Darren Cofer
Alessandro Fantechi (Eds.)

Formal Methods for Industrial Critical Systems

13th International Workshop, FMICS 2008
L'Aquila, Italy, September 2008
Revised Selected Papers

Springer

Lecture Notes in Computer Science 5596

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Darren Cofer   Alessandro Fantechi (Eds.)

Formal Methods for Industrial Critical Systems

13th International Workshop, FMICS 2008
L'Aquila, Italy, September 15-16, 2008
Revised Selected Papers

Springer

Volume Editors

Darren Cofer
Rockwell Collins
7805 Telegraph Rd. 100, Bloomington, MN 55438, USA
E-mail: ddcofer@rockwellcollins.com

Alessandro Fantechi
Università di Firenze, Dipartimento di Sistemi e Informatica
Via S. Marta 3, 50139 Firenze, Italy
E-mail: fantechi@dsi.unifi.it

Preface

The aim of the FMICS workshop series is to provide a forum for researchers who are interested in the development and application of formal methods in industry. In particular, these workshops are intended to bring together scientists and practitioners who are active in the area of formal methods and interested in exchanging their experiences in the industrial usage of these methods. These workshops also strive to promote research and development for the improvement of formal methods and tools for industrial applications.

The topics for which contributions to FMICS 2008 were solicited included, but were not restricted to, the following:

- Design, specification, code generation and testing based on formal methods
- Verification and validation of complex, distributed, real-time systems and embedded systems
- Verification and validation methods that address shortcomings of existing methods with respect to their industrial applicability (e.g., scalability and usability issues)
- Tools for the development of formal design descriptions
- Case studies and experience reports on industrial applications of formal methods, focusing on lessons learned or identification of new research directions
- Impact of the adoption of formal methods on the development process and associated costs
- Application of formal methods in standardization and industrial forums

The workshop included six sessions of regular contributions in the areas of model checking, testing, software verification, real-time performance, and industrial case studies. There were also three invited presentations, given by Steven Miller, Rance Cleaveland, and Werner Damm, covering the application of formal methods in the avionics and automotive industries.

Moreover, a panel was organized on the topic "Formal Methods in Commercial SW Development Tools." The aim of this panel was to promote discussion of current and foreseen applications of formal methods within model-based development frameworks that include formal analysis and generation methods for software design.

Out of the 36 submissions to FMICS 2008, 14 papers were accepted for presentation at the workshop, as well as two short presentations to serve as an introduction to the panel. We wish to thank the members of the Program Committee and the additional reviewers for their careful evaluation of the submitted papers. We also acknowledge the effort of all the members of the Program Committee in constructive discussions during the electronic program selection meeting. Special

thanks for the efforts devoted to the organization of the workshop go to the staff of the ASE 2008 conference, with which this workshop was co-located.

September 2008   Darren Cofer
                                                         Alessandro Fantechi

The FMICS 2008 workshop was hosted by the warm people of L'Aquila, Italy, and by the historic buildings of the city. Workshop participants had the occasion to stroll in the peaceful narrow streets of the old center, and to visit the magnificent monuments and churches that were built in the city several centuries ago.

On Monday, April 6, 2009, a severe earthquake hit the city, followed by more aftershocks in the following days. Hundreds of lives were lost, thousands were injured, and many houses and major historical buildings collapsed or were severely damaged. The vivid images in the memories of the workshop participants have been replaced by pictures of destruction from the media.

It is our hope that the proud, tireless and industrious people of the Abruzzo region will one day be able to bring back the city and the region to what the FMICS guests experienced.

April 2009   Darren Cofer
                                                         Alessandro Fantechi

Organization

FMICS 2008 was organized by the ERCIM Working Group on Formal Methods for Industrial Critical Systems.

Program Chairs

Darren Cofer   Rockwell Collins, USA
Alessandro Fantechi   Università di Firenze and ISTI-CNR, Italy

Program Committee

Maria Alpuente   Universidad Politècnica de Valencia, Spain
Alvaro Arenas   STFC RAL, UK
Lubos Brim   Masaryk University, Czech Republic
Wan Fokkink   Vrije Universiteit Amsterdam, The Netherlands
Patrice Godefroid   Microsoft Research, USA
Leszek Holenderski   Philips Research, The Netherlands
Roope Kaivola   Intel, USA
Stefan Kowalewski   RWTH Aachen, Germany
Stefania Gnesi   ISTI-CNR, Italy
Mark Lawford   McMaster University, Canada
Stefan Leue   University of Konstanz, Germany
Radu Mateescu   INRIA Rhone-Alpes, France
Charles Pecheur   Universitè Catholique de Louvain, Belgium
Francois Pilarski   Airbus, France
Ralf Pinger   Siemens, Germany
Murali Rangarajan   Honeywell, USA
Marco Roveri   IRST, Italy
Ina Schieferdecker   Fraunhofer FOKUS, Germany
Wilfried Steiner   TTTech, Austria

Additional Referees

Jiri Barnat   Masaryk University, Czech Republic
Robert Beers   Intel, USA
Dragan Bosnacki   Eindhoven University of Technology,
                               The Netherlands
Goetz Botterweck   Lero, Ireland
Marco Bozzano   Fondazione Bruno Kessler, Italy
Calame Jens   CWI, The Netherlands

Alessio Ferrari	Università di Firenze, Italy
Jan Friso Groote	Eindhoven University of Technology, The Netherlands
Jose Iborra	Universidad Politècnica de Valencia, Spain
Christophe Joubert	Universidad Politècnica de Valencia, Spain
Dmitry Korchemny	Intel, USA
Alexandre Korobkine	McMaster University, Canada
Frédéric Lang	INRIA Rhone-Alpes, France)
Giovanni Lombardi	ISTI-CNR, Italy
Franco Mazzanti	ISTI-CNR, Italy
Stefan Milius	Siemens, Germany
Francisco Javier Oliver	Universidad Politècnica de Valencia, Spain
Lucian Patcas	McMaster University, Canada
Bas Ploeger	Eindhoven University of Technology, The Netherlands
Erik Reeber	Intel, USA
Viktor Schuppan	Fondazione Bruno Kessler, Italy
Wendelin Serwe	INRIA Rhone-Alpes, France
Andrey Tchaltsev	Fondazione Bruno Kessler, Italy
Maurice H. ter Beek	ISTI-CNR, Italy
Francesco Tiezzi	Università di Firenze, Italy
Stefano Tonetta	Fondazione Bruno Kessler, Italy
Alicia Villanueva	Universidad Politècnica de Valencia, Spain
Michael Whalen	Rockwell Collins, USA
Anton Wijs	INRIA Rhone-Alpes, France

Table of Contents

Invited Presentations

Panel

Research Papers

Formal Methods for Critical Systems

Steven P. Miller

Rockwell Collins, USA

Abstract. Formal methods have traditionally been reserved for systems with requirements for extremely high assurance. However, the growing popularity of model-based development, in which models of system behavior are created early in the development process and used to auto-generate code, are making precise, mathematical specifications much more common in industry. At the same time, formal verification tools such as model checkers continue to grow more powerful. The convergence of these two trends opens the door for the practical application of formal verification techniques early in the life cycle for many systems. This talk will describe how Rockwell Collins has applied both theorem proving and model checking to commercial avionics and security systems to reduce costs and improve quality.

Model-Based Verification of Automotive Control Software

Rance Cleaveland

Department of Computer Science and
Fraunhofer USA Center for Experimental Software Engineering,
University of Maryland, USA

Abstract. This talk will report on the use of an approach, called Instrumentation Based Verification, for checking the correctness of models of control software given in Simulink® and Stateflow®. In IBV, engineers formalize requirements as so-called monitor models, whose purpose is to search executions of the main controller model for violations of required behavior. Testing is then performed on the instrumented controller model in order to check for the possibility of deviations between controller and requirements. Tools such as Reactis® provide automated support for conducting these activities, and the technique has attracted interest in automotive, aerospace and medical-device settings. The presentation will first review model-based development and IBV and their industrial motivations. It will then report on a project between the Fraunhofer Center for Experimental Software Engineering and a major automotive supplier on using IBV to verify models of an exterior-lighting control system.

Contract-Based Analysis of Automotive and Avionics Applications: The SPEEDS Approach

Werner Damm

OFFIS, Germany

Abstract. The Speeds project has developed a layered meta-model of heterogeneous rich components and standardized approaches for the integration of commercial industry standard modeling tools to assemble system-level design models with contract-baed interface specifications by combining models expressed in any authoring tool compliant to the integration standard, including Matlab-Simulink/Stateflow, Rhapsody, and Scade. It is currently integrating a range of analysis methods supporting interface compliance testing and dominance analysis between contracts expressed in extended automata model, subsuming timed automata. The presentation focuses on real-time analysis methods, and demonstrates a methodology for assessing realizability of end-to-end latencies at system level, exploring the design space of possible system configurations meeting vertical resource assumptions, and assessing compliance to such vertical assumptions based on distributed real-time schedulability analysis for FlexRay and CAN bus based target architectures.

