse by calling once the *appCredit(key)* operation with the right secure messaging key.

A subset of the abstract test purposes corresponding to wrong operation sequences can be:

A. After an unsuccessful authentication, the credit initialization fails even with right parameter values.
B. After a successful authentication and an unsuccessful credit initialization, a credit fails.

Other abstract test purposes can be identified, like stress tests, test under limits, card pull-out, etc.

It is important to remark that for a bigger applet, even for our full case study, abstract test purposes can involve much more security mechanisms. These mechanisms usually imply more complicated behaviors. Then, applying automatic test generation tools like UMLAUT enables to optimize test generation process.

These abstract test purposes have to be translated into formal test purposes. In order to use TGV, the test purposes must be expressed in LTS. They are more powerful than UML sequence diagrams, since cycles can be represented. In figure 4, we represent the abstract test purposes A and B by LTS.

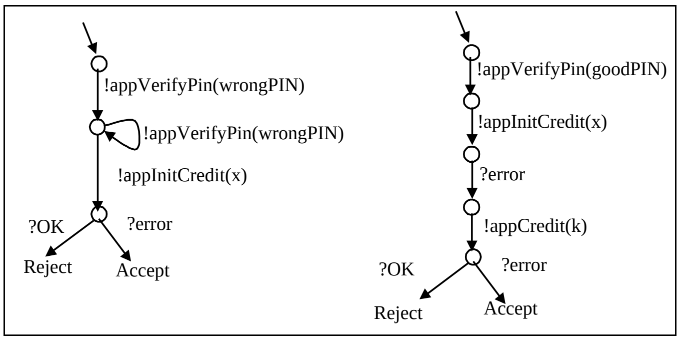


Figure 4. Test purposes A and B represented with LTS

4. Conclusions

In this article, we have briefly summarized a solution to verify properties on the Java Card embedded platform. By assuming the validity of this platform, we have proposed a methodology that integrates software-engineering practice and smart card security needs. This methodology consists in expanding the application UML model in order to increase the conformity verification. In our point of view, this approach allows to offer a high confidence in the application conformity regarding its UML specification.

The next step in our, which has yet started, is to generate concrete test suites that can be applied to the implementation. The aim of these concrete test suites execution on the implementation is to validate our approach and to compare it to a traditional manual testing approach.

References

[BJR99] G. Booch, I. Jacobson, J. Rumbaugh. The Unified Software Development Process. Addison Wesley, 1999.

[FJJ96] J. -C. Fernandez, C. Jard, T. Jéron, C. Viho. Using on-the-fly verification techniques for the generation of test suites. In Conference on Computer-Aided Verification (CAV '96), New Brunswick, New Jersey, USA, A. Alur, T. Henzinger (red.), LNCS 1102, Springer, July 1996.

[JLP98] J. -M. Jézéquel, A. Le Guennec, F. Pennaneac'h. Validating distributed software modelled with UML. In Proc. Int. Workshop UML98, Mulhouse, France, June 1998

[Mar99] H. Martin. Using test hypotheses to build a UML model of object-oriented smart card applications. In ICSSEA'99: International Conference on Software & Systems Engineering and their Applications, Paris, France, December 1999.

[Tre99] J. Tretmans. Testing concurrent systems: A formal approach. In J.C.M Baeten and S. Mauw, editors, CONCUR'99 - 10th Int. Conference on Concurrency Theory, volume 1664 of Lecture Notes in Computer Science, pages 46-65. Springer-Verlag, 1999.

[UML99] Unified Modeling Language Specification, Version 1.3, June 1999. Information available at http://www.omg.org/cgi-bin/doc?ad/99-06-08

[VV99] J.-J. Vandewalle, E. Vétillard. Developing Smart Card-Based Applications with Java Card. In ERSADS'99: Third European Research Seminar on Advances in Distributed Systems, Madeira Island - Portugal, 23-28 April 1999.

Formal data mining and visualization at procedure level

Nikolai Mansurov[1], Djenana Campara[2], Elena Laskavaya[3], Igor Ivanov[4]

[1] Head of Department for CASE tools, Institute for System Programming, Moscow, Russia
email: nick@ispras.ru

[2] Chief Technology Officer, inSight project, Nortel Networks, Ottawa, Canada

[3] Researcher, Department for CASE tools, Institute for System Programming, Moscow

[4] Researcher, Department for CASE tools, Institute for System Programming, Moscow

1. Extended Abstract

One of the biggest challenges for modern telecommunications software development organizations is the need to shorten *time-to-market* of their products. The organization which is "last to market" will find it difficult to acquire market share. Lately, time-to-market has become the dominating factor for industrial success, placing enormous pressure on manufacturers and developers to spend less effort on quality and performance, or to find a much more efficient means of producing high-quality software for their products and services.

One of the most promising approaches to increased productivity is through using more powerful tools. Efficient tools, in turn, require *formal models* in order to allow validation and transformation of software development artifacts. Research into Formal Methods and Formal Description Techniques (FDTs) has only recently begun to focus on the development of powerful, scalable, open tools, which support these methods. Therefore, the current situation creates an opportunity for new generation formal model based Computer Aided Software Engineering (CASE) tools focused on improving time-to-market for development of conventional, not necessarily safety-critical software. The data mining methodology, presented in this paper, is part of the Accelerated Development Methodology (ADM) for specification, design, testing and re-engineering of telecommunications software [2,3]. ADM is an attempt to use powerful tools, standard languages, concurrency in validation and test case creation, automated synthesis and automated code generation to significantly decrease time-to-market without sacrificing the quality of product.

Research into Formal Methods and Formal Description Techniques (FDTs) has only recently began to focus on the development of powerful, scalable, open toolsets which support these methods. Certainly, formal methods have been well-suited and standardized in the world of communications protocols, services and software. In this paper, we consider an integrated suite of languages standardized by the International Telecommunications Union (ITU): *Specification and Description Language* (SDL) [4], a use case scenario description language called *Message Sequence Charts* (MSC) [5]. ITU-T Specification and Description Language (SDL) is one of the most successful telecommunications standard FDTs [2,6].

Many formal methods exist, but were not being adopted due to an *accessibility barrier*, and an *efficiency barrier*. For each formal method, only a few hundred experts were available, certainly not an adequate number for global industrial needs. Computer Aided Software Engineering (CASE) methods and tools used to be rigid and not adaptable to a variety of engineering practices, and therefore could not be efficiently integrated into existing development processes. Thus, these methods could

not be widely adapted (accessibility barrier) or could only be used manually due to rigidity of existing CASE tools (efficiency barrier).

Fortunately, a new generation of much more accessible, scalable, and designer-driven CASE tools have been developed by such organizations as Telelogic [12], ObjecTime, and Rational, among others [2,3]. The tools help to overcome the accessibility and efficiency barriers, although some familiarity with either SDL or UML is expected. This is not a serious problem, however, since most graduates of Information Technology or Software Engineering program are familiar with one or both of these languages.

These industrial-strength commercial tools are able to edit MSCs, analyze SDL specifications, perform validation and verification of SDL specifications based on state-exploration algorithms, automatically and semi-automatically (designer-guided) generate abstract TTCN test cases from SDL specifications and also automatically generate implementations for real-time operating systems [2,3]. A number of industrial case studies have been recently completed, claiming improved quality, much lower development costs and speedup in time-to-market of up to 20-30% due to the use of SDL-based CASE tools [6]. "Success stories" of using SDL in industry mention the phases of system design, detailed design, automatic generation of implementations as well as formal verification and testing [6,12].

However, there is an important issue which needs to be addressed in order for formal methods-based CASE tools for communications software engineering to become common practice. Formal methodologies are only applicable to the so-called "*green-field*" projects, in which the system is developed completely from scratch. However, most projects in the industrial context involve the older, "*legacy*" base software. This software is being maintained, updated by developing new features, or reused in new projects. For the formal methods to be adopted in industry, it is necessary to provide cost-effective methods for integrating CASE-produced components and systems with older, "legacy" base software. Legacy software systems were produced with older development methods, often involving a blend of higher-level code, and system-level code, with heterogeneous languages, architectures, and styles, and often very poorly documented. Up to now, this fact has constituted a "*legacy barrier*" to the cost effective use of formal methods-based development technologies and tools [2,3,8].

In order to overcome the "legacy barrier", there is an increasing demand for developing automatic (or semi-automatic) re-engineering methods which will significantly reduce the effort involved in creating formal specifications of the base software platforms. Cost-effective methods for producing SDL models of the base software platform will allow the following benefits:

- better understanding of the operation of the legacy software through dynamic simulation of the SDL model, which often produces more intuitive results and does not involve the costly use of the target hardware;
- automated generation of regression test cases for the base software platform;
- analysis and validation of the formal specifications of the new features built on top of the SDL model of the base software platform;
- feature interaction analysis including existing and new features;
- automated generation of test cases for new features;

- automatic generation of implementations of the new features. Such implementations are retargetable for different implementation languages (e.g. C, C++, CHILL) as well as for different real-time operating systems (e.g. pSOS, VxWorks, etc.).

This presentation addresses several issues of automatic extraction of formal models from legacy software, primarily within the context of ITU-T standard specification language SDL. This work extends our previous work in automatic extraction of formal models of legacy [8,9,10] by combining the recovery of formal models with less formal approach of data mining and visualization [1].

There exist several levels at which formal models of legacy software can be extracted: *micro level* (single procedure, single data structure, algorithm) to *macro level* (class, component). This process is also complemented by powerful notations for the *middle level*, describing high-level behavior of components, collaborations and interactions between components [5,7]. Current software engineering methodologies are becoming macro level centric (architecture-centric). However, increased focus on the macro level of software does not exclude the requirement to represent algorithms at the micro level. Therefore, modern visual specification languages, for example ITU-T standard SDL, include notations for all three levels: macro level (components, high-level structure of the software system, relations between components), middle level (interactions between components, statecharts) as well as the micro level (flowcharts). The current version of the Unified Modeling Language (UML) [7] has only first two levels; however the synergy between UML and SDL is the subject of on-going discussions [4].

Modern visual languages, such as UML and SDL, are focused on forward engineering activities [4,5,7]. Several industrial-strength CASE tools exist, which support software developments using these notations, e.g. [12]. However, this introduces a gap between support available for forward engineering activities (development of new software from scratch), compared to software maintenance. Developers are getting more and more used to applying visual languages for new development, yet they are forced to deal mostly with linear text when doing debugging and maintenance, or even when developing new features for old (legacy) software. Research indicates, that the vast majority of software vendors spend up to 50% of their budget on maintenance activities, and this share even reaches 70-80% in some companies [11]. Obviously, this number is growing, since more software is being developed and deployed. Therefore we believe, that for wider adoption of visual languages it is imperative to allow developers access to visual diagrams of the existing (legacy) software code. This is particularly important for micro level, because, according to [13], “about 38% of programmer’s time is spent in understanding a software system because code written in multiple files in textual format is hard to read”.

In the presentation we are addressing the issues of formalized data mining and visualization at micro level (procedure level). Firstly, we show that there is a certain conflict between the objective of formal data mining, i.e. extracting formal artifacts, and the objective of visualization, i.e. extracting semi-formal or even informal artifacts (also called diagrams) such that they enhance our understanding of the software. We show that this conflict should be resolved by careful selection of the formal notation for recovery of formal model.

Secondly, we consider the formal artifacts, which can be extracted at procedure level. We focus our presentation on extracting two kinds of artifacts:

- control flow patterns
- paths through the program

We show how these two artifacts resolve the above conflict between formal data mining and visualization and provide a powerful conceptual platform for supporting software maintenance phase. Our approach is based around the standard specification language SDL. We claim, that reverse engineering of legacy code into visual diagrams increases efficiency of software maintenance activities and training new personnel. Graphical patterns are more important for understanding of software than for engineering new software. Automatic reverse engineering of flowcharts from source code is often neglected, yet it makes code inspections and debugging more efficient. It allows better understanding of the structure and functionality of code at procedure level.

Thirdly, we address the issue of integrating the information, obtained at the micro level with the macro level (architecture level) information. This problem is particularly important to achieve the balance between formality and improvements in understanding, because the amount of information available at the micro level is very big. We show, that it is imperative to have on-the-fly synthesis of micro-level information and a navigation capability, which will control on-demand extraction of relevant micro-level diagrams.

All the above was among the main motivations of the inSight project, which was started at Nortel Networks in 1995. Department for CASE tools of the Institute for System Programming (ISP) was contracted to perform R&D for this project. In particular, the project benefited from the language processing and data mining expertize of ISP [10]. The result of the collaboration was the reverse engineering toolkit, called inSight [1], now available commercially. The main objective of inSight toolkit is to extract architecture-level models. However, we have implemented the above ideas to extend the tool for the micro level, since we believed that it is essential to address the micro level of software maintenance. One of our main objectives was to improve efficiency of code inspections.

We describe the architecture of the FlowChart tool of the inSight toolkit, which implements our approach to data mining and visualization at procedure level. We outline the underlying challenges: parsing of the partial source files; presentation of control-flow patterns; navigation capability, controlling on-demand synthesis of micro level information.

We demonstrate the results of a case study, in which we tried to quantify the benefits of using pattern-based flow graphs for solving a typical code inspection problem. Comparison with related research is provided and some future research directions are outlined.

2. References

1. N. Rajala, D. Campara, N. Mansurov, inSight Reverse Engineering CASE Tool, in Proc. of the ICSE'99, Los Angeles, USA, 1998.
2. R. Probert, N. Mansurov, Improving time-to-market using SDL tools and techniques (tutorial), Proc. 9th SDL Forum, Montreal, Canada, June 21-26, 1999.

3. R. Probert, N. Mansurov, Improving time-to-market using SDL tools and techniques, submitted to special issue of Computer Networks and ISDN Systems, 2000
4. ITU-T (1993), CCITT Specification and Description Language (SDL), ITU-T, June 1994
5. Z.120 (1996) CCITT Message Sequence Charts (MSC), ITU-T, June 1992
6. R. Probert, H. Ural, A. Williams, J. Li, R. Plackowski, Experience with rapid generation of functional tests using MSCs, SDL, and TTCN, submitted to Special Issue of Formal Descriptions Techniques in Practice of Computer Communications, 1999
7. J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modelling Language Reference Manual, Addison-Wesley, 1999
8. N. Mansurov, R. Probert, Dynamic Scenario-based Approach to re-engineering of legacy telecommunication software, in Proc. SDL'99 The Next Millenium, ed. R. Dssouli et. at., Elsevier, NY, 1999, pp. 325-340
9. N. Mansurov, Automatic Synthesis of SDL from MSC in Forward and Reverse Engineering, in Proc. 1st Int. Symposium on Visual Formal Methods (VFM'99), Ed. S. Mauw, et.al., Eindhoven, The Netherlands, 1999, Computing Science Reports, Department of Mathematics and Computing Science, Eindhoven University of Technology, report 99-08, pp. 44-64
10. N. Mansurov, E. Laskavaya, A. Ragozin, A. Chernov, On one approach to using SDL-92 and MSC for reverse engineering, in Voprosy kibernetiki: System Programming Applications, N. 3, Moscow, 1997 (in Russian)
11. R. Arnold, "Software Engineering", IEEE Computer Society Press Tutorial, EH0360-8, 1993
12. Telelogic AB (1998), Telelogic ORCA and SDT 3.4, Telelogic AB, Malmoe, Sweden, 1998
13. International Software Automation, Inc, http://www.softwareautomation.com/www/reengin.html

Data Structures for Z Testing Tools

(Extended Abstract)

Mark Utting
Waikato University, Hamilton, New Zealand
Email: `marku@cs.waikato.ac.nz`

1 Introduction

Tools for analyzing Z specifications can be classified into two main groups, based on whether they attempt to show universal or existential properties:

Proof-like Tools: This includes type checkers and theorem provers. The goal of using these tools is to show that the specification has some desirable property *for all possible inputs and outputs.* The main emphasis of these tools is symbolic formula manipulation.

Testing Tools: This includes test-case checkers and animators. The goal of these tools is to check some property of a given set of test data, or to generate some data values that satisfy the specification (or contradict it, in the case of model checkers). Although these tools must perform some symbolic formula manipulation, their main emphasis is on manipulating or evaluating specific ground terms (i.e., terms that contain no variables).

The two classes of tools are complementary. Testing tools are best used early in the system life cycle, to validate specifications against example input-output test sets, or to execute specifications to detect errors (though not all specifications are executable [HJ89]).

Implementation techniques for the proof-like tools are similar to other theorem-proving applications, and there is much literature about theorem proving, type-checking etc. The fundamental data structure of these tools is an abstract syntax tree of the specification, and this is transformed in various ways (e.g., rewriting) to prove the desired results.

Testing tools require different implementation techniques. This paper describes some of the issues that are important for Z testing tools: handling undefined terms (Section 2), managing the balance between general symbolic formula manipulation and evaluation of ground data values (Section 3) and a variety of data structures for representing sets (Section 4)

The techniques described here have been implemented in the *Jaza* tool. Jaza is a new, open-source, animator for the Z specification language that is written in Haskell. It currently provides only a text-based interface (a GUI is planned), but handles most of Spivey Z [Spi92] except for generics. The decision to develop Jaza came from our difficulties with using other animators, such as poor support for quantifiers or various less-often-used Z constructs (μ, λ, θ terms), unpredictable performance characteristics, and inability to handle non-deterministic schemas.

The main contributions of this paper are (1) to show that testing tools need to support a wider variety of data structures for representing sets than proof-like tools, (2) to demonstrate that the multiple set-representation approach described in this paper is a viable technique and (3) to describe some of the architecture and techniques used in Jaza so that other testing tools (for Z and other specification languages) can benefit from the ideas and lessons learnt.

2 Undefinedness

Z uses a 'loose' approach to handling undefined terms [Spi92]. There is no specific *undefined* value. Instead, the result of applying a partial function to a value outside its domain is simply any arbitrary value of the correct type. The logic of Z is two-valued, so $3/0 = 1 \lor 3/0 \neq 1$ is always true, as is $1/0 = 1/0$. However, little else can be deduced about an undefined term. For example, it is not possible to prove whether the set $\{ x : 0 \mathrel{..} 3 \bullet 6/x \}$ has three or four elements.

Provers can support this approach simply by not providing any inference rules for reasoning about undefined terms, other than the basic logical laws shown above. However, for testing tools, we need a more specific solution, so that we can keep track of exactly which terms are defined and which are not.

One possible approach would be to assign a specific return value to each partial function. For example, decide that division by zero will always return 42. However, this would mean that the value returned from evaluating 3/0 would be more specific than in Z, which does not commit to a specific value. This could lead to returning true for a predicate that other Z tools evaluate to false (or vice versa), which is undesirable.

A better approach is to add an explicit *undefined* value to each type, so that we can carefully define how functions deal with undefined inputs. Most functions need to be strict on undefined inputs, but logical operators have more freedom, since $\mathit{true} \lor X$ is true regardless of X.

Jaza implements this approach by defining an *ErrorOr t* type that wraps exception handling around any type of term *t*. The *ErrorOr* type contains an *Ok* alternative, which contains a defined value of type *t*, plus an *Undef* alternative, which means that the result is undefined. It also contains other alternatives which are used for various classes of errors, such as syntax and type errors, 'search space too large', 'undecidable equality', etc. This *ErrorOr* type is wrapped around the evaluation results of all Z schemas, expressions *and predicates*.[1]

3 An Animation Architecture

It is tempting to take a theorem-proving approach to evaluating schema on test cases — repeatedly apply simplification rules until no further simplification is possible. However, this *symbolic rewriting approach* is inefficient compared to how compiled programming languages evaluate expressions (a single bottom-up pass). Jaza attempts to get the best of both approaches by applying a fixed sequence of steps to each schema or expression evaluated. The philosophy is to perform most of the 'expensive' analysis (simplifying terms, determining computation order, reasoning about the free variables of each expression etc.)

[1] So Jaza effectively uses a three-valued logic. This is slightly more strict than Z requires, but has the benefit that it is possible to report if a predicate like $P \lor \neg P$ depends upon undefined terms.

at the beginning, so that the brute-force 'search for a solution' phase at the end can run as quickly as possible. The phases are:

1. **Parse:** The LaTeX input is parsed into an abstract syntax tree (AST). The Z grammar is ambiguous, so some ambiguities are left in the AST.

2. **Unfold:** This phase uses context to resolve all ambiguities, then expands all schema calculus operators, and unfolds all references to other schemas and global definitions. Several Z constructs (e.g., λ, θ and schemas-as-types) are translated into set comprehensions. The resulting term is saved in the specification database until the user calls it.

3. **Substitute Values:** When the user invokes an operation schema, and supplies values for some of the inputs and/or outputs, the term corresponding to that schema is retrieved from the database and the given values are substituted into it.

4. **Simplify:** Simplification rules, such as one-point rules, are applied in a bottom-up pass through the term, so that the schema is specialized to the given input/output values. (If all inputs are supplied, this typically fully evaluates the precondition and narrows a disjunctive schema down to one case).

5. **Optimize:** This reorders declarations and predicates to minimize searching. This phase performs extensive analysis of the free variables of each predicate, and looks for useful consequences of predicates that can help to reduce searching. For example, from the predicate $a \frown b = c$ we can deduce three facts: `c = a \cat b` (which determines a unique value for c once a and b are known); `a \prefix c` (which narrows the search for a to $\#c + 1$ possibilities once c is known); and `b \suffix c` (which similarly narrows the search for b once c is known).

   The result of this phase is a sequence of intermixed declarations and predicates, where each declaration has the form $x : \bigcap\{C_1, C_2 \ldots C_n, T\}$ where C_i are the constraints on x that have been deduced from predicates and T is the original type of x. This narrows the search space for x, often to a single value if there are equalities involving x. Each predicate appears as early as possible in the sequence, immediately after all its free variables have been declared. The goal is to push each filter predicate as early as possible, so that unsuccessful search paths are pruned as early as possible.

6. **Search:** This phase uses the common 'generate-and-filter' approach to search for a solution of the optimized declaration sequence D. Each $x : T$ in D is executed by assigning each member of T in turn to x, with the rest of D being evaluated to determine whether the chosen values are acceptable. When each predicate is evaluated, all its free variables have ground values, so evaluation can be done efficiently in a single bottom-up pass, just like in programming languages.

   This process returns a (lazy) list of all possible bindings that satisfy the schema. If D is not in a context like $(\forall D \bullet P)$ or $\#\{D \bullet E\}$, which requires all solutions to be known, then it is often only necessary to fully evaluate the first one or two solutions. Jaza provides user-level commands for stepping through the solutions of a non-deterministic schema.

If the schema is obviously deterministic and the optimization phase detected this, then no searching will be needed. To ensure reasonable response times for non-deterministic schemas, we limit the search space. The depth of the search is bounded by the length of the declaration sequence and the width is bounded by a user-settable parameter. A useful trick allows us to detect large search spaces in constant time without actually performing the whole search: if we are searching $[a : T_a, b : T_b, c : T_c \mid P]$ with a maximum width of 100,000, and each of the types contain 100 values, we choose the first value of a and note that there are 100 alternatives to try, then choose the first value of b and note that there are now $100*100$ alternatives, then proceed to c and immediately raise an exception, because the search space now contains $100 * 100 * 100$ alternatives.

4 Multiple Set Representations

Breuer and Bowen [BB94] classify Z animators by how they represent sets:
(a) sets must be finite and are modelled by finite arrays/lists;
(b) sets may be countably infinite, are modelled by an enumeration algorithm;
(c) sets are cardinally unbounded and modelled by their characteristic function.

For example: ZANS [Jia] and ZAL [MSHB98] use approach (a) only; Breuer/Bowen use (b) only; Prolog-based Z animators typically use approach (c) [DKC89, DN91] and Z-- supports both 'passive' sets (a) and 'active' sets (c) [Val91].

However, each of these approaches is good for some kinds of sets, but disastrous for others. Approach (a) does not allow infinite sets, which are widely used in Z, but has the advantage that efficient algorithms and data structures can be used for the finite sets. Approach (b) amounts to programming with (potentially infinite) lazy lists ('streams'), so binary operators must address fairness issues and typically suffer from termination problems. Approach (c) is good for membership queries, but makes it difficult to enumerate the members of a set or calculate its cardinality. Note that proof-like tools tend to use only a symbolic representation of sets, which is closest to approach (c). This makes manipulation of finite, enumerated sets awkward and inefficient.

The best solution is to use *all* the above representations for sets, plus other representations that are customized to specific kinds of sets. Jaza supports at least 12 different representations of sets! Each set is kept in its optimal representation, and translated into another representation only when an operator requests it. A typical implementation of an operator handles several representations using special algorithms, then translates any remaining representations into a standard form so that it can apply a standard algorithm. (An error will be raised if the translation is impractical). Let us look at some of Jaza's most important set representations.

ZFSet [ZValue]: A finite set of values that are all in 'canonical' form[2] is represented by a sorted list of values (without duplicates). This makes the

[2] A value is in canonical form iff it is entirely constructed from basic values (integers and members of given sets) and the tuple, binding, free-type and ZFSet constructors. Canonical values have useful properties such as: no further evaluation is possible, no undefinedness can arise, and semantic equality is the same as syntactic equality.

common set operations ($\cap, \cup, \setminus, \#, \in$) linear in the size of the sets.

ZSetDisplay [ZExpr]: Finite sets of arbitrary terms are also represented by lists, but slower $O(N^2)$ algorithms must be used, because the list may contain two elements that are syntactically distinct, yet semantically equal.

ZSetComp [ZGenFilt] ZExpr: Set comprehensions contain a sequence of intermixed declarations and predicates that are optimized as described in Section 3, plus an expression for the outputs of the set. This representation can sometimes be executed to produce a finite set, but if the search space is too large for that, it is left in this ZSetComp form. Membership tests and function applications (λ expressions are always translated into ZSetComp form) can often be processed without enumerating the set. Instead, the alleged member is unified with the output expression of the set and this often narrows the search space. For example: $\{x : \mathbb{N} \bullet (x, x * 2)\}$ is infinite so cannot be enumerated, but when applied (as a function) to the argument 3, then optimized, we get $\{x : \{3\} \bullet (x, x * 2)\}\ 3$, which is easily evaluated.

ZIntSet (Maybe Integer) (Maybe Integer): This represents a contiguous range of integers, with optional upper and lower bounds. For example, if both bounds are missing, it means $\mathbb{Z}$; if the lower bound is 0 it means $\mathbb{N}$ and if both bounds are given, it means $a \mathinner{..} b$. Membership, cardinality and intersection tests can be done in constant time with this representation, and it is very useful for narrowing search spaces by intersecting integer inequalities.

ZFuncSet: This data structure represents function/relation spaces. It stores the domain and range sets, plus seven boolean flags that determine what kind of relational space it is (total, onto, functional, etc.). Typically, this is used as a type, and the most common operation is testing a given set of pairs for membership in the type. However, calculating the intersection of two function spaces is easy with this representation. The predicate $\operatorname{dom} f = s$ can be represented as $f \in \mathit{ZFuncSet}\{\mathit{domset} = s\}$ and this is an elegant way of narrowing the search space for a function value. Similar data structures are used for power sets and cross products.

Relating these to the Breuer/Bowen classification, ZFSet and ZSetDisplay are approach (a), while ZSetComp and ZFuncSet are approach (c). Jaza does represent any sets directly using approach (b), but provides coercion functions that turn the other set representations into a stream of values.

The danger of having multiple set representations is that it can lead to a combinatorial explosion in the algorithms that are needed for each binary operator (12*12 possible input formats). However, our experience so far in Jaza shows that this problem can be overcome by defining a comprehensive set of coercion functions (which raise exceptions when necessary), and by providing query functions that return hints about the size and nature of sets.

5 Experimental Evaluation

This will be an experimental comparison of how several common Z animators, and Jaza, handle a common set of test cases taken from the literature.

References

[BB94] Peter T. Breuer and Jonathan P. Bowen. Towards correct executable semantics for Z. In J.P. Bowen and J.A. Hall, editors, *Z User Workshop, Cambridge 1994*, Workshops in Computing, pages 185–209. Springer-Verlag, 1994.

[DKC89] A. J. J. Dick, P. J. Krause, and J. Cozens. Computer aided transformation of Z into Prolog. In J.E. Nicholls, editor, *Z User Workshop, Oxford 1989*, Workshops in Computing, pages 71–85, Berlin, Germany, 1989. Springer-Verlag.

[DN91] Veronika Doma and Robin Nicholl. EZ: A system for automatic prototyping of Z specifications. In S. Prehn and W. J. Toetenel, editors, *VDM '91: Formal Software Development Methods*, volume 551 of *LNCS*, pages 189–203. Springer-Verlag, 1991.

[HJ89] I. J. Hayes and C. B. Jones. Specifications are not (necessarily) executable. *IEE/BCS Software Engineering Journal*, 4(6):330–338, November 1989.

[Jia] Xiaoping Jia. An approach to animating Z specifications. Available from `ise.cs.depaul.edu` with the ZTC and ZANS tools.

[MSHB98] Ian Morrey, Jawed Siddiqi, Richard Hibberd, and Graham Buckberry. A toolset to support the construction and animation of formal specifications. *The Journal of Systems and Software*, 41(3):147–160, 1998.

[Spi92] J. Michael Spivey. *The Z Notation: A Reference Manual.* International Series in Computer Science. Prentice-Hall, second edition, 1992.

[Val91] Samuel H. Valentine. Z--, an executable subset of Z. In J.E. Nicholls, editor, *Z User Workshop*, Workshops in Computing, pages 157–187, Berlin, Germany, 1991. Springer-Verlag. Proceedings of the Sixth Annual Z User Meeting, York, 16-17 December 1991.

The Korrigan Environment

Christine Choppy[1], Pascal Poizat[2], and Jean-Claude Royer[2]

[1] LIPN, Institut Galilée, Université Paris XIII,
Avenue Jean-Baptiste Clément, F-93430 Villetaneuse, France
Christine.Choppy@lipn.univ-paris13.fr
[2] IRIN, Université de Nantes
2 rue de la Houssinière, B.P. 92208, F-44322 Nantes cedex 3, France
(Pascal.Poizat,Jean-Claude.Royer)@irin.univ-nantes.fr
http://www.sciences.univ-nantes.fr/info/perso/permanents/poizat/

1 Introduction

In this paper, we present a framework to specify mixed systems, *i.e.* systems with both dynamic parts (control, communication, and concurrency) and data types. While the importance of mixed formal specifications is widely accepted, there is still a need for open[1] and extensible tools and environments. Another need is to integrate the formal specification into a software process (*e.g.* an object-oriented one). Therefore in our environment, we need editing and formating tools, verification means, as well as prototyping and code generation tools. Object-orientation is often advocated for programming and we would like to extend this to specifications and specification developments.

Our model of mixed systems is based on the notion of views [4]. This model aims at keeping advantage of the languages dedicated to both aspects (algebraic specifications for data types, and state-transitions diagrams for dynamic behaviour) while providing a unifying model with an operational semantics.

2 The Korrigan Specification Model

Our model focuses on the specification of systems with both static and dynamic aspects and that feature a certain level of complexity that requires the definition of structuring mechanisms. We use two ways to ensure structuring and modularity: a simple form of inheritance and the composition of specification components.

Our model is based on the notion of *view*, an interface to describe components. The key concept behind this notion is the *Symbolic Transition System* (STS) concept. STSs [9] are a general form of finite state-transition diagrams which provides an appropriate level of abstraction and avoids state explosion by the use of open (*i.e.* not ground) terms in states and transitions. The dynamic aspects of components are described in *dynamic views* (cf. Figure 1). The static aspects are described using *static views*. The integration of all aspects of a given component is done using *integration views*. Finally the compositional aspects of components are described by *composition views*. Both integration and composition views use a mixed "glue" (algebraic first-order axioms and temporal formulas) to express the interface of composition as a whole. A great part of the semantics of the model is devoted to explain how to compute a global view structure for the different compositions [4].

[1] Open meaning here, that it should be possible to link them with other existing tools.

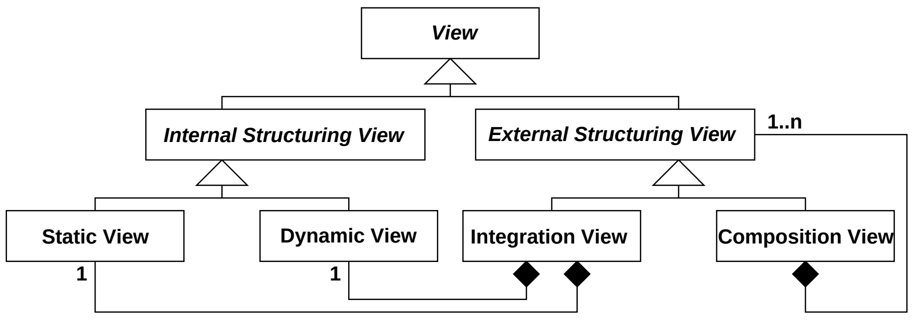


Fig. 1. The View Model Class Diagram

3 The Korrigan Environment

The design of our environment follows several principles. Since our model focuses on mixed systems, we need to interface with several kinds of dedicated tools and environments. The first principle is to interface with some existing tools and environments, for example model checking tools (*e.g.* XTL in CADP [6]), theorem provers (*e.g.* the Larch Prover [7], KIV [20], ELAN [3], PVS [15]), and compilers (*e.g.* Javac [8]). Since such tools are numerous and evolve, our framework has to be extensible. A second principle is to provide general tools which can be useful to other environments or formalisms. For example the CLAP tool (detailed below) can be used to compute (a)synchronous compositions of any state-transition diagrams (automata, Petri Nets, symbolic transition systems). To achieve these two principles we reuse some object-oriented features both in the design and in the implementation of our environment.

The implementation is done in PYTHON [12, 1]. PYTHON is an interpreted object-oriented language, hence it is really useful to produce both quick scripts and prototypes of complex environments. One important feature is that PYTHON is free, open source and portable across several platforms (Unix, Linux, Windows . . .). It is closely related to Lisp, Perl, and C++, but it is much more legible. It is dynamically type-checked, functional and object-oriented. It has a simple meta-object protocol and provides exceptions, powerful built-in data structures and module libraries (parser generation, CORBA programming, XML parsing, ...).

The first environment principle leads to define a classification of specification formalisms (cf. Figure 2). Each class has (at least) methods to parse and print its specific internal format. A general parsing mechanism is defined for the corresponding files. We also provide a library of interface formats, for example to target CASL-LTL [19], Larch Prover [7], LOTOS [11] or Xfig. Whenever one wants to translate KORRIGAN specifications into another format, a cast method to the corresponding class is to be defined.

3.1 Description of the Main Components

Class Library for Specifications. This is an extensible hierarchy mapping the specification classification. It contains classes for the KORRIGAN model, but also for other formalisms. Each class in the hierarchy has an associated parsing class (cf. shadow boxes in Figure 2).

Interface Library. This is a library of formats for documentation (HTML and LaTeX files) and for visualization (DOT [5] and Xfig files). A special format, dedicated to state-transition diagrams is used in the CLAP tool (this original tool is described below). Other kinds

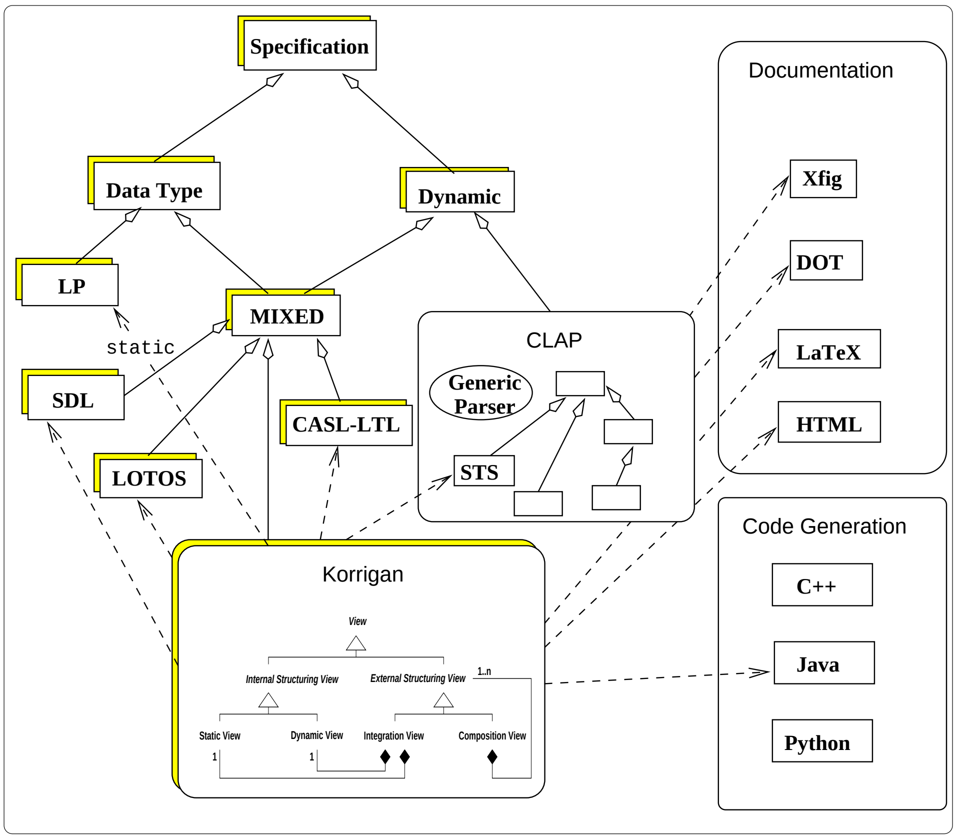


Fig. 2. The KORRIGAN Hierarchy

of formats are defined for code generation in object-oriented languages. We also plan to integrate formats for other verification tools and also for other formal languages.

Code Generation. The code generation is achieved with the following steps (see [17] for more details). The dynamic part, a symbolic transition system with communications and structuration, is implemented by controller structures. These structures are then translated into an object-oriented language (for now Active Java [13]). When the algebraic part is not executable it is refined into an executable one (through interactions with the user). Algebraic specifications are translated into an intermediate object-oriented code based on Formal Classes [2], and implemented into pure Java.

Translation to LOTOS and SDL. We have defined translation mechanisms to generate LOTOS or SDL specification from a KORRIGAN specification [16, 17]. These mechanisms use patterns to generate the dynamic behaviours. The translation of the static part is straightforward.

Verification Tools. It is possible to generate inputs to verification tools using the previous translation mechanisms. For example we can use the CADP environment to verify LOTOS specifications resulting from the translation. However we plan to define and implement specific verification tools. Mixed specifications require specific symbolic verification means

[18, 10]. Another idea is to translate the KORRIGAN specification into a specific formalism where both static and dynamic aspects used the algebraic framework (*e.g.* CASL-LTL), and then verify this resulting specification using a theorem prover.

3.2 The CLAP Tool

CLAP (*Class Library for Automata in Python* [14]) enables one to define different kinds of state-transition diagrams by providing an extensible hierarchy of classes. For example, there are automata with state or transition parameters (initial states, labels, emissions, receipts, colours), Petri Nets... It is easy to add a new class corresponding to some new kind of state-transition diagram. The state-transition diagrams are stored in files following a generic internal format. A parser for this format is provided. The diagrams may be automatically transformed to displaying formats (Xfig or DOT).