Panel Discussion on Formal Methods in Commercial Software Development Tools

Alessandro Fantechi and Alessio Ferrari

DSI - Università di Firenze, Italy

Research on formal methods for software verification currently has decades of history within academia, and it is a central topic in software engineering. Nevertheless, usage of formal methods in companies is still at its embryonic stage. The reasons of this discrepancy have to be explored and solutions have to be found in order to increase adoption of formalized verification within industry.

A panel discussion was held at the closing of the 13th FMICS Workshop in order to investigate this topic with invited speakers from industry and academy. The panel was introduced by two short presentations on the theme, namely:

Model-driven software development: needs and experiences in rail automation, by Stefan Milius and Uwe Steinke, from Siemens AG.

Simulink Design Verifier vs. SPIN   A Comparative Case Study, by Florian Leitner and Stefan Leue from the University of Konstanz, Germany.

The participants to the panel were: Steven Miller of Rockwell Collins, Rance Cleaveland of Reactive Systems Inc., Werner Damm of OFFIS, Mark Lawford from McMaster University and Pedro Merino from the University of Malaga.

The first contribution comes from Mark Lawford. He states that the limited penetration of formal methods within industry is related to the extensive use of design tools which have no formal semantic. Tools like MATLAB are able to ease the specification part of the process, giving the developer a high level of freedom and flexibility; in part because there are no strong formal constraints inherent within these tools. While speeding up the specification and development phase of the process, this approach creates problems in the verification phase: if no formal specification is given, how can formal verification take place? Formal methods are then "bolted onto the side" of existing software development processes, effectively creating two models of the system - the informal one used by the developers and the formal one used by the verifiers. Maintaining consistency of these models becomes an additional burden. The way to achieve a higher rate of adoption of formal methods in the industry is to create integrated methods supported by integrated tools.

Rance Cleaveland has a different opinion: there is no need to disrupt the specification phase, since this is currently well established in companies, and it has a sufficient degree of formalization. The focus has to be raised to the process level, and strategies have to be explored to allow adaptation of the formal verification task into existing processes. A software process is built on the concept of *Who will press Which mouse button, When?*. In industries people live according to the tasks performed by other people, tasks that someone has to undertake, and this chain has to be well defined and strongly consistent. Any

development tool, whether formally based or not, in order to achieve adoption within a company needs to have the process implications explained: how the process can be more efficient, which activities can be removed, which tasks have to be added in order to accommodate the new tool. The problem is again *Who will press Which mouse button When?*, and with the new tool new mouse buttons come, and the path to reach the integration has to be explained. Concerning modeling tools the solution to this issue is currently quite stable. Every year there is a large number of publications on Model Based Development processes, and therefore industries interested in MBD have templates for understanding how to include the modeling tools into their development processes. In the case of verification tools this is not so clear. The incorporation of verification into the workflow is an issue that must be solved. In order to have a predictable development flow the company has to understand when the model checking task takes place, who has to be trained, and who will actually press the button *verify*. Furthermore, the relationship with the existing V&V activities, such as unit testing and integration testing, needs also to be fixed. Solutions to these problems have to be given by the tool vendors, to enable companies understanding when to use the verification tools, and which tool is more suitable for their needs.

Werner Damm enforces the point of view of Rance Cleaveland, stating that technology is nothing from the industrial perspective, and what is required is process integration. He also agrees on the fact that formal verification has to productively deal with industry standard design tools, such as Matlab. But this does not mean that one single modeling and analysis tool can be enough. The categories of design situations which a company has to work on are so diverse, ranging from system architecture design, requirement capturing, code generation, that there will never be a single formalism expressive enough, adaptable enough, to fit with this many different categories of design situation. Therefore, formal verification has to cope with commercial tools, but one tool will not be enough. This need for diversity is also enforced by the issue of the maintainability of the process chain in an industry. A solution to this, is to get to open-source tools and create a community of researchers supporting them. In order to realize this objective another step is required. The key resides in agreeing on a common suite of standard meta-models, and ensure interoperability according to the meta-models. Given this accepted standard, open-source solutions can be created with the support of the research community. On the other hand it is essential to create a market environment where companies are pushed to merge, in order to build a critical mass and in so doing, create commercial alternatives to monopoly situations. The point of arrival is an environment in which commercial solutions are completed by open-source ones, no monopolist is dominating the market and interoperability is guaranteed by the meta-models standard.

Rance Cleaveland agrees on the necessity of defining a standard meta-model, but he completely disagree with Werner Damm on the role of the open source community. His opinion is that, with open source, nobody's leverage depends on the selling of the software and therefore its quality tends to degrade. Therefore a standard is needed in order to let companies comply this standard with their tool.

On the other side Werner Damm points out that the question of maintenance of open source solutions is generally not so negative as described by Rance Cleaveland, since there are companies providing services around open source tools and therefore these companies would be in charge of providing maintenance. Though Pedro Merino agrees on the part that open source tools can play in disrupting monopoly, he is doubtful on the actual role that the meta-modeling standard can play. Pedro Merino has been also working on proposals for such meta-languages, however, according to the past experience with operating systems and network standards, he believes that defining a common language or infrastructure does not imply that software vendors will produce tools supporting it.

As a final argument for the discussion, Steven Miller shows how the interoperability pointed by Werner Damm can be achieved with the existing tools and technologies, and who are the subjects involved. In the Rockwell Collins Translation Framework, SCADE and MATLAB models are translated into the Lustre formal language, and then verified through different analysis tools, both commercial and open source. In the future this approach could be extended, with multiple modeling tools on one side, multiple analysis tools on the other side, but just one standardized modeling language; it being Lustre, SAL, or AP233. The subjects involved on the modeling tools side will probably be the commercial software vendors, while the subjects developing analysis and verification tools will more likely be academies and researchers.

LETO - A Lustre-Based Test Oracle for Airbus Critical Systems

Guy Durrieu[1], Hélène Waeselynck[2], and Virginie Wiels[1]

[1] ONERA, Centre de Toulouse,
2, Avenue E. Belin, BP 74025, 31077 Toulouse Cedex 4, France
{Guy.Durrieu,Virginie.Wiels}@cert.fr
http://www.cert.fr
[2] LAAS-CNRS, Université de Toulouse
7, Av du Colonel Roche, 31077 Toulouse Cedex 4, France
Helene.Waeselynck@laas.fr
http://www.laas.fr

Abstract. This paper presents an approach and an associated tool that have been proposed to automate the test oracle procedure of critical systems developed at Airbus. The target tests concern the early validation of the SCADE design and are performed in a simulated environment. The proposed approach and tool have been successfully applied to several Airbus examples.

Keywords: Test oracle, automation, formal methods, avionics.

1 Introduction

This paper presents the results of an R&D study conducted for Airbus, that aimed to increase their current level of test automation. The target tests concern the early validation of critical systems and are performed in a simulated environment.

The test oracle procedure was identified as a candidate for automation. A *test oracle* is a mechanism for determining whether or not a program produced correct outputs during testing. The availability of such a mechanism has been recognized as problematic for a long time [1]. Considering the importance of the problem, there has been comparatively little work in the testing literature to investigate adequate solutions. Still, some approaches have been proposed (see [2] for an overview), ranging from contract-based assertions to model-based test approaches where a behavioral model is used in both the generation of test cases and the determination of outputs. Such approaches are not widespread in industry yet. In practice, the test result analysis is almost always done manually, the tester playing the role of the oracle. Such is currently the case at Airbus.

We report here on the automated solution we have proposed to Airbus. It has been implemented as prototype tool, LETO (LustrE-based Test Oracle) to demonstrate the concept. Lustre [7] is a formal language for reactive systems. In the spirit, LETO is close to other approaches using Lustre-based synchronous

observers to check the test execution [3] [4]. However, the implementation is different due to the constraints put by the Airbus test environment. In particular, the test traces are analyzed off-line.

The paper starts by a presentation of the industrial context. Section 2 explains which test phase of the Airbus development process is addressed by our work. Section 3 summarizes the insights we gained from the analysis of current practice for this test phase. We identified high-level requirements for an automated oracle. Section 4 presents the oracle approach retained to fulfill these requirements. The approach has been implemented in LETO (Section 5) and experimented on real examples supplied by Airbus (Section 6). Concluding remarks are provided in Section 7.

2 Development Process and Target Test Phase

Figure 1 gives an overview of the Airbus development process. Three levels are distinguished: aircraft, system and equipment. The most critical systems, like flight control systems [9], are designed using the formal language SCADE [10]. A qualified code generator automatically generates most of the embedded code. Some validation activities are then shifted from the code to the SCADE design. Overall, the main V&V activities at Airbus are the following:

- Model tests: the considered system is validated using a desktop simulation environment called OCASIME. The environment provides a panel of commands representing possible pilot actions. It links the SCADE design to code simulating the aircraft movement and accounts for the redundancy levels in the fault-tolerant architecture. It thus makes it possible to simulate the complete system (computers, sensors, actuators, aircraft returns).
- Aircraft level simulation: several systems are validated in a simulated environment.
- Formal verification at code level [11]: Airbus uses abstract-interpretation based tools to verify non functional properties of programs (such as absence of run-time errors) and uses proof based tools to verify functional properties on the parts of the programs that are manually coded.
- Software integration tests.
- Lab tests: first tests with real equipments, on a single system or on several systems.
- Ground tests and Flight tests.

This study is focused on the model tests performed in the OCASIME environment, for the validation of the SCADE design of one system (e.g., the flight control system). Based on the detailed requirements, a test specification document (PGE, for *Programme Général des Essais* in the Airbus terminology) gives the functional test objectives to be covered. The testers then define concrete test scenarios to address the objectives, with possibly several scenarios per objective. The scenarios are implemented and run using OCASIME. The simulation API offers much flexibility to inject stimuli on data, and to select the variables to be
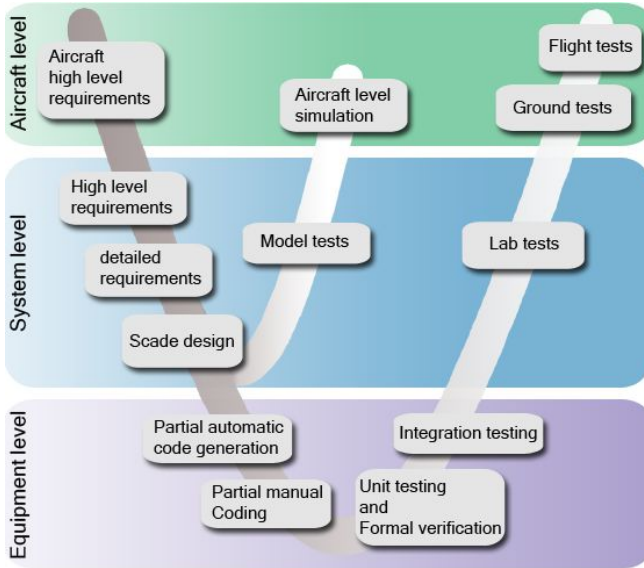
Fig. 1. Airbus development process

monitored. The test trace is then manually analyzed to determine whether the system passes or fails the test. To facilitate analysis, the tester can use the visual support of time diagrams showing the evolution of the monitored variables.

During the life cycle of a system, several versions of the design are to be tested. OCASIME offers facilities to automate the execution of regression tests. However, the analysis of the regression test results remains manual for a large part: as soon as the test trace is not strictly identical to the previously recorded one, manual analysis has to determine whether or not the observed discrepancies reveal a fault. In practice, it turns out that a number of discrepancies are observed, and that most of them are unimportant. For example, the bits of a floating point variable may not be identical from one execution to the other (difference in the value domain), but the corresponding values are actually very close within some epsilon tolerance. Or it may be observed that a Boolean variable changes from False to True at a slightly different simulation step (difference in the time domain), but once again this falls within some time tolerance interval. Hence, the analysis of results is time consuming not only for the first execution of the test, but also for all executions with successive design versions. A significant gain could be expected from the automation of the test oracle procedure.

3   Requirements for an Automated Test Oracle Procedure

Our analysis of the industrial context allowed us to identify some high-level requirements for an automated oracle approach.

In order to simplify the interfacing with the OCASIME environment, it was decided that the oracle tool would perform off-line analysis of test traces. In this way, the tool can be designed as an independent facility, with the only constraint that it should accept the XML file format of OCASIME traces.

The oracle checks should be defined at a higher level than raw expected values. Rather, the tester should be given the possibility of expressing properties relating the input and output values, in a declarative way. The properties are expected to formalize statements from the documents that are currently used as references for the manual analysis, namely the PGE and the detailed system requirements. In order to gain deeper insights into such properties, we analyzed examples extracted from three different functions of flight control systems: sidesticks, autopilot, and ADIRS (Air Data and Inertial Reference System). Related documentation was made available to us, including detailed functional requirements, PGE and concrete scenarios. We also had meetings and interviews with testers.

Analysis showed that a rich specification language is needed for the formalization of properties, with logical, arithmetic as well as temporal operators. An example, to be developed later in this paper, concerns the acquisition of ADIRS data. Some numerical input parameters are acquired from three independent sources. One of the test objectives is to test whether a single faulty source is correctly identified and locked out. The source is identified as faulty if the delivered value departs from the median of the three values for more than a predefined threshold, and does so during a given period of time. Once declared faulty (a Boolean alarm is set to true), the source is locked out (the alarm remains true forever, and the acquired value is no longer considered). The consolidated value of the acquired parameter is then the mean of the two remaining sources. For this test, it can be seen that the oracle checks involve a mix of numerical (calculation of difference from median, of a mean value), logical (determination of the mode for calculating the consolidated value) and temporal (persistence during a time interval, or forever) concerns.

The latter example is representative of a case where the output values are completely determined: the specified properties are actually formulas to derive outputs from the inputs. In other cases, the oracle checks may involve invariant properties to be satisfied whatever the specific values produced during the scenario. For example, in a test objective for sidesticks, a given output parameter has to remain within a safety range, and the calculation for the bounds of the range has to account for both the current and past inputs (numerical and temporal operators). We thus have two classes of oracle checks:

- Checks for equality. A sequence of expected outputs is computed by the oracle according to a specified formula, and compared to actual outputs. Such checks should explicitly accommodate a tolerance in both the value and time domains, as explained in the discussion of regression testing (Section 2 last paragraph).
- Assertion checks. Rather than computing expected values, the oracle checks for the validity of more general properties relating inputs and outputs.

The tester should have flexibility in using the logical, numerical and temporal operators for the specification of both classes of checks.

Finally, an important outcome of our analysis was the observation that a number of test objectives are repeatedly found with parameterization variants. This is due to symmetries in the Flight Control system. An obvious example of symmetry arises from redundancy in the fault-tolerant architecture: variants of a same test objective are found for the three primary computers and their COM/MON channels (see [9] for an overview of Airbus architectures). Also, inside a given computer, similar treatments may be attached to different inputs. For example, the test with a single faulty source is applicable to many ADIRS parameters. Moreover, for a target parameter, each of the sources may play the role of the faulty one. In all these examples, the same tests are repeated with some specialization to the current context (e.g., target computer, target input parameter). It would be convenient to define the corresponding oracle checks once and for all, in a generic way, with some mechanism to instantiate the checks with actual parameters.

The next section presents the approach proposed to fulfill these needs.

4   Proposed Approach

In order to automate the oracle procedure, it is necessary to formalize the test objectives described in the PGE. For this, we need to choose an expressive language and to allow for a form of genericity in the description of objectives.

4.1   Overview

We decided to have two kinds of components to describe test objectives, respectively called *generic test schemas* and *instantiation schemas*. A generic test schema is parameterized and describes a test objective in a generic way. An instantiation schema contains a list of instantiation cases for a generic schema, hence allowing its specialization to different concrete contexts.

In the approach we propose (see Figure 2), the tester writes the generic schemas and instantiation schemas from the PGE. It is a supplementary task but one that will allow the automation of the oracle and that will also facilitate capitalization and reuse of test schemas. The library represented on the figure contains functions that can be used for the definition of test schemas (examples of such functions will be presented in the following), and existing test schemas that can be reused.

The tester also defines the test scenarios that will be executed on the simulator. The automated oracle will decide on the result of the test from the trace file, the generic and instantiation schemas.

4.2   Test Schemas

In this subsection, we give the syntax of the generic and instantiation schemas. The generic schema has the following form:
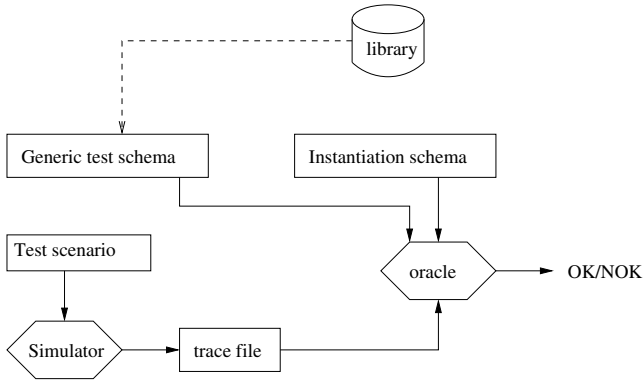
Fig. 2. Proposed approach

```
SCHEMA generic_schema
PARAMETERS : parameters
VARIABLES : internal variables
AUXILIARY : auxiliary computation
TEST : expected results of test
ENDSCHEMA
```

Parameters are the generic variables that will be instantiated in the instantiation schemas. Variables and auxiliary represent internal variables and computation that may be necessary for the expression of the test part. The test part describes the expected results of the test (the oracle checks to be performed).

The instantiation schema has the following form:

```
SCHEMA inst_schema
FOR A in {a1,a2,a3}
  FOR B in {b1,b2,b3}
      generic_schema(A,B)
  ENDFOR
ENDFOR ENDSCHEMA
```

For each generic parameter, a list of actual variables can be given. The schema above means that the generic schema will be considered 9 times (generic_schema(a1,b1), generic_schema(a1,b2), generic_schema(a1,b3), generic_schema(a2,b1), etc).

4.3   Language for the Test Section

The need. Concerning the "test" section, we observed that there was a need for logical, arithmetic and temporal operators. Our first intent was to define a

specific language for the test part, but after analysing the test objectives, we realised that existing languages could answer our needs, and proposed to use the Lustre language [7].