CLAP allows one to define simple formulas in order to compute the initial state, the sets of states and transitions reachable from the initial state, and synchronous products. Amongst the synchronous product parameters, we have: the synchronization formulas for transitions and states, the reachable states formulas. Functions are also used as parameters to define the resulting states and transitions. Therefore it is possible to define the product of two state diagrams that have different types, and obtain a state diagram of a third type.

4 Conclusion

The proposed environment supports our specific model, KORRIGAN, to specify mixed systems. This environment follows two principles: openness and extensibility. According to these principles, it provides translation tools to interface with other formalisms, e.g. LOTOS, SDL, CASL-LTL ... We also have a generic tool to describe state-transition diagrams, to compute their (a)synchronous composition, and to compute their graphical representation. Object-oriented source code is generated from KORRIGAN specifications, and interface to documentation languages like HTML or LaTeX is available.

The KORRIGAN environment is based on a classification of specifications with a general parsing mechanism. New formalisms may be integrated, and translation mechanisms for them may be defined.

References

1. Python.org homepage. `http://www.python.org/`.
2. Pascal André, Dan Chiorean, and Jean-Claude Royer. The Formal Class Model. In *Joint Modular Languages Conference, Modula, Oberon & friends*, ISBN 3-89559-220-X, pages 59–78, Ulm, Germany, September 1994.
3. P. Borovansky, C. Kirchner, H. Kirchner, P.E. Moreau, and C. Ringeissen. An Overview of ELAN. In In C. and H. Kirchner, editors, *2nd International Workshop on Rewriting Logic and its Applications*, September 1998. Pont à Mousson, France.
4. Christine Choppy, Pascal Poizat, and Jean-Claude Royer. A Global Semantics for Views. In *International Conference on Algebraic Methodology And Software Technology, AMAST'2000*, volume 1816 of *Lecture Notes in Computer Science*, 2000.
5. John Ellson, Emden Gansner, Eleftherios Koutsofios, John Mocenigo, Stephen North, Gordon Woodhull, David Dobkin, and Vladimir Alexiev. Graphviz. `http://www.research.att.com/sw/tools/graphviz/`.
6. Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. CADP: A Protocol Validation and Verification Toolbox. In *8th Conference on Computer-Aided Verification*, pages 437–440, New Brunswick, New Jersey, USA, 1996.

7. Stephan Garland and John Guttag. An overview of LP, the Larch Prover. In *Proc. of the third International Conference on Rewriting Techniques and Applications*, volume 355 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.
8. Gosling, Joy, and Steele. *The Java Language Specification*. Addison Wesley, 1996.
9. M. Hennessy and H. Lin. Symbolic Bisimulations. *Theoretical Computer Science*, 138(2):353–389, 1995.
10. Carron Kirkwood and Muffy Thomas. Towards a Symbolic Modal Logic for LOTOS. In *Northern Formal Methods Workshop NFM'96*, eWIC, 1997.
11. L. Logrippo, M. Faci, and M. Haj-Hussein. An Introduction to LOTOS: Learning by Examples. *Computer Networks and ISDN Systems*, 23:325–342, 1992. `ftp::/lotos.csi.uottawa.ca`.
12. Mark Lutz. *Programming Python*. O'Reilly & Associates, 1996.
13. Juan M. Murillo, Juan Hernández, Fernando Sánchez, and Luis A. Álvarez. Coordinated Roles: Promoting Re-usability of Coordinated Active Objects Using Event Notification Protocols. In Paolo Ciancarini and Alexander L. Wolf, editors, *Third International Conference, COORDINATION'99*, volume 1594 of *Lecture Notes in Computer Science*, Amsterdam, The Nederlands, April 1999. Springer-Verlag.
14. Gaël Nedelec, Marielle Papillon, Christelle Piedsnoirs, and Gwen Salaün. CLAP: a Class Library for Automata in Python. Master's thesis, 1999. package available at: `http://www.sciences.univ-nantes.fr/info/recherche/mgl/FRANCAIS/ThemesetProjets/SHES/CLAP/`.
15. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.
16. Pascal Poizat, Christine Choppy, and Jean-Claude Royer. Concurrency and Data Types: a Specification Method. An Example with LOTOS. In J. Fiadero, editor, *Recent Trends in Algebraic Development Techniques, Selected Papers of the 13th Workshop on Algebraic Development Techniques, WADT'98*, volume 1589 of *Lecture Notes in Computer Science*, pages 276–291. Springer-Verlag, 1999. ISBN 3-540-66246-4.
17. Pascal Poizat, Christine Choppy, and Jean-Claude Royer. From Informal Requirements to COOP: a Concurrent Automata Approach. In J.M. Wing and J. Woodcock and J. Davies, editor, *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems*, volume 1709 of *Lecture Notes in Computer Science*, pages 939–962, Toulouse, France, 1999. Springer-Verlag. ISBN 3-540-66588-9.
18. J. Rathke and M. Hennessy. Local Model Checking for Value-Passing Processes (Extended Abstract). In Martín Abadi and Takayasu Ito, editors, *Third International Symposium on Theoretical Aspects of Computer Software TACS'97*, volume 1281 of *Lecture Notes in Computer Science*, pages 250–266. Springer Verlag, 1997. `http://www.cogs.susx.ac.uk/users/matthewh/rresearch.html#value`.
19. G. Reggio, E. Astesiano, and C. Choppy. CASL-LTL: A CASL Extension for Dynamic Reactive Systems – Summary. Technical Report DISI-TR-99-34, DISI – Università di Genova, Italy, 1999. `ftp://ftp.disi.unige.it/person/ReggioG/ReggioEtAll99a.ps`.
20. Wolfgang Rief. The KIV-approach to Software Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software - Final Report*, volume 1009 of *Lecture Notes in Computer Science*. Springer, 1995.

The Coalgebraic Class Specification Language CCSL

JAN ROTHE HENDRIK TEWS
Dep. Theor. Comp. Sci., TU Dresden,
D-01062 Dresden, Germany
{*janr,tews*}*@tcs.inf.tu-dresden.de*

BART JACOBS
Dep. Comp. Sci., Univ. Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
bart@cs.kun.nl

April 27, 2000

Abstract. This note presents the *Coalgebraic Class Specification Language* CCSL that is developed within the LOOP project[1]. CCSL allows the (coalgebraic) specification of behavioral types or classes from object-oriented languages. A front-end to the theorem provers PVS [ORR+96] and ISABELLE [Pau94] compiles CCSL specifications into the logic of the theorem provers and and allows to mechanically reason about the specifications.

1. Introduction

The use of *coalgebras* as semantics for object orientation (explicitly proposed in [Rei95]) is for us the most promising approach towards specification and verification of classes. Coalgebras are the categorical duals to algebras (see [JR97]). They are functions Self $\longrightarrow T(\text{Self})$, where T is an interface type (technically, an endofunctor on the category of sets and total functions). The state space Self is seen as a *black box* and the coalgebra comprises all operations that are available to manipulate objects (or elements of Self). As usual in object-oriented programming we use the term *method* for all the operations. The methods can change the internal state of an object or carry out observations on the state space. Thus, properties of elements of the state space can only be *observed* by the operations of the coalgebra. This observational approach to object semantics is contrary to the constructional view on abstract data types. Their semantics, as represented by (initial) algebras, describes every element as *built* via constructor operations. This is not the way a client observes objects.

Coalgebras can naturally model systems with infinite behavior and partial operations. To reason about coalgebras we use the notions of invariance (closure under the application of methods) and bisimulation (indistinguishability via methods). In an appropriate categorical setting both notions can be directly derived from the interface functor T.

The present note sketches the latest version of the coalgebraic class specification language CCSL (see also [HHJT98]). This language supports nested coalgebraic and algebraic specifications. To restrict the behavior of coalgebras we use a higher-order logic enriched with powerful (method-wise) modal operators; see [Jac99, Rot00] for details. The CCSL compiler developed jointly in Dresden and Nijmegen translates CCSL specifications into the logic of the theorem provers PVS or ISABELLE.

We start in Section 2 with a discussion of an example class specification. Section 3 describes the CCSL compiler, especially the notions which it supports. Finally, in Section 4, we summarize the applications of CCSL and conclude.

2. Example

A coalgebraic specification in CCSL consists of a number of sections, describing the class interface, the assertions and the creation conditions. The class interface consists of a definition of the types of methods and constructors of the class and inheritance information. The assertions describe the behavior of the methods. Finally, the creation conditions establish restrictions on the outcome of a class constructor.

[1]See URL http://www.cs.kun.nl/∼bart/LOOP/.

As an example, we consider a simple FIFO queue. It contains two methods. The method `put` adds a new element to the end of the queue. The method `top` delivers the first element of the queue together with the successor state, in which this first element is removed.

Figure 1 depicts the specification of the queue in CCSL. The first line declares `Queue` as the name of the class specification. The specification is parametric in the type A of the elements of the queue. This type parameter must be instantiated in actual use.

In the declaration of the two methods, $\times$ denotes the Cartesian product and $+$ the coproduct[2] of two types. We use $\mathbf{1}$ to denote the unit type that contains exactly one element $\mathbf{1} = \{*\}$. In the example we use `bot` instead of $\kappa_2(*)$ and $\texttt{up}(a, x)$ for $\kappa_1(a, x)$.

The method `put` is declared on line 3; its type is clear from the preceding explanations: `put` takes a state and a data element and produces a new state with the element added at the end of the queue[3]. The method `top`, which delivers the first element of the queue, is a partial operation because it fails on empty queues. We can model this operation as a coalgebra into $(\mathsf{A} \times \text{Self}) + \mathbf{1}$. If the result of the method `top` is `bot` then we say that the method fails. Otherwise it (successfully) delivers an element of A and a successor state of the queue.

In an algebraic approach the partiality of `top` would be handled via subsorts—which introduce all kinds of complications, see for instance [GD94]. Coalgebraically, the structured output type of `top` handles this partiality quite naturally. This idea scales to more complicated forms like non-determinism or computations that involve exceptions.

Note that, using currying, the method `put` can be transformed into an coalgebra. Then both methods can be combined into one coalgebra of type Self $\longrightarrow$ (Self^A) $\times$ $((A \times \text{Self}) + \mathbf{1})$ to match the view on coalgebraic specification that we presented in the introduction. But in applications it is better to distinguish different operations and to allow a finite number of coalgebras per class specification.

With coalgebras alone it is not possible to construct new objects (elements of Self) out of nothing. Therefore in CCSL we combine a coalgebraic method signature with a (degenerated) algebraic constructor signature. In our example we have one constructor `new_queue`, which is a constant in Self. It serves as initial state.

The assertions restrict the behavior of the two methods. Technically an assertion is a logical formula with one free variable of type Self. This variable is declared with the keyword `SelfVar` on line 7. The basic propositions of the logic are observational equalities $\underline{\leftrightarrow}$. Observational equality is the relation lifting (see [HJ98]) of bisimilarity to arbitrary types. In the example we only use observational equality for the type $(\mathsf{A} \times \text{Self}) + \mathbf{1}$. Two elements u and v of this type are observational equal if and only if they either both equal `bot` or if $u = \texttt{up}(a_1, x)$ and $v = \texttt{up}(a_2, y)$ such that a_1 equals a_2 and x and y are bisimilar (with respect to the interface of `Queue`).

Compared to usual equality, observational equality has several advantages. First, it mirrors the intuition that the state space of a class is seen as black box. Thus, two elements of Self can only be compared up to the observations they yield. Second, as a technical condition the use of observational equality ensures the existence of final models (for consistent specifications).

The first assertion `queue_empty` describes the behavior of an empty queue. A queue is empty, if the `top` method fails, that is, if `x.top = bot`. The implication in lines 9 and 10 describes that, if x denotes an empty queue, then after putting an $a \in \mathsf{A}$ into this queue, the first element of the queue is a and, removing this element, we get a queue that is observationally equal to the original empty one.

The second assertion describes the behavior of nonempty queues. To ensure that x is nonempty, we could have taken the expression `NOT` x`.top = bot`. Because we need the result of this application later in the formula, we bind y and a in the condition of the implication. The assertion describes that the first element of the nonempty queue does not change when adding a new element to its end. Further, for nonempty queues, the two operations of adding an element to the queue and removing the head of the queue commute (up to observational equality).

The final part of a class specification describes properties of objects that are created using the con-

[2]In the category of sets and total functions the coproduct of two sets A and B is given by their disjoint union together with the injections $\kappa_1 : A \longrightarrow A + B$ and $\kappa_2 : B \longrightarrow A + B$.

[3]Note that this style of specification is functional: objects are not handled by reference, but by value; the `put` operation produces a new object.

```
BEGIN Queue[ A : TYPE ] : CLASSSPEC
  METHOD
    put :  Self × A -> Self;
    top :  Self -> (A × Self)+1;
  CONSTRUCTOR
    new_queue:  Self;
  ASSERTION   SelfVar x:  Self
    queue_empty :
      x.top = bot IMPLIES
        FORALL (a:A). x.put(a).top ↮ up(a,x)
    queue_filled:
      FORALL(a:A,y:Self).  x.top ↮ up(a,y) IMPLIES
        FORALL (a':A). x.put(a').top ↮ up(a,y.put(a'))
  CREATION
    queue_cre:
      new_queue.top = bot
END Queue
```

Figure 1: A queue class specification in CCSL.

structor of the class. In our case, there is one creation condition. It asserts, that a queue created by `new_queue` is initially empty.

3. Verification Support

In the LOOP project we are most interested in practical applications. Therefore we did not develop our own theorem prover for coalgebraic class specifications. Instead, we decided to implement a front-end tool (the CCSL compiler) for existing theorem provers (currently ISABELLE and PVS). This section contains an overview of the results of our efforts.

The CCSL compiler is written in OCAML [L$^+$99]. An execution for some class specification performs four major steps: first, the specification is parsed. In this step, an internal representation of the specification is generated. Next, the tool checks type correctness for the logical formulae that appear in the assertions and creation conditions. The third pass produces an internal representation of the theories that have to be generated. Finally, the pretty-printing pass writes these theories into source files for the desired theorem prover.

Supported Notions

After running the CCSL compiler on a source file, you receive the generated theories as an output. Among others, you find theories that define the interface (signature) of the class specification, the notions of bisimilarity and invariance, and the notion of morphisms between coalgebras for the class signature. These definitions together with associated lemmas serve as tools for the following more complex concepts.

Models

A model of a specification is developed in two steps: First, one has to construct a state space and a coalgebra for the signature of the specification. Second, one has to prove that the assertions hold on this coalgebra. The CCSL compiler supports this by generating suitable predicates for proving whether a coalgebra is a model of the specification.

To check whether a class specification matches ones intuition, one can examine the final model (in the category of all models) of the specification. The final model of a class specification comprises all possible behaviors of all models of the specification. Nicely enough it allows to directly access the elements of the state space since bisimilarity boils down to equality on final models. Thus, by showing that one's intended model is final, it is easy to determine whether the specification is too loose in the sense that it allows unintended behavior.

Inheritance

The CCSL compiler supports (multiple) inheritance of specifications. Besides the sections for methods, constructors, assertions, and creation conditions, a class specification can contain an inheritance section like

```
INHERIT FROM Queue[A]
        RENAMING put AS enqueue
```

This has the effect that, provided A is a valid type, the current specification is extended with the methods and the assertions of the queue specification. During this extension the original method `put` is renamed into `enqueue` and the CCSL compiler ensures that the inherited assertions `queue_empty` and `queue_filled` restrict the behavior of `enqueue` instead of `put`.

Components

While inheritance models an *is-a* relation, it is also possible to model the *contains-a* relation between different systems. Imagine a specification with a method declaration like

```
queues :  [Self, nat] -> Queue[A]
```

Then models of this specification are required to *contain* an infinite array of queues over type A. For components we assume a standard semantics. Currently the user can choose between *loose semantics*, where an arbitrary model of the queue specification is used as semantics for the component, or *final semantics*, where the final model is used.

Modal Operators

Modal operators are well suited for the verification of properties of potentially nonterminating systems. The third author recognized in [Jac99] the connection between greatest invariants and the future time necessity operator. This was worked out for CCSL in [Rot00].

The CCSL compiler generates theories that define (method wise) future time necessity and eventuality operators. Using these operators in our example, it is for instance possible to prove that every queue, that is reached from an empty queue, can be emptied again.

Refinement

Refinement is a relation between two specifications. A concrete specification $\mathcal{C}$ *refines* an abstract specification $\mathcal{A}$ if all models of $\mathcal{C}$ can be turned (in a generic way) into models of $\mathcal{A}$. Refinements can be concatenated in a number of steps, starting from a high level abstract specification and leading to an implementation. The properties of refinement ensure that the models of the most concrete specification still fulfill the assertions of the original abstract specification.

For coalgebraic class specifications we have two notions of refinement. *Model theoretic refinement* is a generalizations of [Jac97] and is based on the notion of models. In contrast, *behavioral refinement* is based on bisimilarity of corresponding initial states. Behavioral refinement is more general than model theoretic refinement. Behavioral refinement can also be used if only a part of the abstract class interface (for instance the public part) is refined. Under mild assumptions on the logic used in the assertions, it can be shown that, in a situation where both notions of refinement can be applied, behavioral refinement implies model theoretic refinement.

Binary Methods

The theory of coalgebras is usually developed for (covariant) endofunctors on an arbitrary category. Consider the example of a binary method

```
equal :  [Self, Self] -> bool
```

To model such an operation coalgebraically one would need the functor $F(X) = 2^X$, where 2 is the semantics of type `bool`. But on the categories we are interested in, this F cannot be turned into a covariant endofunctor.

In [Tew00b] the second author shows how the notions of coalgebra and bisimulation can be generalized to *extended polynomial functors* $\mathbf{Set}^{\mathrm{op}} \times \mathbf{Set} \longrightarrow \mathbf{Set}$. Coalgebras for extended polynomial functors can model binary methods for practical applications. The CCSL compiler accepts signatures that correspond to extended polynomial functors.

4. Conclusion

In this note we presented the coalgebraic specification language CCSL and sketched its possible applications. CCSL relies on coalgebras and naturally uses the notions of invariance and behavioral equivalence. Other people from the LOOP project apply the same coalgebraic ideas to reason about JAVA programs [JvdBH+98]. In this work they also develop a front-end to PVS and ISABELLE to use a theorem prover to reason about JAVA programs.

The specification language CCSL has been successfully used in several non-trivial case studies. Meyer specifies in [Mey99] the MSMIE (multiprocessor shared information exchange) protocol in CCSL. He refines this initial specification in three steps and proves the correctness of a Java implementation of the protocol.

In [Tew00a], the second author uses CCSL to formalize parts of the memory management of the micro-kernel FIASCO. He then examines the C^{++} sources of FIASCO. The case study revealed some hidden assumptions about the internal interface of the memory management of FIASCO. The case also study proves that the specification language CCSL together with the theorem prover PVS can well be applied to *real* software.

Currently CCSL admits coalgebraic class specification (as described in Section 2) and ADT definitions. One ongoing project is to investigate the possibility to include algebraic specifications in CCSL.

5. References

[GD94] J.A. Goguen and R. Diaconescu. An Oxford survey of order sorted algebra. *Mathematical Structures in Computer Science*, 4:363–392, 1994.

[HHJT98] U. Hensel, M. Huisman, B. Jacobs, and H. Tews. Reasoning about classes in object-oriented languages: Logical models and tools. In *Programming Languages and Systems*, number 1381 in Lecture Notes in Computer Science, pages 105–121. Springer, 1998.

[HJ98] C. Hermida and B. Jacobs. Structural induction and coinduction in a fibrational setting. *Inf. & Comp.*, pages 107–152, 1998.

[Jac97] B. Jacobs. Invariants, bisimulations and the correctness of coalgebraic refinements. In M. Johnson, editor, *Algebraic Methodology and Software Technology*, volume 1349 of *Lecture Notes in Computer Science*, pages 276–291. Springer, 1997.

[Jac99] B. Jacobs. The temporal logic of coalgebras via Galois algebras. Technical Report CSI-R9906, Computing Science Institute, University of Nijmegen, 1999.

[JR97] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.

[JvdBH+98] B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about classes in Java (preliminary report). In *Object–Oriented Programming, Systems, Languages and Applications*, pages 329–340. ACM Press, 1998.

[L+99] X. Leroy et al. *The Objective Caml system release 2.02*. Insitut National de Recherche en Informatique et Automatique, March 1999.

[Mey99] Dion Meyer. A case study in object oriented specification and verification: The MSMIE protocol. Master's thesis, Katholieke Universiteit Nijmegen, January 1999.

[ORR+96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification*, number 1102 in Lecture Notes in Computer Science, pages 411–414. Springer, Berlin, 1996.

[Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in Lecture Notes in Computer Science. Springer, Berlin, 1994.

[Rei95] H. Reichel. An approach to object semantics based on terminal coalgebras. *Mathematical Structures in Computer Science*, 5:129–152, 1995.

[Rot00] J. Rothe. Modal logics for coalgebraic class specification. Master's thesis, Department of Theoretical Computer Science, TU Dresden, D-01062 Dresden, Germany, 2000.

[Tew00a] H. Tews. A case study in coalgebraic specification: Memory management in the FIASCO microkernel. Technical Report TPG2/1/2000, SFB 358, April 2000. available at URL http://wwwtcs.inf.tu-dresden.de/~tews/vfiasco/.

[Tew00b] H. Tews. Coalgebras for binary methods. In H. Reichel, editor, *Coalgebraic Methods in Computer Science*, volume 33 of *ENTCS*. Elsevier, Amsterdam, 2000.

Program Generation via Declarative Term Graph Attribution
(Extended Abstract)

FRANK DERICHSWEILER WOLFRAM KAHL

Institut für Softwaretechnologie
Universität der Bundeswehr München, 85577 Neubiberg
e-mail: `{deri|kahl}@informatik.unibw-muenchen.de`

1. Introduction

Attribute grammars have been developed by Knuth for specifying and implementing the (static) semantic aspects of programming languages [10, 12]. Since then, attribute grammars have grown into a recognised field of study with numerous applications; for one of many surveys see [14].

Attribute grammar ideas also have found their way into graph transformation research. Early approaches like those of [4,17] attribute graph-grammar parse trees instead of string-grammar parse trees. Most current approaches consider attributed graphs and their derivation or transformation. One of the most well-known frameworks in this context seems to be PROGRES [15,16], where, however, the declarative nature of attribute grammars is given up in favour of an operational approach. In the same way, also the algebraic approach of [13] is not oriented towards a declarative view, but towards describing transformations of attributed graphs.

The approach documented in [1] attempts to move closer to the original attribute grammar setting by concentrating not only on attributions, but also on the traversals necessary to calculate the attributions, but thus necessarily stresses the operational aspects of the attribute grammar view and also abandons pure declarativity.

In this paper we present a purely declarative approach to term graph attributions in a formalism which is essentially a straight-forward transfer of the attribute-grammar paradigm to the slightly more general setting of term graphs. Furthermore, we present applications of an implementation of this approach to program generation in Smalltalk, Ada, and Haskell.

2. From Syntax Tree Attributions to Term Graph Attributions

Attribute grammars are an extension of context-free grammars which consists of *semantic rules* added to the syntactic rules of the context-free grammar. In our view, a semantic rule consists of

– a *tree pattern* P determining applicability of the rule,
– an *attribute name* N for the attribute to be defined, and
– an *expression* E defining the values of the N-attributes of those nodes where the pattern P matches; this expression
  • is written in an *attribute definition language*, and
  • contains *attribute references* to other attributes, written in an *attribute reference language* which allows access to attributes of nodes accessible via navigation primitives or as images of nodes in the pattern p.

In purely declarative attribute grammars, attribute definition languages are referentially transparent, i.e., cannot express side-effects.

Such an attribute grammar is then used to define *attributions* of *syntax trees*, and since syntax trees can be considered as a special kind of graphs, we consider an attribution as a (partial) function mapping a node of the graph and an attribute name to an attribute value from an attribute value set depending on the attribute name.

Usually only certain attribute values at the root of the tree are relevant, but we may as well consider the attribution of the whole tree as the result of following the semantic rules of an attribute grammar.

This view of attributions carries over to graphs without any problems, and the definition of semantic rules also does not need noteworthy adaptation. What changes, however, is the semantics of an attribute grammar, since the interpretation of the attribute reference language will have to change, according to the following discussion.

If we consider the attribute flow over the syntax tree in a conventional attribute grammar, then attribute values usually flow along single edges, and they may flow along an edge in either direction. So a single node may have *synthesised* attribute values coming in from below (i.e., via its outgoing edges), and it is relevant via which edge each attribute value arrives, and *inherited* attribute values come from above, and here, too, it may be relevant which edge among the node's parent's outgoing edges this is.

When considering trees as term graphs, then the ordering among outgoing edges is replaced by edge labels attached to these edges, and for every node, no two outgoing edges have the same label. Since every tree node has only at most one incoming edge, it is also true that for every tree node, no two incoming edges have the same label. Generalising, we now consider the direction in which an edge is attached to a node together with its label as one "*input channel*" of the node in question.

Where conventional attribute grammars are always confronted with single attribute values coming from any input channel, a corresponding formalism for general term graphs has to cope with arbitrary numbers of attribute values in the input channels coming in from above, i.e., from the direction of the parent nodes.

Furthermore, in contrast with the operational definition of [1], where different values are arrived at in a sequential manner (see also Sect. 4) and can therefore be considered to be organised in lists, in our purely declarative formalism there is no such obvious structure organising the different attribute values — the only fact that must not be hidden is the possibility of multiple occurrences of the same attribute value. Therefore, multisets are exactly the structure that naturally organises attribute values in general labelled graphs.

As a result, the interpretation of those elements of the attribute reference language that refer to inherited attributes changes its type from single elements of the respective attribute value set to multisets, and the attribute definition language will have to provide primitives to implement *declarative* functions from these multisets into single values that can be used for the defined attributes.

Therefore it turns out that given appropriate machinery for representation and manipulation of multisets the move from syntax tree attributions to term graph attributions can be realised without violating the purely declarative attribute grammar principles.

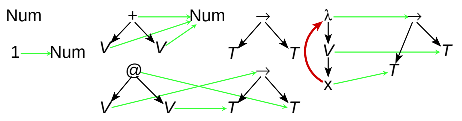
When turning to attribute value sets that are CPOs, then even cyclic attribute dependencies and cyclic term graphs can be dealt with easily by defining the attribution as a fixed-point as usual [3].

3. Declarative Term Graph Attribution in HOPS

The **H**igher **O**bject **P**rogramming **S**ystem HOPS [9, 2, 7, 5, 19] is a graphically interactive term graph programming system designed for transformational program development. In the spirit of Literate Programming [11], HOPS modules are documents containing program fragments, and in HOPS, these are mostly typing elements, transformation rules, and attribution definitions.

HOPS manipulates arbitrary second-order term graphs, where all the structure usually encoded via name and scope is made explicit. Term graphs in HOPS therefore feature nameless variables, explicit variable binding (to denote which node binds which variable), explicit variable identity (to denote which nodes stand for the same variable) and metavariables with arbitrary arity. For the purposes of attribution, variable binding and variable identity can be considered as edges with additional labels.

HOPS types are integrated into the HOPS program term graph structure, and term graph nodes are connected to their types via special *typing edges*. The basis for the typing system are *typing elements*, i.e., simple term graphs that introduce a new node label together with its typing schema making explicit how the typing of a node with this new label is related to the typing of its successors and bound variables. We provide six example typing elements for simply-typed λ-calculus and for arithmetics — the typing function is denoted by thin, light arrows with tiny heads, and the typing elements for $\rightarrow$ and Num, although they do not contain any typing arrows, are necessary for introducing their respective label and for fixing its arity (the thick curved arrow in the typing element for λ-abstraction denotes variable binding, and different nodes labelled with "T" are different type variables):



A term graph G is *well-typed* if for every node there is a homomorphism from the typing element for that node's label into G at that node; the details of this type system have been introduced in [8]. Since typing elements thus define the language of HOPS term graphs, they also serve as attribute rule patterns.

Each attribution definition consists of a target at which it is directed and the definition text proper. These definition texts take on the shape of series of Haskell definitions interspersed with attribute reference expressions written in a syntax similar to that of FunnelWeb, a powerful literate programming system [18].

This means that Haskell here serves as the *attribute definition language* used to produce the real result values, which will usually be strings or functions delivering strings, and these strings may then be interpreted in the target language. However, the dependence of this attribution mechanism on Haskell as its attribute definition language is very weak, and adapting this mechanism to a different attribute definition language would be extremely easy.

The FunnelWeb-like *attribute reference language* is a typed functional language with only the *String* type allowed for embedding into the Haskell code, but also featuring *Node* and function types.

A typical fragment might be the following (the attribute *type* is defined in term of the *type* attributes of the successors, and the *label* of the node is used inside a string constant):

```
@<type@>@(@1@) = Constr "@<label@>@(@1@)" (map tp succtypes)
 where succtypes = @<successors@>@(@1@,@<type@>@)
```

In the document output, however, and for easier reading, HOPS renders the attribute reference language without "@"-characters and in a different font:

```
type(1) = Constr "label(1)" (map tp succtypes)
 where succtypes = successors(1,type)
```

To give a flavour of how this attribution mechanism is used, we show here the definitions of two attributes *expr* and *pexpr* in a simplified Haskell conversion that does not respect sharing in any way and takes care of parenthesisation in only a rather crude way — the value of the attribute *expr* at a node *n* is a string containing a Haskell expression corresponding to the subgraph induced by the node *n*, and this Haskell expression is not parenthesised on the outside if easily avoidable; *pexpr* has parentheses added at least if this makes a difference. The definition for the conversion of function application shows how to use the natural numbering of the nodes of the typing element for referring to the attributes of different nodes:

HaskellAttrib for $\boxed{\textbf{Standard.@}}$

```
    expr(1) = expr(2) ++ ' ' : pexpr(3)
    pexpr(1) = '(' : expr(1) ++ ")"
```

In the definition for λ-abstraction, we have to take care whether there is a bound variable or not; we choose to use different conversions for this purpose. This is implemented via the built-in macro *bvar* which takes five arguments: The first argument refers to a node; if this node has a bound variable, then the call evaluates to the fourth argument in an environment where the second argument, considered as a macro name, is bound to the result of applying the third argument to the bound variable. Otherwise it evaluates to the fifth argument. E.g., if `@1` refers to a binder having a bound variable node with the built-in *number* attribute being `1005,` then the macro call "*bvar*(*1*,`bv`,*number*,`"x`*bv*`"`,`[]`)" evaluates to "`"x1005"`". If `@1` however refers to a binder that does not bind any variable (as e.g. in λ *x*. 3), then that macro call evaluates to "`[]`".

Here this is used to implement the distinction between a λ-abstraction in Haskell and an application of `const` — since λ-abstractions already need parentheses in our context, we make the distinction on the top-level:

HaskellAttrib for $\boxed{\lambda}$

```
    bvar(1,bv,expr,
    expr(1) = "(\\ " ++ bv ++ " -> " ++ expr(2)
    ++ ")"
    pexpr(1) = expr(1)
    ,
    expr(1) = const pexpr(2)
    pexpr(1) = '(' : expr(1) ++ ")"
    )
```

Since HOPS is a language-independent term graph programming framework, it is easy to define term graph languages for different purposes and oriented at different paradigms. We now present two applications where programs in other languages are generated from more or less specialised HOPS languages.

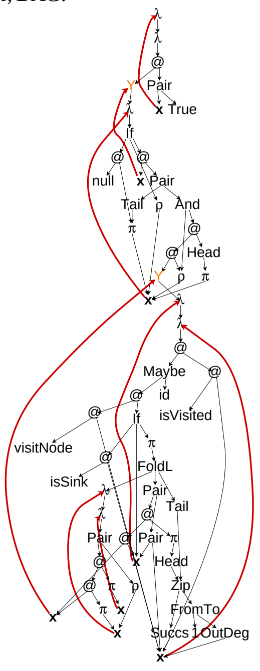
4. Specialised Graph Traversals in Smalltalk

Many problems on graphs can be solved via algorithms that are instances of a depth-first traversal coupled with inherit-synthesise attribute calculations [1]. The complete algorithm schema is represented by a HOPS graph with about 250 nodes, including 13 parameter functions.

For dealing with a given problem, we first instantiate the parameters functions and then use partial evaluation to derive a simplified instance of this algorithm schema. Partial evaluation as a variant of program transformation is realised using the rule application mechanism [6] and transformation strategies [2] of HOPS.

Examples for instances of the inherit-synthesise schema are the computation of strongly connected components of a graph, or unification. Here we present an instance that checks a directed graph for acyclicity by marking every node while traversing it the first time, and clearing the mark after having traversed all nodes reachable from the node in question. Reaching a marked node during the traversal means that a cycle has been found. Using non-strict boolean operators ensures that in the case of a cyclic graph the result is propagated as fast as possible and that no unnecessary traversal is performed.

The optimised algorithm is given by the following, much smaller, DAG:



Now the code generation mechanism is used in oder to produce the following methods in the programming language Smalltalk, implementing an efficient cycle-check based on simple graph navigation primitives (manually uglified for reasons of space):

```
cycle_check: n_359 with: dummy_n_363
 ^self lf_n_294: (Array with: n_359 with: true)

lf_n_294: n_290
 n_290 first isEmpty
  ifTrue: [^n_290 snd]
  ifFalse: [^self lf_n_294: (Array
    with: (n_290 first cpRemIdx: 1)
    with: (n_290 snd and:
     [self lf_n_304: n_290 snd with: n_290 fst fst]))]

lf_n_304: n_303 with: n_299
 ^(self visValOrNil: n_299)
  isNil: [self visVal: n_299 with: (n_299 isSink
   ifTrue: [n_303]
   ifFalse: [(((n_299 succs zip: (1 to: n_299 succsSize))
     cpRemIdx: 1) foldLeft: [:zero_n_335 :value_n_335 |
     self lf_n_314: zero_n_335 with: value_n_335]
    zero: (Array with: (self lf_n_304: n_303 with:
     (n_299 succs zip: (1 to: n_299 outDeg)) fst fst)
    with: (Array with: n_303 with: n_299))) fst])]
  orApply: [:value_n_341 | value_n_341]

lf_n_314: n_305 with: n_308
  ^Array with: (self lf_n_304: n_305 fst with: n_308 fst)
         with: n_305 snd
```

5. Ada Code Transformation

Another example, which focuses on the HOPS transformation mechanism was used to transform Ada code for a primeness predicate on natural numbers. The starting point is a given Ada algorithm with unnecessary parameters within local functions. First we translate the given program into HOPS rules; then, by using fold/unfold techniques and the transformation facilities, a new and optimised version of the function is generated. Finally the attribution mechanism is used to produce Ada again.

On the whole, the initial Ada code

```
function Isprim (n : Nat) return Boolean is
 function Isdiv (k, n : Nat) return Boolean is
  function Divides (k, n : Nat) return Boolean is
  begin if n < k then return (n = 0);
        else return Divides (k, n - k); end if;
  end Divides;
 begin
  if k <= 1 then return False;
  else return (Divides (k, n) or Isdiv (k - 1, n ));
  end if;
 end Isdiv;
begin return not (Isdiv (n / 2, n )) and (2 <= n);
end Isprim;
```

is thus transformed into the following version:

```
function Isprim(n2 : Nat) return Boolean is
  function f4(n9 : Nat) return Boolean is
    function f6(n3 : Nat) return Boolean is
      begin if n9 > n3 then return n3 = 0;
            else return f6(n3 - n9); end if;
      end f6;
  begin if 2 > n9 then return False;
        else return (f6(n2)) or (f4(n9 - 1));
        end if;
  end f4;
begin return (not (f4(n2 div 2))) and (n2 > 1);
end isPrim;
```

6. Let Sharing Make a Difference

The translation to Haskell as sketched in Sect. 3 could equally well be handled via expanding the term graph to a syntax tree first, and then applying conventional attribution techniques — largely this also applies to the Smalltalk and Ada examples of the last two sections.

Term graphs are, however, not in all contexts equivalent to terms, and therefore we now present an application that crucially depends on the possibility to recognise and classify different sharing situations. Although superficially an ad-hoc solution using an operational approach such as in [1] might seem simpler to use, we claim that, already in the example we present here, the declarative approach proves to be much easier to understand and to argue about.

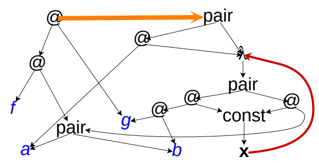
The problem we shall tackle is the generation of Haskell definitions from term graph rules with the constraint that *sharing* be reflected in the formulation of the Haskell rule.

The following cases have to be distinguished:

- If a shared node has no successors, then it will be represented by a single identifier and the sharing may be ignored. All other cases therefore assume that the shared node has successors.
- If the shared node is only reachable from the rule's right hand side, then:
  - If there is no λ-bound variable occurring free below the shared node, then the expression represented by that node should be the right-hand side of a definition for an identifier for that node in a `where`-clause.
  - Otherwise the corresponding definition has to be put inside a `let`-binding inside the innermost λ-abstraction binding one such free variable.
- If a node with successors is reachable from both rule sides, and if the expression represented by the shared node is a pattern, then this should be converted into an as-pattern on the left-hand side and into a reference to the variable bound by the whole pattern on the right-hand side.

  This also covers the cases that the left hand side is a recursively defined pattern occurring somewhere within the right-hand side, and that the right-hand side is a pattern reachable in the left-hand side (as in some projection rules).
- Otherwise, the sharing is ignored.

A few of these effects are illustrated in the following term graph rule:



This rule is translated automatically into the following Haskell definition:

```
  f (v20 @ (a, b)) g = (a v30, v30)
    where v30 = \ x -> let v26 = const x
                       in ((g b v26), v26 v20)
```

We only sketch the solution: We start from an attribution with values for three attributes:

1. Every node has an attribute containing a unique Haskell identifier for the case that that node needs to be represented by a variable (in the above example, three of these are used: `v20`, `v26`, and `v30`).
2. One attribute contains a Boolean value indicating for each node whether it is the source node of the left-hand side,
3. another Boolean attribute indicates the source node of the right-hand side.

The nodes of the left-hand side have attributes indicating whether they might be patterns, so that in the case of sharing an as-pattern can be generated. Every node has two attributes that indicate which rule sides it is reachable from. This together with the reference to the multisets of the corresponding attributes of its predecessors enables each node to determine exactly its sharing status. Bindings for sharings are accumulated in synthesised attributes which are scanned at λ-nodes for those bindings which contain the bound variable; these are then integrated into a local `let`-binding; the remainder is further transmitted via the synthesised attribute and, at the top of the right-hand side, turned into a `where`-clause. Every node has an attribute that prescribes how it is to be turned into Haskell syntax depending on the strings that represent its successors; this syntax prescription is used for building expressions on the right-hand side and patterns on the left-hand side.

The details are rather involved — all in all, this only seemingly simple conversion is implemented via ten attributes only for managing all aspects of determining rule sides and sharing status, eight for expression generation and expressions on both rule sides, and three more for the administration of local bindings — but for most of these attributes, their dependencies on other attributes are very simple; quite a few are only used for being able to share the results of intermediate calculations between different attributes. At first sight, an operational approach might seem to offer simpler solutions. However, in our opinion the declarative approach helps to make the analysis of the problem and the structure of the solution much more explicit.

The whole definition is included in the HOPS user manual [7]. The base language for defining attribute definition functions in HOPS is Haskell, with an embedded syntax for attribute references. Where we use multisets in this paper, the mechanism currently returns lists, and so the user is responsible to only use the multiset structure, most typically via `fold`s with commutative and associative operators. The implementation converts the resulting attribute value dependencies for an attributed graph into a set of Haskell definitions; since Haskell considers top-level and `let`-bindings as mutually recursive, this implements the lazy attribution semantics discussed in Sect. 2.

7. Concluding Remarks

We presented a straightforward extension of the attribute grammar approach to cover term graph attributions in close analogy to the original syntax tree attributions, but unlike the mostly operational approaches to be found in the literature, our approach is purely declarative and includes a natural treatment of sharing and the resulting multiplicity of inherited attributes.

Since term graphs are a popular data structure in all kinds of symbolic computation systems, including interpreters, compilers, theorem provers, and proof assistants, the declarative way of defining term graph attributions as presented in this paper is an attractive means of defining output from term graphs.

References

[1] Rudolf Berghammer. Wiederverwendbare Algorithmenschemata in ML am Beispiel von Graphdurchlauf-Problemen. *Informatik, Forschung und Entwicklung* **11** (4), 179–190 (November 1996).

[2] Frank Derichsweiler. Strategy Driven Program Transformation within the **H**igher **O**bject **P**rogramming **S**ystem HOPS. In A. Poetzsch-Heffter, J. Meyer (eds.), *Programmiersprachen und Grundlagen der Programmierung*, pages 165–172. Informatik Berichte 263 – 1/2000. Fernuniversität Hagen, 2000.

[3] R. Farrow. Automatic Generation of Fixed-Point-Finding Evaluators for Circular, but Well-Defined, Attribute Grammars. *SIGPLAN Notices* **21** (7), 85–98 (1986).

[4] H. Göttler. Attribute Graph Grammars for Graphics. In Hartmut Ehrig, Manfred Nagl, Grzegorz Rozenberg (eds.), *Graph-Grammars and Their Application to Computer Science, 2nd International Workshop*, pages 130-142. LNCS. Springer-Verlag, 1983.

[5] Wolfram Kahl. Can Functional Programming Be Liberated from the Applicative Style?. In Bjørn Pehrson, Imre Simon (eds.), *Technology and Foundations, Proceedings of the IFIP 13th World Computer Congress, Hamburg, Germany, 28 August – 2 September 1994, Volume I*, pages 330–335. IFIP Transactions. North-Holland. IFIP, 1994.

[6] Wolfram Kahl. *Algebraische Termgraphersetzung mit gebundenen Variablen*. Reihe Informatik. Herbert Utz Verlag, München, 1996. ISBN 3-931327-60-4, zugleich Dissertation an der Fakultät für Informatik, Universität der Bundeswehr München.

[7] Wolfram Kahl. The **H**igher **O**bject **P**rogramming **S**ystem — User Manual for HOPS, Fakultät für Informatik, Universität der Bundeswehr München, 1998. URL *http://ist.unibw-muenchen.de/kahl/HOPS/*.

[8] Wolfram Kahl. Internally Typed Second-Order Term Graphs. In Juraj Hromkovic, Ondrej Sýkora (eds.), *Graph Theoretic Concepts in Computer Science, 24th International Workshop, WG '98, Smolenice Castle, Slovak Republic, June 1998, Proceedings*, pages 149–163. LNCS 1517. Springer-Verlag, 1998.

[9] Wolfram Kahl. The Term Graph Programming System HOPS. In Rudolf Berghammer, Yassine Lakhnech (eds.), *Tool Support for System Specification, Development and Verification*, pages 136–149. Advances in Computing Science. Springer-Verlag, Vienna, March 1999. ISBN: 3-211-83282-3

[10] Donald E. Knuth. Semantics of Context-Free Languages. In *Mathematical Systems Theory, Vol. 2*, pages 127–145. Springer-Verlag, New York, 1968.

[11] Donald E. Knuth. Literate Programming. *The Computer Journal* **27** (2), 97–111 (1984).

[12] Donald E. Knuth. The Genesis of Attribute Grammars. In Pierre Deransart, Martin Jourdan, *Attribute Grammars and their Applications (WAGA)*, pages 1–12. LNCS 461. Springer-Verlag, New York–Heidelberg–Berlin, 1990.

[13] Michael Löwe, Martin Korff, Annika Wagner. An Algebraic Framework for the Transformation of Attributed Graphs. In M.R. Sleep, M.J. Plasmeijer, M.C.J.D. van Eekelen (eds.), *Term Graph Rewriting: Theory and Practice*, pages 185–199. John Wiley, 1993.

[14] Jukka Paakki. Attribute Grammar Paradigms — A High-Level Methodology in Language Implementation. *ACM Computing Surveys* **27** (2), 196–255 (1995).

[15] Andy Schürr. Introduction to PROGRES, an Attribute Graph Grammar Based Specification Language. In M. Nagl (ed.), *Graph-Theoretic Concepts in Computer Science*, pages 151–165. LNCS 411. Springer-Verlag, 1990.

[16] Andreas Schürr. Programmed Graph Replacement Systems. In Grzegorz Rozenberg (ed.), *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1*, pages 479–546. World Scientific, Singapore, New Jersey, London, Hong Kong, 1997.

[17] A. Schütte. *Spezifikation und Generierung von Übersetzern für Graphsprachen durch attributierte Graphgrammatiken*. Dissertation, EWH Koblenz. EXpress Edition, 1987.

[18] Ross N. Williams. FunnelWeb User's Manual, May 1992. URL *http://www.ross.net/funnelweb/introduction.html*. Part of the FunnelWeb distribution

[19] Hans Zierer, Gunther Schmidt, Rudolf Berghammer. An Interactive Graphical Manipulation System for Higher Objects Based on Relational Algebra. In Gottfried Tinhofer, Gunther Schmidt (eds.), *Proc. 12th International Workshop on Graph-Theoretic Concepts in Computer Science, WG '86, Bernried, Starnberger See*, pages 68–81. LNCS 246. Springer-Verlag, 1986.

`Fred`: An approach to generating real, correct, reusable programs from proofs.

John Crossley[*] and Iman Poernomo[**]

School of Computer Science and Software Engineering
Monash University, Australia

Abstract. In this paper we describe our system `Fred` for automatically extracting "correct" programs from proofs using a development of the Curry-Howard process. (The word "correct" in this paper means "meeting its specification".) Our system 1. uses Henkin's technique [6] to reduce higher-order logic to many-sorted first-order logic, and 2. extensively uses previously programmed functions and abbreviations so as to imitate normal mathematical practice. As an example of our system we consider a constructive proof of the well-known theorem that every graph of even parity can be decomposed into a list of disjoint cycles. Given such a graph as input, the extracted program produces a list of such cycles as promised.

The well-known Curry-Howard isomorphism (see e.g. [7] or [4]) has been used by a number of people including Constable [2], Hayashi [5]and Coquand and Huet [8], to produce a program in the form of a term of a lambda calculus from a (constructive) proof of a formula. Thus, in arithmetic a constructive proof of a formula of the form $\forall x \exists y \alpha(x, y)$ (where $\alpha(x, y)$ is quantifier free) yields an algorithm for computing a function f such that $\alpha(\overline{n}, f(\overline{n}))$ holds for every natural number n. ($\overline{n}$ is the numeral for n.) Our ultimate goal is to produce readable, resuable and correct programs that people will actually use. We have therefore developed a system `Fred` that is user friendly and generates such programs.

We employ an extension of the Curry-Howard isomorphism to a first-order, many-sorted, predicate calculus that also allows the use of previously programmed functions (and predicates). This has previously been done successfully in various higher order systems. Our approach avoids the use of higher order logic. We also try to mirror, as far as possible, normal mathematical practice. Therefore we are happy to use programs that the user already possesses, provided that the user will guarantee that they are correct. (Alternatively, we could say that the programs we produce are at least as reliable as the ones provided by the user.) In order to do this we add a computational type theory to our logical type theory so that we can admit the use of pre-programmed functions and predicates. We have defined a protocol in [10] between the computational type theory and the logical type theory of the Curry-Howard isomorphism. This allows us to 1. easily

[*] `jnc@csse.monash.edu.au`
[**] `ihp@csse.monash.edu.au`

axiomatize and use pre-programmed functions in our proofs, and 2. investigate and describe constructive proof "idioms" (analogous to programming "idioms" or "patterns").

Our software system, written in C++ and currently called `Fred`, (see `http://www.csse.monash.edu.au/fred`) for "FREge-style Dynamic [system]", is our implementation. At present `Fred` produces programs in *ML*. It has a LaTeX output feature, so that we can easily include proofs written in `Fred` in a document such as the present paper.

In addition to the purely logical framework our system will allow the use of rules in a natural deduction style enabling modifications of the underlying specifications. These were first introduced by Wirsing (see e.g. [11]) and our present version of them may be found in [3].

We use a standard logical type theory (*LTT*) of many-sorted intuitionistic logic. The types are many-sorted intuitionistic formulae and the terms ("Curry-Howard" terms) are essentially terms in a lambda calculus with dependent sum and product types that represent proofs.

The technique of reducing everything to first-order logic, albeit with many sorts, is, we believe, first described in Henkin [6]. For each sort s we have a set of axioms that are Harrop formulae. (The axioms one would normally employ in constructive mathematics are Harrop formulae.) The restriction is a natural one and also has a significant effect on reducing the size of our extracted programs.[1] axioms. We associate with each many-sorted formula a Curry-Howard term (essentially a term of a lambda calculus) representing the derivation of the formula. In order to normalize these Curry-Howard terms we have *reduction rules* whose application corresponds to proof normalization. (See [4] and [1] for the full list of rules.)

Our computational type theory *CTT* is the programming language *ML*, although it might just as easily be LISP or C++. Any language $\mathcal{L}$ for which there is a mapping from terms of Church's simple typed lambda calculus with parameterized types into $\mathcal{L}$ will work.

New predicates and functions. The *LTT* respects the operational meaning of its function terms. That is, each function term of the LTT corresponds to a program in the *CTT*. So there must always be agreement between the *LTT* axioms for function terms and the *CTT* definitions of the corresponding programs. Function terms can be defined in whatever way we wish, as long as they satisfy the axioms of the *LTT*. The user is required to guarantee that these programs are "correct". Thus we retain a distinction between *extensional meaning* (given by the axioms they must satisfy) and *intensional meaning* (how they are coded in the computational type theory). For instance the function $length_\alpha : List\ (\alpha) \rightarrow N$ may be

[1] Harrop formulae are defined as follows: 1. An atomic formula or $\perp$ is a Harrop formula. 2. If α and β are Harrop, then so is $\alpha \wedge \beta$. 3. If α is a Harrop formula and γ is any formula, then $\gamma \rightarrow \alpha$ is a Harrop formula. 4. If α is a Harrop formula, then $\forall x \alpha$ is a Harrop formula.

introduced in the LTT by the axioms

$$length_{\alpha}(\epsilon_{\alpha}) = 0$$
$$length_{\alpha}(\langle a\rangle :: l) = \overline{1} + length_{\alpha}(l)$$

These axioms define a total function $length_{\alpha}$ in the LTT. A corresponding program in the CTT may be specified as follows:

```
let rec length_{\alpha} = function
                [ ] -> 0
                | a::l -> 1+length_{\alpha}(l)
;;
```

In ordinary mathematics, we often abbreviate a formula by a predicate. This is a useful way of encapsulating information, aids readability and helps us to identify and to use common "proof patterns". In Fred, it is permitted to introduce a predicate abbreviation for a formula.

We have defined an *extraction mapping* from Curry-Howard terms in the LTT to terms of our CTT ML in a way similar to that in [1]. The full details of this map can be found in [3].

Theorem 1. *Given a proof $p^{\forall x:s_1 \exists y:s_2 \alpha(x,y)}$ in the logical type theory, there is a program f in the computational type theory ML such that $\alpha(x : s_1, f(x) : s_2)$ is a theorem and the extracted program, $f =$ **extract**(p), has ML type $\mathtt{s}_1 - > \mathtt{s}_2 * \mathtt{s}_3$ where s_3 is the type of the computational content of $\alpha(x, y)$.*

Just as every $f \in F_s$ has a corresponding program in the CTT, every program f in the CTT has a corresponding unique constant, f, in the LTT. Thus, as we had a rule for predicates, so we now have a structural rule (*Skolemization*) to allow us to introduce new functions. From the perspective of the associated Curry-Howard terms, this means that if we have a *proof* t of $\forall x \exists y \alpha(x, y)$, then (the universal closure of) $\alpha(x, f_{\alpha}(x))$ can be treated as an *axiom.*

New induction rules. Adding a sort s with constructors often gives rise to a structural induction rule in the usual manner. This may introduce a new Curry-Howard term recursion operation with the usual fixed point semantics, and an obvious set of reduction rules. In our graph-theoretic example we reduce cases by a function g where g is a function giving a "simpler" list, or else a base case list.

Cycles in even parity graphs. We consider a standard axiomatization of the theory of graphs, $\mathcal{G}$, in terms of vertices (represented by positive integers) and edges. The properties we need are expressible in our formal system for $\mathcal{G}$ with the aid of certain extra function symbols for handling, e.g., lists and lists of lists. These functions have associated (Harrop) axioms in the LTT and computational definitions in the CTT (*cf.* the example of $length_{\alpha}$ above).

Once all the above functions are defined in Fred, we can introduce a predicate $graph(l)$ (defined in Fred by the conjunction of four Harrop formulae, see [9])

```
let rec rho_Chaininduction_LList (ll,A,BASE) =
begin match ll with
[[]] -> BASE
| _ -> A ll (rho_Chaininduction_LList((g ll),A,BASE))
end;;

let main =
(let recFunX33 l =
rho_Chaininduction_LList(l,
(let funX30 x =
(let funX31 X10 = (case3 (Cgrmain (g x))
((let funX32 d = ((lappend d (F x))) in funX32) X10) ([(F x)]) )
in funX31) in funX30),[[]]) in recFunX33)
;;
```

Fig. 1. *ML* program extracted from proof of the theorem:$\forall L(\mathit{evenpar}(L)\ \&\ \mathit{start}(L) \neq 0 \rightarrow \exists M(\mathit{listcycyle}(M, L)))$

to mean that a list l of sort $List(List(N))$ represents a graph. A graph has *even parity* if the number of vertices adjacent to each vertex is even.

In [9] we gave a proof and showed how to extract a program for finding a cycle in an even parity graph from it. In our paper [10] we extended this proof to show that every even parity graph can be decomposed into a *list of disjoint cycles.* We use **extract** (see Theorem 1 above) on the latter proof to obtain a program that gives the decomposition into a list of disjoint cycles. The theorem that we prove is

$$H \vdash \forall L(\mathit{evenpar}(L)\ \&\ \mathit{start}(L) \neq 0 \rightarrow \exists M(\mathit{listcycyle}(M, L))) \qquad (1)$$

where L, M are lists (of lists of natural numbers) and $\mathit{listcycle}(M, L)$ holds if M is a maximal list of disjoint cycles in the graph represented by the list L. The assumption formula H describes the predicate *listcycle*. (See [9] for the function *start*.)

The idea behind the proof is that if we start with an even parity graph L and apply our previous algorithm to obtain a cycle $F(L)$ in L. Then, by deleting the edges of this cycle from L, we are left with another *even parity* graph, $g(L) = L - F(L)$. We repeat the process until we are left with a graph that contains no edges. The resulting list of disjoint cycles of L is given by $\langle F(L), F(g(L)), F(g(g(L))), \ldots\rangle$. The base case for a list L is the trivial list obtained from L by deleting all the cycles.

The `Fred` *Environment.* `Fred` provides an advanced GUI for modular proof development.

The underlying "look and feel" of the environment is based on popular Integrated Development Environments for programming languages (e.g., the Microsoft Visual Suite or Borland C++ Builder). The environment provides the following features: 1. `Fred` uses a Multiple Document Interface, allowing for many proofs to be edited at the same time in separate child windows. This visually reflects the manner in which mathematicians solve a theorem – by breaking it into separate related lemmas. 2. Proofs are presented in tree form, using a "navigation directory tree" visual component that is familiar to users of Windows or MacOS operating systems. 3. Formulae are validated by a "drag and drop" action.

In `Fred`, just as in mathematics practised by mathematicians, we can examine a proof at various levels of "granularity". We layer the proof and the full details may be found in [9].

The *ML* program extracted for the is displayed in Fig.1 where `Cgrmain` is a program corresponding to other lemmas used in the proof of the theorem and `g` and `f` are pre-programmed functions – see Appendix C of [9] for a full listing of their programs.

References

1. Albrecht, D. and J.N. Crossley. Program extraction, simplified proof-terms and realizability. Technical Report 96/275, Monash University Department of Computer Science, 1996.
2. Constable, R.L., S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R. Harper, D.J. Howe, T.B. Knoblock, N.P. Panangaden, J.T. Sasaki, and S.F. Smith *Implementing Mathematics with the Nuprl Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
3. Crossley, J.N., I. Poernomo and M. Wirsing, Extraction of Structured Programs from Specification Proofs, To appear in WADT99.
4. Crossley, J.N. and J.C. Shepherdson. Extracting programs from proofs by an extension of the Curry-Howard process. pp. 222-288 in J. N. Crossley, J. B. Remmel, R. Shore, and M. Sweedler, editors, *Logical Methods: Essays in honor of A. Nerode.* Birkhäuser, Boston, Mass., 1993.
5. Hayashi, S. and H. Nakano. *PX, a computational logic.* MIT Press, Cambridge, Mass., 1988.
6. Henkin, L. Completeness in the Theory of Types. *Journal of Symbolic Logic*, **15** (1950) 81–91.
7. Howard, W. A. The formulae-as-types notion of construction. pp. 479–490 in J.R.Seldin and R.J.Hindley (eds), *To H.B. Curry : essays on combinatory logic, lambda calculus, and formalism.* Academic Press, London,New York, 1980.
8. Huet, G., G. Kahn, and C. Paulin-Mohring. The Coq Proof assistant Reference Manual: Version 6.1. Coq project research report RT-0203, Inria, 1997.
9. Jeavons, J.S., I. Poernomo, B. Basit and J.N. Crossley. A layered approach to extracting programs from proofs with an application in Graph Theory Paper presented at the Seventh Asian Logic Conference, Hsi-Tou, Taiwan, June 1999.
10. Jeavons, J.S., I. Poernomo, J.N. Crossley and B. Basit. `Fred`: An implementation of a layered approach to extracting programs from proofs. Part I: an application in Graph Theory. *Proceedings of the AWCL Workshop, Canberra, 2000.*
11. Wirsing, M. Structured specifications: syntax, semantics and proof calculus. Pp. 411–442 in F.L.Bauer *et al.* (eds), *Logic and Algebra of Specification.* Springer, 1994.

Java2CSP: A System for Verifying Concurrent Java Programs

Hui Shi

Bremen Institute for Safe Systems, University of Bremen
Postfach 33 04 40, 28334 Bremen, Germany
shi@informatik.uni-bremen.de

1 Introduction

More and more software systems are used in safety critical areas. In practice, the quality of such software, or its correctness, is still assured by inspecting its source code, or testing. Experience in the areas of railway control systems [5] and space station control systems has shown the formal verification of concurrent software systems is necessary and feasible, especially if one concentrates on those properties relevant for synchronisation. Verified Systems GmbH and the Bremen Institute for Safe System (BISS) have successfully analysed a fault-tolerant data management system [3] for the International Space Station (ISS) (under the contract with DaimlerChrysler Aerospace, DASA). In that project, some properties like deadlock [1] and livelock freedom [2] have been investigated based on CSP specifications and the model-checking tool FDR.

Until now there exist only a few practical tools to support the automatic analysis of existing software systems, let alone tools for translating them into a special formal method like CSP so that the associated model-checking tools can be applied. An important reason for choosing them, except for some previous experiences with CSP and FDR, is to take advantage of the HOL-CSP system [10], a semantic encoding of CSP in Higher-Order Logic within the theorem prover Isabelle [7], for performing those verification tasks that exceed the power of a model-checker. In the industrial project mentioned above, existing programs were manually transferred to CSP specifications. In this procedure, errors can be introduced, regardless how carefully it is done. In addition, this process is very inefficient and requires deep knowledge of the formal method used.

This paper presents the system Java2CSP which translates concurrent Java programs into CSP processes. Our goal is to verify automatically the synchronization behaviour, such as deadlock and livelock, of the original Java programs with the model-checking tool FDR.

2 Implementation Approach

In contrast to programming languages CSP was designed as a notation and theory for describing and analysing systems whose primary interest arises from the ways in which different components interact at the level of communication. CSP differs from concurrent programming languages in three important aspects:

– CSP was designed to be a notation for purely communicating systems: the computations internal to the component state (variables, assignments, etc.) are ignored.
– CSP supports no structured types, like strings, arrays, etc.
– The CSP notation for describing concurrency is quite simple: a set for communication events and a set of parallel operators like || for synchronised or ||| for asynchronised composition.

To interpreter concurrent Java programs in CSP the system Java2CSP makes an *abstract representation* of a Java program by removing all parts that do not influence its synchronisation behaviour, in order to reduce it to a manageable size. Then, CSP *process patterns* are introduced for simulating the concurrency concepts of Java, like shared variables, threads, and monitors.

For the verification of synchronisation behaviours of concurrent programs it is not necessary to inspect every detail of the original code, since only a subset of the programmed statements have impact on properties like deadlock and livelock freedom. Theoretically, every program with finite states can be verified by a model-checker like FDR, but it is still necessary to consider the problem of state space explosion. In fact, the main obstacle of using model checking for industrial projects is to solve this problem. It is therefore necessary to generate a CSP specification which represents an abstract version of the original Java program showing only the amount of detail which is relevant for the verification of its synchronisation behaviour. The important abstractions we applied are: *abstract object-model*, *abstract interpretation of types* and *abstract representation of algorithms*.

CSP processes which simulate those mechanisms that are supported by Java but not by CSP such as shared variables, threads and monitors, are called *process patterns*. The following CSP-process, for example, simulates a shared integer variable, where channels *readNum* and *writeNum* are for the reading and writing operation, respectively:

```
VarNum(var,val) = writeNum.var?x -> VarNum(var,x)
                  []
                  readNum.var!val -> VarNum(var,val)
```

The parameters are for the variable's name and its current value. The reading operation does not change its value, while after a writing operation the variable gets a new value.

3 The System

Java2CSP is implemented in Java. Figure 1 gives an overview of all system components. The system is divided into three phases: the bytecode analysis, the syntax-tree analysis and the code generation.

Bytecode analysis
The Java bytecode, instead of the Java source text, of a Java program has be used as the source code for the translation, since

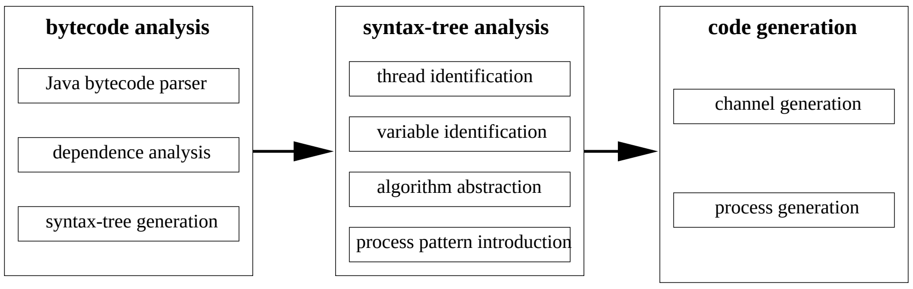


Fig. 1. The system architecture

- The Java Class File Format is a standard determined by Sun Microsystem.
- The Java Class File Format can not only be used for user applications written in Java, but for applications which can be compiled into this format.
- A Java program can include some auxiliary sources from libraries or from other users, which are sometimes only available in their bytecode representations.

The *bytecode parser* consists of two elementary parts. One reads a Java class file and writes the class information from a code segment into a file and one initiates the parsing process and resolves the dependency relation between classes (*dependence analysis*). The generated constructs maintain a shallow branch hierarchy and use only references as a structure. With such information it is difficult to make lower level analysis later. For this reason the parsed information will be transformed into a syntax-tree at an abstract level by the *syntax-tree generation*.

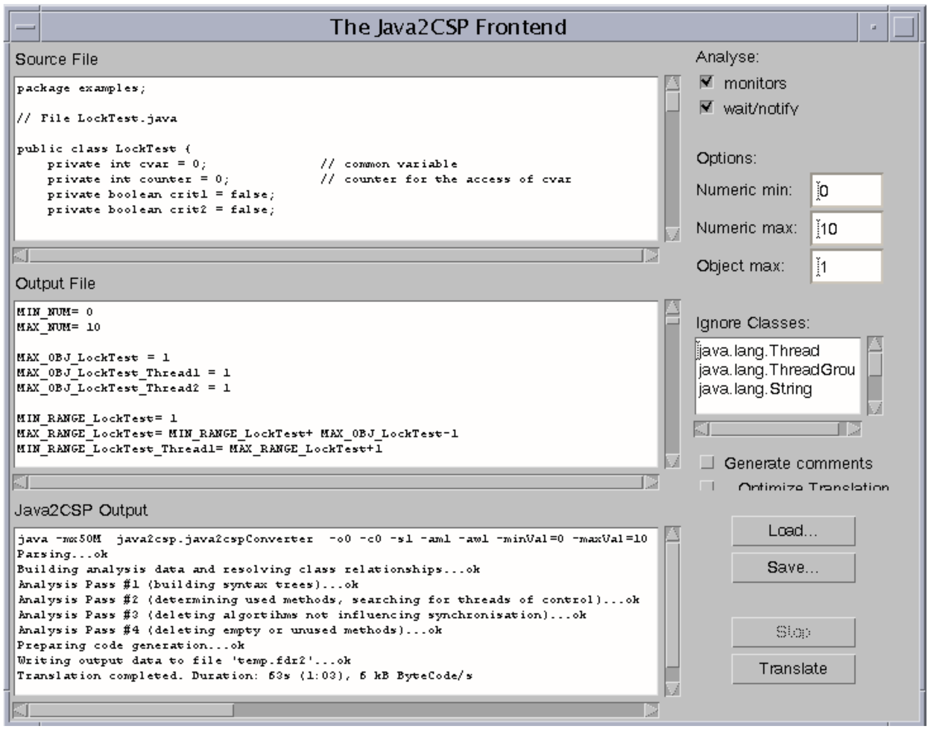
Syntax-tree analysis
For the evaluation of concurrent Java programs it is important to *identify threads* which can run concurrently, and to *identify variables* which influence the behaviour of threads in different ways. A *structure variables*, for example, influences the control flow of some threads; or a *synchronization variable* called by *wait-* and *notify*-method can change the state of some threads.

A program contains not only code to control its concurrent and synchronous behaviour, it may contain parts which take over the functional tasks and have no influence on its control flow. It is not necessary to translate them into CSP specifications for the verification of synchronization properties of the original program. *Abstracting algorithms* of a Java program from such codes is another important step for generating compact CSP specifications and for efficiently model-checking.

Code generation
The code-generation phase converts the above generated syntax-trees with all necessary information into a CSP specification. Such a CSP specification consists usually of a set of channel definitions, a set of auxiliary processes and a main process.

Within a generated specification the main process specifies always the main-method of the original Java program, and all in the Java program created threads will be translated into processes, as well. The whole system is composed of all these single generated processes which synchronise with process patterns specifying shared variables, threads and monitors, etc.

Figure 2 shows a screenshot of Java2CSP's simple graphical user interface. On the left are three windows showing the original java file (*Source File*), the corresponding CSP specification (*Output File*) and the system output (*Java2CSP Output*), respectively. The generated CSP specification can be saved in a FDR file using the *Save* button. Together with the specification two assertions will always be generated for the model-checker FDR to prove the deadlock and livelock behaviours of the original Java program.



Source File

```
package examples;

// File LockTest.java

public class LockTest {
    private int cvar = 0;               // common variable
    private int counter = 0;            // counter for the access of cvar
    private boolean crit1 = false;
    private boolean crit2 = false;
```

Output File

```
MIN_NUM= 0
MAX_NUM= 10

MAX_OBJ_LockTest = 1
MAX_OBJ_LockTest_Thread1 = 1
MAX_OBJ_LockTest_Thread2 = 1

MIN_RANGE_LockTest= 1
MAX_RANGE_LockTest= MIN_RANGE_LockTest+ MAX_OBJ_LockTest-1
MIN_RANGE_LockTest_Thread1= MAX_RANGE_LockTest+1
```

Java2CSP Output

```
java -mx50M  java2csp.java2cspConverter  -o0 -c0 -s1 -am1 -aw1 -minVal=0 -maxVal=10
Parsing...ok
Building analysis data and resolving class relationships...ok
Analysis Pass #1 (building syntax trees)...ok
Analysis Pass #2 (determining used methods, searching for threads of control)...ok
Analysis Pass #3 (deleting algortihms not influencing synchronisation)...ok
Analysis Pass #4 (deleting empty or unused methods)...ok
Preparing code generation...ok
Writing output data to file 'temp.fdr2'...ok
Translation completed. Duration: 63s (1:03), 6 kB ByteCode/s
```

Analyse:
[✓] monitors
[✓] wait/notify

Options:
Numeric min: 0
Numeric max: 10
Object max: 1

Ignore Classes:
java.lang.Thread
java.lang.ThreadGrou
java.lang.String

[ ] Generate comments
[ ] Optimize Translation

Fig. 2. The Graphical User Interface of Java2CSP

4 Conclusion

When we began with this work we thought the most difficult part of the system should be in the area of compiler construction, because the translation of Java bytecode to efficient structures for the analysis is in no means trivial. In reality, it turned out, however, that the representation of Java programs as CSP specifications is more difficult. For this purpose some abstraction methods have been implemented, such as the abstract representation of Java object-model, of Java types and algorithms. A set of CSP process patterns have been developed as well, which specify the general behaviour of a set of Java mechanisms, like shared variables, threads and monitors. Both of them are the most important parts of the system.

At present the system Java2CSP is in an experimental stage. The representation of floats and strings has not been implemented, nor the timed synchronisation. Although it can translate a lot of concurrent Java programs, like any other compiler output format, the output of Java2CSP is again the input for FDR, the improvement of the quality of the output format with respect to the readability and the recoverability is also an indispensable effort to make the system practical useful for large Java programs.

References

1. B. Buth, M. Kouvaras, J. Peleska and H. Shi: Deadlock analysis for a fault-tolerant system. In M. Johnson (ed.), *Algebraic Methodology and Software Technology. Proceedings of the AMAST'97.* LNCS 1394, (1997).
2. B. Buth, J. Peleska and H. Shi: Livelock analysis for a fault-tolerant system. In A. M. Haeberer (Ed.): *Algebraic Methodology and Software Technology. Proceedings of the AMAST'98.* LNCS 1548, (1998).
3. DaimlerChrysler Aerospace: DMS-R FTC Detailed Design Document Volume 3 (FML Software).
4. Formal Systems: *Failures Divergence Refinement FDR2.* FDR2 User Manual. Formal Systems (Europe) Lts (1998).
5. A. Haxthausen, J. Peleska: Formal Development and Verification of a Distributed Railway Control System. In *Proceedings of the 1st FMERail Workshop*, (1998).
6. C. A. R. Hoare: *Communicating Sequential Processes.* Prentice-Hall International (1985).
7. L. C. Paulson. *Isabelle - A Generic Theorem Prover.* LNCS 828, Springer Verlag, (1994).
8. A. W. Roscoe: *The Theory and Practice of Concurrency.* The Prentice-Hall International (1998).
9. H. Shi, J. Peleska and M. Kouvaras: Combining Methods for the Analysis of a Fault-Tolerant System. In B. Werner (ed.) *Proceedings of 1999 Pacific Rim International Symposium on Dependable Computing.* IEEE Computer Society, (1999).
10. H. Tej and B. Wolff: A Corrected Failures-Divergence Model for CSP in Isabelle/HOL. In J. Fitzgerald, C. B. Jones, and P. Lucas (eds.) *Proceedings of the FME '97 — Industrial Applications and Strengthened Foundations of Formal Methods.* LNCS 1313, Springer Verlag, (1997).

Using PAX to Verify Parameterized Networks

K. Baukus[1] Y. Lakhnech[2] K. Stahl[1]

[1] Institute of Computer Science and Applied Mathematics
University of Kiel
Preusserstr. 1–9, D-24105 Kiel, Germany
{kba, kst}@informatik.uni-kiel.de

[2] VERIMAG, Centre Equation
2 Av. de Vignate, 38610 Giéres, France
lakhnech@imag.fr

Recently there has been much interest in the automatic and semi-automatic verification of parameterized networks, i.e., verification of a family of systems $\{\mathcal{P}_i \mid i \in \omega\}$, where each $\mathcal{P}_i$ is a network consisting of i finite-state processes. Apt and Kozen show in [AK86] that the verification of parameterized networks is undecidable. Nevertheless, automated and semi-automated methods for the verification of restricted classes of parameterized networks have been developed.

In [BBLS00] we first transform a given infinite family of networks of finite processes into a bisimilar single transition system whose variables are set variables and whose transitions are described in WS1S, the weak monadic second order logic of one successor. We call such systems WS1S transition systems.

The idea of representing sets of states of parameterized networks by regular languages is applied in [KMM$^+$97], where additionally finite-state transducers are used to compute predecessors. The work presented in [ABJN99] extends the idea by considering the effect of applying infinitely often a transition that satisfies certain restrictions.

Contrary, in our approach we do not try to compute the exact set of reachable states. Instead, our tool PAX computes a finite abstraction of the obtained WS1S transition system. The abstract system gives us an over-approximation of the set of reachable states, but also maintains some properties of the original control flow. These properties can be analyzed using model-checking techniques. Therefore, PAX provides a reachability analysis and translations to input languages of some model checkers.

Since our abstraction is guaranteed to be conservative, i.e., the abstract system exhibits for every behavior of the WS1S system a corresponding abstract behavior, we are able to verify universal path quantified properties of the WS1S system. These properties also include liveness properties.

But a problem in the verification of liveness properties is that often they do not hold on the abstract level since abstraction introduces a lot of new cycles that have no concrete counterparts. Therefore, PAX offers an algorithm to add fairness conditions to the abstract system which are guaranteed to hold for the concrete system, and which rule out these cycles as possible counterexamples.

Moreover, in [BLS00] we show how to deal with fairness requirements already given for the concrete system. Combining these techniques allows us to verify some interesting liveness properties of non-trivial parameterized networks.

We first shortly explain our verification approach and give some details about the functionality of the PAX tool afterwards.

1 Approach

The verification problem we would like to tackle is the following: Given a parameterized network $P_1 \parallel \ldots \parallel P_n$ and a quantifier-free linear-time temporal formula $\psi(p_1, \ldots, p_k)$, we want to prove $P_1 \parallel \ldots \parallel P_n \models \forall p_1, \ldots p_k \leq n : \psi(p_1, \ldots, p_k)$, for every $n \in \omega$.

Our approach is the verification by abstraction and we proceed as follows:

1. Transformation of the given infinite family of networks of finite processes into one bisimilar transition system with set variables; for details see [BBLS00]. To be able to compute abstractions automatically, we restrict ourselves to the class of parameterized networks which can be described in WS1S. In this paper we assume the parameterized network to be given as a WS1S system.

2. Finding an abstraction relation. We use boolean systems as abstract systems. For heuristics choosing a suitable abstraction relation see [BBLS00, BLS00].

3. Construction of the abstract system using PAX. A conservative abstraction is computed, hence, the concrete property is guaranteed to hold for the concrete system when the corresponding abstract property can be established on the abstract level.

4. Enrich the abstract system with fairness conditions which can be safely added, since they are guaranteed to hold for the concrete system.

5. Model-check the abstract system to prove that it satisfies the abstract property.

2 The pax Tool

2.1 Constructing the abstract system

In this section, we show how to automatically construct a finite abstract transition system that can be model-checked from a given abstraction relation and a concrete system.

Let $\mathcal{S} = (\mathcal{V}, \Theta, \mathcal{T})$ be a given WS1S system consisting of a set of variables $\mathcal{V}$, an initial state predicate Θ, and a set of transitions $\mathcal{T}$, each transition τ given as a predicate ρ_τ. Let α be an abstraction relation given as predicate $\widehat{\alpha}(\mathcal{V}, \mathcal{V}_A)$. Notice that since the abstract variables are booleans, the abstract system we construct is finite, and hence, can be subject to model-checking techniques. Moreover, we make use of the fact that both $\widehat{\alpha}(\mathcal{V}, \mathcal{V}_A)$ and the transitions in $\mathcal{T}$ are expressed in WS1S to give an effective construction of the abstract system.

The abstract system we construct contains for each concrete transition τ an abstract transition τ_A, which is characterized by the formula

$$\exists \mathcal{V}, \mathcal{V}' : \widehat{\alpha}(\mathcal{V}, \mathcal{V}_A) \wedge \rho_\tau(\mathcal{V}, \mathcal{V}') \wedge \widehat{\alpha}(\mathcal{V}', \mathcal{V}'_A)$$

with free variables $\mathcal{V}_A$ and $\mathcal{V}'_A$.

The initial states of the abstract system we construct can be described by the formula

$$\exists \mathcal{V} : \widehat{\alpha}(\mathcal{V}, \mathcal{V}_A) \ .$$

To compute the abstract transitions and initial states, one has to find the set of all models of these formulae, which is possible since we only have WS1S formulae. This is done by PAX, and it uses the decision procedures of MONA [KM98, HJJ+96] for this purpose.

The input for the PAX tool consists basically of a predicate describing the initial states, predicates describing the transitions of the system, and the abstraction relation given by a set of predicates, one predicate ψ_a for each abstract boolean variable a. The abstraction relation is $\bigwedge_{a \in V_A} (a \Leftrightarrow \psi_a)$. All these predicates are only allowed to have concrete variables as free variables.

Example 2.1 *Here we give an example transition.* Pci *is the set variable for control location i. A process j is at control location i, iff $j \in$* Pci *and $j \notin$* Pck *for all $k \neq i$. The primed variables refer to the post-state of the transition, while the unprimed versions refer to the pre-state.*

```
ex1 i: Pc3 = empty & Pc5 = empty & Pc6 = empty & Pc7 = empty
       & i in Pc1 & i in Pc2'
       & all1 j: j ~= i =>
             (j in Pc1 <=> j in Pc1') & (j in Pc2 <=> j in Pc2')
           & (j in Pc3 <=> j in Pc3') & (j in Pc4 <=> j in Pc4')
           & (j in Pc5 <=> j in Pc5') & (j in Pc6 <=> j in Pc6')
           & (j in Pc7 <=> j in Pc7')
```

The next example shows part of a result of a PAX computation.

Example 2.2 *Some example transitions computed by* PAX *are given below. Here, the abstraction relation relates an abstract variable* aPci *to a concrete state by the formula* aPc$i \Leftrightarrow$ Pc$i \neq \emptyset$.

```
Variables: aPc1 aPc2 aPc3 aPc4 aPc5 aPc6 aPc7 a_error a_inv
Abstract transitions:
t12 : 100000000 -> 010000000
t12 : 110000000 -> 010000000
t12 : 100000000 -> 110000000
t12 : 110000000 -> 110000000
```

The PAX tool is able to do a forward state exploration to verify invariance properties of the abstract system. Moreover, it is possible to translate the abstract system to Promela (the input language of the model-checker Spin) and to the input language of SMV/NuSMV. Thus, these model-checkers can also be used for verification. Another option is given in Section 2.2.