Using Lustre as a test oracle language. Using Lustre brings many advantages:

- it has a well defined semantics;
- it is well suited for the targeted type of systems, and widely used for designing critical embedded systems, in particular at Airbus: the SCADE language is actually a graphical language based on Lustre;
- there is an active working community around Lustre and the associated tools;
- it is able to handle all identified needs;
- it is a modular and hierarchical language, which could be useful to ease reuse and composition of schemas, although this aspect was not developed nor implemented in the current version of LETO;
- it is a formal declarative language, which is interesting for several reasons:
  - it makes the automation of the oracle easier,
  - from a more global perspective, it should be useful for establishing links with formal verification of properties that is experimented also at Airbus.

One possible inconvenience of choosing Lustre is the use of the same language for describing the system and for specifying the test objectives, this could hinder the necessary independence between development and verification. This inconvenience will have to be evaluated during experiments, it is however diminished by the fact that we use the textual version of the language that feels more declarative than the graphical one thanks to the the equational form.

Overview of Lustre – selected subset for the test oracle. As previously said, Lustre is a functional declarative language, the application field of which is the specification of control and signal processing systems. It belongs to the family of *synchronous languages* for which, different tools oriented towards the design of safety-critical embedded systems have been developed during the last decade [8], the Scade suite among others [10]. Synchrony divides time into *discrete instants*, and a synchronous program progresses according to successive *atomic reactions*, which implies a notion of *clock* defining the instants where reactions occur. Data are represented as infinite flows of typed values, each value corresponding to an instant, and operations on data are specified as flow equations. A Lustre program is thus a set of equations with neither a notion of control nor of sequentiality: the equation set is reevaluated at each instant on a data flow control basis (maybe concurrently). A specific function, called `pre`, allows us to get the value of a flow at the previous instant; another specific function, `->`, allows us to specify, if necessary, the value of a flow at the first instant.

Besides the arithmetic, logic, conditional and temporal operations required for writing equations Lustre includes:

- the notion of *assertion*, allowing us to specify a condition that must hold at all instants of the execution of a Lustre program;

- the notion of *node*, supporting modularity and hierarchy: this notion simply generalizes the notion of operator;
- a set of operations allowing *calculus on clocks*.

For a first version of our test language, we only needed the basic equation specification part of Lustre, and some additional basic temporal operators, allowing us to easily express temporal properties, such as `during`, the result of which is `true` when a given condition holds for a given time interval. The syntax of Lustre equations was embedded in the `TEST` part of the schemas. The notion of assertion was also included in our test language. Finally, syntactic constructs allowing the specification of value and time tolerances (see Section 5 "Oracle") were added to the basic equation syntax.

The semantics, however, is here slightly different from the semantics of standard Lustre, since we refer to the contents of a trace file instead of the execution of a program; that is, an instant corresponds to a simulation step (associated to a time value), and the value of a tested variable corresponds to the measured value at this time (which must agree with the expected value defined by the corresponding equation); the value returned by the `pre` function for a tested variable is the value of the tested variable at the previous measure; an assertion allows to verify that a given condition holds for all the measures recorded in the considered trace file.

In a later version, the test language would include the Lustre notion of node, which would be useful for reusing and composing schemas, or for extending the set of basic operators.

4.4   Illustration

We illustrate here the concepts of generic and instantiation schemas on a case study.

Case Study. We present here a case study extracted from the ADIRS PGE. ADIRS deals with the acquisition of several parameters necessary for the flight control system (such as altitude, speed, angle of attack). For each of these parameters, redundant sensors exist and a consolidated value is computed from the set of input values available. The treatment is the same for a certain number of parameters, we will consider the test objectives defined for the nominal case and the case where one of the input is out of range. Thanks to our generic approach, we were able to define a generic schema that was applicable for several parameters.

- Nominal case: the PGE says "Verify that the consolidated value is equal to the median of the three input values".
- Faulty case: the PGE says "Inject a divergence on each of the three input values (one at a time). For each case, verify that the consolidated value is equal to the average of the two remaining values".

The case study presented is of course only part of the real treatment. We have simplified it for the paper but our approach was able to handle the complete treatment and other complex examples (see Section 5).

Generic schemas. The generic schema for the nominal case is the following.

```
SCHEMA Nominal
----------------------------------------------
-- Test Schema for the nominal computation --
----------------------------------------------
PARAMETERS :
   input : real^3 ;                   -- the three inputs
   consolidated : real ;             -- the consolidated value
VARIABLES :
AUXILIARY :
TEST :
   consolidated = median(input[0], input[1], input[2]) ;
ENDSCHEMA Nominal
```

The schema has two generic parameters *input* which is an array of three inputs and *consolidated* the consolidated value. The consolidated value should be the median of the three input values.

The generic schema for the faulty case is the following.

```
SCHEMA Faulty
--------------------------------------------------------------------
-- In this schema, we suppose it is the first input that is faulty --
--------------------------------------------------------------------
-- the functions:
--          - median
--          - half_sum
-- are defined in a standard library
PARAMETERS :
   input : real^3 ;                   -- the 3 inputs
   s, confirm :  real ;              -- threshold, confirmation delay
   consolidated : real  ;            -- consolidated value
VARIABLES :
   fault : bool ;
   t  : bool ;
   m  : real ;
   hs : real ;
AUXILIARY :
   m = median(input[0], input[1], input[2]) ;
   t  = abs(input[0] - m) > s  ;
   hs = half_sum(input[1], input[2]) ;
   fault = false -> during(confirm, t) or pre fault;
TEST :
   consolidated = if fault then hs else m ;
ENDSCHEMA Faulty
```

For the faulty case, two generic parameters are added that represent the threshold and the confirmation delay. A fault is defined as the fact that the difference between the input value and the median is greater than the threshold during a time that is greater or equal to the confirmation delay.

In this schema, we suppose that it is the first input that is faulty, we will see in the instantiation schema how to apply this generic schema in order to handle the fault for each of the three inputs.

Tolerances. If necessary, tolerances can be added on real values or time for the change of boolean value, the syntax is the following:

```
fault = false -> during(confirm, t) or pre fault timetolerance 1;
consolidated = if fault then hs else m realtolerance 0.02;
```

Assertions. It is also possible to define assertions inside test schemas to verify that a given expression is true. In the ADIRS PGE, the test objectives include verification on the value of a certain number of booleans representing alarms in case of faults. The schema above gives an example of an assertion that states that the boolean should becomes true between time 4 and time 8 (between is an operator encoded in the tool).

```
SCHEMA Alarm
PARAMETERS :
   bool_alarm : bool ;
VARIABLES :
AUXILIARY :
TEST :
   assert between(4.0, 8.0, bool_alarm) ;
ENDSCHEMA Alarm
```

Instantiation Schemas. An instantiation schema for the nominal case could be the following.

```
SCHEMA InstantiatedNominal
FOR COMPUTER IN { "1", "2", "3" }
   FOR UNIT IN { COM, MON }
      FOR consolidated IN { ALPHA },
          input IN { [ALPHA1, ALPHA1, ALPHA3] },
          TRACEFILE IN
                {"${AUT_ORACLE}/ALPHA_P123.ras"}
              Nominal(input, consolidated)
      ENDFOR
   ENDFOR
ENDFOR ENDSCHEMA InstantiatedNominal
```

The instantiation schema states that the generic schema will be applied for the parameter alpha (angle of attack), for three primary computers and two units of each computer.

An instantiation schema for the faulty case could be the following.

```
SCHEMA InstantiatedFaulty
FOR TRACEFILE IN {"${AUT_ORACLE}/Faulty_ALPHA_P123.ras" }
   FOR s IN { 3.6 }
      FOR confirm IN { 100 }
         FOR consolidated IN { ALPHA }
               FOR COMPUTER IN { "1", "2", "3" }
                     FOR UNIT IN { COM, MON }
                           FOR input IN { [ALPHA1, ALPHA2, ALPHA3],
                                          [ALPHA2, ALPHA1, ALPHA3],
                                          [ALPHA3, ALPHA1, ALPHA2]  },
                                 Faulty(input, s, confirm, consolidated)
                           ENDFOR
                     ENDFOR
               ENDFOR
         ENDFOR
      ENDFOR
   ENDFOR
ENDFOR
ENDSCHEMA InstantiatedFaulty
```

In addition to the different computers and units as before, this instantiated schema allows the application of the generic schema to handle the divergence of each of the three inputs, as specified in the PGE.

5   Implementation of the Approach

An experimental oracle prototype, based on the principles described in the previous sections, was developed in Java. Figure 3 shows the main features of this prototype, called LETO (LustrE-based Test Oracle). The main components are briefly described below.

Schema Parsing. The oracle prototype requires as inputs:
- a file containing the generic test schema, which specifies the expected relations between the measured flows.
- a file containing the instantiation cases, which binds the formal names used within the generic schema to the real names found in the trace files, and allow to iterate the use of a generic schema on different measure sets.

JavaCC [5][6] has been used for the development of a parser accepting the syntax of the generic test schemas and instantiation cases. JavaCC is a parser generator which takes as input a grammar at EBNF (*Extended Backus Naur Form*) format with additional syntactic and semantic predicates, and produces a *LL(k)* recursive descending parser implemented as a finite state automaton. JavaCC also provides tools for automatic construction of abstract syntax trees (AST).

The schema parser is launched by the oracle, with the name of the schema files to be parsed; the AST obtained after parsing are sent back for processing to the oracle core.
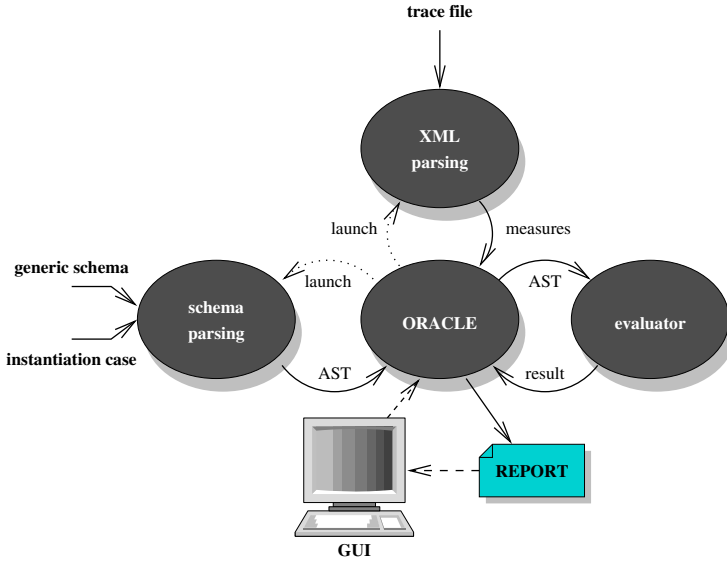
Fig. 3. The experimental oracle prototype

XML Parsing. In addition to test schema and instantiation cases, LETO obviously accesses the trace files to be checked. These trace files adopt an XML format, with a header specifying the registered signals and a sequence of records containing the measured values at given instants.

The XML parser was developed using the SAX2 toolkit. The result is a set of callback methods, activated when encountering the associated XML tags, which in turn call the appropriate processing functions within the oracle. The XML parser is launched by the oracle, with the name of the trace file to be analyzed.

Oracle. The oracle core performs three functions:

- it analyzes the generic test schema, the instantiation cases, and successively carries out the specified instantiations for the set of generic flows used in the generic test schema. The analysis of the generic test schema implies an analysis of dependencies in order to get an evaluation order for the specified equations.
- for each instantiation it launches the parsing of the specified trace file; for each measure record, it performs the verification specified in the test schema with help of the evaluator, that is, given the current set of measures, it computes the expected value of the tested flows and compares the result to their measured value with some specified tolerance (see below); it is also able to check that a given condition (or assertion) is verified on the whole trace file; if necessary, it issues warning messages. For each measure record, it manages the current and previous values of each flow.
- at the end of each iteration, it produces a report summarizing the results obtained while processing the trace file.

As suggested above, some differences between the expected and measured values may be meaningless from the test validity point of view; this can be taken into account by associating *tolerances* to the relevant equation. Two kinds of tolerance are considered:

- *value tolerance*: if the difference between the expected and measured values is less than a given bound, no discrepancy message is issued; this kind of tolerance is mainly used for floating point measures (e. g. a sensor measure).
- *time tolerance*: if the duration of the discrepancy between the expected and measured values is less than a given number of clock ticks, no discrepancy message is issued; this kind of tolerance is mainly used for boolean measures (e. g. an alarm signal).

Evaluator. Given a set of value for the flows defined in the generic test schema, the evaluator is able to compute, on request of the oracle core, the results of any arithmetic and logical expression specified in the schema.

Graphical User Interface. In order to help in the design and experimentation of generic test schemas and instantiation cases, a graphical user interface has
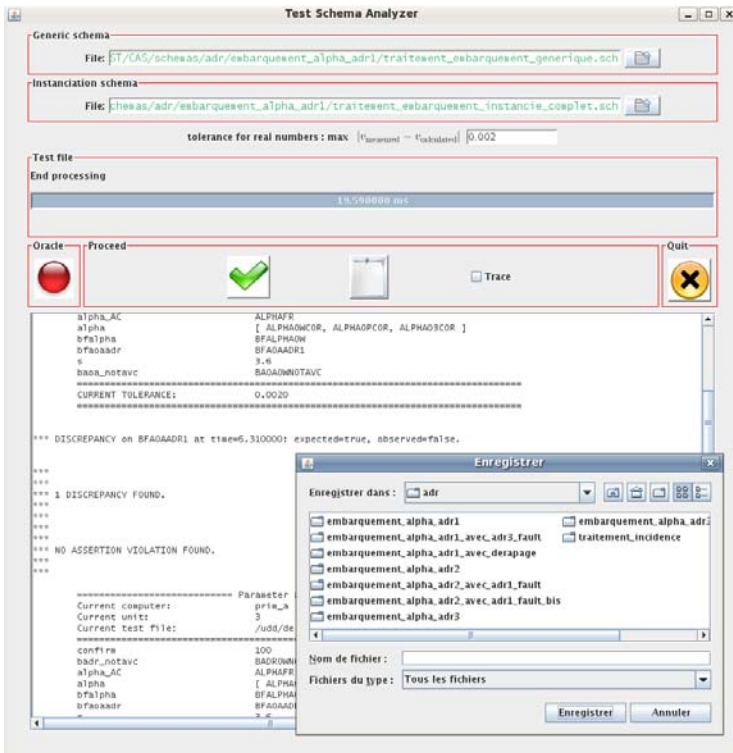
Fig. 4. Screen shot of the LETO graphical user interface

I :   instantiation loop; for each instantiation case the R loop is processed.
R: record loop; for a given instantiation, the set of test records is processed.
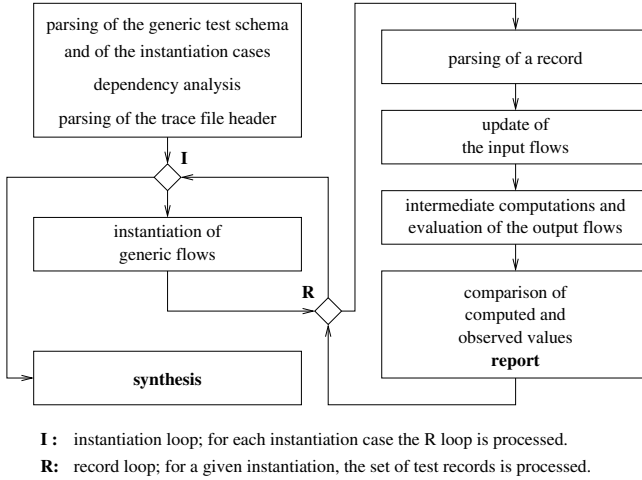
Fig. 5. General algorithm of the oracle prototype

been developed, allowing us to easily specify the names of the involved files, global tolerance, to start the analysis process and to display the results. However, the use of this graphical interface is not mandatory, and the normal mode for exploiting the automated interface is the batch mode.

Figure 4 shows a screen shot of the graphic user interface of the oracle prototype.

The upper part allows us to specify the generic schema and the instantiation cases which are to be used. The middle part contains several buttons for starting the analysis, enter debug modes and save the log pane contents located in the lower part of the GUI. At the end of the analysis, the visual indicator located at the left of the middle part is green if no discrepancy has been detected, and red otherwise.

Figure 5 summarizes the general algorithm of the oracle prototype.

6   Experiments and Results

The approach described in this paper was successfully used on several industrial cases coming from the avionic world, specifically from the AIRBUS A380. The checked trace files, for example, were dealing with:

- the Air Data & Inertial Reference System: it was to be verified that in the trace file the right value was correctly selected by the logic managing the redundancy in each operational case (nominal, one or more faulty sources).
- the pilot environment (side sticks): it was to be verified that the electrical signal generated by the command devices always evolved within the expected domain.

The experiments carried out showed a good proximity between the values computed by the oracle and the values found in the trace files. Few discrepancies were detected by the oracle, which all were explained either by a too restrictive tolerance or by a slight time shift between the oracle and the trace file; once these discrepancies explained, it was possible to parameterize the oracle (tolerances on value and on time) in order to avoid irrelevant alerts.

It was also important from the project point of view to evaluate the perception of the tool, and more specifically of the test language, by Airbus people in charge of carrying out the test campaigns. This was done through some "practical exercises" which allowed several of these Airbus test experts to experiment the Lustre specifying style as well as the whole test automation approach on the above practical industrial cases, and to observe the potential effort savings.

7   Conclusion

We have proposed an approach to automate the test oracle procedure for a given industrial context. The approach has been implemented in a tool, applied to a set of representative case studies and has been successful in answering the identified needs. In particular, it is very useful to handle non regression tests in an efficient way: our approach verifies the simulation results against expected I/O properties (which are quite stable) and not against the raw results of the previous simulations. An industrial deployment of the approach would now necessitate further methodological work: choice of the best granularity for the test schemas, definition of the library of functions necessary for the test schemas definition.

The existing industrial context imposed some constraints on the solutions that could be proposed. For example, the automated oracle checks had to be done off-line. They could not be done during test execution, for technical reasons regarding the implementation of the OCASIME simulator. However, the approach and tool we proposed may be used in other industrial contexts. The specific part is the format of the trace files, but given it is provided in an XML format, the necessary changes to the LETO tool are not prohibitive. The principle of the test schemas and the chosen language for these schemas is intended to be well adapted to critical embedded systems in general. The test schemas are generic and can be instantiated for several concrete cases. They also establish a formal traceability between the test specification document (e.g., PGE in the Airbus context) and the corresponding tests. By essence, test schemas are less ambiguous and more informative than informal test specification documents. Their definition requires supplementary work but this work can be capitalized and reused.