2.2 Constructing abstractions of abstract systems

If one succeeds in computing the abstract system as shown in Section 2.1, this abstract system can be model-checked, or it can be first augmented with fairness conditions as explained in Section 2.3 and then analyzed.

The decision procedure for WS1S is based on constructing a finite automaton over finite words that recognizes the set of models of the considered formula. In practice, however, it can happen that the automaton cannot be constructed because of its size. Assume we have a concrete system $\mathcal{S}$ and an abstraction relation α. Let $\mathcal{S}_A$ be the abstract system, though it is not computable for us. In this case, we propose to go on as follows to obtain an abstraction $\widetilde{\mathcal{S}_A}$ of $\mathcal{S}_A$.

The states of $\widetilde{\mathcal{S}_A}$ are sets of states of $\mathcal{S}_A$. We assume that each such state q is given by a characteristic predicate $\widehat{q}$.

We start with a state representing the set of initial states of $\mathcal{S}_A$. This set is usually much simpler to compute than the abstract transitions. Given now a state q and concrete transition τ, we construct new states by computing for each abstract variable a_i the set $M_i \subseteq \{\text{true}, \text{false}\}$ of fulfilling values for a_i' of

$$\exists \mathcal{V}_A, \mathcal{V}, \mathcal{V}' : \widehat{q}(\mathcal{V}_A) \wedge \widehat{\alpha}(\mathcal{V}, \mathcal{V}_A) \wedge \rho_\tau(\mathcal{V}, \mathcal{V}') \wedge (a_i' \Leftrightarrow \varphi_i(\mathcal{V}')) \ .$$

In case M_i is empty for some i, then there does not exist any post state, and hence, the computation for the other variables can be omitted. Otherwise, one takes as abstract post-state of $\widetilde{\mathcal{S}_A}$ the set $\{s \mid \forall i : s(a_i) \in M_i\}$. This set contains at least all possible abstract post states w.r.t. τ_A. It is not difficult to see that this method computes an abstraction of $\mathcal{S}_A$, and hence, is also an abstraction of $\mathcal{S}$.

In case some of the transitions can be computed as shown in Section 2.1, these abstract transitions can be used to compute the precise post-states for this transitions instead of the over-approximation.

2.3 Marking Algorithm

It is well known that an obstacle to the verification of liveness properties using abstraction, is that often the abstract system contains cycles that do not correspond to fair computations of the concrete system. A way to overcome this difficulty is to enrich the abstract system with fairness conditions or more generally ranking functions over well-founded sets that eliminate undesirable computations.

In this section we present a marking algorithm implemented in our tool that given an abstraction of a WS1S system enriches the abstract system with strong fairness conditions while preserving the property that to each concrete computation corresponds an abstract *fair* one.

The method uses a marking algorithm that labels each edge of the considered abstract system with one of the symbols $\{+_X, -_X, =_X\}$ for each set variable X of the original WS1S system. Intuitively, the labels $-_X$ resp. $=_X$ express whether the transitions at the concrete level reduce resp. maintain the cardinality of a set X, the label $+_X$ represents all other cases.

Marking Algorithm

Input: WS1S system $\mathcal{S} = (\mathcal{V}, \Theta, \mathcal{T})$, abstract system $\mathcal{S}_A = (\mathcal{V}_A, \Theta_A, \mathcal{T}_A)$. For each $\tau_A \in \mathcal{T}_A$ let τ be the corresponding concrete transition.

Output: Labeling of $\mathcal{T}_A$

Description: For each $X \in \mathcal{V}$ and each edge $\tau_A \in \mathcal{T}_A$, let $\Delta(X, \tau, \prec)$, with $\prec \in \{\subset, =\}$, denote the WS1S formula:

$$\widehat{\alpha}(\mathcal{V}, \mathcal{V}_A) \wedge \widehat{\alpha}(\mathcal{V}', \mathcal{V}'_A) \wedge \rho_\tau(\mathcal{V}, \mathcal{V}') \Rightarrow X' \prec X.$$

Then, mark τ_A with $-_X$, if $\Delta(X, \tau, \subset)$ is valid, mark τ_A with $=_X$, if $\Delta(X, \tau, =)$ is valid; otherwise mark τ_A with $+_X$.

Now, for a set variable X we denote with $\mathcal{T}_X^+$ the set of edges labeled with $+_X$. Then, we add for each such X and each transition τ_A labeled with $-_X$ the fairness condition $(\tau_A, \mathcal{T}_X^+)$ which states that τ_A can only be taken infinitely often when one of the transitions in $\mathcal{T}_X^+$ are taken infinitely often.

The generated fairness conditions can be expressed as LTL formulae and the model-checker can check certain properties under the assumption that these conditions are not violated. This is necessary in most cases to establish liveness properties.

3 Conclusions

We presented a method for the verification of universal properties of parameterized networks. Our method is based on the transformation of an infinite family of systems into a single WS1S transition system and applying abstraction techniques on this system. To be able to prove liveness properties we presented a method to add fairness requirements to the abstract system. We have applied this method, which has been implemented in our tool PAX, to a number of parameterized protocols. To compute the finite abstract system we use MONA to decide the WS1S formulae. The following table shows the time and memory needed for the construction of some examples. We used a Sun Ultra 5/10 UPA/PCI (UltraSPARC-IIi 440MHz) with 1024 MB of memory.

Example	Time	Memory
Szymanski	3 sec	20 MB
Dijkstra	1 min 30 sec	260 MB
Simple D.	1 min 7 sec	214 MB

Table 1: PAX construction of abstract transition systems

Although the PAX tool is in an experimental stage, the first results obtained using our methods and PAX are very encouraging and can be found at
`http://www.informatik.uni-kiel.de/~kba/pax/`

References

[ABJN99] P.A. Abdulla, A. Bouajjani, B. Jonsson, and M. Nilsson. Handling Global Conditions in Parameterized System Verification. In N. Halbwachs and D. Peled, editors, *CAV '99*, volume 1633 of *LNCS*, pages 134–145. Springer, 1999.

[AK86] K. Apt and D. Kozen. Limits for Automatic Verification of Finit-State Concurrent Systems. *Information Processing Letters*, 22(6):307–309, 1986.

[BBLS00] K. Baukus, S. Bensalem, Y. Lakhnech, and K. Stahl. Abstracting WS1S Systems to Verify Parameterized Networks. In *TACAS 2000*, volume 1785 of *LNCS*, pages 188–203. Springer, 2000.

[BLS00] K. Baukus, Y. Lakhnech, and K. Stahl. Verifying Universal Properties of Parameterized Networks. Submitted to *FTRTFT 2000*, 2000.

[HJJ$^+$96] J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic Second-Order Logic in Practice. In *TACAS '95*, volume 1019 of *LNCS*. Springer, 1996.

[KM98] N. Klarlund and A. Møller. MONA Version 1.3 User Manual. BRICS, 1998.

[KMM$^+$97] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic Model Checking with Rich Assertional Languages. In O. Grumberg, editor, *Proceedings of CAV '97*, volume 1256 of *LNCS*, pages 424–435. Springer, 1997.

Verifying Reactive Systems Using Predicate Diagrams[⋆]

Dominique Cansell[1], Dominique Méry[2], and Stephan Merz[3]

[1] Université de Metz & LORIA, cansell@loria.fr
[2] Université Henri Poincaré, Nancy & LORIA, mery@loria.fr
[3] Institut für Informatik, Universität München, merz@informatik.uni-muenchen.de

1 Motivation

Verification techniques for reactive systems are traditionally classified as either *deductive* or *algorithmic*. Deductive techniques can handle complex systems, but their implementation based on interactive proof assistants requires careful guidance by expert users. Algorithmic techniques such as model checking have found wide acceptance because they promise full automation, but they are usually restricted to finite-state models. Abstraction techniques [2, 3] promise to integrate the two paradigms. In this paper we propose a particular approach to abstraction centered around the notion of *predicate diagrams*, finite-state transition systems whose nodes are labelled with state predicates and represent the set of system states that satisfy these predicates. Diagrams serve as intermediaries between specifications and properties: on the one hand, non-temporal proof obligations ensure that the diagram represents the specification; these must in general be discharged using theorem proving. On the other hand, properties can be established from the diagram using model checking, obviating the need for tedious temporal-logic reasoning that is typical of liveness proofs. We also show that techniques from abstract interpretation can be used to generate diagrams that can at least serve as a starting point for verification.

Predicate diagrams can be shown to be as powerful as traditional deductive verification. Pragmatically, the use of predicate diagrams may be preferable because of the potential for substantial automation. They may also be attractive because similar notations are used in semi-formal design methods [1] and can aid documentation. Similar diagrammatic techniques have been proposed by the STeP group [8]; they can also be used as the basis for deductive model checking [11]. Our definition is somewhat different in the treatment of fairness conditions. It is also slightly more general in that several actions can be represented within a single diagram, which is useful for the proof of refinement relations, a topic that we do not consider in this paper.

2 Predicate Diagrams

We express system specifications and properties in a variant of linear-time temporal logic whose formulas are built from *state predicates* and *actions*, which may contain primed state variables. For example, $x > 3$ is a state predicate, and $x \leq y' + 1$ is an action. For an action A, we denote by ENABLED A the state predicate obtained from A by existential quantification over the primed state variables. For a state predicate P, we denote by P' the action obtained from P by replacing all flexible variables v by v'. Temporal formulas are then built using boolean connectives, the *always* operator $\Box$, and quantification over rigid (state-independent) variables.

The semantics of state formulas is defined with respect to a *state*, i.e. an assignment of values to state variables, and a valuation of the rigid variables. Actions are interpreted relative to a pair (s, t) of states, where s and t interpret respectively the unprimed and primed state variables. Temporal formulas are interpreted over *behaviors*, which are ω-sequences $\sigma = s_0 s_1 \ldots$ of states [6, 9]. Derived

[⋆] This work has been partly supported by a PROCOPE grant from A.P.A.P.E. and DAAD.

connectives include the *eventually* operator defined by $\Diamond F \equiv \neg\Box\neg F$ and, for an action A, the formulas

$$\mathrm{WF}(A) \equiv \Diamond\Box\,\text{ENABLED } A \Rightarrow \Box\Diamond A \qquad \mathrm{SF}(A) \equiv \Box\Diamond\,\text{ENABLED } A \Rightarrow \Box\Diamond A$$

that assert weak and strong fairness conditions for A. This logic is similar to Lamport's Temporal Logic of Actions [7] except that it is not invariant under stuttering. We consider system specifications written in the form $\mathit{Init} \wedge \Box\mathit{Next} \wedge L$ where Init is a state predicate characterizing the system's initial state, Next is an action representing the next-state relation, and L is a conjunction of formulas $\mathrm{WF}(A)$ or $\mathrm{SF}(A)$.

We assume the underlying assertion language to contain a finite set $\mathcal{O}$ of binary relation symbols $\prec$ that are interpreted by well-founded orderings. For $\prec \in \mathcal{O}$, we denote by $\preceq$ its reflexive closure. We write $\mathcal{O}^=$ to denote the set of relation symbols $\prec$ and $\preceq$, for $\prec$ in $\mathcal{O}$.

The definition of predicate diagrams is relative to finite sets $\mathcal{P}$ and $\mathcal{A}$ that contain the state predicates and actions of interest. We denote by $\overline{\mathcal{P}}$ the set containing the predicates in $\mathcal{P}$ and their negations.

Definition 1. *A* predicate diagram $G = (N, I, \delta, o, \zeta)$ *over* $\mathcal{P}$ *and* $\mathcal{A}$ *consists of*

- *a finite set* $N \subseteq 2^{\overline{\mathcal{P}}}$ *of* nodes,
- *a finite set* $I \subseteq N$ *of* initial nodes,
- *a family* $\delta = (\delta_A)_{A\in\mathcal{A}}$ *of relations* $\delta_A \subseteq N \times N$,
- *a family* $o = (o_A)_{A\in\mathcal{A}}$ *of edge labellings* o_A *that associate finite sets* $\{(t_1, \prec_1), \ldots, (t_k, \prec_k)\}$ *of terms* t_i *paired with a relation* $\prec_i \in \mathcal{O}^=$ *with the edges* $(n, m) \in \delta_A$,
- *a mapping* $\zeta : \mathcal{A} \to \{\mathrm{NF}, \mathrm{WF}, \mathrm{SF}\}$ *that associates a fairness condition with every action in* $\mathcal{A}$ *(*NF *indicates "no fairness").*

We say that the action $A \in \mathcal{A}$ can be taken *at node* $n \in N$ *iff* $(n, m) \in \delta_A$ *holds for some* $m \in N$.

Definition 2. *Let* $G = (N, I, \delta, o, \zeta)$ *be a predicate diagram over* $\mathcal{P}$ *and* $\mathcal{A}$*. The set* $tr(G)$ *of* traces through G *consists of all* ω*-sequences* $\sigma = s_0 s_1 \ldots$ *of states such that there exist sequences* $n_0 n_1 \ldots$ *and* $A_0 A_1 \ldots$ *of nodes* $n_i \in N$ *and actions* $A_i \in \mathcal{A}$ *such that all of the following conditions hold:*

- $n_0 \in I$ *is an initial node,*
- $(n_i, n_{i+1}) \in \delta_{A_i}$ *holds for all* $i \in \mathbb{N}$,
- $s_i \models n_i$ *holds for all* $i \in \mathbb{N}$,
- $(s_i, s_{i+1}) \models A_i$ *holds for all* $i \in \mathbb{N}$,
- $(s_i, s_{i+1}) \models t' \prec t$ *holds for all* $i \in \mathbb{N}$ *and* $(t, \prec) \in o_{A_i}(n_i, n_{i+1})$,
- *for every action* $A \in \mathcal{A}$ *such that* $\zeta(A) = \mathrm{WF}$ *there are infinitely many* $i \in \mathbb{N}$ *such that either* $A_i = A$ *or* A *cannot be taken at* n_i*, and*
- *for every action* $A \in \mathcal{A}$ *such that* $\zeta(A) = \mathrm{SF}$*, either* $A_i = A$ *holds for infinitely many* $i \in \mathbb{N}$ *or there are only finitely many* $i \in \mathbb{N}$ *such that* A *can be taken at* n_i.

Predicate diagrams are finite labelled transition systems whose nodes are sets of state predicates that hold at the system states represented by the node. (We indifferently write n for the set and the conjunction of its elements.) Traces through a diagram are behaviors that correspond to fair runs, where enabling conditions of actions $A \in \mathcal{A}$ are identified with the existence of A-labelled edges. We define the notion of *conformance* between diagrams and temporal formulas by trace inclusion. The following theorem gives a set of non-temporal proof conditions that ensure that a diagram conforms to a system specification.

Theorem 3. *Let* $G = (N, I, \delta, o, \zeta)$ *be a predicate diagram over* $\mathcal{P}$ *and* $\mathcal{A}$*, and* $\mathit{Spec} \equiv \mathit{Init} \wedge \Box\mathit{Next} \wedge L$ *be a system specification. If all of the following conditions hold, then* $\sigma \in tr(G)$ *holds for all models* σ *of Spec.*

$$\begin{aligned}
\mathit{Init} &\equiv n \in \mathit{Nat} \land n \neq 0 \land c_0 = \text{“t”} \land c_1 = \text{“t”}\\
\mathit{Eat}_0 &\equiv c_0 = \text{“t”} \land \text{even}(n)\\
&\quad \land c_0' = \text{“e”} \land c_1' = c_1 \land n' = n\\
\mathit{Thk}_0 &\equiv c_0 = \text{“e”} \land c_0' = \text{“t”}\\
&\quad \land n' = n \text{ div } 2 \land c_1' = c_1\\
\mathit{Eat}_1 &\equiv c_1 = \text{“t”} \land \neg\text{even}(n)\\
&\quad \land c_1' = \text{“e”} \land c_0' = c_0 \land n' = n\\
\mathit{Thk}_1 &\equiv c_1 = \text{“e”} \land c_1' = \text{“t”}\\
&\quad \land n' = 3 * n + 1 \land c_0' = c_0\\
\mathit{Next} &\equiv \mathit{Eat}_0 \lor \mathit{Thk}_0 \lor \mathit{Eat}_1 \lor \mathit{Thk}_1\\
\mathit{DM} &\equiv \mathit{Init} \land \Box\mathit{Next} \land \text{WF}(\mathit{Next})
\end{aligned}$$

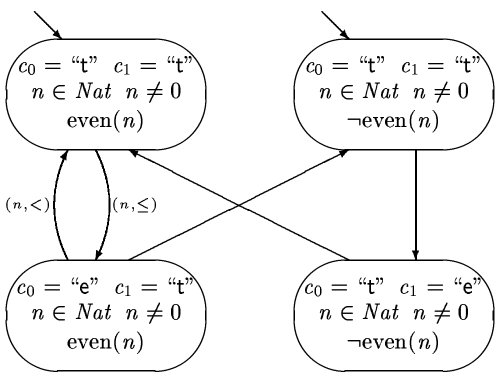

$c_0 = \text{“t”}$ $c_1 = \text{“t”}$
$n \in \mathit{Nat}$ $n \neq 0$
$\text{even}(n)$
$c_0 = \text{“t”}$ $c_1 = \text{“t”}$
$n \in \mathit{Nat}$ $n \neq 0$
$\neg\text{even}(n)$
$(n, <)$
$(n, \leq)$
$c_0 = \text{“e”}$ $c_1 = \text{“t”}$
$n \in \mathit{Nat}$ $n \neq 0$
$\text{even}(n)$
$c_0 = \text{“t”}$ $c_1 = \text{“e”}$
$n \in \mathit{Nat}$ $n \neq 0$
$\neg\text{even}(n)$

Fig. 1. The “dining mathematicians” example.

1. $\models \mathit{Init} \Rightarrow \bigvee_{n \in I} n$
2. $\models n \land \mathit{Next} \Rightarrow \bigvee_{\{(A,m):(n,m)\in\delta_A\}} A \land m'$ *holds for every node* $n \in N$.
3. $\models n \land A \land m' \Rightarrow t' \prec t$ *holds for all* $(n, m) \in \delta_A$ *and all* $(t, \prec) \in o_A(n, m)$.
4. *For every action* $A \in \mathcal{A}$ *such that* $\zeta(A) \neq \text{NF}$:
   (a) *If* $\zeta(A) = \text{WF}$ *then* $\models \mathit{Spec} \Rightarrow \text{WF}(A)$.
   (b) *If* $\zeta(A) = \text{SF}$ *then* $\models \mathit{Spec} \Rightarrow \text{SF}(A)$.
   (c) $\models n \Rightarrow \text{ENABLED } A$ *holds whenever* A *can be taken at node* n.
   (d) $\models n \land A \Rightarrow \neg m'$ *holds for all* $n, m \in N$ *such that* $(n, m) \notin \delta_A$.

If the conditions of theorem 3 holds, we say that G *represents* the specification *Spec*. As an example, Fig. 1 contains the specification of the “dining mathematicians” example, a mutual-exclusion protocol taken from [3], and a predicate diagram (w.r.t. the predicates shown and the single action *Next*) that represents that specification. The variables c_0 and c_1 represent the control states of two processes that alternate between “thinking” and “eating”. The integer variable n is shared between the processes to ensure mutual exclusion.

As finite transition systems, predicate diagrams can easily be encoded into the input languages of standard model checkers such as Spin [5], recording the active node and the last action taken. For every predicate $p \in \mathcal{P}$ we define the atomic propositions b_p and $b_{\neg p}$ such that they are true in precisely those nodes that contain p, respectively $\neg p$. We assume the fairness conditions indicated by the diagram. For every term t and relation $\prec\, \in \mathcal{O}$ such that $(t, \prec)$ appears in some ordering annotation o_A, we add an assumption that ensures that whenever edges labelled by $(t, \prec)$ are taken infinitely often, then edges that are labelled by neither $(t, \prec)$ nor $(t, \preceq)$ are also taken infinitely often. This condition can be expressed by a Streett-type formula.

The encoding of a predicate diagram in a model checker can be used to verify temporal formulas built from predicates in $\mathcal{P}$, substituting b_p and $b_{\neg p}$ for occurrences of p and $\neg p$. For example, we can verify the following properties for the “dining mathematicians” specification from the diagram of Fig. 1:

$$\begin{array}{llll}
(\mathit{Pos}) & \Box(n \in \mathit{Nat} \land n \neq 0) & (\mathit{Excl}) & \Box\neg(c_0 = \text{“e”} \land c_1 = \text{“e”})\\
(\mathit{Live}_0) & \Box\Diamond(c_0 = \text{“e”}) & (\mathit{Live}_1) & \Box\Diamond(c_1 = \text{“e”})
\end{array}$$

The first two properties are invariants that assert that n remains a positive natural number throughout any run of specification DM, and that mutual exclusion is ensured. The remaining properties are

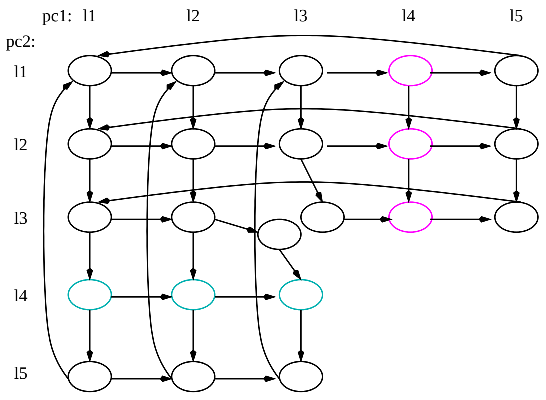


Fig. 2. Predicate diagram for the Bakery algorithm.

liveness properties that assert starvation-freedom for both processes. Note that property $(Live_1)$ can be established despite the cycle between the two leftmost states in the diagram, because the ordering annotations forbid that cycle to be followed indefinitely.

3 Generation of Diagrams by Abstract Evaluation

Instead of verifying the conditions of theorem 3 with the help of a proof assistant, we can try to generate a diagram that represents a given specification by abstract evaluation of the initial condition and the next-state relation based on user-defined (conditional) rewrite rules. Rules appropriate for the dining-mathematicians example include

$$\begin{array}{ll}
\text{even}(0) & \neg\text{even}(1) \\
\text{even}(x) \equiv \text{even}(y) \Rightarrow \mathit{even}(x+y) & \text{even}(x) \not\equiv \text{even}(y) \Rightarrow \neg\mathit{even}(x+y) \\
x \in \mathit{Nat} \wedge \text{even}(x) \wedge x \neq 0 \Rightarrow x \text{ div } 2 \neq 0 & x \in \mathit{Nat} \Rightarrow \text{even}(x) \vee \neg\text{even}(x)
\end{array}$$

These rules are first applied to the initial condition Init. The resulting formula is transformed to disjunctive normal form; each disjunct gives one possible initial node. State expansion is done similarly by evaluating the action Next based on the information about which predicates are true, resp. false, in the source node, and collecting information about the target nodes. The same technique can be used to evaluate other action formulas, in particular those with non-trivial fairness conditions.

It may occasionally happen during state expansion that there is insufficient information about the source node n (represented by unprimed variables) to evaluate some predicate P that appears as a guard in the action currently considered. We can then abort the construction, asking the user to supply more rewrite rules. Alternatively, we may treat P as if it were true; this respects condition (2) of theorem 3 because it can only add disjuncts on the right-hand side. On the other hand, it may invalidate condition (4c), so we are not allowed to use this approach for actions that have non-trivial fairness conditions. In any case, we mark such edges as “maybe” edges to alert the user that they could be obvious culprits when model checking fails. As a third alternative, we have found it useful to reconsider the predecessors, say $n_1, \ldots, n_k$, of n in the part of the diagram that has been constructed so far, and to check whether $n_i \wedge \mathit{Next} \wedge n'$ implies P' or its negation, using an automatic theorem prover. In that case, we add P or $\neg P$ to the set of predicates contained in n, splitting the node if necessary.

If the procedure does not fail, the generated diagram is guaranteed to represent the given specification. In particular, the existence of an edge (other than edges marked as “maybe”) leaving a given node n implies the enabledness of the action at n. We have prototypically implemented the procedure outlined above, using the rewrite engine from Atelier B [10], the automatic prover Simplify [4], and the model checker Spin. Fig. 2 shows a predicate diagram generated for the two-process version of Lamport's "Bakery" algorithm, a standard benchmark problem for the verification of infinite-state systems. We have omitted the predicate labels, but have indicated the control locations ("l4" is the critical section). One process moves horizontally, the other one vertically. The diagram contains two nodes where both processes are at location "l3"; these are distinguished by which process holds the lower "ticket" and may therefore proceed. In producing this diagram, we have only included rules to distinguish between the "ticket" variables t_1 and t_2 being zero or not. The remaining predicates have been introduced "on the fly" via backtracking as explained above. Both mutual exclusion and liveness have been verified by Spin from the diagram.

4 Discussion

We consider predicate diagrams as an interface between specifications and properties. In this paper, we have emphasized the "bottom-up" construction of diagrams starting from the specification. Dually, deductive model checking [11] relies on a "top-down" approach where an initial diagram that ensures the desired property is interactively refined until the conditions of theorem 3 can be established. The bottom-up approach could be characterized as "abstract model checking" because it generates the full state space of the model modulo the abstraction defined by the predicates in $\mathcal{P}$.

The results reported in this paper, though encouraging, are still preliminary. It remains to consider more systematically the annotation of generated diagrams with fairness and ordering information. Another interesting option to explore is the verification of refinement relations between specifications. This is possible because we do not formally distinguish between specifications and properties, and because we allow transitions that correspond to different actions to be represented within the same diagram. We also intend to study appropriate generalizations for parametric systems.

References

1. G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modelling Language: User Guide*. Addison Wesley, 1999.
2. Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
3. Dennis Dams, Orna Grumberg, and Rob Gerth. Abstract interpretation of reactive systems: Abstractions preserving $\forall$CTL*, $\exists$CTL* and CTL*.
4. D. Detlefs, G. Nelson, and J. Saxe. Simplify: the ESC theorem prover. Technical report, Systems Research Center, Digital Equipment Corporation, Palo Alto, CA, nov 1996.
5. Gerard Holzmann. The Spin model checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, may 1997.
6. Fred Kröger. *Temporal Logic of Programs*, volume 8 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1987.
7. Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
8. Z. Manna, A. Browne, H.B. Sipma, and T.E. Uribe. Visual abstractions for temporal verification. In A. Haeberer, editor, *AMAST'98*, volume 1548 of *Lecture Notes in Computer Science*, pages 28–41. Springer-Verlag, 1998.
9. Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems—Specification*. Springer-Verlag, New York, 1992.
10. STERIA Méditerrannée. *Atelier B, Manuel de Référence du Langage B*. GEC Alsthom Transport and STERIA and SNCF and INFRETS and RATP, 1997.
11. H.B. Sipma, T.E. Uribe, and Z. Manna. Deductive model checking. In *8th International Conference on Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 208–219, New Brunswick, N.J., 1996. Springer-Verlag. to appear in *Formal Methods in System Design*, 1999.

Accelerating Techniques for OBDD-based Formal Verification of Sequential Systems

Christoph Meinel
FB Informatik
University of Trier
meinel@uni-trier.de

Christian Stangier
FB Informatik
University of Trier
stangier@uni-trier.de

1 Introduction

Model checking has been proven to be a powerful tool in formal verification of sequential circuits, reactive systems, protocols, etc. The model checking of systems with huge state spaces is possible only if there is an efficient representation of the model. Reduced Ordered Binary Decision Diagrams (shortly: OBDDs) [1] allow an efficient *symbolic* representation of the model [8].

In the following we present techniques that attack two mayor problems of symbolic model checking:

- Finding a good partitioning of the transition relation of the system under consideration.
- Efficient dynamic variable reordering during the computation.

All techniques have in common that *high level information* is utilized. For the first problem RTL information is used to improve the quality of the partitioning. While, for the second problem detailed knowledge about the model checking process is used to accelerate variable reordering.

2 RTL based Partitioning Heuristic

The computation of the reachable states (RS) of a sequential circuit is an important task for synthesis, logic optimization and formal verification. If the RS are computed by using OBDDs, the system under consideration is represented in terms of a transition relation (TR). Since the monolithic representation of the circuit's TR usually leads to unmanageable large OBDD-sizes, the TR has to be partitioned [2, 4]. The quality of the partitioning is crucial for the efficiency of the RS computation. If the TR is divided into too many parts the computation of transitions will be unnecessarily time consuming. On the other hand a number of partitions that is too small will lead to a blow-up of OBDD-size and hence, memory consumption. Partitioning the TR is usually done without utilizing any external information.

A common strategy for partitioning of the TR as it is used e.g. by VIS [3] proceeds in three steps:

1. **Order latches**. First, the latches are ordered by using a benefit heuristic [5] that performs a structural analysis of the transition relations of the latches to address an effective AndExist [4]. operation. Hence, the heuristic considers: variables that may be quantified out, highest index in the function, etc.

2. **Cluster latches**. The single latch relations are clustered by following a greedy strategy. Latches are added to a OBDD (i.e. by performing AND) until the size of the OBDD exceeds a certain threshold.

3. **Order clusters**. In the last step the clusters are ordered similarly to the latches by using a benefit heuristic (VIS uses the same heuristic as in Step 1).

Although the standard method optimizes the partitioning twice, its main disadvantage is that it uses only structural information to optimize the partitioning for an efficient schedule for the AndExist operation during the image computation.

Our new heuristic improves this optimization by including additional semantical information about the represented functions. This information is taken from the register transfer level (RTL) description of the design given in Verilog [6].

As experimental results show, there is a close conjunction between the RTL description and an efficient image computation.

The RTL heuristic proceeds in three steps:

1. **Group latches**. The latches are grouped according to the modules given in the top module of the RTL description in Verilog. Within the groups the latches are ordered by a lexicographic order that takes into account submodule names and bit numbers.

2. **Cluster groups**. The groups represent borders for the clusters. There is no cluster containing latches from different groups. To control the BDD size of the clusters, the greedy partitioning strategy is applied within the groups.

3. **Order clusters**. In the last step the clusters are ordered by using the benefit heuristic from the standard method.

Experiments were performed using real life Verilog benchmarks. The RTL partitioning method significantly outperforms the standard method and reduces CPU time as well as memory consumption by more than 50%.

3 Dynamic Variable Reordering

Due to the huge number of operations applied to the OBDDs during symbolic model checking, the computation time is strongly related to the size of the OBDDs. As the order of the input variables has a strong influence on the size of the OBDDs, well suited variable orders have to be found. Since it is NP-hard to find the optimal variable order for a given function, much effort is spent on finding reasonable good orders or improving given ones. In practice, techniques that improve the size of a given OBDD by changing the variable order dynamically during the computation have been proven to be most powerful. Many common *dynamic reordering* approaches are based on swapping the position of neighboured variables in a given OBDD. This operation can be performed locally and thus, can be computed efficiently. The *sifting* algorithm [11] that is based on this idea moves each variable to the top and to the bottom of the order to find its best position. This algorithm has been proven to be one of the most efficient reordering strategies.

Dynamic reordering strategies are especially useful for symbolic model checking, since the represented functions (e.g. reachable state sets) are changing during computation. As a consequence, the variable order has to be adapted to fulfill the new requirements. Although, dynamic variable reordering may drastically reduce the OBDD size, often it is very time consuming and sometimes does not lead to substantially smaller OBDD sizes.

In the following we present adaptions of reordering techniques originally intended for combinatorial verification to the specific requirements of symbolic model checking. The techniques are orthogonal in the way that they use either structural information about the OBDDs or semantical information about the represented functions. The application of these techniques substantially accelerates the reordering process and makes it possible to finish computations, that are too time consuming, otherwise.

3.1 Block Restricted Sifting

Our goal is to accelerate the variable reordering process while simultaneously retaining reasonable OBDD sizes. To manage this, we adapted a method called *block restricted sifting* (BRS) [10] to the needs of model checking. The idea behind BRS is to move the variables during reordering only within fixed blocks instead of moving them through the complete order. From theory it is known, that changing the variable order of a block does not affect the size of the other blocks.

The determination of the block boundaries follows from a communication complexity argument. A small information flow between two parts of an OBDD indicates a good candidate for a block boundary. If there is only little information flow between two blocks the distribution of variables to these blocks is well chosen. Improving the variable order inside the blocks might lead to a significant reduction of the OBDD size. The information flow is best indicated by the number of subfunctions that cross one level. The *subfunction profile* of an OBDD counts not only the number of nodes per level, it also adds the edges that cross a level without having a node on it to the profile.

With the aid of this profile we get easily computable structure information of the represented function. For a successful application of BRS to symbolic model checking, we have to find solutions for the following problems:

1. Restricting the search space. Sifting only within fixed blocks significantly accelerates the reordering process, but it may keep one away from good orders. In conventional applications like combinational verification the larger number of reordering with changing block boundaries compensates this negative effect. For symbolic model checking this is not true, because of the small number of reorderings. Therefore, we have changed the concept of block boundaries. We now allow a small overlapping of the blocks, i.e. a few levels beside the boundaries of the block are also incorporated in the reordering. This concept partially remedies the problems stated above.

2. Acceleration power. To take full advantage of the BRS approach one should restrict the size of blocks. The native BRS searches for local minima in the subfunction profile. This may lead to unnecessary large blocks in the lower part of the OBDD, where the number of nodes and represented subfunctions naturally decreases. We have changed this strategy to a *first-fit* strategy, i.e the first level that fulfills the given properties is chosen.

3. Settings. The parameter, that is mostly responsible for the trade-off between reordering time and the quality of the computed order is the minimal fraction of variables that a block must contain. This fraction is denoted MINBLOCK. In contrast to combinatorial verification, where MINBLOCK = 10% is an average setting for model checking larger blocks are appropriate.

3.2 Sample Sifting

Sampling is a common heuristic technique applied to optimization problems with huge search spaces. The idea behind the sampling strategy is to choose a relevant sample from the given problem instance to solve the optimization problem for the chosen subset and to generalize the solution to the complete instance.

Applied to the problem of reordering OBDD variables the sampling strategy can be described as follows [9, 7]: (1) Choose some OBDDs or subOBDDs from the common shared OBDD. (2) Copy these OBDDs to a different location. (3) Reorder only the Sample. (4) Shuffle the variables of the original BDD to the newly computed order of the sample.

Step 1 is the most critical during this sample sifting process. As mentioned before, one should choose a relevant sample. If there is some knowledge about the represented functions, it can be used for the choice of samples. The chosen OBDDs should not be too small, so that as many variables of the representation as possible are contained in the sample. If there is no knowledge about the represented functions the sample may be chosen randomly from the single roots of the shared OBDDs. The reordering (Step 3) can be done with any common reordering technique (we used the standard sifting algorithm). If the OBDD size increases after Step 4 the OBDD is reshuffled to the original order, but one may repeat the complete process to obtain better results.

A successful application of the sampling method to model checking is challenging, because the two main problems (Time and Quality) of variable reordering for model checking instantiate as follows:

T1. Small samples. The size of the sample is the most important parameter of sample sifting. Choosing a smaller sample will reduce the computational overhead for copying the sample. But even more important: The accelerating effect of sample sifting results from the fact that only a small OBDD is reordered, also resulting in smaller intermediate OBDD sizes during the reordering. To fulfill the quality requirements of model checking the sample has to be chosen larger than for combinatorial applications.

T2. Number of Trials. More than one trial per sample reordering might be a good idea for combinatorial application but it is not for model checking. Here only one trial is possible due to time limitations.

Q1. Sample without semantical information. If no external semantical information is available one may at least use some structural information about the represented functions. We used a pseudo-random strategy proposed by [9]: Starting from the top level of the OBDD nodes are chosen randomly as roots of subOBDDs for the sample.

Q2. Sample with semantical information. One should make use of the semantical information about represented functions provided by the model checker. In [9] it is proposed to use *recently-used-roots*, i.e. roots involved in operations in the last steps of the computation. Again, this strategy is not suitable for model checking, since the huge number of operations will result in a random choice of roots. Instead, we use *recently-used-important-roots*, i.e. roots involved in elementary model checking operations like Exist-Abstract, Universal-Abstract and And-Abstract (see [8]). If we cannot fulfill the size requirements for the sample by using important roots we fall back to the method of choosing random roots. Using this strategy we obtain the best results for sampling.

Q3. Methods for copying. In [9] copying a fraction of an OBDD is done by copying nodes in DFS postorder way. We replaced this method by a DFS preorder way. Our experience has shown that the preorder method works more stable and produces better results than the postorder method.

Both methods (BRS and Sampling) were able to reduce CPU time by more than 30% with only small additional memory consumption (5% to 10%). For detailed results see [12]

References

[1] R. E. Bryant, *Graph-Based Algorithms for Boolean Function Manipulation*, IEEE Transactions on Computers, C-35, 1986, pp. 677-691.

[2] J. R. Burch, E. M. Clarke, D. E. Long, *Symbolic Model Checking with partitioned transition relations*, Proc. of Int. Conf. on VLSI, 1991.

[3] R.K. Brayton, G. D. Hachtel, A. L. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. A. Edwards, S. P. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, T. Villa, *VIS: A System for Verification and Synthesis*, Proc. of Computer Aided Verification CAV'96, 1996, pp. 428-432.

[4] O. Coudert, C. Berthet and J. C. Madre, *Verification of Synchronous Machines using Symbolic Execution*, Proc. of Workshop on Automatic Verification Methods for Finite State Machines, LNCS 407, Springer, 1989, pp. 365-373.

[5] D. Geist and I. Beer, *Efficient Model Checking by Automated Ordering of Transition Relation Partitions*, Proc. of Computer Aided Verification CAV'94, 1994, pp. 294-310.

[6] D.E. Thomas and P. Moorby, *The Verilog Hardware Description Language*, Kluwer, 1991.

[7] J. Jain, W. Adams and M. Fujita, Sampling schemes for computing OBDD variable orderings, *Proc. IEEE International Conference on Computer-Aided Design*, 331–638, 1998.

[8] K. L. McMillan, *Symbolic model checking*, Kluwer Academic Publishers, 1993.

[9] A. Slobodová and C. Meinel, Sample Method for Minimization of OBDDs. In *Proc. of the International Workshop on Logic Synthesis*, 311–316, 1998.

[10] C. Meinel and A. Slobodová, Speeding up Variable Reordering of OBDDs, in *Proc. of the International Conference on Computer Design*, 338–343, 1997.

[11] R. Rudell, Dynamic Variable Ordering for Ordered Binary Decision Diagrams, in *Proc. IEEE International Conference on Computer-Aided Design* , 42–47, 1993.

[12] C. Meinel and C. Stangier, Speeding Up Symbolic Model Checking by Accelerating Dynamic Variable Reordering, in *Proc. Of 10th Great Lakes Symposium on VLSI*, pp. 39–42, 2000

Parallel Model Checking for the Alternation-Free μ-Calculus

Martin Leucker
Lehrstuhl für Informatik II
Aachen University of Technology, Germany
leucker@i2.informatik.rwth-aachen.de

Abstract

In this extended abstract we describe the design and implementation of a parallel model checking algorithm for the alternation free fragment of the μ-calculus. The algorithm is based on a characterisation of the model checking problem for this fragment of the μ-calculus in terms of games or (equivalently) a certain class of alternating Büchi automata. Strictly speaking, we present a parallel algorithm for checking the emptiness of so called 1–simple–weak–alternating–Büchi automata. It is designed to run on a cluster of workstations. A prototype implementation within the verification tool `Truth` shows promising results.