Several themes are interesting for future work. Composition operations could be investigated to further facilitate the definition of test schemas from existing ones. We are also studying a similar notion of schemas for the activation part of the test: generic and instantiated schemas could be used to formalize, store and reuse activation functions for test scenarios. Finally and in a more distant perspective, existing lustre-based tools could allow automatic generation of test scenarios.

References

1. Weyuker, E.: On Testing Non-Testable Programs. The Computer Journal 25(4), 465–470 (1982)
2. Baresi, L., Young, M.: Test oracles. Technical Report CIS-TR01-02, Univ. of Oregon (2001)
3. Raymond, P., Weber, D., Nicollin, X., Halbwachs, N.: Automatic Testing of Reactive Systems. In: 19th IEEE Real-Time Systems Symposium (RTSS 1998), pp. 200–209. IEEE CS Press, Los Alamitos (1998)
4. Parissis, I., Vassy, J.: Strategies for Automated Specification-based Testing of Synchronous Software. In: 16th IEEE Int. Conf. on Automated Software Engineering (ASE 2001), pp. 364–367. IEEE CS Press, Los Alamitos (2001)
5. Copeland, T.: Generating Parsers with JavaCC. Centennial Books, Alexandria (2007)
6. javacc Project home, `http://javacc.dev.java.net/`
7. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous dataflow programming language LUSTRE. Proceedings of the IEEE 79(9), 1305–1320 (1991)
8. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Le Guernic, P., de Simone, R.: The Synchronous Languages 12 Years Later. Proceedings of the IEEE 91(1) (January 2003)
9. Traverse, P., Lacaze, I., Souyris, J.: Airbus Fly-By-Wire: A Total Approach to Dependability. In: Building the Information Society, 18th IFIP World Computer Congress (WCC 2004), pp. 191–212. Kluwer, Dordrecht (2004)
10. Esterel Scade Suite,
    `http://www.esterel-technologies.com/products/scade-suite`
11. Duprat, S., Souyris, J., Favre-Flix, D.: Formal Verification Workbench for Airbus Avionics Software. In: Embedded Real-Time Software (ERTS 2006). SIA (2006)

Extending Structural Test Coverage Criteria for LUSTRE Programs with Multi-clock Operators

Virginia Papailiopoulou, Laya Madani, Lydie du Bousquet, and Ioannis Parissis

University of Grenoble - Laboratoire d'Informatique de Grenoble
BP 72 - 38402 Saint-Martin d'Hères Cedex France
{Virginia.Papailiopoulou,Laya.Madani,Lydie.du-Bousquet,
Ioannis.Parissis}@imag.fr

Abstract. LUSTRE is a formal synchronous declarative language widely used for modeling and specifying safety-critical applications in the fields of avionics, transportation or energy production. Testing this kind of applications is an important and demanding task during the development process. It mainly consists in generating test data and measuring the achieved coverage. A hierarchy of structural coverage criteria for LUSTRE programs have been recently defined to assess the thoroughness of a given test set. They are based on the operator network, which is the graphical representation of a LUSTRE program and depicts the way that input flows are transformed into output flows through their propagation along the program paths. The above criteria definition aimed at demonstrating the opportunity of such a coverage assessment approach but doesn't deal with all the language constructions. In particular, the use of multiple clocks has not been taken into account. In this paper, we extend the criteria to programs that use multiple clocks. Such an extension allows for the application of the existing coverage metrics to industrial software components, which usually operate on multiple clocks, without negatively affecting the complexity of the criteria.

1   Introduction

Synchronous software is normally part of safety-critical applications in such domains as avionics, transportation and energy. Formal specification is usually required to model the system behavior along the different levels of the development process. Such a specification not only describes the correct function of the system but also it defines the conditions under which that correct function is reached. That specification can be further used to automatically generate test data.

Several programming languages have been proposed to specify and implement synchronous applications, such as Esterel [2], Signal [8] or Lustre [5,1]. LUSTRE is a declarative, data-flow language, which is devoted to the specification of real-time applications. It provides formal specification and verification facilities and ensures efficient C code generation. It is based on the synchronous approach which demands that the software reacts to its inputs instantaneously. In practice, that means that the software reaction is sufficiently fast so that every change in the external environment is taken into account. As soon as the

order of all the events occurring both inside and outside the program is specified, time constraints describing the behavior of a synchronous program can be expressed [6]. These characteristics make it possible to efficiently design and model synchronous systems.

A graphical tool dedicated to the development of critical embedded systems and often used by industries and professionals is SCADE (Safety Critical Application Development Environment). SCADE is a graphical environment used in the development of safety-critical embedded software. It is based on the LUSTRE language and it allows the hierarchical definition of the system components and the automatic code generation. From the SCADE functional specifications, C code is automatically generated, though this transformation (SCADE to C) is not standardized. This graphical modeling environment is used mainly in the aerospace field (Airbus, DO-178B); however its capabilities serve also transportation, automotive and energy.

In major industrial applications, the testing process usually consists in producing test cases based on the functional requirements of the system under test. Test objectives and test data are constructed with regard to the system requirements and the coverage evaluation is applied on the generated C code. For programs written in sequential languages, several adequacy criteria have been presented in the past, such as path/branch coverage criteria, LCSAJ (Linear Code Sequence And Jump) [10] and MC/DC (Modified Decision Condition Coverage).

These criteria are not conformed with the synchronous paradigm and cannot be applied on LUSTRE programs to assess how thoroughly the produced test data have tested the corresponding specification. Furthermore, it is difficult to formally relate the coverage measurement results with the system specification and the test objective. To deal with this problem, especially designed structural coverage criteria for LUSTRE programs have been proposed [7]. Although these criteria are comparable to the existing data-flow based criteria [9,3], they are not the same. They aim at defining intermediate coverage objectives and estimating the required test effort towards the final one. These criteria are based on the notion of the *activation condition* of a path, which informally represents the propagation of the effect of the input edge through the output edge.

However, the above coverage criteria can be applied only on specifications that are defined under a unique global clock. The global clock is a boolean flow that always values *true* and defines the frequency of the program execution cycles. Other, slower, clocks can be defined through boolean-valued flows. They are mainly used to prevent useless operations of the program and to save computational resources by forcing some program expressions to be evaluated strictly on specific execution cycles. Thus, nested clocks may be used to restrict the operation of certain flows when this is necessary, without affecting at the same time the rest of the program variables. In LUSTRE, using multiple clocks is made through two specific operators, `when` and `current`. In this paper, we propose the extension of the existing coverage criteria taking into account the `when` and `current` operators. In fact, we define the activation conditions for the paths containing these operators in order that the coverage criteria are applicable on

such paths. The complexity of the criteria, in terms of the cost of computing the paths and their activation conditions, is not increased.

The paper is structured in three main sections. Section 2 provides a brief overview of the essential concepts on LUSTRE language. Section 3 presents the existing coverage criteria for LUSTRE programs while in section 4 we thoroughly demonstrate their extension to the use of multiple clocks. Section 5 concludes and shows some perspectives for future work.

2   Overview of LUSTRE

LUSTRE [5] is a data-flow language. Contrary to imperative languages which describe the control flow of a program, LUSTRE describes the way that the inputs are turned into the outputs. Any variable or expression is represented by an infinite sequence of values and take the n-th value at the n-th cycle of the program execution, as it is shown in Figure 1. At each tick of a global clock, all inputs are read and processed simultaneously and all outputs are emitted, according to the synchrony hypothesis.

A LUSTRE program is structured into nodes. A node is a set of equations which define the node's outputs as a function of its inputs. Each variable can be defined only once within a node and the order of equations is of no matter. Specifically, when an expression E is assigned to a variable X, $X=E$, that indicates that the respective sequences of values are identical throughout the program execution; at any cycle, X and E have the same value. Once a node is defined, it can be used inside other nodes like any other operator.

The operators supported by LUSTRE are the common arithmetic and logical operators (`+`, `-`, `*`, `/`, `and`, `or`, `not`) as well as two specific temporal operators: the *precedence* (`pre`) and the *initialization* (`->`). The `pre` operator introduces to the flow a delay of one time unit, while the `->` operator -also called *followed by* (`fby`)- allows the flow initialization. Let $X = (x_0, x_1, x_2, x_3, \ldots)$ and $(e_0, e_1, e_2, e_3, \ldots)$ be two LUSTRE expressions. Then `pre`(X) denotes the sequence $(nil, x_0, x_1, x_2, x_3, \ldots)$, where nil is an undefined value, while X`->`E denotes the sequence $(x_0, e_1, e_2, e_3, \ldots)$.
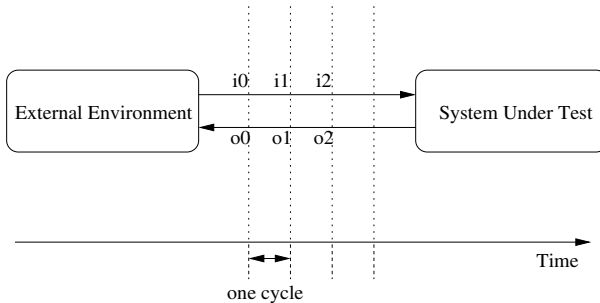
Fig. 1. Synchronous software operation

```
node Never(A: bool) returns (never_A: bool);
let
  never_A = not(A) -> not(A) and pre(never_A);
tel;
```

	c_1	c_2	c_3	c_4	...
A	*false*	*false*	*true*	*false*	...
never_A	*true*	*true*	*false*	*false*	...