1 Introduction

Formal methods are becoming more and more popular for the specification and verification of complex hardware and software systems. The term *formal methods* usually denotes the application of mathematical methods for specifying and verifying the underlying systems. The formal specification of a system helps to understand the system under development. Furthermore, a common and formal basis for reasoning about the system is given.

Two approaches for the verification of systems can be distinguished. Model Checking ([Eme90] and Theorem Proving. In this extended abstract we focus on model checking. Numerous case studies have shown that especially model checking improves the detection of errors during the design process (see [CW96] for an overview).

Despite the developments in the last years, the so–called *state space explosion* limits its application. While *partial order reduction* ([Pel98]) or *symbolic model checking* ([McM93]) reduce the state space by orders of magnitude, typical verification tasks still last days on a single workstation (see for example [GLL$^+$00]) or are even undecidable due to memory restrictions.

On the other hand, simple and cheap but powerful parallel computer networks can be build–up by connecting workstations. Libraries such as the message–passing–interface (MPI, [For93]) and a corresponding implementation (e.g. for LINUX) allow a significant speed–up for solving problems provided a suitable parallel algorithm can be formulated. Even more important for model checking, a cluster of workstations offers an enlargement of the total memory available. Hence, it is a fundamental goal to find parallel model checking algorithms which then may be combined with well–known techniques to avoid the state space explosion.

In this extended abstract we present a parallel algorithm for checking the emptiness of so called 1–letter–simple–weak–alternating–Büchi automata (1SWABA). This class of automata was introduced in [BVW94] and it was shown that model checking the alternation–free fragment of the μ-calculus can be reduced to the emptiness problem for this class of automata. While this fragment is already important on its own, it subsumes the logic CTL which is employed in many practical verification tools.

This class of automata can be described in terms of two party games. It can be shown that by solving the emptiness problem for a 1SWABA corresponding to a model checking problem one can determine a winning strategy for the winner of a corresponding model checking game. The latter may be employed by the user of a verification tool for debugging the underlying system interactively. The relation between this class of automata and certain (model checking) games is deeply investigated in [Leu99].

In this way, we get, to our knowledge, the first parallel model checking algorithm for this fragment which even supports interactive debugging. A first prototype implementation within the verification tool `Truth` ([LLNT99]) shows promising results. A drawback of our algorithm is that it *global.* This means, that the whole underlying transition system is constructed and analysed before the model checking problem is answered. However, since model checking the alternation–free μ-calculus is *inherently sequential*, it is unlikely to avoid this problem in the context of parallel algorithms.

2 Alternating Büchi Automata

Nondeterminism gives an automaton the power of existential choices: A word w is accepted by an automaton iff there exists an accepting run on w. Alternation gives a machine the power of universal choices and was studied in [BL80, CKS81] (in the context of automata). In this section we recall the notion of alternating automata along the lines of [Var96] where alternating Büchi automata are used for model checking LTL. For an introduction to Büchi automata we refer to [Tho90].

For a finite set X of variables let $\mathcal{B}^+(X)$ be the set of **positive Boolean formulas** over X, i.e., the smallest set such that

- $X \subseteq \mathcal{B}^+(X)$
- $\texttt{true}, \texttt{false} \in \mathcal{B}^+(X)$
- $\varphi, \psi \in \mathcal{B}^+(X) \Rightarrow \varphi \wedge \psi \in \mathcal{B}^+(X), \varphi \vee \psi \in \mathcal{B}^+(X)$

The dual of a formula $\varphi \in \mathcal{B}^+(X)$ denoted by $\overline{\varphi}$ is the formula where `false` is replaced by `true`, `true` by `false`, $\vee$ by $\wedge$ and $\wedge$ by $\vee$.

We say that a set $Y \subseteq X$ **satisfies** a formula $\varphi \in \mathcal{B}^+(X)$ ($Y \models \varphi$) iff φ evaluates to *true* when the variables in Y are assigned to *true* and the members of $X \backslash Y$ are assigned to *false*. For example, $\{q_1, q_3\}$ as well as $\{q_1, q_4\}$ satisfy the formula $(q_1 \vee q_2) \wedge (q_3 \vee q_4)$.

Let us consider a Büchi automaton (BA). For a state q and an action a let $\{q_1, \ldots, q_k\} = \{q' \mid q \xrightarrow{a} q'\}$ be the set of possible next states for (q, a). The key idea for alternation is to describe the nondeterminism by the formula $q_1 \vee \cdots \vee q_k \in \mathcal{B}^+(Q)$. Hence, we write $q \xrightarrow{a} q_1 \vee \cdots \vee q_k$. If $k = 0$ we write $q \xrightarrow{a} \texttt{false}$. An alternation is introduced by allowing an arbitrary formula of $\mathcal{B}^+(Q)$. Let us be more precise:

Definition 2.1
An ***Alternating Büchi Automaton*** *(ABA) over an alphabet Σ is a tuple $\mathcal{A} = (Q, \delta, q_0, \mathcal{F})$ such that Q is a finite nonempty set of* ***states****, $q_0 \in Q$ is the* ***initial state****, $\mathcal{F} \subseteq Q$ is a set of accepting states and $\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$ is the* ***transition function****.*

Because of universal quantification a run is no longer a sequence but a tree. A ***Q*-labelled tree** τ is a pair (t, T) such that t is a tree and $T : \mathit{nodes}(t) \to Q$. For our needs, we can restrict T to be one–to–one for children of the same parent. Hence, we identify nodes in the following canonical way: The root is marked with ε, if a node s is marked with w then a child s' labelled with q is marked with wq.

For a node s let $|s|$ denote its **height**, i.e., $|\varepsilon| = 0$, $|wq| = |w| + 1$. A branch of τ is a maximal sequence $\beta = s_0, s_1, \ldots$ of nodes of τ such that s_0 is the root of τ and s_i is the parent of s_{i+1}, $i \in \mathbb{N}$.

A **run** of an alternating BA $\mathcal{A} = (Q, \delta, q_0, \mathcal{F})$ on a word $w = a_0 a_1 \ldots$ is a (possibly infinite) Q-labelled tree τ such that $T(\varepsilon) = q_0$ and the following holds:

> if x is a node with $|x| = i$, $T(x) = q$ and $\delta(q, a_i) = \varphi$ then either $\varphi \in \{\mathtt{true}, \mathtt{false}\}$ and x has no children or x has k children $x_1, \ldots, x_k$ for some $k \leq |Q|$ and $\{T(x_1), \ldots, T(x_k)\}$ satisfies φ.

The run τ is **accepting** if every finite branch ends on `true` (i.e., $\delta(T(x), a_i) = \mathtt{true}$ where x denotes the maximum element of the branch wrt. the height and i denotes its height) and every infinite branch of τ hits an element of $\mathcal{F}$ infinitely often. The language $L(\mathcal{A})$ is, as usual, the set of all words for which an accepting run of $\mathcal{A}$ exists.

It is obvious that every Büchi automaton can be turned into an equivalent (wrt. to the accepted language) alternating Büchi automaton in the way described above. The converse is also true and is described for example in [Var96]. However, the construction involves an exponential blow up. This yields an exponential algorithm for checking the emptiness of the language of an ABA. For a subclass of ABAs suitable for our needs a linear non-emptiness decision procedure can be given.

3 1SWABA

An ABA $\mathcal{A} = (Q, \delta, q_0, \mathcal{F})$ over an alphabet Σ is called **1-letter** iff the alphabet contains just one letter, i.e., $|\Sigma| = 1$. Hence, the language of $\mathcal{A}$ is either empty or a single (infinite) word.

A formula $\varphi \in \mathcal{B}^+(X)$ is **simple** if it is any one of `true`, `false`, atomic, a conjunction or a disjunction. The latter means that it has the form $x_1 * \cdots * x_k$ where $* \in \{\vee, \wedge\}$ and $x_i \in X$. An ABA is **simple** if all its transitions are simple.

A **weak** ABA is a tuple $\mathcal{A} = (Q, \delta, q_0, \mathcal{F}^+, \mathcal{F}^-)$ such that $(Q, \delta, q_0, \mathcal{F})$ is an ABA. Furthermore, there exists a partition of Q into disjoint sets $Q_1, \ldots, Q_m$ such that for each set Q_i, either $Q_i \cap \mathcal{F}^+ \neq \emptyset$ or $Q_i \cap \mathcal{F}^- \neq \emptyset$. That means, each Q_i contains either one (or more) **supporter(s)** or one (or more) **spoiler(s).** Q_i is called **accepting set** or **rejecting** set, respectively. In addition, there exists a partial order $\leq$ on the collection of the Q_i's such that for every $q \in Q_i$ and $q' \in Q_j$ for which q' occurs in $\delta(q, a)$ for some $a \in \Sigma$, we have $Q_j \leq Q_i$. Thus, transitions from a state in Q_i lead to states in either the same Q_i or a lower one. It follows that every infinite path of a run of a weak ABA ultimately gets "trapped" within some Q_i. The path then satisfies the acceptance condition if and only if Q_i is an accepting set. Indeed, a run visits infinitely many states in $\mathcal{F}^+$ iff it gets trapped in an accepting set.

This observation can be employed for a sequential algorithm which decides whether the language of a 1SWABA $\mathcal{A}$ is empty or not. We give a sketch of such an algorithm. It constructs a graph whose nodes are the states of $\mathcal{A}$. It labels a state q by *green* or *red* depending on whether there is an accepting run or not for the automaton starting in q.

Let (Q, E, l) be the graph where the nodes are the states of the automaton $\mathcal{A}$ and $(q, q') \in E \subseteq Q \times Q$ iff the formula $\delta(q)$ contains q'. Furthermore, let $l : Q \rightarrow \{\bigvee, \bigwedge\}$ be the mapping denoting whether $\delta(q, a)$ is a disjunction or a conjunction, resp. This graph directly corresponds to the game graph defined when considering the game–based approach of model checking. Hence, we call this graph **game graph**.

As $\mathcal{A}$ is weak, there exists a partition of Q into disjoint sets Q_i such that each set Q_i, either is an **accepting set** or a **rejecting** set. Furthermore, there exists a partial order $\leq$ on the Q_i such that for every $q \in Q_i$ and $q' \in Q_j$ for which q' occurs in $\delta(q, a)$, we have that $Q_j \leq Q_i$. Thus, transitions from a state in Q_i lead to states in either the same Q_i or a lower one. It is easy to see that the Q_i correspond to strongly connected components in the game graph.

The graph can be coloured by processing these strongly connected components (or, a little bit imprecise, the Q_i) up according to the partial order. To make the algorithm deterministic, enlarge the partial order on the Q_i to a total order. Let Q_i be minimal wrt. to $\leq$. Hence, every transition for every state of Q_i leads to Q_i. If Q_i is accepting its nodes are labelled by *green* otherwise by *red*. In particular, if Q_i only consists of a state q with $\delta(q, a) = \texttt{true}$ ($\texttt{false}$) it is labelled by *green* (*red*).

Let Q_j be the next set of states wrt. to the total order. Then all states in $Q_i \leq Q_j$ are already coloured by either *red* or *green*. Now we distinguish two cases. Suppose Q_j is a rejecting set. If there is an $\bigvee$–node x leading to a lower component Q_i which is labelled by *green* then all the nodes are labelled by *green*. Otherwise, every run gets trapped within this rejecting set or within lower rejecting one. Thus there no possibility to successfully leave the rejecting set and all the nodes are coloured by *red*. If Q_j is an accepting set, one has to look for an $\bigwedge$–node leading to a lower *red*-coloured component. Then the component is coloured by *red*, otherwise by *green*. In this way, all nodes can be coloured by either *green* or *red*.

Note that this colouring can be employed to obtain winning strategies for corresponding model checking games which play an important role for interactive debugging specifications ([Leu99, SS98]). In the next section, we explain how to find the labelling in parallel.

4 Parallel Model Checking

The idea of our algorithm is, given a 1SWABA, to construct the game graph in parallel as well as to determine the colour of its nodes in parallel. It is obvious that the construction of the game graph can be carried out in parallel by a typical breadth first strategy. Assuming a shared memory architecture, one has only to care about a well–suited load sharing. The parallel construction and distribution for distributed memory is sketched at the end of this section.

The parallel colouring process is carried out *speculative*. For example, given a supporter $q \in \mathcal{F}^+$, it determines an accepting component. This component is coloured green unless there is an $\bigwedge$–node with an edge to a lower component which is coloured *red*. Anyway, the supporter q will be labelled by *green*. Furthermore, a notification is send to its direct ancestors $q_1, \ldots, q_k$. This notification tells each q_i that one of their children changed their colour. Hence, they recompute there own colour according to following obvious rule: If q_i is an $\bigvee$–node then it is labelled by *green* if one of its successors is *green*, otherwise *red*. If q_i is an $\bigwedge$–node then the dual is carried out. Note, that it is allowed that some successors are not labelled at all. If the colour of q_i has changed, it sends a notification to its predecessors where the same procedure starts again. Otherwise, the procedure is done. It is clear, that the ancestors can be processed in parallel. The whole algorithm stops, if all notifications are processed.

The correctness of the algorithm can easily be seen by recalling that the game graph can be divided into strongly connected components which are partially ordered and a correct labelling can be obtained by processing the components according to this order. The labelling remains correct, if the labelling of higher (wrt. the partial order) components is done speculative and corrected as soon as the correct colour of the lower component is determined. Note, that the colour of *leafs* and *leaf components* (i.e. components which are minimal for the partial order) are correctly labelled from the beginning.

It should be mentioned that for an implementation, the two steps of constructing the game graph and labelling the nodes are carried out concurrently.

While the above presentation is ideal for parallel machines with shared memory, it can be modified for distributed memory machines easily. Let f be a function mapping the states of the automaton to a processor of our network. Usually, one takes a function in the spirit of a hash function assigning to every state an integer and subsequently its value modulo the number of processors. Then f determines the location of every state within the network deterministicly. In a breadth first manner, starting with the initial state q_0 of the automaton, the game graph can be constructed in parallel with the help of f in the following way. Given a state q (and possibly some of its direct predecessors), send it to its processor p_q. If q is already in the local store of p_q, then q is reached a second time, hence the procedure stops. If predecessors of q were send together with q, the list of predecessors is augmented accordingly. If q is not in the local memory of p_q, it is stored there together with the given predecessors as well as all its successors $q_1, \dots, q_k$, the states within the formula $\delta(q, a)$ which are computed. These are send in the same manner to their processors, together with the information that q is the direct predecessor. The corresponding processes update their local memory similarly.

It should be clear, how to combine the parallel construction of the game graph and the labelling procedure described before.

5 Conclusion

In this paper we presented a *parallel* game–based model checking algorithm for an important fragment of the μ-calculus. The demand for parallel algorithms becomes visible by considering the memory and run–time consumptions of sequential algorithms. Since the employed fragment of the μ-calculus subsumes the well–known logic CTL it is of high practical interest. At the moment, we implement the algorithm within our verification platform `Truth`. We have already implemented the described parallel construction of the state space of the automaton and are extending this implementation by the labelling routines. We have constructed automata with several millions of states within minutes on a workstation cluster consisting of up to 52 workstations. We found out, that the algorithm scales very well wrt. run–time and memory consumption when enlarging the number of workstations. Furthermore, the number of states on each processor is nearly the same. Within the next weeks, we will have more results concerning the labelling routine. For the further future, we will analyse how to combine our algorithm with partial order reduction and symbolic model checking.

References

[BL80] J.A. Brzozowski and E. Leiss. On equations for regular languages, finite automata, and sequential networks. *Theoretical Computer Science*, 10:19–35, 1980.

[BVW94] O. Bernholtz, M.Y. Vardi, and P. Wolper. An automata–theoretic approach to branching–time model checking. In D.L. Dill, editor, *Proceedings of the 6th International Conference on Computer–Aided Verification (CAV'94)*, volume 818 of *Lecture Notes in Computer Science*, pages 142–155. Springer–Verlag, 1994.

[CKS81] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, January 1981.

[CW96] E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. Technical Report CMU-CS-96-178, Carnegie Mellon University (CMU), September 1996. URL: `ftp://reports.adm.cs.cmu.edu/usr/anon/1996/CMU-CS-96-178.ps`.

[Eme90] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 14, pages 996–1072. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, New York, N.Y., 1990.

[For93] The Message Passing Interface Forum. Document for a Standard Message-Passing Interface. CS-93-214, University of Tennessee, 11 1993.

[GLL$^+$00] S. Gnesi, D. Latella, G. Lenzini, C. Abbaneo, A. Amendola, and P. Marmo. A formal specification and validation of a critical system in presence of byzantine errors. In *TACAS*, number ?? in LNCS, Vancouver, BC, Canada, 2000. Springer.

[Leu99] M. Leucker. Model checking games for the alternation free mu-calculus and alternating automata. In Harald Ganzinger, David McAllester, and Andrei Voronkov, editors, *Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning*, volume 1705 of *Lecture Notes in Artificial Intelligence*, pages 77–91. Springer, 1999.

[LLNT99] M. Lange, M. Leucker, T. Noll, and S. Tobies. Truth – a verification platform for concurrent systems. In *Tool Support for System Specification, Development, and Verification*, Advances in Computing Science. Springer-Verlag Wien New York, 1999.

[McM93] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[Pel98] Doron Peled. Ten years of partial order reduction. In *CAV, Computer Aided Verification*, number 1427 in LNCS, pages 17–28, Vancouver, BC, Canada, 1998. Springer.

[SS98] Perdita Stevens and Colin Stirling. Practical model-checking using games. In B. Steffen, editor, *Proceedings of the 4th International Conforence on Tools and algorithms for the construction and analysis of systems (TACAS '98)*, volume 1384 of *Lecture Notes in Computer Science*, pages 85–101, New York, NY, USA, 1998. Springer-Verlag Inc.

[Tho90] Wolfgang Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 4, pages 133–191. Elsevier Science Publishers B. V., 1990.

[Var96] Moshe Y. Vardi. *An Automata-Theoretic Approach to Linear Temporal Logic*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer-Verlag Inc., New York, NY, USA, 1996.

Model Checking Statecharts with Keeping Hierachies

G. Kwon
Artificial Intelligence Institute
Department of Computer Science
Dresden University of Technology
Germany

Abstract

Hierarchy plays an important role in every phases of software development from specification to testing. However, it is an excuse for contemporary model checkers such as SMV [McMillian 92] which consider only the flat-structure of finite transition system as an input. Thus hierarchical structures are flatten-out first before commencing model checking it. To overcome this problem, some researches have recently done on hierarchical model checking which is able to handle hierarchical structures of finite transition systems directly [Alur 98]. But it is still immature and need more considerations on this field.

In this paper we will concentrate on model checking statecharts with the symbolic model checker SMV. We regard this works as stepping-stone to the research on hierarchical model checking. The reason we consider statecharts as the target specification is that it provides highly expressive constructs including hierarchy for specifying large-scale specifications with ease. Already, there has been some works on model checking statecharts as shown in Table 1.

Table 1 : Previous works on model checking statecharts or very similar to it

Author	Specifications	Model Checker	Property	Extent	Translation	Hier-archy
[Chan, 98]	RSML Machine	SMV	CTL	Basic	Manual	No
[Damm, 98]	Harel Statecharts	SIEMENS AG	STD	Full	Automatic	No
[Mikk, 98]	Harel Statecharts	SPIN	LTL	Basic	Automatic	No
[Latella, 99]	UML Statecharts	SPIN	LTL	Basic	Manual	No
[Gnesi, 99]	UML Statecharts	JACK	ACTL	Basic	Manual	No
[Lilius, 99]	UML Statecharts	SPIN	LTL	Full	Automatic	No

We classified them into six categories shown in the above. To our surprise, no one mentions about how to keep hierarchical structure of statecharts when model checking it. To the best of our knowledge, all previous works deploy bulldozer to flatten out hierarchical structures of statecharts before starting model checking it.

However, such a flattening can cause transition relations to be very messy. As an example, consider the following statecharts for the operation mode of dish washer shown in Figure 1. Only 9 transition relations are there in the original hierarchical model. If we flatten out, then we get 25 transition relations: 7 for normal operations, 6 for transitions from operating states to closed, 6 for recording its recent history, and 6 for restoring the previous status when the door closed. The more we have transition relations, the more SMV has BDD nodes for representing them.

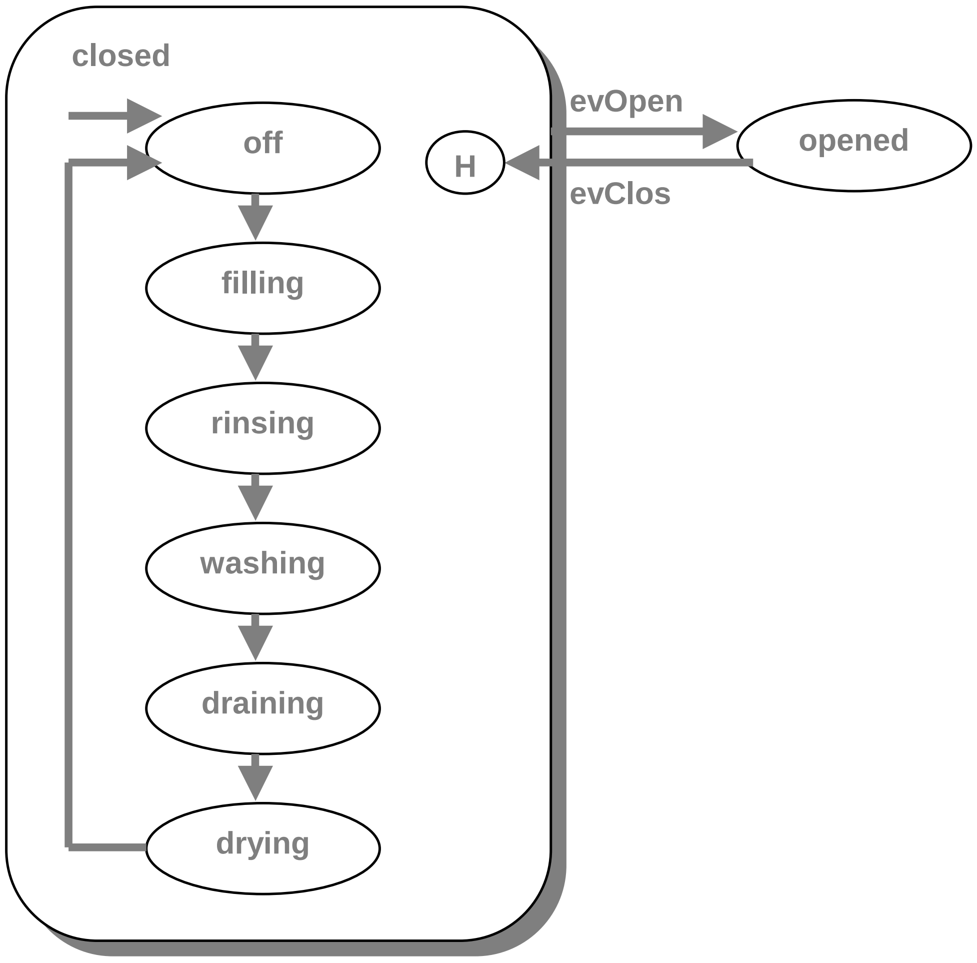


Figure 1: Statecharts for the operation mode in dish washer from [iLogix, 99]

On the other hand, problems with flattening is not over here. It can cause a blow-up of states, particularly when there is a lot of sharing. As an example of this, consider the hierarchical model of a digital clock from [Alur 98]. In the top-level, there are 24 superstates for representing hours of the day. Each such state, in turn, is a hierarchical state consisting of a cycle through 60 superstates counting minutes, each of which, in turn, is a superstate consisting of a cycle counting seconds. Such a hierarchical statecharts of a digital clock has 24 + 60 + 60 = 144 states. On the other hand, the flattened statecharts has 24 X 60 X 60 = 84,600 states.

For these two reasons, we are interested in keeping hierarchies when model checking statecharts. Fortunately, SMV provides a module mechanism for the sake of localization as well as reusability. We believe that it is worthy to make a hierarchical SMV program for statechart specification with the use of modules. Although modules support only hierarchy with respect to the syntactic viewpoint not the semantic one, the hierarchical SMV program for statecharts to be model checked is at least better than the flattened one. We will show the formalization of statecharts and its semantics. Then the translation rules are described according to the semantics. And we discuss its advantages and disadvantages of our works.

References

[Alur 98] R. Alur and M. Yannakakis, Model checking of hierarchical state machines. Sixth ACM Symposium on the Foundations of Software Engineering, pp.175-188, 1998

[Chan, 98] W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, J. D. Reese, Model checking large software specifications, IEEE Trans. on Software Engineering, Vol.24, No.7, pp.498-520, 1998

[Damm, 98] Werner Damm, Bernhard Josko, Hardi Hungar, and Amir Pnueli. A compositional real-time semantics of STATEMATE designs. In W.-P. de Roever, H. Langmaack, and A. Pnueli, editors, Proceedings COMPOS'97, Lecture Notes in Computer Science 1536, pages 186-238. Springer-Verlag, 1998

[Gnesi, 99] S. Gnesi, D. Latella, M. Massink, Model checking UML satetchart diagrams using JACK, 1999

[iLogix, 99] iLogix, Rhapsody Totorial, 1999

[Latella, 99] D. Latella, I. Majzik and M. Massink, Towards a formal operational semantics of UML statechart diagrams, The 3rd International Conference on Formal Methods for Open Object-Oriented Distributed Systems, Kluwer Academic Publishers, 1999

[Lilius, 99] J. Lilius and I.P. Paltor, The Semantics of UML State Machines, In the proceedings of <<UML>>'99, Springer-Verlag, 1999

[McMillian 92] K.L. McMillan, Symbolic Model Checking: An approach to the state explosion problem, PhD thesis, Carnegie Mellon University, 1992

[Mikk, 98] E. Mikk, Y. Lakhnech, M. Siegel, G.J. Holzmann, Implementing statecharts in Promela/SPIN, Technical Report, Bell Labs, Lucent Technologies, 1998

Reflections on Ken Thompson's *Reflections on Trusting Trust* (Extended Abstract)

Wolfgang Goerigk*

Introduction

In 1984, Ken Thompson, the inventor of Unix, devoted his Turing Award lecture [20] to security problems due to Trojan Horses intruded by compiler implementations. Source level verification is not sufficient in order to provide *trusted* compiler executables.

In this paper[1] we formally and mechanically prove that the previous sentence is true, using the ACL2 theorem prover [11]. Furthermore, we sketch a correct way out that involves an explicit compiler target level correctness proof [12, 7, 9].

Scenario and the Challenge Suppose we construct and verify a compiler written in its own source language **SL** and use an initial **SL** compiler executable to bootstrap it, producing a new executable, now a machine implementation of the correct compiler. Then, as usual, we run the bootstrap test [21], that is we execute the new compiler implementation and apply it to the correct source code again. Suppose this test succeeds, i.e., the new compiler executable, applied to its own verified source code, reproduces itself instruction by instruction.

This gives us two identical new compiler executables, generated by applying implementations of the verified source code to the verified source code. Can we give a rigorous mathematical argument that proves our procedure to finally produce a correct compiler machine implementation?

Unfortunately not. The final executable can be as incorrect as the initial implementation. It is the challenge of this paper to construct such a bad guy, which will pass nearly every test, compiler validation suites, the very practical and valuable strong bootstrap test, and nevertheless it might eventually cause a catastrophe.

We use the ACL2 [11] to formalize an abstract stack machine an to mechanically verify a Lisp compiler. We prove *preservation of partial correctness* (L-simulation) [5, 17, 7, 3]. ACL2 allows for (efficient) execution of machine programs and to formally talk about compiler bootstrapping, i.e., about executing compiled compilers compiling compilers. We construct and execute a malicious machine implementation of the verified compiler, bootstrapping the latter. We prove mechanically that – after any precaution on source level – this executable compiles any but two programs correctly, but passes the strong compiler test and compiles exactly one additional source program incorrectly. Finally, we sketch a technique for target level compiler correctness proofs, and finish with conclusions and some remarks on related work.

The Source Language SL

Our source language is a small subset of ACL2 Lisp, with only a few built in Lisp functions and a restricted syntax. It is similar to the language L3 [13] of first order mutually recursive functions. A program is a list of function definitions, followed by a list of *input* variables and a *main* program expression which may use the input variables.

$$
\begin{array}{lcl}
p & ::= & \texttt{((}d_1 \ \ldots \ d_n\texttt{)} \ \texttt{(}x_1 \ \ldots \ x_k\texttt{)} \ e\texttt{)} \\
d & ::= & \texttt{(defun } f \ \texttt{(}x_1 \ \ldots \ x_n\texttt{)} \ e\texttt{)} \\
e & ::= & c \mid x \mid \texttt{(if } e_1 \ e_2 \ e_3\texttt{)} \mid \texttt{(}f \ e_1 \ \ldots \ e_n\texttt{)} \mid \\
 & & \texttt{(}op \ e_1 \ \ldots \ e_n\texttt{)}
\end{array}
$$

*Institut für Informatik und Praktische Mathematik, University of Kiel, Olshausenstraße 40, D-24098 Kiel, Germany. wg@informatik.uni-kiel.de

[1]The work reported here has been supported by the Deutsche Forschungsgemeinschaft (DFG) in the *Verifix* and *VerComp* projects on *Correct Compilers* and *Techniques for Compiler Implementation Verification*

Expressions e (forms) are either constants c, variables x (symbols not equal to `nil` or `t`), conditional expressions, user defined function or operator calls. Functions and operators have a fixed number of arguments.

We omit the set of operators, an example, and also the definition of well-formed expressions and programs in this abstract. Due to lack of space we also shift the definition of an operational semantics (a call-by-value semantics defined by an interpreter called `evaluate` written in ACL2) to the appendix (section A).

The Target Machine

The target machine is an abstract stack machine. Its configuration consists of a `code` part and a separate state (or memory) `stack`, which is a data stack containing Lisp s-expressions. The `code` is never changed.

The machine has six machine instructions. We can push a constant c onto the stack (`PUSHC` c), push the stack content (the variable) at a particular stack position (`PUSHV` i), pop the n stack elements below the top (`POP` n). There is a subroutine call (`CALL` f), that executes the code associated to a subroutine name within `code`, and the (`OPR` *op*) instruction applies an operator to the topmost (one or two) stack cell(s). Moreover, we have a structured (`IF` *then else*) instruction, that removes the top of stack and executes the instruction sequence *else* if the top has been `NIL`, *then* otherwise.

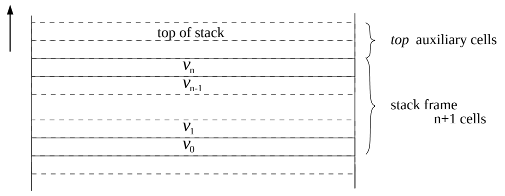
Machine programs (m) are sequences of (mutually recursive) subroutine declarations (d) together with a *main* instruction sequence which is to be executed on an initial `stack` after downloading the list of declarations into `code`.

$$\begin{array}{lcl} m & ::= & (d_1 \ \ldots \ d_n \ (ins_1 \ \ldots \ ins_k)) \\ d & ::= & (\texttt{defcode} \ f \ (ins_1 \ \ldots \ ins_k)) \end{array}$$

In the appendix (section B) we define this machine, i.e. an operational semantics of machine programs, in ACL2.

Compiling SL to TL

The principle idea of executing **SL** programs on such a machine is quite simple and known as the *stack principle*. Arguments are found on the stack; for a given expression e we generate a sequence of instructions that pushes the value of e onto the stack. Functions or operators consume (pop) their arguments and push the result.



In order to execute a function call (f e_0 ... e_n), we compute the argument forms e_0 ... e_n from left to right and push result by result onto the stack. After invoking f and using *top* auxiliary variables, we find the value v_i of the formal parameter x_i at position ($top + n - i$). Using the current formal parameter list as a compiletime environment, we can find the variable positions and compute their *relative addresses.*

Compiling expressions and programs

Due to lack of space we shift the compiler program $\mathcal{C}_{\mathbf{SL}}$ to the appendix (section C) and just give some explanations on how expressions, definitions and programs are compiled:

Constants are pushed onto the stack using `PUSHC`. For a variable we push the content of the stack at its relative address using `PUSHV`. For a function or operator call we subsequently compile the argument forms, thereby incrementing the number *top* of used stack cells, and then generate a `CALL` or `OPR`. For a conditional, we compile the condition and then use the machine conditional containing the compiled code for the two alternatives.

For a function definition, we compile the body in a new environment, which is the formal parameter list, say of length n. The stack-frame will be on top initially, so `top` is zero. The final instruction (`POP` n) removes the arguments from the stack and leaves the result on top.

The function `compile-program` has three arguments corresponding to the three parts of an **SL** program, `defs`, `vars`, and `main`. It compiles the definitions in `defs` and appends the result to the compiled `main` expression. Finally, we generate a `(POP (len vars))`. Thus, if executed on an initial stack, the generated program will remove its inputs and either return a stack with the result on top, or an `error`.

Compiler Source Level Correctness

Before we start proving the correctness of $\mathcal{C}_{\mathbf{SL}}$, let us first give some remarks on our notion of correct compilation. We call a (machine) program m a *correct implementation* of a (source) program p, if every non-erroneous result of m is also a possible result of p. The machine may fail, but it will never return an unexpected non-erroneous result[2]. Consequently, we call a compiler, e.g. $\mathcal{C}_{\mathbf{SL}}$, *correct* or say that it *preserves partial correctness*, if it at most generates correct implementations m of source programs p in the above sense.

Compared to specification refinement in VDM [10], or to former work on compiler verification using ACL2 resp. its predecessor Nqthm ([14, 22, 15]), there is a subtle difference in the notion of *correct compilation.* In [15] for instance J Moore proves, that every non-erroneous result of (the Piton machine on) p will also be computed by m (on the FM9001), that m is more defined than p. This allows for optimizations, but trusted execution of m requires total correctness of p.

The two notions are incompatible. Preservation of partial correctness allows for executing target programs with a well-placed trust in their results, even if the source program is not totally correct. But it requires for instance complete runtime error checking. However, it is quite close to the every day programmer's intuition. The target program execution either returns the correct result, or signals an error, or it returns no result at all.

The Correctness Proof

Due to lack of space in this abstract we can only briefly sketch the mechanically checked correctness proof that $\mathcal{C}_{\mathbf{SL}}$ preserves partial correctness. Essentially, we prove two theorems simultaneously by a combined computational and structural induction, the correctness theorem for expressions, and a similar theorem for expression lists. We prove them for well-formed expressions in well-formed programs.

Theorem 1 *(correctness for form (lists))*
If the machine, executed on a compiled `form` *(list), is defined on a* `stack` *for an* `n`*, then the following three conjectures hold:*

1. *The semantics of the* `form` *(list) – in the given function environemnt and with the free variables bound to their values in the current stack-frame – is defined for the same* `n`*,*
2. *the machine returns a new stack with the value(s) of the* `form`*(s) on top (in reverse order), and*
3. *the stack just below the result value(s) remaines unchanged.*

The full paper will contain the ACL2 formulations of these theorems and we will present a more elaborated description of the proof. We need a lot of lemmas. The combined induction is suggested by a large admissible ACL2 function which explicitly lists the entire set of induction hypothesises we need for the proof to succeed in every single case.

It is interesting that we have to prove the definedness of the source code semantics and the correctness of the machine result simultaneously. The reason is the conditional. It needs not be strict in both alternatives, so its definedness inductively depends on the value of the condition. The conditional has actually been the challenging case to find this proof.

The Compiler Correctness Theorem

The correctness theorem for programs is, after some technical lemmas, a simple consequence of the previous theorems, applied to the main expression of the program. If the source program is well-formed, and if the target program (applied to an initial stack containing the correct number of `inputs` in reverse order on top) returns a non-erroneous result on top (is `defined`), then this result is `equal` to the semantics (`evaluate`) of the program applied to the inputs.

[2] *Preservation of partial program correctness* [5, 16]. In case of non-determinism we additionally allow the target program to be more deterministic than the source program.

Theorem 2 *(compiler correctness)*

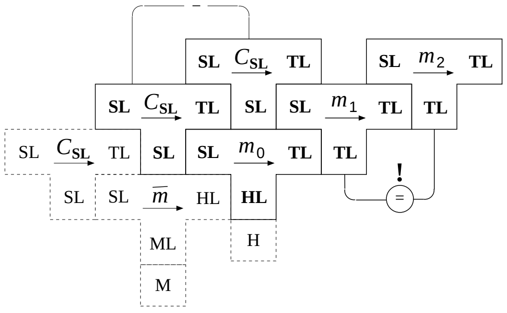
```
(defthm compiler-correctness-for-programs
 (let ((new-stack
        (execute
         (compile-program program)
         (append (rev inputs) old-stack) n))
       (value
        (car (evaluate program inputs n))))
  (implies
   (and (wellformed-program program)
        (defined new-stack)
        (true-listp inputs)
        (equal (len vars) (len inputs)))
   (equal new-stack
          (cons value old-stack)))))
```

The above theorem is stronger and hence implies that $\mathcal{C}_{\mathbf{SL}}$ *preserves partial program correctness.*

This finishes the first part of the paper. Section C shows a concrete compiler source program $\mathcal{C}_{\mathbf{SL}}$ proved to correctly compile well-formed **SL** programs to **TL** programs with respect to their operational semantics as formally defined in Section A and B, respectively. The machine semantics is executable, so that we can run compiled programs. The compiler is written in **SL** itself, so that we can compile it to **TL** (somehow), run it on the machine, and apply it to itself.

Compiler Bootstrap Test

We are now in a typical situation in a compiler development. We have a correct source program, but not yet a correct executable. Suppose that we use an existing (but usually unverified) compiler executable $\overline{m}$ from **SL** to **TL** in order to generate an initial implementation m_0 of $\mathcal{C}_{\mathbf{SL}}$, maybe running on a different machine M or generating (different) host machine code **HL**. We can then run m_0 on machine H, compile $\mathcal{C}_{\mathbf{SL}}$ again and generate an initial **TL** implementation m_1.



If $\mathcal{C}_{\mathbf{SL}}$ and hence m_1 are deterministic programs, and if we repeat this procedure and apply m_1 to $\mathcal{C}_{\mathbf{SL}}$ again, and if all compilers work correctly, we get m_1 back, i.e. $m_2 = m_1$. The bootstrap test (or strong compiler test) [21] succeeds.

Therefore, compiler constructors hold this test in high esteem in order to uncover bugs. If the compilers are correct (and deterministic), we can prove that the bootstrap test will succeed [4]. Hence, if it does not, something has gone wrong.

Theorem 3 *(Bootstrapping Theorem)*
If m_0 and $\mathcal{C}_{\mathbf{SL}}$ are both correct, if m_0, applied to $\mathcal{C}_{\mathbf{SL}}$, terminates with regular result m_1, and if the underlying hardware worked correctly, then m_1 is correct. □

Theorem 4 *(Bootstrap Test Theorem)*
If m_0 and $\mathcal{C}_{\mathbf{SL}}$ are both correct and deterministic, if m_0, applied to $\mathcal{C}_{\mathbf{SL}}$, terminates with regular result m_1, if m_1, applied to $\mathcal{C}_{\mathbf{SL}}$, terminates with regular result m_2, and if the underlying hardware worked correctly, then $m_1 = m_2$. □

We are now exactly at the point where practical compiler construction at present stops. We have implemented a correct compiler on the desired target machine, we have tested it, maybe even on some more test programs or a validation suite. And we did a lot more than just hand-waving or source code inspection for $\mathcal{C}_{\mathbf{SL}}$. We proved it correct. But our story does not end here.

Self-reproduction and Reflection

The full paper will contain a section on a quite general technique to construct self-reproducing and reflective programs. Due to lack of space we have to omit this here. Just for a taste of it, consider the following *conditionally self-refroducing* function, which is an incorrect implementation of the identity function that works correctly in any but exactly two particular cases:

```
(defun ident (x)
 (let ((b '(defun ident (x)
            (let ((b '2000))
              (cond ((equal x 'ident)
                     (subst b (+ 1999 1) b))
                    ((equal x 'login) 'Oops)
                    (t x))))))
   (cond ((equal x 'ident)
          (subst b (+ 1999 1) b))
         ((equal x 'login) 'Oops)
         (t x))))
```

It returns its own code, if applied to the argument `ident` (*reproduction*). It returns `Oops`, if the argument is `login` (*catastrophe*), and otherwise it returns its argument (*normal*).

The Bad Guy

We will now construct an incorrect machine executable $\overline{m}_0$ by correctly compiling an incorrect source program $\overline{\mathcal{C}_{\mathbf{SL}}}$, which we program according to

$$\overline{\mathcal{C}_{\mathbf{SL}}}\,[\,\pi\,] \;=\; \begin{cases} \mathcal{C}_{\mathbf{SL}}\,[\,\overline{\mathcal{C}_{\mathbf{SL}}}\,] \;=\; \overline{m}_0 & ,\ \textit{if}\ \pi = \mathcal{C}_{\mathbf{SL}} \\ \overline{m}_l & ,\ \textit{if}\ \pi = \mathcal{L}_{\mathbf{SL}} \\ \mathcal{C}_{\mathbf{SL}}\,[\,\pi\,] & \textit{otherwise} \end{cases}$$

Whenever applied to $\mathcal{C}_{\mathbf{SL}}$, it returns itself (*reproduction*) and hence passes the bootstrap test (arbitrarily often). It incorrectly compiles $\mathcal{L}_{\mathbf{SL}}$ to $\overline{m}_l$ (*catastrophe*), and in any other (*normal*) case it behaves like $\mathcal{C}_{\mathbf{SL}}$. The reproduction case does not even show an effect unless we apply the result in the catastrophic case (and wait for the catastrophe to happen). It is highly unlikely to find such bug by testing.

The full paper contains the entire construction process and the program $\overline{\mathcal{C}_{\mathbf{SL}}}$. Moreover, we prove formally, that source level verification is not sufficient to guarantee the correctness of compiler executables.

Full Compiler Correctness

The incorrect initial **TL** implementation $\overline{m}_0$ must (syntactically) differ from what we would expect as the correct result m_0 of compiling $\mathcal{C}_{\mathbf{SL}}$. If we carefully look through $\overline{m}_0$ and compare it with m_0, we would find the mismatch. This is the idea of an explicit compiler target level verification.

Let $\mathcal{CC}_{\mathbf{SL},\mathbf{TL}}$ be a (semantically) correct compiling relation between **SL** and **TL**, and let $\mathcal{C}_{\mathbf{SL}}$ be a correct implementation of $\mathcal{CC}_{\mathbf{SL},\mathbf{TL}}$[3]. The following theorem from [9] can easily be proved by transitivity of correct implementation:

Theorem 5 *(Semantics to Syntax)*
If $\mathcal{CC}_{\mathbf{SL},\mathbf{TL}}$ is correct, if $\mathcal{C}_{\mathbf{SL}}$ is a correct implementation of $\mathcal{CC}_{\mathbf{SL},\mathbf{TL}}$, and if $(\mathcal{C}_{\mathbf{SL}}, m) \in \mathcal{CC}_{\mathbf{SL},\mathbf{TL}}$, then m is a correct implementation of $\mathcal{CC}_{\mathbf{SL},\mathbf{TL}}$ as well. Thus, m is a correct compiler (executable) from **SL** *to* **TL**. □

That means, that if the bootstrap test succeeds in a stronger sense, if we can assure that this one execution of the compiler applied to itself generated the expected target code $m = m_0$, then we can guarantee that m, if successful, correctly compiles any program. Note, that this theorem reduces the semantical question of correct compilation to a final purely syntactical *a posteriori code inspection based on code comparison* between $\mathcal{C}_{\mathbf{SL}}$ and m. m might be mechanically generated by an arbitrary initial unsafe implementation of $\mathcal{C}_{\mathbf{SL}}$. However, if we would try to generate it by applying $\overline{m}_0$ to $\mathcal{C}_{\mathbf{SL}}$, the test would fail.

It turns out that there is a technique for such proofs based on code inspection, which additionally exploits modularization into adequate intermediate layers. A *diagonal argument* allows for trusted machine support to generate large parts without need for checking at all [9, 4].

5.1 Conclusions and Related Work

We have formalized and proved the impact of a problem due to *Trojan Horses* in compiler implementations. In the operating system community, the problem is known at least since Ken Thompson's Turing Award Lecture in 1984 [20]. In the compiler community, the need for compiler implementation verification has, as far as we know,

[3]We do not see a difference between $\mathcal{CC}_{\mathbf{SL},\mathbf{TL}}$ and $\mathcal{C}_{\mathbf{SL}}$ in our ACL2 scenario, but we can look at $\mathcal{C}_{\mathbf{SL}}$ as a correct implementation of a corresponding specification.

first been mentioned in 1986 [2] and later in 1988 [14] and again in [15]. But even today the problem is too often neglected by many authors:

Unless we seriously and rigorously bridge the gap between source level program correctness and running code, we hopelessly remain sitting in the present situation which is best characterized by the *moral* of Ken Thompson's Turing Award lecture in 1984: *"You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me.) No amount of source-level verification or scrutiny will protect you from using untrusted code."*

We have sketched a solution [7, 9], a technique for explicit compiler target level verification based on syntactical code comparison. It is closely related to *runtime result verification* [6, 19] or *program checking* [1]. It can also be seen as a *translation validation* [18], however, we check sufficient *syntactical* conditions for the result of one particular compiler bootstrap, whereas in [18] every compiler result is semantically validated at runtime. Semantical translation validation might have limitations with respect to real-world optimizing compilers, and furthermore it produces a new implementation correctness problem for the checker routines. However, the latter might be much easier than proving correctness a priori [6].

In our approach, source code verification and one additional syntactical target code inspection together guarantee the (semantical) correctness and hence *trustworthiness* of the generated compiler executable.

References

[1] M. Blum and S. Kannan. Program correctness checking ... and the design of programs that check their work. In *Proceedings 21st Symposium on Theory of Computing*, 1989.

[2] L.M. Chirica and D.F. Martin. Toward Compiler Implementation Correctness Proofs. *ACM Transactions on Programming Languages and Systems*, 8(2):185–214, April 1986.

[3] Wolfgang Goerigk. Compiler Verification Revisited. In Matt Kaufmann, Pete Manolios, and J Strother Moore, editors, *Computer Aided Reasoning: ACL2 Case Studies.* Kluwer Academic Publishers, 1999. In preparation.

[4] Wolfgang Goerigk. On Trojan Horses in Compiler Implementations. In F. Saglietti and W. Goerigk, editors, *Proc. des Workshops* Sicherheit und Zuverlässigkeit softwarebasierter Systeme, ISTec Report ISTec-A-367, ISBN 3-00-004872-3, Garching, August 1999.

[5] Wolfgang Goerigk, Axel Dold, Thilo Gaul, Gerhard Goos, Andreas Heberle, Friedrich W. von Henke, Ulrich Hoffmann, Hans Langmaack, Holger Pfeifer, Harald Ruess, and Wolf Zimmermann. Compiler Correctness and Implementation Verification: The *Verifix* Approach. In P. Fritzson, editor, *Proceedings of the Poster Session of CC '96 – International Conference on Compiler Construction*, pages 65 – 73, IDA Technical Report LiTH-IDA-R-96-12, Linkøping, Sweden, 1996.

[6] Wolfgang Goerigk, Thilo Gaul, and Wolf Zimmermann. Correct Programs without Proof? On Checker-Based Program Verification. In *Proceedings ATOOLS'98 Workshop on "Tool Support for System Specification, Development, and Verification"*, Advances in Computing Science, Malente, 1998. Springer Verlag.

[7] Wolfgang Goerigk and Ulrich Hoffmann. Rigorous Compiler Implementation Correctness: How to Prove the Real Thing Correct. In Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullmann, editors, *Applied Formal Methods - FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 122 – 136, 1998.

[8] Wolfgang Goerigk and Markus Müller-Olm. Erhaltung partieller Korrektheit bei beschränkten Maschinenressourcen. – Eine Beweisskizze –. Technical Report Verifix/CAU/2.5, CAU Kiel, 1996.

[9] Ulrich Hoffmann. *Compiler Implementation Verification through Rigorous Syntactical Code Inspection.* PhD thesis, Technische Fakultät der Christian-Albrechts-Universität zu Kiel, Kiel, 1998.

[10] C.B. Jones. *Systematic Software Development Using VDM, 2nd ed.* Prentice Hall, New York, London, 1990.

[11] M. Kaufmann and J S. Moore. Design Goals of ACL2. Technical Report 101, Computational Logic, Inc., August 1994.

[12] H. Langmaack. Contribution to Goodenough's and Gerhart's Theory of Software Testing and Verification: Relation between Strong Compiler Test and Compiler Implementation Verification.

Foundations of Computer Science: Potential-Theory-Cognition. *LNCS*, 1337:321–335, 1997.

[13] Jacques Loeckx and Kurt Sieber. *The Foundations of Program Verification (Second edition)*. John Wiley and Sons, New York, N.Y., 1987.

[14] J S. Moore. Piton: A verified assembly level language. Technical Report 22, Comp. Logic Inc, Austin, Texas, 1988.

[15] J S. Moore. *Piton: A Mechanically Verified Assembly-Level Language*. Kluwer Academic Press, Dordrecht, The Netherlands, 1996.

[16] Markus Müller-Olm. Three Views on Preservation of Partial Correctness. Technical Report Verifix/CAU/5.1, CAU Kiel, October 1996.

[17] Markus Müller-Olm. *Modular Compiler Verification: A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, volume 1283 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, New York, 1997.

[18] A. Pnueli, M. Siegel, and E. Singerman. Translation Validation. In *Proc. 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lisbon, Portugal, March 1998.

[19] Amir Pnueli and Paolo Traverso, editors. *Proceedings of the FLoC'99 International Workshop on "Runtime Result Verification"*, Trento, Italy, 1999.

[20] Ken Thompson. Reflections on Trusting Trust. *Communications of the ACM*, 27(8):761–763, 1984. Also in ACM Turing Award Lectures: The First Twenty Years 1965-1985, ACM Press, 1987, and in Computers Under Attack: Intruders, Worms, and Viruses Copyright, ACM Press 1990.

[21] Niklaus Wirth. *Compilerbau, eine Einführung.* B.G. Teubner, Stuttgart, 1977.

[22] W.D. Young. A verified code generator for a subset of gypsy. Technical Report 33, Comp. Logic. Inc., Austin, Texas, 1988.

A SL Semantics

The semantics (`evaluate`) of an **SL** program is as expected: the top-level expression is evaluated after binding the input variables to some (given) inputs. Functions may not terminate, so the semantics of a program in general is a partial mapping from the inputs to the program result. The ACL2 formalization requires to add a *termination argument*, a natural number `n`, to define `evaluate` as a total ACL2 function making partiality explicit. The functions either return a list containing the value, or `'error`, if evaluation exhausts `n`.

The semantics of a form is defined by the mutually recursive interpreter functions `evl` and `evlist`. It depends on a function environment `genv` mapping function names to parameter lists and a body expression, a local environment `env` mapping free variables to values, and the termination argument `n` which decreases if and only if the body of a user defined function is interpreted. The function `evlop` evaluates operator calls.

```
(defun evlop (op args genv env n)
  (cond
   ((equal op 'CAR) (list (CAR (car args))))
   ((equal op 'CDR) (list (CDR (car args))))
   ((equal op 'CADR) (list (CADR (car args))))
   ((equal op 'CADDR) (list (CADDR (car args))))
   ((equal op 'CADAR) (list (CADAR (car args))))
   ((equal op 'CADDAR) (list (CADDAR (car args))))
   ((equal op 'CADDDR) (list (CADDDR (car args))))
   ((equal op '1-) (list (1- (car args))))
   ((equal op '1+) (list (1+ (car args))))
   ((equal op 'LEN) (list (LEN (car args))))
   ((equal op 'SYMBOLP) (list (SYMBOLP (car args))))
   ((equal op 'CONSP) (list (CONSP (car args))))
   ((equal op 'ATOM) (list (ATOM (car args))))
   ((equal op 'CONS) (list (CONS (car args) (cadr args))))
   ((equal op 'EQUAL) (list (EQUAL (car args) (cadr args))))
   ((equal op 'APPEND) (list (APPEND (car args) (cadr args))))
   ((equal op 'MEMBER) (list (MEMBER (car args) (cadr args))))
   ((equal op 'ASSOC) (list (ASSOC (car args) (cadr args))))
   ((equal op '+) (list (+ (car args) (cadr args))))
   ((equal op '-) (list (- (car args) (cadr args))))
   ((equal op '*) (list (* (car args) (cadr args))))
   ((equal op 'LIST1) (list (LIST1 (car args))))
   ((equal op 'LIST2) (list (LIST2 (car args) (cadr args))))
   ))

(mutual-recursion

(defun evl (form genv env n)
  (cond
   ((zp n) 'error)
   ((equal form 'nil) (list nil))
   ((equal form 't) (list t))
   ((symbolp form) (list (cdr (assoc form env))))
   ((atom form) (list form))
   ((equal (car form) 'QUOTE) (list (cadr form)))
   ((equal (car form) 'IF)
    (let ((cond (evl (cadr form) genv env n)))
      (if (defined cond)
          (if (car cond)
              (evl (caddr form) genv env n)
            (evl (cadddr form) genv env n))
        'error)))
   (t (let ((args (evlist (cdr form) genv env n)))
        (if (defined args)
            (if (operatorp (car form))
                (evlop (car form) args genv env n)
              (evl (caddr (assoc (car form) genv))
                   genv
                   (bind (cadr (assoc (car form) genv)) args env)
                   (1- n)))
          'error)))))

(defun evlist (forms genv env n)
  (cond ((zp n) 'error)
        ((endp forms) nil)
        (t (let ((f (evl (car forms) genv env n))
                 (r (evlist (cdr forms) genv env n)))
             (if (and (defined f) (defined r))
                 (cons (car f) r)
               'error)))))
)

(defun construct-genv (defs)
  (if (consp defs)
      (cons (cons (cadar defs) (cddar defs)) ;; same as (cdar defs)
            (construct-genv (cdr defs)))
    nil))

(defun evaluate (defs vars main inputs n)
  (evl main (construct-genv defs) (bind vars inputs nil) n))
```

The function `evaluate` takes an **SL** program consisting of the declarations `defs`, the input variable list `vars`, the main expression `main`, and returns the value of `main` after binding `vars` to `inputs` and constructing (`construct-genv`) a true association list (`genv`) from `defs` mapping the function names to their extended bodies.

B Machine Semantics

The function `opr` applies operators to the one or two topmost stack cells. For the `stack` we use a list that grows to the left, i.e. we use `cons` to push an item onto the stack, and `nth` or `nthcdr` to read the contents or pop elements. The function `download` downloads the declarations into `code`, which is constructing a true association list from `dcls`. We can not guarantee machine programs to terminate. Consequently we can not guarantee the machine to terminate. So we again add a *termination argument*, a natural number `n`, in order to force the machine to stop execution after at most `n` subroutine calls.

```
(defun opr (op code stack)
  (cond
   ((equal op 'CAR) (cons (MAR (car stack)) (cdr stack)))
   ((equal op 'CDR) (cons (MDR (car stack)) (cdr stack)))
   ((equal op 'CADR) (cons (CADR (car stack)) (cdr stack)))
   ((equal op 'CADDR) (cons (CADDR (car stack)) (cdr stack)))
   ((equal op 'CADAR) (cons (CADAR (car stack)) (cdr stack)))
   ((equal op 'CADDAR) (cons (CADDAR (car stack)) (cdr stack)))
   ((equal op 'CADDDR) (cons (CADDDR (car stack)) (cdr stack)))
   ((equal op '1-) (cons (1- (car stack)) (cdr stack)))
   ((equal op '1+) (cons (1+ (car stack)) (cdr stack)))
   ((equal op 'LEN) (cons (LEN (car stack)) (cdr stack)))
   ((equal op 'SYMBOLP) (cons (SYMBOLP (car stack)) (cdr stack)))
   ((equal op 'CONSP) (cons (CONSP (car stack)) (cdr stack)))
   ((equal op 'ATOM) (cons (ATOM (car stack)) (cdr stack)))
   ((equal op 'CONS) (cons (CONS (cadr stack) (car stack)) (cddr stack)))
   ((equal op 'EQUAL) (cons (EQUAL (cadr stack) (car stack)) (cddr stack)))
   ((equal op 'APPEND) (cons (APPEND (cadr stack) (car stack)) (cddr stack)))
   ((equal op 'MEMBER) (cons (MEMBER (cadr stack) (car stack)) (cddr stack)))
   ((equal op 'ASSOC) (cons (ASSOC (cadr stack) (car stack)) (cddr stack)))
   ((equal op '+) (cons (+ (cadr stack) (car stack)) (cddr stack)))
   ((equal op '-) (cons (- (cadr stack) (car stack)) (cddr stack)))
   ((equal op '*) (cons (* (cadr stack) (car stack)) (cddr stack)))
   ((equal op 'LIST1) (cons (CONS (car stack) nil) (cdr stack)))
   ((equal op 'LIST2) (cons (CONS (cadr stack) (CONS (car stack) nil)) (cddr stack)))
   ))


(mutual-recursion

(defun mstep (form code stack n)
  (cond
   ((or (zp n) (not (true-listp stack))) 'ERROR)
   ((equal (car form) 'PUSHC) (cons (cadr form) stack))
   ((equal (car form) 'PUSHV) (cons (nth (cadr form) stack) stack))
   ((equal (car form) 'CALL)
    (msteps (cdr (assoc (cadr form) code)) code stack (1- n)))
   ((equal (car form) 'OPR) (opr (cadr form) code stack))
   ((equal (car form) 'IF)
    (if (car stack)
        (msteps (cadr form) code (cdr stack) n)
      (msteps (caddr form) code (cdr stack) n)))
   ((equal (car form) 'POP) (cons (car stack) (nthcdr (cadr form) (cdr stack))))))

(defun msteps (seq code stack n)
  (cond ((or (zp n) (not (true-listp stack))) 'ERROR)
        ((endp seq) stack)
        (t (msteps (cdr seq) code (mstep (car seq) code stack n) n))))
)

(defun download (dcls)
  (if (consp dcls)
      (cons (cons (cadar dcls) (caddar dcls))
            (download (cdr dcls)))
    nil))

(defun execute (prog stack n)
  (let ((code (download (butlst prog))))
    (msteps (car (last prog)) code stack n)))
```

C The Compiler Program from SL to TL

The function operatorp identifies operators. The two mutually recursive functions compile-form and compile-forms compile expressions and expression lists, respectively. Compile-forms iterates compile-form over forms, thereby incrementing the number top of used stack cells. The function compile-def generates a **TL** subroutine, and compile-defs maps compile-def over defs. Finally, compile-program compiles the function definitions and appends them to the compiled main program expression, which additionally pops the input values off the stack. The compiler **SL** program is:

```
(((defun operatorp (name)
    (member name '(car cdr cadr caddr cadar caddar cadddr
                  1- 1+ len symbolp consp atom cons equal
                  append member assoc + - * list1 list2)))

  (defun compile-forms (forms env top)
    (if (consp forms)
        (append (compile-form (car forms) env top)
                (compile-forms (cdr forms) env (1+ top)))
      nil))

  (defun compile-form (form env top)
    (if (equal form 'nil) (list1 '(PUSHC NIL))
    (if (equal form 't) (list1 '(PUSHC T))
    (if (symbolp form)
        (list1 (list2 'PUSHV (+ top (1- (len (member form env))))))
    (if (atom form) (list1 (list2 'PUSHC form))
    (if (equal (car form) 'QUOTE) (list1 (list2 'PUSHC (cadr form)))
    (if (equal (car form) 'IF)
        (append (compile-form (cadr form) env top)
          (list1 (cons 'IF
                  (list2 (compile-form (caddr form) env top)
                         (compile-form (cadddr form) env top)))))
    (if (operatorp (car form))
        (append (compile-forms (cdr form) env top)
                (list1 (list2 'OPR (car form))))
        (append (compile-forms (cdr form) env top)
                (list1 (list2 'CALL (car form))))))))))))

  (defun compile-def (def)
    (list1 (cons 'defcode
            (list2 (cadr def)
              (append (compile-form (cadddr def) (caddr def) 0)
                (list1 (list2 'POP (len (caddr def)))))))))

  (defun compile-defs (defs)
    (if (consp defs)
        (append (compile-def (car defs))
             (compile-defs (cdr defs)))
    nil))

(defun compile-program (defs vars main)
  (append (compile-defs defs)
          (list1 (append (compile-form main vars 0)
                         (list1 (list2 'POP (len vars)))))))
(defs vars main)
(compile-program defs vars main))
```

Refinement Tool in the Synthesis of Asynchronous Circuits

Rimvydas Rukšėnas[1,2]

[1] Turku Centre for Computer Science (TUCS)
[2] Dept. of Computer Science, Åbo Akademi University

1 Motivation and background

Theorem provers have been extensively used in the context of hardware verification especially when verifying properties of synchronous circuits. However, asynchronous techniques are gaining increasing attention, also in an industrial setting. A delay-insensitive circuit is an asynchronous device whose operation is based on local communication events, called *handshakes*, between different parts of a circuit.

A design method for delay-insensitive circuits based on the action system and the refinement calculus formalisms was introduced recently [1]. We explore possibilities to use the Refinement Calculator as a tool performing transformations and required proofs within this framework.

The Refinement Calculator [2] is an user-friendly environment for program development using the refinement calculus. It uses the HOL theorem-proving system as an underlying inference engine and produces as the result of a refinement step a theorem stating that the refinement in question holds. In particular, the Refinement Calculator contains a package for generic data refinement transformations [2] based on the calculational approach.

The action systems formalism is based on Dijkstra's language of *guarded commands*. An action system is essentialy a collection of actions; one of the enabled actions is nondeterministally selected and executed. Several action systems can be composed to form a larger system using the parallel composition operator. The refinement relation used in stepwise derivations of circuits is trace refinement. We rely on the fact that data refinement implies trace refinement of action systems.

2 Handshake circuits in the Refinement Calculator

In the action system approach to the synthesis of delay-insensitive circuits, communication between components is first described using channels. Each channel is modelled as a boolean variable, called a channel variable. Then *handshaking expansion* which is the most central transformation in this approach introduces *request* and *acknowledgment* variables, called handshaking variables, needed in the delay-insensitive interaction. In the the Refinement Calculator, the transformation is naturally modelled as data refinement. The variables replaced represent channels between communicating systems, thus they are not local to a particular system. Therefore, there are, essentially, two options to perform handshaking expansion. The first one is to treat a parallel composition of action systems as one system with all channel variables as its local variables. However in such setting, the size of the objects involved (action systems and the abstraction relation) makes it very difficult to handle refinement proofs in the Refinement Calculator. Instead, we have chosen another option which is to consider the refinement of each system separately.

Refinement in context We consider a system, called a *component*, and its environment as a parallel composition of two action systems Since the variables introduced while the component in question is refined are shared by a system and its environment, such compositional approach requires to take into account context of the system. This is done in standard way [3] by introducing two predicates, say I and J, which represent a pair of rely and guarantee conditions, respectively. The predicate J describes state changes that the system $\mathcal{C}$ guarantees to obey, denoted by $\langle I \rangle\ \mathcal{C}\ \langle J \rangle$, provided its environment obeys state changes specified by the predicate I. Then refinement in a parallel context is defined in the following way.

Definition 1. Let $\mathcal{A}$ and $\mathcal{C}$ be action systems and let R be an abstraction relation. Assume that I and J is a rely-guarantee pair. Then the system $\mathcal{C}$ refines the system $\mathcal{A}$ under the relation R in any parallel context that obeys I, denoted by $\langle I \rangle\ \mathcal{A} \preceq_R \mathcal{C}\ \langle J \rangle$, if:

(i) Local data refinement under the relation R: $\mathcal{A} \preceq_R \mathcal{C}$
(ii) Abstraction relation: $R \ \wedge\ I \ \wedge\ (a' = a) \ \wedge\ (c' = c) \ \Rightarrow\ R'$
(iii) Guarantee condition: $\langle I \rangle\ \mathcal{C}\ \langle J \rangle$

where a and c are the local variables of the systems $\mathcal{A}$ and $\mathcal{C}$, respectively, while R' is the abstraction relation R with all variables primed.

The second condition in the definition ensures that the abstraction relation R is preserved by the environment.

Rely-guarantee pair In general, rely (guarantee) predicate can be any binary state predicate that desribes state changes allowed (obeyed) by a component. In our case, rely-guarantee conditions are used to describe handshake communication between components and, therefore, they are very concrete and uniform. Let a be a channel variable that is replaced by two concrete variables, req_a and ack_a, in the handshaking expansion. Then the rely and guarantee conditions associated with these variables are as follows:

$$\begin{aligned} J_{ack}(a) &\ \hat{=}\ ack_a \neq ack'_a \ \Rightarrow\ ack_a \neq req_a \\ J_{req}(a) &\ \hat{=}\ req_a \neq req'_a \ \Rightarrow\ ack_a = req_a \end{aligned}$$

Composition of handshake components Essentially, handshaking expansion can be applied to any system that communicates with its environment via channel variables according to certain rules. As an example, let us consider a system $\mathcal{A}$ with a channel variable, say a. Assume, that the system $\mathcal{A}$ initiates a communication by setting a to *true*, while its environment is supposed to acknowledge the communication by setting a to *false*. Assume that the system $\mathcal{A}^x$ is a handshaking expansion of $\mathcal{A}$. Then the guarantee condition for the system $\mathcal{A}^x$ is written as follows: $\langle J_{ack}(a)\rangle\ \mathcal{A}^x\ \langle J_{req}(a)\rangle$. Later on we show how proofs of such guarantee conditions are handled in the Refinement Calculator.

If there are more channels in the component $\mathcal{A}^x$ the above guarantee condition is to be extended by adding an appropriate conjunct ($J_{ack}(b)$ or $J_{req}(b)$) to the rely or guarantee predicate for every channel b. We call such systems $\mathcal{A}$ and $\mathcal{A}^x$ a handshake component and an expanded handshake component, respectively.

Now consider a circuit $\mathcal{A}$ described as a parallel composition of handshake components $\mathcal{A}_1, \ldots, \mathcal{A}_n$. The handshaking expansion as a refinement of the entire system $\mathcal{A}$ is established using the following theorem.

Theorem 2. *Let $\mathcal{A}$ be a parallel composition of handshake components $\mathcal{A}_1, \ldots, \mathcal{A}_n$. Assume that $\mathcal{A}_1^x, \ldots, \mathcal{A}_n^x$ are the corresponding expanded handshake components. Let C_{env} be a set of channels between the system $\mathcal{A}$ and its environment. Then the following refinement holds:*

$$\langle I\rangle\ \mathcal{A} \preceq_R \mathcal{A}^x\ \langle J\rangle$$

where $\mathcal{A}^x$ is a parallel composition of $\mathcal{A}_1^x, \ldots, \mathcal{A}_n^x$ and R is a conjunction of the corresponding abstraction relations. Also, $J = \wedge_{c \in C_{env}} J(c)$, $\overline{J} = \wedge_{c \in C_{env}} \overline{J}(c)$, and each $J(c)$ is either $J_{req}(c)$ or $J_{ack}(c)$ depending on which variable is assigned in $\mathcal{A}$ while $\overline{J}(c)$ is its complement. [1]

The proof of the theorem relies on the theorem for composing refinements in context [3]. It also exploits the specific form of the rely-guarantee predicates.

Such compositional approach allows to have a library of basic handshake components such as sequencer, parallelizer, mixer, synchronizer, iteration composer, etc. with the corresponding refinement theorems and use them for the synthesis of circuits. However, new handshake components can always be introduced whenewer needed. This only requires to prove the handshaking expansion (in context) for the individual component in question to ensure refinements of systems with such components later on.

3 Verification

For a number of reasons we model action systems as loop statements in the guarded commmand language. As an example, a handshake component, called a mixer, is represented in the Refinement Calculator as the following program:

[1] The complement of $J_{req}(c)$ is $J_{ack}(c)$, and vice versa.

```
program $mix2$ var $\cdot$
    |[ var $a_1, a_2, b : bool; pc : num \mid \neg a_1 \;\wedge\; \neg a_2 \;\wedge\; \neg b \;\wedge\; pc = 0 \cdot$
    do
     # $pc = 0 \;\wedge\; a_1 \rightarrow b, pc := true, 1$
     # $pc = 1 \;\wedge\; \neg b \rightarrow a_1, pc := false, 0$
     # $pc = 0 \;\wedge\; a_2 \rightarrow b, pc := true, 2$
     # $pc = 2 \;\wedge\; \neg b \rightarrow a_2, pc := false, 0$
    od ]|
```

where a_1, a_2 and b are channel variables while pc is a program counter. We briefly discuss the verification of the three conditions given in the definition 1 which are needed to establish refinement in a parallel context.

Local refinement Here the system is considered in izolation. To simplify calculation of concrete systems, the handshaking expansion as a refinement is proven in several interactive steps each of them dealing with the different aspects of the transformation. In each of those steps a new system is calculated from the old one and the abstraction relation using the following rule (or its modifications) provided by the data refinement package in the Refinement Calculator:

$$\begin{aligned}&\mathcal{D}_R(\textbf{do } A \textbf{ od}) = \\ &\quad \textbf{do } \mathcal{D}_R(A) \# h \rightarrow c := c' \bullet t[c := c'] < t \;\wedge\; (\forall a \cdot R \Rightarrow R[c := c']) \textbf{ od}\end{aligned}$$

where c are newly introduced variables and R is the abstraction relation. The rule allows to add new actions, called stuttering actions, to the system which is essential in the handshaking expansion. In the rule, the second action in the refining statement is a stuttering one, while t is an expression ranging over the natural numbers; it guarantees that the stuttering action cannot be repeated infinitely. The relation R, the guard h and the termination function t are to be given by the user.

In the first refinement step, a stuttering action is added for each of the handshaking variables. Adding new actions really means data refinement of the program counter pc. Therefore, an abstraction relation, say PC, which proves the data refinement has to specify a relationship between the abstract and concrete program counters. The relation PC is then used by the above rule for the calculation of a new system. Usually, the calculated system requires further refinements (simplifications) performed interactively.

In the second step, the handshaking variables are introduced. The relationship between a channel, say c, and the corresponding handshaking variables, req_c and ack_c, is defined as the following handshaking relation H_c:

$$H_c \;\hat{=}\; c = req_c \wedge \neg ack_c$$

The abstraction relation R_c also includes invariant CI_c: $H_c \wedge CI_c$. The invariant describes how the handshaking variables are related to the program counter of the refined system. The calculated system normally requires simplifications. The handshaking variables are introduced in a separate step for each channel.

Abstraction relation The result of the refinements described above is an expanded handshake component which is a data refinement of the original component under the abstraction relation R: $PC \wedge H \wedge CI$. Here H and CI are conjunctions (for all channels c) of H_c and CI_c, respectively. The second condition in the definition 1 requires to show that the relation R is preserved by any environment that obeys state changes specified by the rely condition of the component. For the moment being we have not developed special support for such verification, since it can be performed using existing libraries of the HOL system (simplifier, decision procedures, etc.).

Guarantee conditions Due to their specific form, proving guarantee conditions amounts to establishing certain total correctness formulas. Thus, to establish the guarantee condition $\langle J_{ack}(a)\rangle\ \mathcal{A}_x\ \langle J_{req}(a)\rangle$ we have to show that the following two total correctness assertions hold for all actions A in the system $\mathcal{A}^x$:

$$CI_a \wedge gA \wedge req_a \wedge \neg ack_a\ \langle sA\rangle\ req_a$$
$$CI_a \wedge gA \wedge \neg req_a \wedge ack_a\ \langle sA\rangle\ \neg req_a$$

Here gA and sA are the guard and the body of the action A. The Refinement Calculator contains the package for establishing correctness assertions which, in the case of handshake components, usually is able to do this automatically.

4 Concluding remarks

For the moment being we have concentrated on the verification of refinements of individual handshake components in the Refinement Calculator. Combining within the Refinement Calculator results of these verifications to derive refinement theorems corresponding to the composition theorem (Theorem 2) remains to be dealt with in the future. In particular, this would require to treat action systems in the Refinement Calculator in a proper way instead of just modelling them as ordinary programs in the guarded command language. Such treatment is necessary to be able to define a notion of the parallel composition of action systems. Also, the handshaking refinements and the related proofs are to be incorporated into the user interface provided by the Refinement Calculator.

References

1. J. Plosila, R. Rukšėnas, and K. Sere. Synthesis of delay-insensitive circuits. In J. Grundy, M. Schwenke, and T. Vickers, editors, *International Refinement Workshop & Formal Methods Pacific'98*, pages 286–305. Springer-Verlag, 1998.
2. R. Rukšėnas and J. von Wright. A tool for data refinement. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics: 11th International Conference*, volume 1479 of *LNCS*, pages 423–441. Springer-Verlag, 1998.
3. Q. Xu. On compositionality in refining concurrent systems. In J. He, J. Cooke, and P. Wallis editors, *Proceedings of the BCS FACS 7th Refinement Workshop, Bath UK, 1996.* Springer-Verlag, 1996.

MOPS: Verifying Modula-2 programs specified in VDM-SL

Thomas Kaiser[1] Bernd Fischer[2] Werner Struckmann[3]

Introduction

Almost all computer programs contain errors, at least initially. The traditional approach to discover these errors is testing. However, since testing can only be used to show the presence of errors but not their *absence*, other approaches as *program verification* are pursuit. It is an exact, formal method to prove for all possible inputs the consistency between the specification of a program and its implementation. A *verification system* automates parts of the verification task. The architecture of verification systems usually comprises two different tiers, a predicate transformer or *verification condition generator*, and a prover. The verification condition generator takes the program and the specification and computes a set of logical expressions called *proof obligations*. These are then *discharged*, either automatically, by the prover, or manually, by the software engineer. If all obligations are discharged the program is proven correct with respect to the specification (assuming that the underlying calculus is sound). However, the failure to discharge an obligation does not always mean that the program contains an error. It may also indicate that the specification is incomplete or not adequate, or that the prover is too weak. The reason for the two-tiered architecture is purely pragmatic. Any specification language which is expressive enough to capture "interesting" requirements (and thus to describe "interesting" programs) is undecidable. Hence, any prover is too weak for a fully automatic system. In contrast to that, the generation of verification conditions is decidable and a fully automatic verification condition generator can be implemented, even for real programming languages.

The *Modula Proving System* (MOPS) is a Hoare-calculus based program verification system for a large subset of the programming language Modula-2 which uses VDM-SL [6] as specification language. The main goal of MOPS is to demonstrate the feasibility and viability of a Hoare-style verification system for a real imperative programming language, including pointers, arrays, and other data structures. MOPS also provides support for the modular and partial verification of large systems and includes hooks for specification-based code reuse systems as for example NORA/HAMMR [3]. Finally, MOPS demonstrates the combination of a verification system with an established specification language which exists outside the verification system itself.

MOPS is built according to the two-tiered architecture outlined above and comprises a weakest precondition predicate transformer and a rather weak rewrite-based prover; however, stronger off-the-shelf provers can be incorporated relatively easy. The predicate transformer used in MOPS supports only proofs of partial correctness, i.e., reasoning about termination cannot be done within MOPS. However, this allows us to use a simpler calculus and also yield simpler proof obligations.

[1] dvg, Postfach 72 11 07, D-30539 Hannover
[2] RIACS/NASA Ames Research Center, fisch@ptolemy.arc.nasa.gov
[3] Technische Universität, Institut für Software, Abteilung Programmierung, D-38092 Braunschweig

MOPS essentially follows the more traditional approach to verify programs after the implementation is completed instead of developing proof and program hand-in-hand, as for example advocated by the KIV-system [11]. However, we believe that the traditional approach is better suited for the incremental or even partial verification of large systems as the users can easily restrict the verification to the critical parts of a system.

The current version of MOPS supports almost the entire Modula-2 programming language as defined in [12], including pointers and data structures. The only language constructs not yet supported are variant record types, procedure types, and procedures as parameters, i.e., higher-order procedures cannot be verified. The verification of REAL-arithmetics is idealized and ignores possible rounding errors. Modula-2 also relies heavily on the use of standard libraries, e.g., for input/output, systems programming, and parallel programming. MOPS does not provide specific support for these modules but programs built on top of them can be verified as usual (except for input/output) after these modules have been re-specified using the modular verification techniques described below.

Calculus

MOPS is built upon the Hoare-calculus. The theoretical foundations and the fundamental verification algorithms based on this calculus can be found in, e.g., [1, 2, 5]. We extended these foundations into a calculus for the programming language Modula-2 by adding further proof rules and extending the underlying logic. Adding new statements to the language means adding new proof rules to the calculus. This is relatively straightforward and as long as the new rules are sound and the statements are disjoint from the core, the extended calculus remains obviously sound. Adding data types, however, extends the underlying logic and can easily compromise its soundness.

The starting point for the verification of arrays, records and pointers has been the proof system given in [10]. For MOPS, this system was extended to support explicit memory deallocation via the `DISPOSE`-procedure in the Modula-2 system module. Obviously, pointers introduce the same *aliasing problem* as arrays, i.e., a memory location can be addressed by different names. The main idea in [10] is to treat all pointers of a particular type as a single dynamic array and thus to handle pointer aliasing with the same mechanism as array aliasing. This approach, however, critically relies on Modula-2's pointer discipline which guarantees that two pointers refer to the same memory location only if one of them has—directly or indirectly—been assigned to the other. It can thus not be applied to languages as C which allow pointer arithmetics. The complete axioms and proof rules for this approach are given in [7].

Hoare-style calculi are usually defined over the classical, two-valued predicate calculus. This implies that expressions are always assumed to be defined which in turn requires all semantic functions to be total. Since MOPS uses VDM-SL as specification language, it is natural to base the calculus on the logic LPF (Logic of Partial Functions) underlying VDM-SL. This does not affect the verification condition generator; however, the proof obligations are now LPF-formulae. Semantically, this provides an encapsulation of all partiality reasoning within the proof theory for LPF or an off-the-shelf translation from LPF to the classical predicate calculus. Moreover, partial correctness becomes a stronger result than in the classical case as it implies the absence of run-time errors caused by application of partial functions to arguments outside their domain, e.g., division by zero.

Intuitively, our calculus should be sound and relatively complete with respect to LPF; we expect the formal proofs to be straightforward adaptations from the classical proofs in the literature. Obviously, however, the calculus is not relatively complete with respect to the classical predicate logic.