Fig. 2. Example of a LUSTRE node

LUSTRE does not support loops (operators such as `for` and `while`) nor recursive calls. Consequently, the execution time of a LUSTRE program can be statically computed and the satisfaction of the synchrony hypothesis can be checked.

A simple LUSTRE program is given in Figure 2, followed by an instance of its execution. This program has a single input boolean variable and a single boolean output. The output is *true* if and only if the input has never been *true* since the beginning of the program execution.

2.1   Operator Network

The transformation of the inputs into the outputs in a LUSTRE program is done via a set of operators. Therefore, it can be represented by a directed graph, the so called *operator network*. An operator network is a graph with a set of N operators which are connected to each other by a set of $E \subseteq N \times N$ directed edges. Each operator represents a logical or a numerical computation. With regard to the corresponding LUSTRE program, an operator network has as many input edges (respectively, output edges) as the program input variables (respectively, output variables).

Figure 3 shows the corresponding operator network for the node of Figure 2.
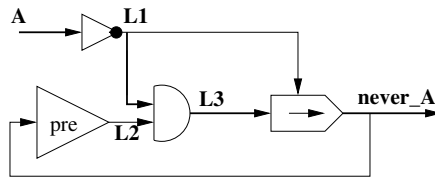
Fig. 3. The operator network for the node `Never`

An operator represents a data transfer from an input edge into an output edge. There are two kinds of operators:

a) the basic operators which correspond to a basic computation and

b) the compound operators which correspond to the case where in a program, a node calls another node[1].

---

[1] For the time being, we only consider basic operators.

A basic operator is denoted as $\langle e_i, s \rangle$, where e_i, $i = 1, 2, 3, \ldots$, stands for its inputs edges and s stands for the output edge.

2.2   Clocks in LUSTRE

In LUSTRE, any variable and expression denotes a flow, i.e. each infinite sequence of values is defined on a clock, which represents a sequence of time. Thus, a flow is the pair of a sequence of values and a clock.

The clock serves to indicate when a value is assigned to the flow. That means that a flow takes the n-th value of its sequence of values at the n-th time of its clock. Any program has a cyclic behavior and that cycle defines a sequence of times, i.e. a clock, which is the *basic clock* of a program. A flow on the basic clock takes its n-th value at the n-th execution cycle of the program. Slower clocks can be defined through flows of boolean values. The clock defined by a boolean flow is the sequence of times at which the flow takes the value *true*.

Two operators affect the clock of a flow: `when` and `current`.

`when` is used to **sample** an expression on a slower clock. Let E be an expression and B a boolean expression with the same clock. Then `X=E when B` is an expression whose clock is defined by B and its values are the same as those of E's only when B is *true*. That means that the resulting flow X has not the same clock with E or, alternatively, when B is *false*, X is not defined at all.

`current` operates on expressions with different clocks and is used to **project** an expression on the immediately faster clock. Let E be an expression with the clock defined by the boolean flow B which is not the basic clock. Then `Y=current(E)` has the same clock as B and its value is the value of E at the last time that B was *true*. Note that until B is *true* for the first time, the value of Y will be *nil*.

The sampling and the projection are two complementary operations: a projection changes the clock of a flow to the clock that the flow had before its last sampling operation. Trying to project a flow that was not sampled produces an error. Table 1 provides the use of the two temporal LUSTRE operators in more details.

Table 1. The use of the operators `when` and `current`

E	e_0	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	...
B	*false*	*false*	*true*	*false*	*true*	*false*	*false*	*true*	*true*	...
X=E when B			$x_0 = e_2$		$x_1 = e_4$			$x_2 = e_7$	$x_3 = e_8$	...
Y=current(E)	$y_0 = nil$	$y_1 = nil$	$y_2 = e_2$	$y_3 = e_2$	$y_4 = e_4$	$y_5 = e_4$	$y_6 = e_4$	$y_7 = e_7$	$y_8 = e_8$	...

An example [4] of the use of clocks in LUSTRE is given in Figure 4.

The LUSTRE node `mux` receives as input the signal m. Starting from this input value when the clock c is true, the program counts backwards until zero; from this moment, it restarts from the current input value and so on.

```
node mux(m:int) returns (c:bool; y:int);
var (x:int) when c;
let
 y = if c then current(x) else pre(y)-1;
 c = true -> (pre(y)=0);
 x = m when c;
tel;
```
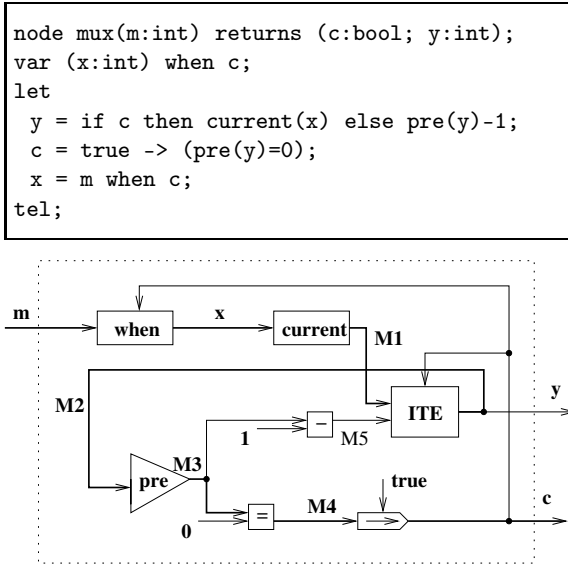

Fig. 4. The mux example and the corresponding operator network

3   Coverage Criteria for LUSTRE Programs

3.1   Activation Conditions

Given an operator network N, paths can be defined in the program. That is, the possible directions of flows from the input through the output. More formally, a path is a finite sequence of edges $\langle e_0, e_1, \ldots, e_n \rangle$, such that for $\forall i \epsilon [0, n-1]$, e_{i+1} is a successor of e_i in N. A *unit path* is a path with two successive edges. For instance, in the operator network of Figure 3, there can be found the following paths.

$p_1 = \langle A, L_1, never\_A \rangle$
$p_2 = \langle A, L_1, L_3, never\_A \rangle$
$p_3 = \langle A, L_1, never\_A, L_2, L_3, never\_A \rangle$
$p_4 = \langle A, L_1, L_3, never\_A, L_2, L_3, never\_A \rangle$

Obviously, one could discover infinitely many paths in an operator network depending on the number of cycles repeated in the path (i.e. the number of pre operators in the path). However, we only consider paths of finite length by limiting the number of cycles. That is, a path of length n is obtained by concatenating a path of length *n-1* with a unit path (of length 2). Thus, beginning from unit paths, longer paths could be built; a path is finite if it contains no cycles or if the number of cycles is limited.

A boolean LUSTRE expression is associated with each pair $\langle e, s \rangle$, denoting the condition on which the data flows from the input edge e through the output s.

Table 2. Activation conditions for all LUSTRE operators

Operator	Activation condition
$s = NOT\,(e)$	$AC\,(e, s) = true$
$s = AND\,(a, b)$	$AC\,(a, s) = not\,(a)\ or\ b$
	$AC\,(b, s) = not\,(b)\ or\ a$
$s = OR\,(a, b)$	$AC\,(a, s) = a\ or\ not\,(b)$
	$AC\,(b, s) = b\ or\ not\,(a)$
$s = ITE\,(c, a, b)$	$AC\,(c, s) = true$
	$AC\,(a, s) = c$
	$AC\,(b, s) = not\,(c)$
relational operator	$AC\,(e, s) = true$
$s = FBY\,(a, b)$	$AC\,(a, s) = true\ \text{->}\ false$
	$AC\,(b, s) = false\ \text{->}\ true$
$s = PRE\,(e)$	$AC\,(e, s) = false\ \text{->}\ pre\,(true)$

This condition is called *activation condition*. The evaluation of the activation condition depends on what kind of operators the paths is composed of. Informally, the notion of the activation of a path is strongly related to the propagation of the effect of the input edge through the output edge. More precisely, a path activation condition shows the dependencies between the path inputs and outputs. Therefore, the selection of a test set satisfying the paths activation conditions in an operator network leads to a notion for the program coverage. Since covering all the paths in an operator network could be impossible, because of their potentially infinite number and length, in our approach, coverage is defined with regard to a given path length.

Table 2 summarizes the formal expressions of the activation conditions for all LUSTRE operators (except for `when` and `current` for the moment). In this table, each operator `op`, with the input e and the output s, is paired with the respective activation condition $AC\,(e, s)$ for the unit path $\langle e, s \rangle$. Noted that some operators may define several paths through their output, so the activation conditions are listed according to the path inputs.