Specification and Verification

MOPS supports the verification of arbitrary program segments and not only, e.g., procedures or modules. This precludes considering the implementation as the final refinement of a specification module as for example in KIV but requires a direct embedding of the VDM-SL specification into the Modula-2 code. Syntactically, this is achieved by enclosing the VDM-SL expressions within formal comments `(*{` and `}*)` such that the annotated program can still be compiled and executed by any Modula-2 compiler. MOPS thus assumes the syntactic correctness of the Modula-2 program. Since the VDM-SL specification can be extracted from the annotated program automatically and shown consistent using external tools, MOPS also assumes the syntactic correctness and internal consistency of the VDM-SL specification. Such embedding approaches date back at least to the ANNA-system [9] and have also been used in the specification languages in the Larch-tradition, e.g., in the Penelope-system [4].

MOPS uses `entry`/`exit`-tags as shown below to mark the verification segments; these can be nested to break large proofs into manageable pieces. Loop invariants, which must be provided as usual in Hoare-style calculi, and additional `assert`-tags are used to aid the proof construction. Joint scoping allows the specification to refer to program variables but not vice versa.

```
...
(*{ entry sum_loop
     pre  sum  = 0
     post sum' = n * (n+1) div 2      }*)
(*{ loopinv sum = ((i - 1) * i) div 2 }*)
FOR i := 1 TO n DO
    sum := sum + i;
END;
(*{ exit sum_loop }*)
...
```

Verification segments also provide convenient hooks for specification-based retrieval as the `pre`/`post`-pair already comprises the crucial part of a retrieval query. By changing the `entry`-tag into the VDM-SL operation signature `sum_loop(n:int) ext rw sum:int` a retrieval system as NORA/HAMMR [3] (which also uses VDM-SL as specification language) can extract a full query and search a library for semantically matching, verified components. This allows a smooth integration of reuse without compromising program correctness, thus reducing the overall verification effort.

The main problem of embedding an existing specification language into a verification system (as opposed to defining a specialized behavioral interface specification language) is to define a suitable between the constructs of the implementation and specification languages. Fortunately, VDM-SL's meta-language heritage makes this task easier and most constructs (e.g., base types) map rather straightforward. Procedures, however, are slightly more complicated. A Modula-2 `PROCEDURE` with a return value and call-by-value-parameters only but without side effects can be specified via a VDM-SL *function.* Procedures with side effects are specified by *operations*; all global variables of a Modula-2 module are automatically mapped on a single *state.* Call-by-reference parameters have no direct correspondence in VDM-SL; they require generating a (local) state.

Large systems are inevitably split into several separate modules and MOPS supports the verification of such modular systems. Procedure specifications can be separated from their corresponding implementations by including them into the definition modules only. The implementations are then verified against their definitions. Client modules which import a specified procedure automatically import the associated function specification and thus need to verify only the particular call. Thus, the verification can be modularized. Figure 1 illustrates this concept.

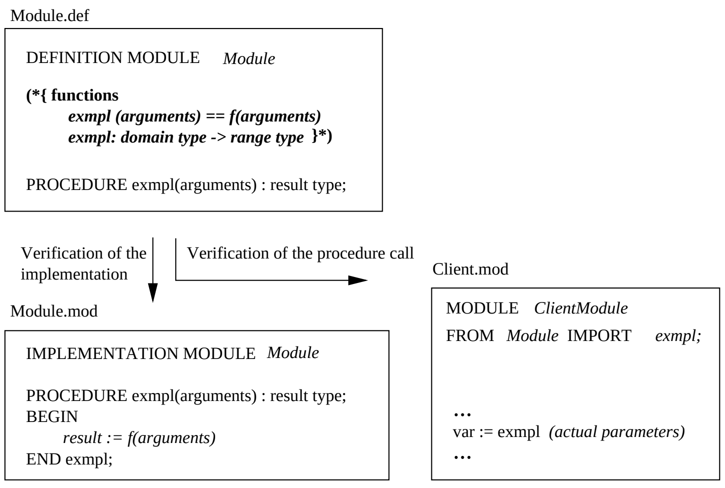

Module.def
DEFINITION MODULE Module
(*{ functions
exmpl (arguments) == f(arguments)
exmpl: domain type -> range type }*)
PROCEDURE exmpl(arguments) : result type;
Verification of the implementation
Verification of the procedure call
Client.mod
Module.mod
IMPLEMENTATION MODULE Module
PROCEDURE exmpl(arguments) : result type;
BEGIN
result := f(arguments)
END exmpl;
MODULE ClientModule
FROM Module IMPORT exmpl;
...
var := exmpl (actual parameters)
...

Figure 1: *Modular Verification*

If a procedure contains no call-by-reference parameters, its specification can be separated entirely from the Modula-2 declaration, even beyond the file boundary of the definition module, and moved into a completely seperated specification file containing a pure VDM-SL module. The correspondence of these files is guaranteed by extending the Modula-2 naming conventions (see figure 2). This allows a subsequent specification of existing modules, e.g., standard library modules, without any changes to the definition modules. This is required for the timestamp-based module consistency mechanism employed by most Modula-2 compilers.

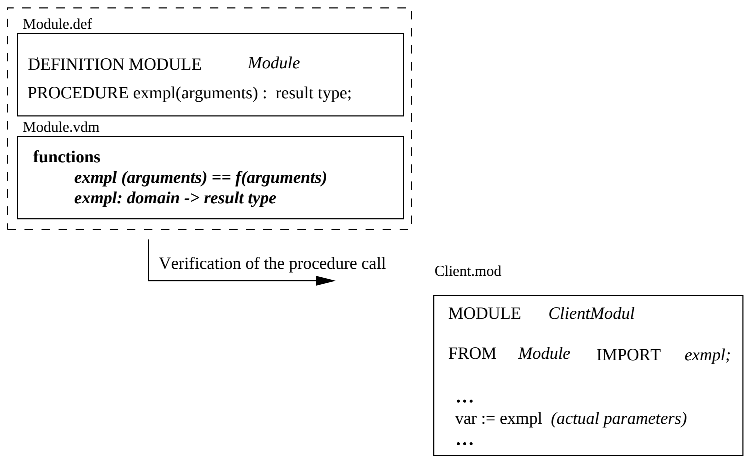

Module.def
DEFINITION MODULE Module
PROCEDURE exmpl(arguments) : result type;
Module.vdm
functions
exmpl (arguments) == f(arguments)
exmpl: domain -> result type
Verification of the procedure call
Client.mod
MODULE ClientModul
FROM Module IMPORT exmpl;
...
var := exmpl (actual parameters)
...

Figure 2: *Subsequent specification*

In MOPS, a Modula-2 client module can import arbitrary objects from arbitrary other modules. In particular, it can also access symbols from pure VDM-SL modules which are not associated with any definition or implementation modules. Hence, VDM-SL can be used as shared language to define theories supporting the verification.

Experiences and Conclusions

The MOPS-system is implemented in the functional programming language SML. We have tested it successfully on small and mid-size programs, including the usual sorting examples. [7, 8] contain a series of increasingly sophisticated variants of the quicksort-algorithm, including the median-of-three pivot selection strategy and the use of selection sort and bubblesort for small subarrays. The quicksort-implementations work on open arrays of element-records and sort by one of the record components. The base variant consists of more than 300 lines of Modula-2 code and VDM-SL specification. MOPS generates 23 proof obligations and discharges 14 by plain rewriting. By encapsulation of the variation into separate verification segments, the number of emerging proof obligations for the variants can generally be kept small; however, MOPS does not provide any proof management. Currently, we work on the specification and verification of the well known LZW compression and decompression algorithms [13].

MOPS has deliberately been designed as a "small tool". It combines established techniques as Hoare-style reasoning and specification-based reuse with established implementation and specification languages as Modula-2 and VDM-SL. This conceptual simplicity is—in our opinion—a major contribution of MOPS and makes it also suitable for educational purposes. Future work on MOPS includes the combination with fully automated theorem provers and the migration from the programming language Modula-2 to Java.

References

[1] K. R. Apt. Ten years of Hoare's logic: A survey—part I. *ACM Trans. on Prog. Lang. and Systems*, 3:431–483, 1981.

[2] K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer, New York, 1991.

[3] B. Fischer, J. M. Ph. Schumann, and G. Snelting. Deduction-based software component retrieval. In *Automated Deduction - A Basis for Applications*, pages 265–292, Dordrecht, 1998. Kluwer.

[4] D. Guaspari, C. Marceau, and W. Polak. Formal verification of Ada. *IEEE Trans. Software Engineering*, 16(9):1058–1075, 1990.

[5] B. Hohlfeld and W. Struckmann. *Einführung in die Programmverifikation.* BI-Wissenschaftsverlag, Mannheim, 1992.

[6] C. B. Jones. *Systematic Software Development using VDM.* Prentice Hall International Series in Computer Science. Prentice Hall, New York, 2nd edition, 1990.

[7] Th. Kaiser. Behandlung von Datenstrukturen in einem VDM-basierten Prädikatentransformer für Modula-2, September 1998. Diplomarbeit, Technische Universität Braunschweig.

[8] Th. Kaiser, B. Fischer, and W. Struckmann. The Modula-2 Proving System MOPS. Informatik-Bericht Nr. 2000-01, Technische Universität Braunschweig, Mai 2000.

[9] D. Luckham, F. W. von Henke, B. Krieg-Brückner, and O. Owe. *ANNA - A Language for Annotating Ada Programs*, volume 260 of *Lect. Notes Comp. Sci.* Springer, 1987.

[10] D. C. Luckham and N. Suzuki. Verification of Array, Record and Pointer Operations in PASCAL. *ACM Trans. on Prog. Lang. and Systems*, 1(2):226–244, 1979.

[11] W. Reif. Formale Methoden für sicherheitskritische Software – Der KIV-Ansatz. *Informatik Forsch. Entw.*, 14(4):193–202, 1999.

[12] N. Wirth. *Programmieren in Modula-2.* Springer, Berlin, 1991.

[13] J. Ziv and A. Lempel. A Universal Algorithm for Data Compression. *IEEE Trans. Information Theory*, 23:337–343, 1977.

A Sound Integration of Some UML Structural and Behavioural Diagrams Using Rewriting Logic

Nasreddine Aoumeur   Gunter Saake
Institut für Technische Informationssysteme
Otto-von-Guericke-Universität Magdeburg
Postfach 4120, D–39016 Magdeburg
E-mail: {aoumeur|saake}@iti.cs.uni-magdeburg.de

1 Extended abstract

Among the main features that make UML [BJR97]widely accepted is the sufficient number of diagrams allowing it for covering all possible facets in developing (particularly modelling) complex systems. However, one of the price of this generality is the lack of a complete *coherence* of different diagrams. This induce in particular duplication, incompatibility, and especially absence of a uniform view of whole modelled system; which imply difficulties of verifying its properties, its refinement, etc. On the other hand, most of semantics proposed for different diagrams ignore the (full-) distribution and concurrent nature of most of present-day complex (information) systems. Moreover, real-word applications—following our experience with distributed complex information systems [AS00, Aou00, AS99b]— often necessitate only some of these diagrams; in particular: (1) object and class diagrams for structural aspects; (2) state-chart and life-cycle diagrams with component diagrams for dynamic and behavioural features.

Starting from these motivations, the present work propose a coherent and sound view of above diagrams. More precisely, our approach for soundly unifying object- and class-diagram, state-chart and component diagrams may be summarized in the following:

- Our first point are the class- (and object-) diagrams. With each attribute we associate a corresponding variable(s) which will stands for its current values. In the same sense, we also enrich each method invocation parameter a corresponding variable (in addition to extra-parameter variables to object identifiers). Second, using a very appealing Petri-net-like notations, we depict how such methods act on attributes. With these user-friendly notations we capture the dynamic of each method, which in usual partly represented in the state-chart model and partly encoded in programming phase. Third, we perceive each association as an external or observed (active or passive) method, and using the same notations we specify their dynamic (only for association regarded to be active). From the two constructions (internal and observed methods), we may deduce which attributes in each class have to be declared as internal or observed. This step is illustrated using a simplified account specification as depicted in Figures 1 and 2.

- For soundly interpreting these constructions, we propose to use an adequate extension of the rewriting logic based MAUDE language [Mes92, Mes93], we proposed in [AS99a]. This interpretation not only captures straightforwardly these constructions, but also allow for generating rapid-prototypes with full exhibition of intra- as well as inter-object concurrency and explicit distinction between internal class dynamic and interaction between different classes.

- For life-cycle specification, we propose to capture state-chart behaviour using reflection in rewriting logic [CM96](and MAUDE in particular). Indeed, a state-chart model allows globally to capture different 'processes' (i.e. sequence, choice and parallelization of different method applications under appropriate conditions) to be respected by object behaviour. But this is exactly what the reflection in rewrite logic may intrinsically cover.

- other advantages of our approach is the natural way for covering real-time constraints on the basis of timed rewriting logic [KW97].

This work is supported by prototypes that is still under development. The tool is implemented using C++ programming language, and it covers the first step and partially the second one. However, apart from the first step, current implementation of the MAUDE language [CDE+99] may be easily adapted for this purpose.

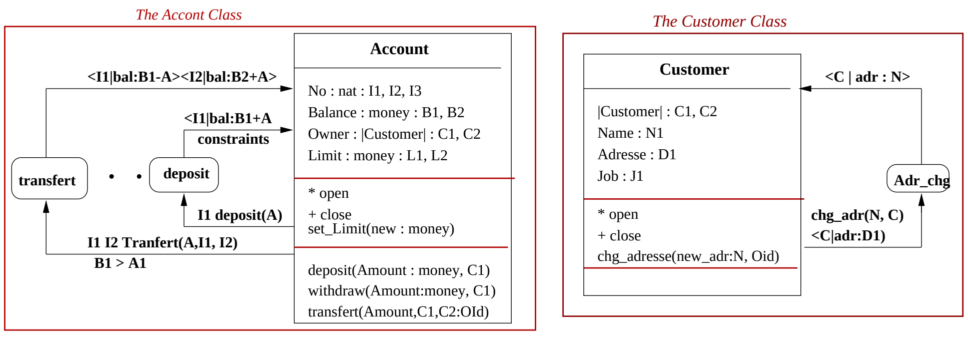


Figure 1: a Sketch of the Account and Customer dynamic specification

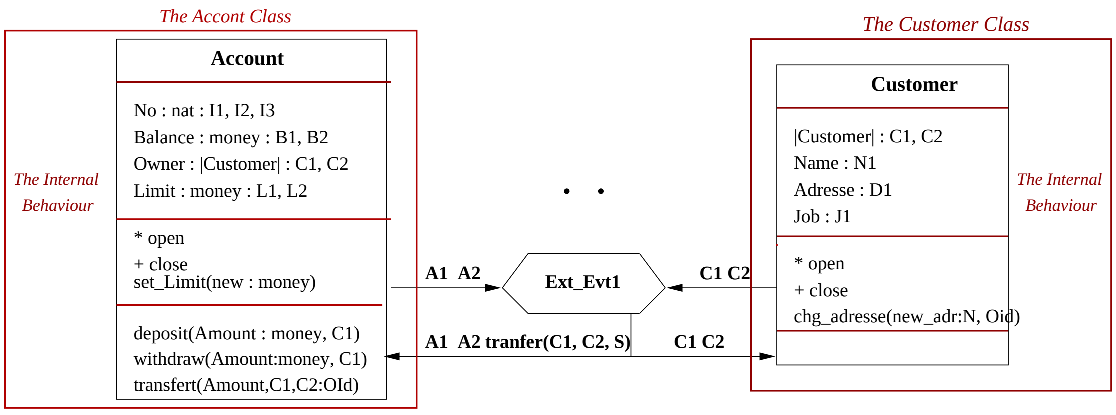


Figure 2: interaction between different components in the account specification

References

[Aou00] N. Aoumeur. Specifying Distributed and Dynamically Evolving Information Systems Using an Extended CO-Nets Approach. In G. Saake, K. Schwarz, and C Türker, editors, *Transactions and Database Dynamics*, volume 1773 of *Lecture Notes in Computer Science, Berlin*, pages 91–111. Springer-Verlag, 2000. *Selected papers form the 8th International Workshop on Foundations of Models and Languages for Data and Objects, sep. 1999, Germany.*

[AS99a] N. Aoumeur and G. Saake. On the Specification and Validation of Cooperative Information Systems Using an Extended MAUDE. In K. Futatsugi, J. Goguen, and J. Meseguer, editors, *Proc. of 1st Int. OBJ/CafeOBJ/Maude Workshop, at FM'99 Conference, Toulouse, France*, pages 197–211. The Theta Foundation Bucharest, Romania, 1999.

[AS99b] N. Aoumeur and G. Saake. Towards an Object Petri Nets Model for Specifying and Validating Distributed Information Systems. In M. Jarke and A. Oberweis, editors, *Proc. of the 11th Int. Conf. on Advanced Information Systems Engineering, CAiSE'99*, volume 1626 of *Lecture Notes in Computer Science*, pages 381–395. Springer-Verlag, 1999.

[AS00] N. Aoumeur and G. Saake. Cooperative Information Systems Modelling and Validation Using the CO-NETS Approach: The Chessmen Making Shop Case Study. In S. Gnesi, I. Schieferdecker, and A. Rennoch, editors, *Proc. of the 5th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'00), Berlin, Germany*, pages 361–383, 2000.

[BJR97] G. Booch, I. Jacobson, and J. Rumbaugh, editors. *Unified Modeling Language, Notation Guide, Version 1.0.* 1997.

[CDE+99] M. Clavel, F. Duran, S. Eker, J. Meseguer, and M. Stehr. Maude : Specification and Programming in Rewriting Logic. Technical report, SRI, Computer Science Laboratory, March 1999. URL : http://maude.csl.sri.com.

[CM96] M. Clavel and J. Meseguer. Reflection and Strategies in rewriting logic. In G. Kiczales, editor, *Proc. of Reflection'96*, pages 263–288. Xerox PARC, 1996.

[KW97] P. Kosiuczenko and Wirsing. Timed Rewriting Logic with an Application to Object-Based Specification. *Science of Computer Programming*, 28:225–246, 1997.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model for concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[Mes93] J. Meseguer. A Logical Theory of Concurrent Objects and its Realization in the Maude Language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Object-Based Concurrency*, pages 314–390. The MIT Press, 1993.

The KeY Approach: Integrating Object-oriented Design and Formal Verification[*]

Wolfgang Ahrendt[†] Thomas Baar[†] Bernhard Beckert[†] Martin Giese[†]
Elmar Habermalz[†] Reiner Hähnle[‡] Wolfram Menzel[†] Peter H. Schmitt[†]

[†] Universität Karlsruhe
Inst. f. Logik, Komplexität und Dedukt.-Syst.
D-76128 Karlsruhe, Germany

[‡] Chalmers University of Technology
Department of Computing Science
S-41296 Gothenburg, Sweden

i12www.ira.uka.de/˜key

Abstract

This paper reports on the ongoing KeY project aimed at bridging the gap between (a) object-oriented software engineering methods and tools and (b) deductive verification. A distinctive feature of our approach is the use of a commercial CASE tool enhanced with functionality for formal specification and deductive verification.

1 Introduction

1.1 Analysis of the Current Situation

While formal methods are by now well established in hardware and system design, usage of formal methods in software development is still (and in spite of exceptions [8], [9]) more or less confined to academic research. This is true though case studies clearly demonstrate that computer-aided specification and verification of realistic software is feasible [14]. The real problem lies in the excessive demand imposed by current tools on the skills of prospective users:

1. Tools for formal software specification and verification are not integrated into industrial software engineering processes.
2. User interfaces of verification tools are not ergonomic: they are complex, idiosyncratic, and are often without graphical support.
3. Users of verification tools are expected to know syntax and semantics of one or more complex formal languages. Typically, at least a tactical programming language and a logical language are involved. And even worse, to make serious use of many tools, intimate knowledge of employed logic calculi and proof search strategies is necessary.

Successful specification and verification of larger projects, therefore, is done separately from software development by academic specialists with several years of training in formal methods, in many cases by the tool developers themselves. It is unlikely that formal software specification and verification will become a routine task in industry under these circumstances.

The future challenge for formal methods is to make their considerable potential feasible to use in an industrial environment. This leads to the requirements:

1. Tools for formal software specification and verification must be integrated into industrial software engineering procedures.
2. User interfaces of these tools must comply with state-of-the-art software engineering tools.
3. The necessary amount of training in formal methods must be minimized. Moreover, techniques involving formal software specification and verification must be teachable in a structured manner. They should be integrated in courses on software engineering topics.

To be sure, the thought that full formal software verification might be possible without any background in formal methods is utopian. An industrial verification tool should, however, allow for *gradual* verification so that software engineers at any (including low) experience level with formal methods may benefit. In addition, an integrated tool with well-defined interfaces facilitates "outsourcing" those parts of the modeling process that require special skills.

Another important motivation to integrate design, development, and verification of software is provided by modern software development methodologies which are *iterative* and *incremental*. *Post mortem* verification would enforce the antiquated waterfall model. Even worse, in a linear model the extra effort needed for verification cannot be parallelized and thus compensated by greater work force. Therefore, delivery time increases considerably and would make formally verified software decisively less competitive.

But not only must the extra time for formal software development be within reasonable bounds, the cost of for-

[*]The KeY project is supported by the Deutsche Forschungsgemeinschaft (grant no. Ha 2617/2-1).

mal specification and verification in an industrial context requires accountability:

4. It must be possible to give realistic estimations of the cost of each step in formal software specification and verification depending on the type of software and the degree of formalization.

This implies immediately that the mere existence of tools for formal software specification and verification is not sufficient, rather, formal specification and verification have to be fully integrated into the software development process.

1.2 The KeY Project

Since November 1998 the authors work on a project addressing the goals outlined in the previous section; we call it the KeY project (read "key").

In the principal use case of the KeY system there are actors who want to implement a software system that complies with given requirements and formally verify its correctness. The system is responsible for adding formal detail to the analysis model, for creating conditions that ensure the correctness of refinement steps (called proof obligations), for finding proofs showing that these conditions are satisfied by the model, and for generating counter examples if they are not. Special features of KeY are:

- We concentrate on object-oriented analysis and design methods (OOAD)—because of their key role in today's software development practice—, and on Java as the target language. In particular, we use the Unified Modeling Language (UML) [20] for visual modeling of designs and specifications and the Object Constraint Language (OCL) for adding further restrictions. This choice is supported by the fact, that the UML (which contains OCL since version 1.3) is not only an OMG standard, but has been adopted by all major OOAD software vendors and is featured in recent OOAD textbooks [18].

- We use a commercial CASE tool as starting point and enhance it by additional functionality for formal specification and verification. The current tool of our choice is TogetherSoft LLC's TogetherJ.

- Formal verification is based on an axiomatic semantics of the *real* programming language Java Card [23] (soon to be replaced by Java 2 Micro Edition, J2ME).

- As a case study to evaluate the usability of our approach we develop a scenario using smart cards with Java Card as programming language [12, 13]. Java smart cards make an extremely suitable target for a case study:
  - As an object-oriented language, Java Card is well suited for OOAD;
  - Java Card lacks some crucial complications of the full Java language (no threads, fewer data types, no graphical user interfaces);
  - Java Card applications are small (Java smart cards currently offer 16K memory for code);
  - at the same time, Java Card applications are embedded into larger program systems or business processes which should be modeled (though not necessarily formally verified) as well;
  - Java Card applications are often security-critical, thus giving incentive to apply formal methods;
  - the high number (usually millions) of deployed smart cards constitutes a new motivation for formal verification, because, in contrast to software run on standard computers, arbitrary updates are not feasible.[1]

- Through direct contacts with software companies we check the soundness of our approach for real world applications (some of the experiences from these contacts are reported in [3]).

The KeY system consists of three main components:

- The *modeling component*: this component is based on the CASE tool and is responsible for all user interactions (except interactive deduction). It is used to generate and refine models, and to store and process them. The extensions for precise modeling contains, e.g., editor and parser for the OCL. Additional functionality for the verification process is provided, e.g., for writing proof obligations.

- The *verification manager*: the link between the modeling component and the deduction component. It generates proof obligations expressed in formal logic from the refinement relations in the model. It stores and processes partial and completed proofs; and it is responsible for correctness management (to make sure, e.g., that there are no cyclic dependencies in proofs).

- The *deduction component*. It is used to actually construct proofs—or counter examples—for proof obligations generated by the verification manager. It is based on an interactive verification system combined with powerful automated deduction techniques that increase the degree of automation; it also contains a part for automatically generating counter examples from failed proof attempts. The interactive and automated techniques and those for finding counter examples are fully integrated and operate on the same data structures.

Although consisting of different components, the KeY system is going to be fully integrated with a uniform user interface.

A first KeY system prototype has been implemented, integrating the CASE tool TogetherJ and a deductive component (it has only limited capabilities and lacks the verification manager component). Work on the full KeY system is under progress.

[1]While Java Card applets on smart cards can be updated in principle, for security reasons this does not extend to those applets that verify and load updates.

2 Designing a System with KeY

2.1 The Modeling Process

Software development is generally divided into four activities: analysis, design, implementation, and test. The KeY approach embraces verification as a fifth category. The way in which the development activities are arranged in a sequential order over time is called software development *process*. It consists of different phases. The end of each phase is defined by certain criteria the actual model should meet (milestones).

In some older process models like the waterfall model or Boehm's spiral model no difference is made between the main activities—analysis, design, implementation, test—and the process phases. More recent process models distinguish between phases and activities very carefully; for example, the Rational Unified Process [15] uses the phases inception, elaboration, construction, and transition along with the above activities.

The KeY system does neither support nor require the usage of a *particular* process. However, it is taken into account that most modern processes have two principles in common. They are *iterative* and *incremental*. The design of an iteration is often regarded as the refinement of the design developed in the previous iteration. This has an influence on the way in which the KeY system treats UML models and additional verification tasks (see Section 2.3). The verification activities are spread across all phases in software development. They are often carried out after test activities.

2.2 Specification with the UML and the OCL

The diagrams of the Unified Modeling Language provide, in principle, an easy and concise way to formulate various aspects of a specification, however [25, foreword]: "[...] there are many subtleties and nuances of meaning diagrams cannot convey by themselves." This was a main source of motivation for the development of the Object Constraint Language (OCL), part of the UML since version 1.3 [20]. Constraints written in this language are understood in the context of a UML model, they never stand by themselves. The OCL allows to attach preconditions, postconditions, invariants, and guards to specific elements of a UML model.

When designing a system with KeY, one develops a UML model that is enriched by OCL constraints to make it more precise. This is done using the CASE tool integrated into the KeY system. To assist the user, the KeY system provides menu and dialog driven input possibility. Certain standard tasks, for example, generation of formal specifications of inductive data structures (including the common ones such as lists, stacks, trees) in the UML and the OCL can be done in a fully automated way, while the user simply supplies names of constructors and selectors. Even if formal specifications cannot fully be composed in such a schematic way, considerable parts usually can.

In addition, we have developed a method supporting the extension of a UML model by OCL constraints that is based on enriched design patterns. In the KeY system we will provide common patterns that come complete with predefined constraint schemata. These schemata are formulated in a language that is a slight extension of OCL. They are flexible and allow the user to easily generate well-adapted constraints for the different instances of a pattern. The user needs not write formal specifications from scratch, but only to adapt and complete them. A detailed description of this technique and of experiences with its application in practice is given in [4].

2.3 The KeY Module Concept

The KeY system supports modularization of the model in a particular way. Those parts of a model that correspond to a certain component of the modeled system are grouped together and form a *module*. Modules are a different structuring concept than iterations and serve a different purpose. A module contains all the model components (diagrams, code etc.) that refer to a certain system component. A module is not restricted to a single level of refinement.

There are three main reasons behind the module concept of the KeY system:

Structuring: Models of large systems can be structured, which makes them easier to handle.

Information hiding: Parts of a module that are not relevant for other modules are hidden. This makes it easier to change modules and correct them when errors are found, and to re-use them for different purposes.

Verification of single modules: Different modules can be verified separately, which allows to structure large verification problems. If the size of modules is limited, the complexity of verifying a system grows linearly in the number of its modules and thus in the size of the system. This is indispensable for the scalability of the KeY approach.

In the KeY approach, a hierarchical module concept with sub-modules supports the structuring of large models. The modules in a system model form a tree with respect to the sub-module relation.

Besides sub-modules and model components, a module contains the refinement relations between components that describe the same part of the modeled system in two consecutive levels of refinement. The verification problem associated with a module is to show that these refinements are correct (see Section 3.1). The refinement relations must be provided by the user; typically, they include a signature mapping.

To facilitate information hiding, a module is divided into a public part, its *contract*, and a private (hidden) part; the user can declare parts of *each* refinement level as public or private. Only the public information of a module A is visible in another module B provided that module B implicitly or explicitly *imports* module A. Moreover, a component of module B belonging to some refinement level can only *see* the visible information from module A that belongs to the same level. Thus, the private part of a module can be changed as long as its contract is not affected. For the description of a refinement relation (like a signature mapping) all elements of a module belonging to the initial model or the refined model are visible, whether declared public or not.

As the modeling process proceeds through iterations, the system model becomes ever more precise. The final step is a special case, though: the involved models—the implementation model and its realization in JAVA—do not necessarily differ in precision, but use different paradigms (specification vs. implementation) and different languages (UML with OCL vs. JAVA).

The ideas of refinement and modularization in the KeY module concept can be compared with (and are partly influenced by) the KIV approach [21] and the B Method [1, 17], but still follow different guidelines.

2.4 The Internal State of Objects

The formal specification of objects and their behavior requires special techniques. One important aspect is that the behavior of objects depends on their state that is stored in their attributes, however, the methods of a JAVA class can in general not be described as functions on their input as they may have side effects and change the state. To fully specify the behavior of an object or class, it must be possible to refer to its state (including its initial state). Difficulties may arise if methods for observing the state are not defined or are declared private and, therefore, cannot be used in the public contract of a class. To model such classes, *observer methods* have to be added. These allow to observe the state of a class without changing it.

3 Formal Verification with KeY

Once a program is formally specified to a sufficient degree one can start to formally verify it. Neither a program nor its specification need to be complete in order to start verifying it. In this case one suitably weakens the postconditions (leaving out properties of unimplemented or unspecified parts) or strengthens preconditions (adding assumptions about unimplemented parts). Data encapsulation and structuredness of OO designs are going to be of great help here.

The verification process will be automated as much as possible with the help of deduction techniques based on previous work [2] done in our group on integrating our automated [6] and interactive theorem provers [21].

3.1 Proof Obligations and Program Logic

For obtaining the proof obligations to be justified, we employ design by contract [19] with the same restriction as [25]: run-time aspects are completely ignored.

The logic we use is dynamic logic (DL) [16]. It is a full logic with first-order quantification, built from basic blocks of the form $\langle\alpha\rangle Q$ with the meaning: program α terminates and afterwards formula Q holds. We decided to take a bold step and allow any legal JAVA CARD program to occur in the place of α. A more detailed description of KeY-DL is given in [5]. The central point is, of course, to deal with features of OO languages such as side effects and exception handling.

3.2 The Deduction Component

The KeY system comprises a deductive component that can handle KeY-DL. This KeY prover combines interactive and automated theorem proving techniques. Experiences with the KIV system [21] have shown how to cope with DL proof obligations: The original goal is reduced to first-order predicate logic using DL rules, as described in [5].

Our deductive system uses a technique of *schematic theory specific rules*, which combine purely logical knowledge, information on how this knowledge should be used, and information on when and where this knowledge should be presented for interactive use. This technique has been implemented in the interactive proof system IBIJa[2].

Interactive proving is greatly enhanced by intermediate automated steps based on proof search in the style of analytic tableaux [11]. Also, a component of *disproving* formulas by finding counterexamples is being developed.

4 Related Work

There are many projects dealing with formal methods in software engineering including several ones aimed at JAVA as a target language. There is also work on security of JAVA CARD and ACTIVEX applications as well as on secure smart card applications in general. We are, however, not aware of any project quite like ours. We mention some of the more closely related projects:

- The COGITO project [24] resulted in an integrated formal software development methodology and support system based on extended Z as specification language and Ada as target language. It is not integrated into a CASE tool, but stand-alone.

- The FUZE project [10] realized CASE tool support for integrating the FUSION OOAD process with the formal specification language Z. The aim was to formalize OOAD methods and notations such as the UML, whereas we are interested to derive formal specifications with the help of an OOAD process extension.

- The goal of the QUEST project [22] is to enrich the CASE tool AUTOFOCUS for description of distributed systems with means for formal specification and support by model checking. Applications are embedded systems, description formalisms are state charts, activity diagrams, and temporal logic.

- Aim of the SYSLAB project is the development of a scientifically founded approach for software and systems development. At the core is a precise and formal notion of hierarchical "documents" consisting of informal text, message sequence charts, state transition systems, object models, specifications, and programs. All documents have a "mathematical system model" that allows to precisely describe dependencies or transformations [7].

[2]More information on IBIJa is available at i11www.ira.uka.de/˜ibija.

- The goal of the PROSPER project was to provide the means to deliver the benefits of mechanized formal specification and verification to system designers in industry (www.dcs.gla.ac.uk/prosper/index.html). The difference to the KeY project is that the dominant goal is hardware verification; and the software part involves only specification.

5 Conclusion and the Future of KeY

We described the current state of the KeY project and its ultimate goal: To facilitate and promote the use of formal verification in an industrial context for real-world applications. It remains to be seen to which degree this goal can be achieved.

Our vision is to make the logical formalisms transparent for the user with respect to OO modeling. That is, whenever user interaction is required, the current state of the verification task is presented in terms of the environment the user has created so far and not in terms of the underlying deduction machinery. The situation is comparable to a symbolic debugger that lets the user step through the source code of a program while it actually executes compiled machine code.

References

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

[2] W. Ahrendt, B. Beckert, R. Hähnle, W. Menzel, W. Reif, G. Schellhorn, and P. H. Schmitt. Integration of automated and interactive theorem proving. In W. Bibel and P. Schmitt, editors, *Automated Deduction: A Basis for Applications*, volume II, chapter 4, pages 97–116. Kluwer, 1998.

[3] T. Baar. Experiences with the UML/OCL-approach to precise software modeling: A report from practice. Available at i12www.ira.uka.de/~key, 2000.

[4] T. Baar, T. Sattler, R. Hähnle, and P. H. Schmitt. Entwurfsmustergesteuerte Erzeugung von OCL-Constraints. In *Proceedings, Softwaretechnik 2000, Berlin, Germany*, 2000. To appear. Available at i12www.ira.uka.de/~key.

[5] B. Beckert. A dynamic logic for java card. In *Proceedings, 2nd ECOOP Workshop on Formal Techniques for Java Programs, Cannes, France*, 2000. To appear. Available at i12www.ira.uka.de/~key.

[6] B. Beckert, R. Hähnle, P. Oel, and M. Sulzmann. The tableau-based theorem prover $_3T^AP$, version 4.0. In *Proceedings, 13th International Conference on Automated Deduction (CADE), New Brunswick/NJ, USA*, LNCS 1104, pages 303–307. Springer, 1996.

[7] R. Breu, R. Grosu, F. Huber, B. Rumpe, and W. Schwerin. Towards a precise semantics for object-oriented modeling techniques. In H. Kilov and B. Rumpe, editors, *Proceedings, Workshop on Precise Semantics for Object-Oriented Modeling Techniques at ECOOP'97*. Technical University of Munich, Technical Report TUM-I9725, 1997.

[8] E. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[9] D. L. Dill and J. Rushby. Acceptance of formal methods: Lessons from hardware design. *IEEE Computer*, 29(4):23–24, 1996. Part of: Hossein Saiedian (ed.). *An Invitation to Formal Methods*. Pages 16–30.

[10] R. B. France, J.-M. Bruel, M. M. Larrondo-Petrie, and E. Grant. Rigorous object-oriented modeling: Integrating formal and informal notations. In M. Johnson, editor, *Proceedings, Algebraic Methodology and Software Technology (AMAST), Berlin, Germany*, LNCS 1349. Springer, 1997.

[11] M. Giese. Integriertes automatisches und interaktives Beweisen: Die Kalkülebene. Diploma Thesis, Fakultät für Informatik, Universität Karlsruhe, June 1998. In German. Available at i11www.ira.uka.de/~giese/da.ps.gz.

[12] S. B. Guthery. Java Card: Internet computing on a smart card. *IEEE Internet Computing*, 1(1):57–59, 1997.

[13] U. Hansmann, M. S. Nicklous, T. Schäck, and F. Seliger. *Smart Card Application Development Using Java*. Springer, 2000.

[14] M. G. Hinchey and J. P. Bowen, editors. *Applications of Formal Methods*. Prentice Hall, 1995.

[15] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Object Technology Series. Addison-Wesley, 1999.

[16] D. Kozen and J. Tiuryn. Logic of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 14, pages 789–840. Elsevier, Amsterdam, 1990.

[17] K. Lano. *The B Language and Method: A guide to Practical Formal Development*. Springer Verlag London Ltd., 1996.

[18] J. Martin and J. J. Odell. *Object-Oriented Methods: A Foundation, UML Edition*. Prentice-Hall, 1997.

[19] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, 2nd edition, 1997.

[20] Object Management Group, Inc., Framingham/MA, USA, www.omg.org. *OMG Unified Modeling Language Specification, Version 1.3*, June 1999.

[21] W. Reif. The KIV-approach to software verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*, LNCS 1009. Springer, 1995.

[22] O. Slotosch. Overview over the project QUEST. In *Applied Formal Methods, Proceedings of FM-Trends 98, Boppard, Germany*, LNCS 1641, pages 346–350. Springer, 1999.

[23] Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.1 Application Programming Interfaces, Draft 2, Release 1.3*, 1998.

[24] O. Traynor, D. Hazel, P. Kearney, A. Martin, R. Nickson, and L. Wildman. The Cogito development system. In M. Johnson, editor, *Proceedings, Algebraic Methodology and Software Technology (AMAST), Berlin*, LNCS 1349, pages 586–591. Springer, 1997.

[25] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Object Technology Series. Addison-Wesley, 1999.

On the Automatic Visualization of Statecharts: The ViSta tool

R. Castelló, R. Mili [†], I. G. Tollis and V. Benson
Computer Science Program
The University of Texas at Dallas
P.O. Box 830688
Richardson, TX 75083-0688, USA
1 972 883 2091
{castello, rmili, tollis, vlada}@utdallas.edu

April 28, 2000

1 Introduction

Statecharts are widely used for the requirements specification of reactive systems. This notation captures the requirements attributes that are concerned with the behavioral features of a system, and models these features in terms of a hierarchy of diagrams and states. The usefulness of statecharts depends primarily on their readability, that is the capability of conveying the meaning of the drawings quickly and clearly. Several visualization tools for reactive system specification and design are available in the market [2, 6, 5, 9]. Even though these tools are helpful in organizing designers' thoughts, they are mostly sophisticated graphical editors, and therefore are severely inadequate for the modeling of complex reactive systems. Specifically, hand made diagrams become easily unreadable when the specification complexity increases. Therefore computer assistance is of paramount importance for the graphical representation of complex reactive systems.

In this paper we present a tool that automatically generates statechart layouts. We proceed in two steps: we first extract information from a textual description of requirements and store it into interactive templates; then we automatically generate graphs that model statecharts in a hierarchical fashion. The resulting drawings enjoy several properties: they have a low number of arc crossings; they emphasize the natural hierarchical decomposition of states into substates; and they cover an optimal drawing area.

The automatically produced graphical representation is an effective requirements assessment tool since it allows the specifier to shift focus from organizing the mental or physical structure of the requirements to its analysis. In addition, the interdependence between the textual, template and graphical representations ensures consistency between the different documents and therefore facilitates the V&V effort.
In Section 2 we describe our automatic visualization tool. In Section 3, we assess our results and discuss our prospects for future work.

[*]Research supported in part by Sandia National Labs and by the Texas Advanced Research Program under grant number 009741-040.
[†]Correspondence author.

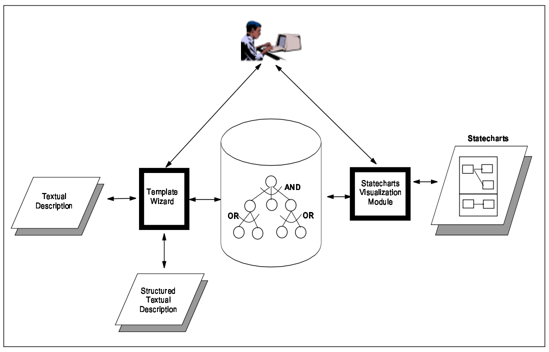

Textual Description
Template Wizard
Structured Textual Description
AND
OR
OR
Statecharts Visualization Module
Statecharts

Figure 1: System Overview.

2 The ViSta tool

The *ViSta* tool consists of three main components: the *template wizard*, the *central database* and the *statechart visualization module* (see Figure 1).

2.1 Template Wizard

The template wizard guides the user through the steps necessary for the construction of a structured requirements document. It offers an elaborate user interface that facilitates requirements capture. The user inputs a textual description by either opening an existing document or creating a new file. Then he/she selects information from the textual document and dynamically introduces it into a set of templates. Selected parts turn into a distinct color to inform the user that they were successfully accepted by the templates.

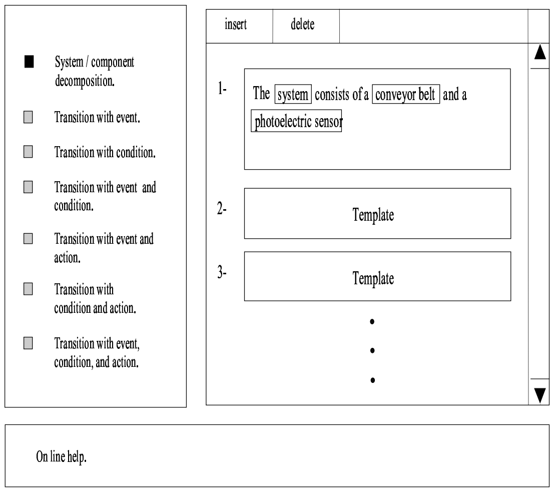
A template is a form-based component that has a predefined structure. It consists of structured propositions with text fields to be filled in with information specific to the requirements. Figure 2 shows various types of templates offered by our system.

It is possible to modify the contents of the templates at any time by traversing the Wizard backward and forward. The user can add, delete or rearrange the order of templates to best fit his/her needs. Insertion is performed by positioning the mouse on a specific template type (left side of the view); the corresponding template type gets highlighted and the user can insert the new template at a position he/she desires. A deletion operation is performed by selecting a template on the right side of the view and clicking **delete**. Rearrangement of the templates is achieved through the "Drag-and Drop" feature.

The collected data are dynamically stored into a central database that is used to generate and update the graphical and structured textual representations.

2.2 Decomposition Tree

The data stored in our central database is summarized in a formal structure called a *decomposition tree*. This structure reflects the decomposition of superstates into substates. A node in the decomposition tree includes the following information: its name; its width and height; the coordinates of its origin point; a pointer to its parent; the list of its children; its decomposition type


■ System / component decomposition.
□ Transition with event.
□ Transition with condition.
□ Transition with event and condition.
□ Transition with event and action.
□ Transition with condition and action.
□ Transition with event, condition, and action.

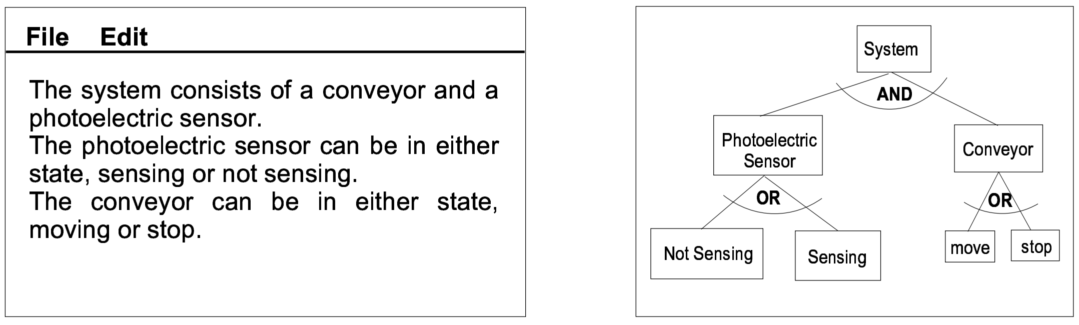
1- The system consists of a conveyor belt and a photoelectric sensor

2- Template

3- Template

On line help.

Figure 2: Types of Template.


File Edit

The system consists of a conveyor and a photoelectric sensor.
The photoelectric sensor can be in either state, sensing or not sensing.
The conveyor can be in either state, moving or stop.

a) Textual Description

b) Decomposition Tree

Figure 3: Decomposition Tree.

(e.g., *AND*, *OR* or *leaf*); the list of incoming arcs; the list of outgoing arcs; a list of attributes; and finally its aliases. The root of a decomposition tree corresponds to the system superstate; leaves correspond to atomic states. Each node in the tree can be decomposed through the *AND* or *OR* decomposition. The *OR* decomposition reflects the hierarchical structure of state machines; the *AND* decomposition reflects concurrency. Figure 3(b) shows the decomposition tree that corresponds to the text given in Figure 3(a).

2.3 Statechart Visualization Module

This module automatically draws a graph that models a statechart in a hierarchical fashion. In our approach, a statechart is treated as a graph. Nodes [1] in the graph correspond to states, and arcs correspond to transitions between states.

[1] In the remainder of this paper we will use the words *node* and *object* interchangeably.

Basic Algorithm

Our algorithm proceeds as follows: first, the decomposition tree is traversed in order to determine the dimensions (and origin point) of every node in a recursive manner. If a node v is a leaf then a drawing procedure is called. This procedure produces a labeled rectangle and returns the dimensions of the rectangle. If v is an AND node then a recursive algorithm constructs the drawings of each child of v and places the drawings contiguously. If v is an OR node then a recursive algorithm constructs the drawings of v's children in a hierarchical fashion by assigning each child to a specific layer. This algorithm is a variant of Sugiyama's algorithm [8] tailored to statecharts. Figure 5 shows the statechart diagram that was automatically generated by the ViSta tool, based on the decomposition tree depicted in Figure 4.

Edge Crossing Reduction

Our approach to the edge crossing reduction problem is based on the *layer by layer sweep* paradigm [1]. Specifically, given a hierarchy of layers, we first select two adjacent layers either starting from the first layer of the hierarchy or the last. Let us assume that we traverse the hierarchy left to right. We select layers L_1 and L_2 and assign a unique number, called *position_in_layer*, to each vertex of these layers. This number corresponds to the relative position of the vertices in the selected layers. Then, for each node α of L_2, we compute the number of crossings between the edges incident to α and the edges incident to nodes β_i (cn_{α,β_i}) such that $\alpha, \beta_i \in L_2$ and $position\_in\_layer(\alpha) < position\_in\_layer(\beta_i)$. This information is captured in the upper triangular portion of a table called *crossing number matrix*. The crossing number matrix is fed into the *adjacent exchange* algorithm [1] that analyzes the data and repeatedly exchanges the positions of two adjacent vertices α, β whenever $c_{\alpha,\beta} > c_{\beta,\alpha}$. Finally, the edge crossing reduction heuristic updates the crossing number matrix and proceeds with the next two adjacent layers (e.g., L_2 and L_3). Figures 6 and 7 show statechart drawings before and after the application of our edge crossing reduction algorithm.

Edge Labeling

Edge labels are crucial in describing transitions in statecharts. Previously, edge labeling techniques were described for graph drawings, and geographical and technical maps with fixed geometry [4, 3]. In our work, we address the problem of graph drawings with flexible geometric features.
In the statecharts notation, an edge label consists of three components namely *event*, *condition* and *action*. We have defined the following steps for the placement of edge labels in statecharts:

1. We fix the maximum length of the label to a constant, and we write the three components (i.e., *events*, *conditions* and *actions*) on three separate lines. If the size of a component is greater than the maximum length of the label, then we write it on several lines.

2. At the beginning of the execution of the drawing algorithm, we assign labels to sublayers.

3. We traverse the hierarchy from left to right, considering two adjacent layers L_1 and L_2 at a time. For each vertex a in L_1, we identify the set of edges E_a between a and the vertices in L_2. We order E_a in such a way that potential crossings are removed.

Figures 8 and 9 show statechart diagrams before and after the application of our edge labeling algorithm.

3 Conclusion and Future Work

Summary and Assessment. In this paper we presented a tool that automatically generates statecharts. We proceed in two steps: we first extract information from a textual description of requirements and store it into interactive templates; then we draw graphs that model statecharts in a hierarchical fashion. Our drawings enjoy several properties:1) they emphasize the natural recursive hierarchical decomposition of states into substates; 2) nodes are placed on layers according to their distance from the local initial state; 3) the number of arc crossings is low.

Future work. Our prospects for future work include the optimization of the drawing algorithm by using floor planning techniques; and the development and implementation of methods that will allow the automatic translation of decomposition trees into a formal notation (e.g., Z [7]).

References

[1] GuiseppeDi Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs.* Prentice Hall, 1999.

[2] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, May 1990.

[3] Konstantinos G. Kakoulis and Ioannis G. Tollis. An algorithm for labeling edges of hierarchical drawings. In Giuseppe DiBattista, editor, *Graph Drawing (Proceedings GD'97)*, pages 169–180. Springer-Verlag, 1997. Lecture Notes in Computer Science 1353.

[4] Konstantinos G. Kakoulis and Ioannis G. Tollis. On the edge label placement problem. In Stephen North, editor, *Graph Drawing (Proceedings GD'96)*, pages 241–256. Springer-Verlag, 1997. Lecture Notes in Computer Science 1190.

[5] Rory O'Donnel, Bjǒrn Waldt, and Johan Bergstrand. Automatic code for embedded systems based on formal methods. Available from Telelogic over the Internet. http://www.Telelogic.se/solution/techpap.asp. Accessed on April 5, 1999.

[6] J. Peterson. Overcoming the crisis in real-time software development. Available from Objectime over the Internet. http://www.Objectime.on.ca/otl/technical/crisis.pdf. Accessed on April 5, 1999.

[7] J. M. Spivey. *The Z Notation, A reference Manual.* Prentice Hall, 1989.

[8] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, February 1981.

[9] Artisan Software Tools. Real-time studio: The rational alternative. Available from Artisan Software Tools over the Internet. http://www.artisansw.com/rtdialogue/pdfs/rational.pdf. Accessed on April 5, 1999.

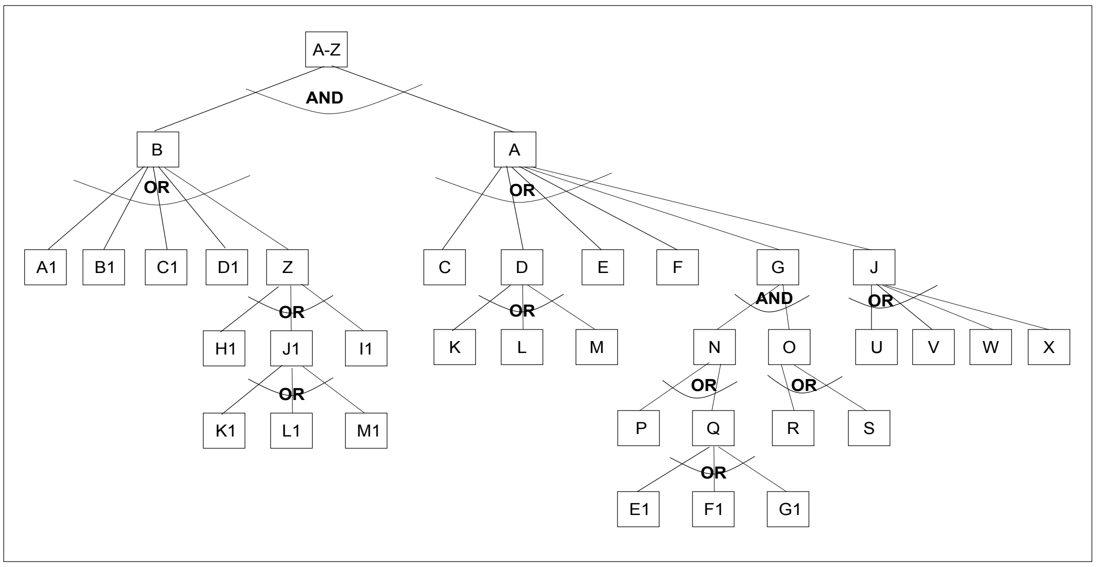


Figure 4: Decomposition Tree.

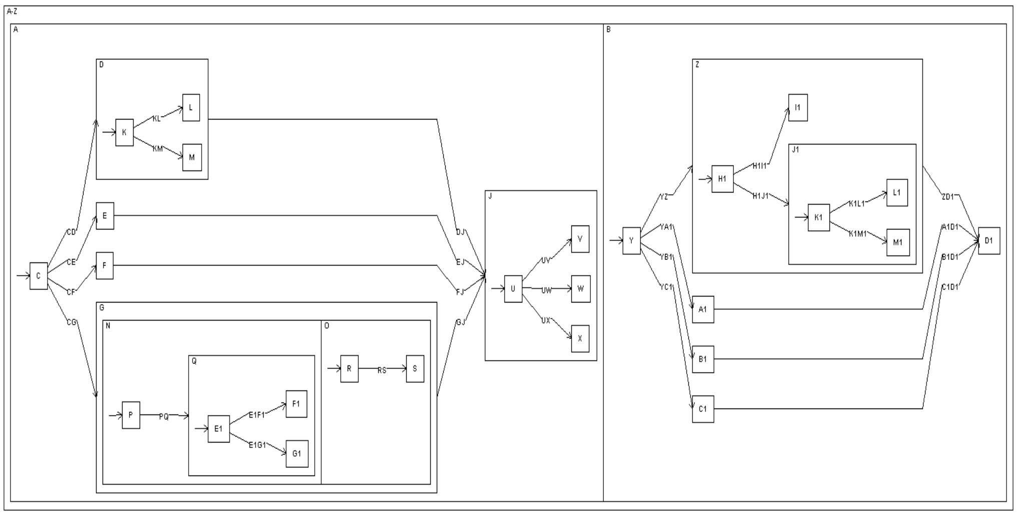


Figure 5: Automatically generated Statechart.

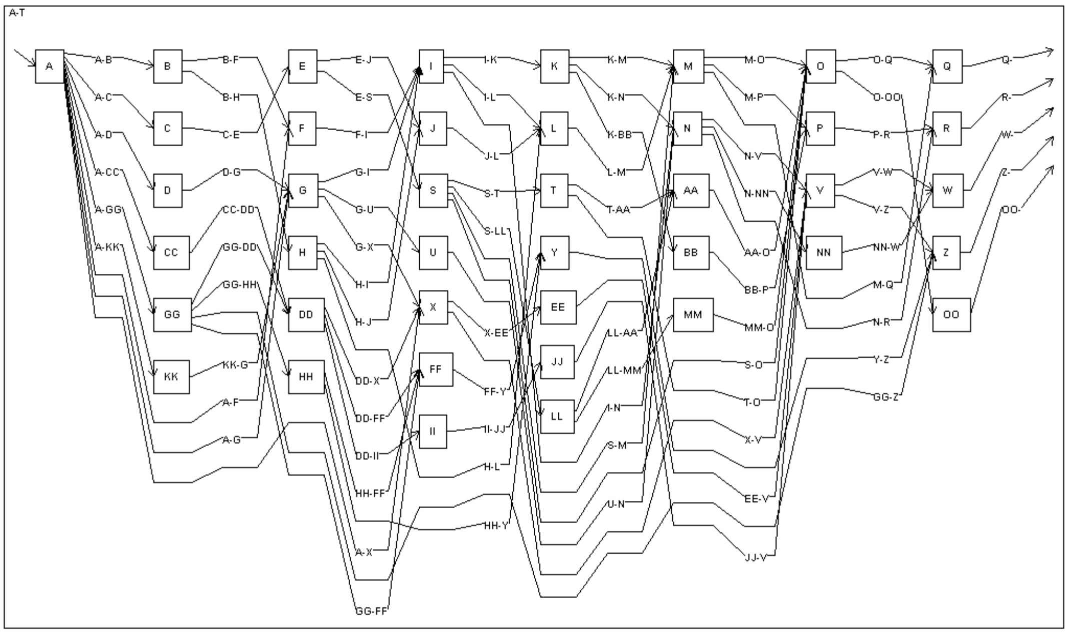


Figure 6: Statechart Diagram Before Edge Crossing Reduction.

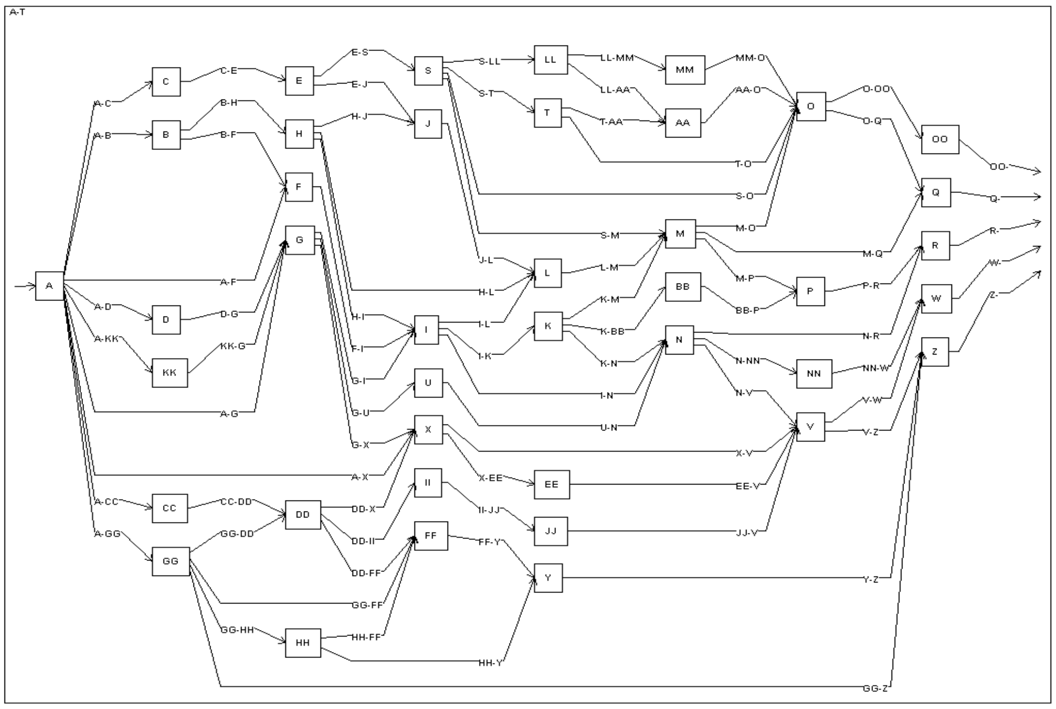


Figure 7: Statechart Diagram After Edge Crossing Reduction.

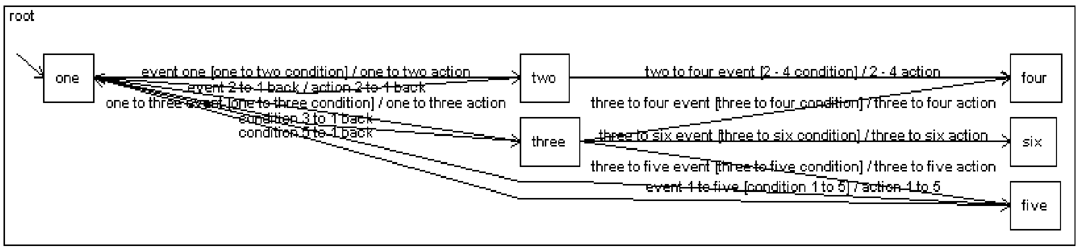

root
one
two
three
four
six
five
event one [one to two condition] / one to two action
event 2 to 1 back / action 2 to 1 back
one to three event [one to three condition] / one to three action
condition 3 to 1 back
condition 5 to 1 back
two to four event [2 - 4 condition] / 2 - 4 action
three to four event [three to four condition] / three to four action
three to six event [three to six condition] / three to six action
three to five event [three to five condition] / three to five action
event 1 to five [condition 1 to 5] / action 1 to 5

Figure 8: Statechart Diagram Before Edge Labeling Algorithm.

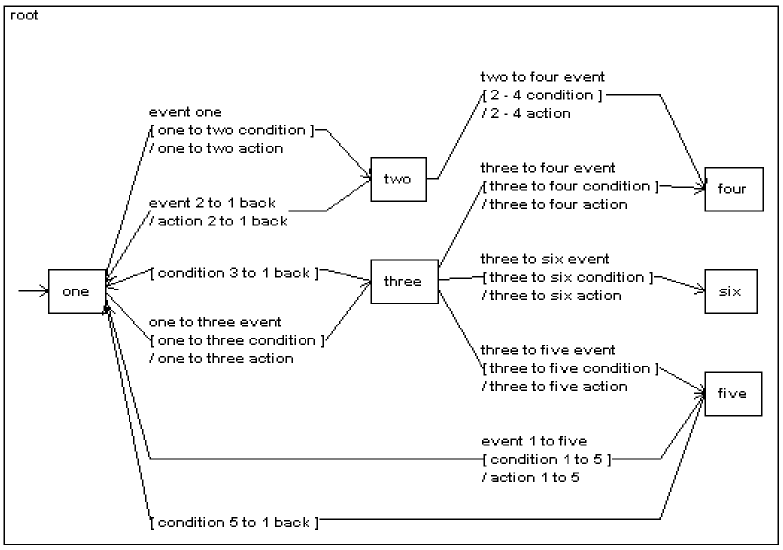


Figure 9: Statechart Diagram After Edge Labeling Algorithm.

Synchronous Object-Oriented Programming: sE2.0 *

Reinhard Budde, Axel Poigné, Karl-Heinz Sylla
German National Research Centre for Information Technology

1 Introduction

With sE, we provide a design environment which supports the following design and programming paradigms:

- Synchronous modelling, for constructing reactive real-time components, and for enabling system validation by model-checking.
- Object-oriented modelling in a strongly typed lan-guage, which is well suited for a robust and flexible design of complex systems.

We claim as highlights that

- in contrast to other synchronous languages, data operations and reactive behaviour are tightly integrating, giving full control, for instance, over time races, and that
- the reactive sublanguage is a fine-grained integration of synchronous notations [2, 3, 1] combining means to specify spontaneous and periodic synchronous behaviour.

We outline the main design decisions, and give an overview of the presently available tools.

2 Synchronous Computation

The economic importance of embedded software design is beyond dispute, as is the object-oriented design paradigm. Synchronous languages such as ESTEREL[2], LUSTRE[3], and SYNCCHARTS[1] are well accepted though not that familiar for the general programming community.

An embedded reactive system interacts with its environment via signals. Input signals may carry sensor data and output signals may trigger actuators,

*Partially funded by the ESPRIT IiM "CRISYS (Project No. 25514) for instance. A system reaction depends on its internal state and on the value of input signals. It generates output signals and changes the internal state.

The synchronous execution model reflects the basic idea of digital hardware design and of many engineering formalisms: Processing proceeds in steps controlled by a trigger signal, a "clock". When the trigger signal is present, or "when the clock ticks", a system starts to react and the reaction is finished in time before the next trigger signal is present. Such a processing step is called an instant.

There are several benefits of the synchronous computation model, to name a few:

- a mathematically precise model [2, 3, 6, 7],
- deterministic scheduling at compile time implying that behaviour is reproducible, a condition sine-qua-non for testing,
- highly efficient target code for different targets, including micro processors, even hardware, and
- several high level source languages based on quite different formalisms such as data flow [3, 6], hierarchical automata [1, 4], preemptive imperative programming [2].

We have demonstrated with the Synchrony Workbench [5] that these languages can be accomodated in a single framework. However, two drawbacks surfaced:

- the lack of control with regard to data operations, and
- the user's difficulty to handle three quite different notations.

sE resolves these shortcomings by tightly unifying the data and control model, as discussed in Section 3, and by a fine-grained combination of notations discussed in Section 4.

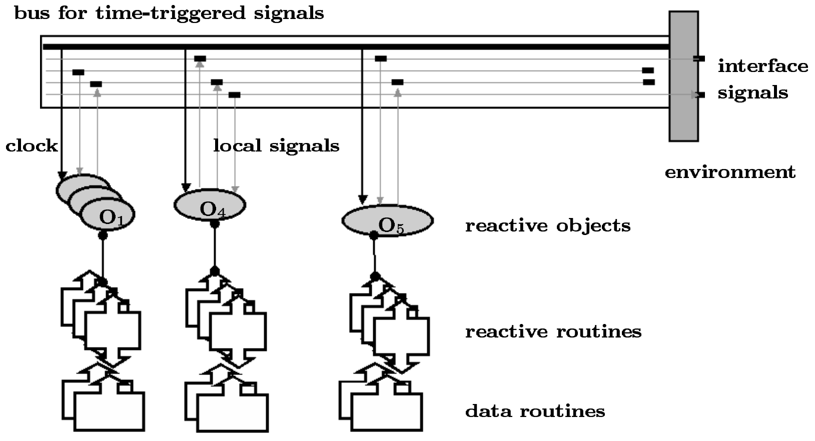
3 Reactive Objects and Temporal Firewalls

Synchronous languages are crafted to deal with control. Except for some base types, data routines must be defined in a host language, such as C, being triggered by signals via the external interface. Since many such data routines may be called in an instant, this may result in *time races*, hence in non-determinism.

In contrast, sE integrates reactive behaviour and data routines in a uniform, object-oriented framework enhanced by *reactive objects* that is: objects which encapsulate (synchronous) reactive behaviour. For avoiding the unwanted interaction of control and data operations, we restrict

- communication between reactive objects to signals only, and
- access to data routines in an reactive object: only the reactive routines can access the data routines (i.e. data routines in a reactive object are private).

We give a schematic view of the architecture.



The reactive objects read signals from and broadcast signals to the *signal bus*. The basic pulse or *clock* is part of the signal bus. If the clock is present, all reactive objects evaluate in parallel. They read from and write to the signal bus. The compiler will schedule the read and write action deterministically. This scheduling is an inherent part of the compilation of synchronous programs ("causaility analysis"). Hence time races with regard to different objects are reduced to read/write conflicts on the signal bus that are dealt with by the synchronous technology anyway. We speak of a first *temporal firewall*. To given example:

```
class TimerApp
-- signal interface and signal bus
    start : signal is const map start = t1.start = t2.start end; (*)
    clock : signal is const map clock = t1.clock = t2.clock end;
    t1_elapsed : signal is map t1_elapsed = t1.elapsed end;
    t2_elapsed : signal is map t2_elapsed = t2.elapsed end;
-- reactive objects
    t1, t2 : Timer;(**)
-- creation routine
    run : none is creation
      do t1.set (800);
         t2.set (1000);
    end;
end class TimerApp
```

The external interface and the "wiring" the signal bus are defined simultaneously, e.g. the declaration (∗) defines the input signal "`start`" and connects it to the respective "`start`" signals of the two timer objects "`t1`" and "`t2`" of class "`Timer`" declared in line (∗∗).

Next we explain the interaction of reactive control and data routines.

```
reactive class Timer
-- creation routine
    set (d:int) : none is public creation
do latch := d end;
-- signal declarations
    start, clock : signal is public const;
    elapsed      : signal is public;
-- reactive routines
    react is reactive
     do loop
          await ?start; reset();
          next;
          cancel
             loop
                await ?clock;
                decrement();
                next;
             end
          when is_elapsed() then emit elapsed
       end
    end;
-- data routines and attributes
    latch          : int is const;
    counter        : int;
    reset(d: int)  : none is do  counter := d  end;
    decrement      : none is
       do if counter > 0 then counter := counter - 1 end;
    is_elapsed : bool is  do result := (counter = 0) end ;
-- scheduling for avoiding time races
    sequence decrement < is_elapsed
end class Timer
```

The reactive routine "`react`" should be rather self-explaining: A timer object initially waits for the "`start`" signal. If the "`start`" signal is present, the counter is set to a start value by calling the (data) routine "`reset()`". At each "`clock`" "tick" the counter decrements until the condition "`is_elapsed()`" holds. It is important to notice that "time passes" only when waiting. Reactions are considered as instantaneous: If the "`clock`" signal is present the counter decrements at the same instant as the termination condition "`is_elapsed()`" is evaluated. The creation routine "`set`" sets the constant attribute "`latch`".

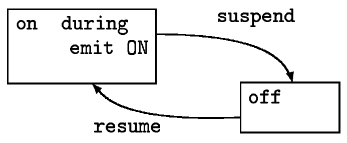
There is a "time race" between the data routine "`decrement`" and "`is_elapsed`"; that is, the execution order of data routines may affect the result. To avoid non-determinism we have to specify the desired execution order (here: "`decrement`" before "`is_elapsed`"). Time races will be detected by the sE compiler based on the analysis of the data flow and the state transitions. The programmer is required to specify the scheduling of routine calls through *sequence clauses*, if necessary. We refer to the enforced deterministic scheduling of data action as the second temporal firewall. Overall the combination of both the firewalls guarantee a deterministic scheduling of control and data.

4 The Combination of Styles

The engineering of embedded systems, let it be for control or signal processing, is concerned with several types of models of quite different nature such as state automata or periodic data flow. Though it sometimes may be possible to model parts of a system within one single paradigm, more often even parts of a system can be adequately modelled only by combining different aspects.

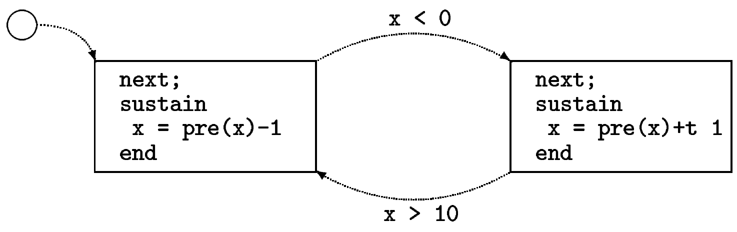
It is an outstanding feature of sE that it supports ostensible different computational paradigms for reactive behaviour. We present the styles using a taxonomy of "time" versus "state", and "discrete" versus "continuous".

	discrete state	continuous state

discrete time		`ufl := ( 0.0 -> 0.9*pre(ufl)`
		`+ 0.1*sensor`
		`) when ON;`
		`dfl := current(ufl)`
		`drv := 0.0 -> dx / dy.to_double;`

continuous time	`[[ await (ON.timestamp-now)>2sec;`
	`emit FROZEN;`
	`|| elapsed::=now+300millisec;`
	`await now > elapsed`
	`]]`
		`dy  := 0.0  -> flt - pre(flt);`
		`dx  := 0sec -> now - pre(now);`
		`drv := 0.0  -> dy / dx.to_double;`

The left upper square displays a notation for (hierarchical) automata, the right upper square a slight variant of LUSTRE data flow equations. Data flow equations are encapsulated by a "`sustain ...end`" context. These statements may be freeely used as in[1]



This is essentially the fine-grained integration we spoke of earlier.

The "continuous" time dimension is interpreted in a rather specific way in that we relate it to real time; the signal "`now`" provides a time stamp in terms of system time at the beginning of an instant (presently in micro seconds). Hence the difference "`now - pre(now)`" defines the real time measured between two instants, i.e. dx.

The unification of the idioms is achieved on the level of signals. We do not have the space to present the semantic model, however the essentials are thus:

[1]Note that only one flow definition should be apply at every instant to avoid non-determinism, hence the "`next`" which delays evaluation by one instant.

signal uniformly have a *presence* and a *value*. A value may only change if a signal is present. a signal may be present because it is emitted (as in "`emit FROZEN`"), or because it is specified as a periodic signal. The periodicity of the signal is specified in its declaration. The format

```
raising_edge :  signal(bool) is public at true
```

states that a signal "`raising_edge`" is present whenever the expression succeeding the "`at`" is true at an instant. The value of such a signal is constrained by a *flow definition*, e.g.

```
raising_edge = false -> x and not pre(raising_edge)
```

5 The Development Environment

sE targets the design of embedded software for small micro processors but scales to large systems. It is a strong typed object-oriented language. We have designed it in accordance with the language Eiffel. Design by contract, multiple inheritance and generic types are supported. Compared to Eiffel subclassing is restricted to introducing subtypes. The environment supports compilation, configuration, simulation, test, and verification of synchronous object-oriented programs. Behavioural descriptions may be edited in graphical or in textual form. Code generators for efficient and compact code in C and diverse hardware formats, e.g. Verilog, are available. For model checking, code is generated which is accepted either by the VIS or SMV model checkers. Specification of properties either in temporal branching time logic (CTL) and Past Time Logic (PTL) are supported. The compilation and also the optimisation of reactive behaviour is compositional and may be performed separately for each class. Thus the size of the model of an application can be reduced considerably. This enlarges the size of applications which can be validated by model checking. For validation also, so called, "synchronous observers" may be defined. An observer is a reactive program constructed to detect defective conditions. The non occurrence of a defect may be model checked for the application combined with the observer, or the observer program is executed in parallel to the application, thus serving as a watchdog.

6 Further Features and Outlook

sE supports clusters of distributed synchronous processes using blackboards as a generalisation of shared memory. We are about to modify syntax to Java. The resulting "sJ" system will offer a compilation of Java to C with mild restrictions (for efficiency) but with the add-on of all the reactive behaviour discussed above.

References

[1] C. André, Representation and Analysis of Reactive Behaviors: A Synchronous Approach, CESA'96, IEEE-SMC, Lille(F), 1996,

[2] G. Berry and G. Gonthier, The synchronous programming language Esterel: design, semantics, implementation, *Science of Computer Programming*, 19:87–152, 1992.

[3] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, The synchronous data flow programming language Lustre, *Proceedings of the IEEE*, 79(9):1305–1321, Sep. 1991.

[4] D. Harel, STATECHARTS: A Visual Formalism for Complex Systems, *Science of Computer Programming*, 8(3)231-274,1987

[5] L. Holenderski, and A. Poigné, Synchrony Workbench, In: R. Berghammer, Y. Lakhnech (eds.), *Tool Support for System Specification, Development, and Verification*, Advances of Computing Science, 1999

[6] P. Le Guernic, T. Gautier,M. Le Borgne,C. Le Maire, Programming Real-time Applications with SIGNAL, *Proceedings of the IEEE*, 79(9), Sept. 1991

[7] A. Poigné, and L. Holenderski, On the Combination of Synchronous Languages, In: W.P. de Roever (ed.), *Workshop on Compositionality, The Significant difference*, LNCS 1536, Springer, Heidelberg, pp. 490 - 514, 1998.

Functional Extension of Decision Diagrams in Practice

Harald Sack, Christoph Meinel
FB IV - Informatik
Universität Trier
D-54286 Trier
{sack,meinel}@uni-trier.de

a b s t r a c t

In computer aided design of very large scale integrated circuits (CAD for VLSI) Ordered Binary Decision Diagrams (OBDDs) [1] have been established as the state-of-the-art data structure. They are applied in synthesis as well as in formal verification of combinatorial or sequential designs. This is due to the fact that almost every design step can be mapped to the task of manipulating Boolean functions. For performing these tasks efficiently in an automated way with a computer, OBDDs are very well suited, because they are compact, efficiently to manipulate, and canonical, i.e. there exists a unique OBDD for every Boolean function. The compactness property of OBDDs holds for most Boolean functions that are used in practice, but unfortunately not for all. The multiplication of two binary encoded numbers can only be represented with an OBDD of exponential size related to the number of inputs. This restriction is responsible for the research and development of more general data structures, based on extensions of OBDDs.

Besides relaxing the ordering restriction [2], easing the read-once property of the input variables, or the usage of different decomposition types for Boolean functions, we are focusing on the extension of OBDDs with functional operator nodes, esp. Parity-OBDDs (POBDDs), i.e. OBDDs with additional operator nodes computing the Boolean parity of their successors [3]. By introducing parity nodes the representation has the potential of being more compact while on the other hand giving up canonicity. Therefore, the identification of two POBDDs representing the same Boolean function becomes an essential operation. We present an efficient probabilistic equivalence test for POBDDs that admits working with POBDDs in an professional environment [4]. Due to the fact that the size of Decision Diagrams crucially depends on the order of the input variables we show how to apply heuristics for POBDD minimization based on dynamic changes in the variable order and the relocation of parity operator nodes inside the data structure.

Many problems in practice require the transformation of symbolic variables to a binary encoding for getting accessible with OBDDs or POBDDs. Extending the OBDD data structure from the binary domain to a finite domain results in so called Multi-valued Decision Diagrams (MDDs) [5] and a binary encoding of symbolic variables is not necessary anymore. The already introduced POBDDs can now be extended towards Mod-p-Decision Diagrams (Mod-p-DDs), i.e. MDDs with additional operator nodes representing an integer addition modulo p, p -

prime. Such decision diagrams have a potential of being more space-efficient than MDDs. However, they are not a canonical representation and thus, the equivalence test of two Mod-p-DDs is more difficult than the test of two MDDs. To overcome this problem, we design a fast probabilistic equivalence test for Mod-p-DDs based on the transformation of integer functions represented by Mod-p-OBDDs to polynomials over a finite domain [6] and show how to apply heuristics for their minimization.

References

[1] R.E. Bryant, Graph-based algorithm for Boolean function manipulation, *IEEE Transactions on Computers* **C-35** No. 8 (1986), 677-691.

[2] J. Gergov, C. Meinel, Frontiers of feasible and probabilistic feasible Boolean manipulation with branching programs, *Proc. 10th Annual Symp. on Theoretical Aspects of Computer Science*, **665** of LNCS, Springer (1993), 576-585.

[3] J. Gergov, C. Meinel, Mod2-OBDDs: A data structure that generalizes EXOR-sum-of-products and Ordered Binary Decision Diagrams, *in Formal Methods in System Design* **8** (1996), Kluwer Academic Publishers, 273-282.

[4] C. Meinel, H. Sack, $\oplus$-OBDDs - a BDD structure for probabilistic verification, *in Proc. Workshop on Probabilistic Methods in Verification (PROBMIV'98)* (1998), 141-151.

[5] D. M. Miller, Multiple-valued logic design tools, *Proc. 23rd Int. Symp. on MVL* (1993), 2-11.

[6] E. Dubrova, H. Sack, Probabilistic verification of multiple-valued functions, Tech. Report 99-23, University of Trier, FB IV-Informatik, Nov. 1999.