Let us consider the path $p_2 = \langle A, L_1, L_3, never\_A \rangle$ in the corresponding operator network for the node `Never` (Figure 3). The condition under which that path is activated is represented by a boolean expression showing the propagation of the input A through the output $never\_A$. To calculate its activation condition, we progressively apply the rules for the activation conditions of the corresponding operators according to Table 2[2]. Starting from the end of the path, we reach

---

[2] In the general case (path of length n), the path p containing the `pre` operator is activated if its prefix p' is activated at the previous cycle of execution, that is $AC\,(p) = false\ \text{->}\ pre\,(AC\,(p'))$. Similarly in the case of the initialization operator `fby`, the given activation conditions are respectively generalized in the forms: $AC\,(p) = AC\,(p')\ \text{->}\ false$ (i.e. the path p is activated if its prefix p' is activated at the initial cycle of execution) and $AC\,(p) = false\ \text{->}\ AC\,(p')$ (i.e. the path p is activated if its prefix p' is always activated except for the initial cycle of execution).

the beginning, moving one step at a time along the unit paths. Therefore, the necessary steps would be the following:

$AC\,(p_2) = false \rightarrow AC\,(p')$, where $p' = \langle A, L_1, L_3 \rangle$
$AC\,(p') = not\,(L_1)\ or\ L_2\ and\ AC\,(p'') = A\ or\ pre\,(never\_A)\ and\ AC\,(p'')$, where $p'' = \langle A, L_1 \rangle$
$AC\,(p'') = true$

After backward substitutions, the boolean expression for the activation condition of the selected path is:

$$AC\,(p_4) = false \rightarrow A\ or\ pre\,(never\_A).$$

In practice, in order for the path output to be dependent on the input, either the input has to be *true* at the current execution cycle or the output at the previous cycle has to be *true*; for the first cycle of the execution, the input needs to be *false*.

3.2   Coverage Criteria

A LUSTRE/SCADE program is compiled into an equivalent C program. Provided that the format of the generated C code depends on the compiler, it is hard to fix a formal relation between the original LUSTRE program and the final C one. In addition, major industrial standards, such as DO-178B in the avionics field, demand coverage to be measured on the generated C code. Therefore, three coverage criteria specifically defined for LUSTRE programs have been proposed [7]. They are specified on the operator network according to the length of the paths and the input variable values.

Let $\mathcal{T}$ be the set of test sets (input vectors) and $P_n = \{p | length(p) \leq n\}$ the set of all paths in the operator network whose length is inferior or equal to n. Hence, the following families of criteria are defined for a given and finite order $n \geq 2$. The input of a path p is denoted as $in\,(p)$ whereas a path edge is denoted as e.

1. **Basic Coverage Criterion (BC).** This criterion is satisfied if there is a set of test input sequences, $\mathcal{T}$, that activates at least once the set P_n. Formally, $\forall p \in P_n,\ \exists t \in \mathcal{T}\colon AC\,(p) = true$. The aim of this criterion is basically to ensure that all the dependencies between inputs and outputs have been exercised at least once. In case that a path is not activated, certain errors such as a missing or misplaced operator could not be detected.
2. **Elementary Conditions Criterion (ECC).** In order that an input sequence satisfies this criterion, it is required that the path p is activated for both input values, *true* and *false* (taking into account that only boolean variables are considered). Formally, $\forall p \in P_n,\ \exists t \in \mathcal{T}\colon in\,(p) \wedge AC\,(p) = true$ and $not\,(in\,(p)) \wedge AC\,(p) = true$. This criterion is stronger than the previous one in the sense that it also takes into account the impact that the input value variations have on the path output.

3. **Multiple Conditions Criterion (MCC).** In this criterion, the path output depends on all the combinations of the path edges, also including the internal ones. A test input sequence is satisfied if and only if the path activation condition is satisfied for each edge value along the path. Formally, $\forall p \in P_n, \forall e \in p, \exists t \in \mathcal{T}: e \wedge AC(p) = true$ and $not(e) \wedge AC(p) = true$.

The above criteria form a hierarchical relation: MCC satisfies all the conditions that ECC does, which also subsumes BC.

4 Extension of Coverage Criteria to `when` and `current` Operators

The aim of this paper is to extend the above criteria in order to support the two temporal LUSTRE operators `when` and `current`, which handle the use of multiple clocks since this is the case for many industrial applications.

The use of multiple clocks implies the filtering of some program expressions. It consists in changing their execution cycle, activating it only at certain cycles of the basic clock. Consequently, the associated paths are activated only if the respective clock is true. As a result, the tester must adjust this rarefied path activation rate according to the global *timing*.

In this section, we present the definition for the path activation conditions for `when` and `current`, followed by their formal verification. Then, we demonstrate the application of the extended criteria as well as the coverage evaluation, using the simple example of the inverse counter of Section 2.2.

4.1 Activation Conditions for `when` and `current`

Informally, the activation conditions associated with the `when` and `current` operators are based on their intrinsic definition. Since the output values are defined according to a condition (i.e. the *true* value of the clock), these operators can be represented by means of the conditional operator `if-then-else`. For the expression E and the boolean expression B with the same clock,

– `X=E when B` could be seen as `X=if B then E else NON_DEFINED` and similarly,
– `Y=current(X)` could be seen as `Y=if B then X else pre(X)`.

Hence, the formal definitions of the activation conditions result as follows:

Definition 1. *Let e and s be the input and output edges respectively of a* `when` *operator and let b be its clock. The activation conditions for the paths $p_1 = \langle e, s \rangle$ and $p_2 = \langle b, s \rangle$ are:*
$AC(p_1) = b$
$AC(p_2) = true$

Definition 2. *Let e and s be the input and output edges respectively of a* `current` *operator and let b be the clock on which it operates. The activation condition for the path* $p = \langle e, s \rangle$ *is:*
$$AC(p) = b$$

As a result, to compute the paths and the associated activation conditions of a LUSTRE node involving several clocks, one has just to replace the `when` and `current` operators by the corresponding conditional operator (see Figure 5). At this point, two basic issues need to be farther clarified. The first one concerns the `when` case. Actually, there is no way of defining the value of the expression X when the clock B is not *true* (branch *NON_DEF* in Figure 5(a)). By default, at these instants, X does not occur and such paths (beginning with a non defined value) are infeasible[3]. In the `current` case, the operator implicitly refers to the clock parameter B, without using a separate input variable (see Figure 5(b)). This hints at the fact that `current` always operates on an already sampled expression, so the clock that determines its output activation should be the one on which the input is sampled.

(a)

(b)
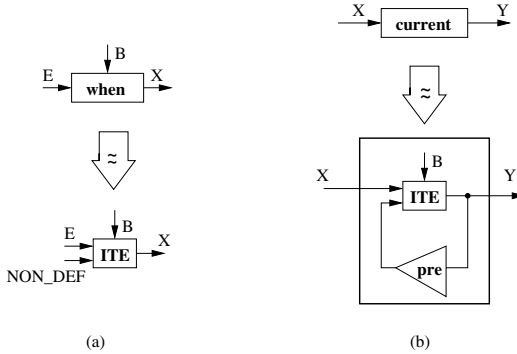
Fig. 5. Modeling the `when` and `current` operators using the `ITE`

Let us assume the path $p = \langle m, x, M_1, M_2, M_3, M_4, c \rangle$ in the example of Section 2.2, displayed in bold in Figure 4. Following the same procedure for the activation condition computation and starting from the last path edge, the activation conditions for the intermediate unit paths are:

$AC(p) = false$ -> $AC(p_1)$, where $p_1 = \langle m, x, M_1, M_2, M_3, M_4 \rangle$
$AC(p_1) = true$ *and* $AC(p_2)$, where $p_2 = \langle m, x, M_1, M_2, M_3 \rangle$
$AC(p_2) = false$ -> $pre(AC(p_3))$, where $p_3 = \langle m, x, M_1, M_2 \rangle$
$AC(p_3) = c$ *and* $AC(p_4)$, where $p_4 = \langle m, x, M_1 \rangle$
$AC(p_4) = c$ *and* $AC(p_5)$, where $p_5 = \langle m, x \rangle$
$AC(p_5) = c$

---

[3] An infeasible path is a path which is never executed by any test cases, hence it can never be covered.

After backward substitutions, the activation condition of the selected path is:

$$AC\,(p) = false \ \text{->} \ pre\,(c)\,.$$

This condition corresponds to the expected result and is compliant with the above definitions, according to which the clock must be true to activate the paths with `when` and `current` operators.

In order to evaluate the impact of these temporal operators on the coverage assessment, we consider the operator network of Figure 4 and the paths:

$p_1 = \langle m, x, M_1, y \rangle$
$p_2 = \langle m, x, M_1, M_2, M_3, M_4, c \rangle$
$p_3 = \langle m, x, M_1, M_2, M_3, M_5, y \rangle$

Intuitively, if the clock c holds true, any change of the path input is propagated through the output, hence the above paths are activated. Formally, the associated activation conditions to be satisfied by a test set are:

$AC\,(p_1) = c$

$AC\,(p_2) = false \ \text{->} \ pre\,(c)$

$AC\,(p_3) = not\,(c)\ and\ false \ \text{->} \ pre\,(c).$

Eventually, the input test sequences satisfy the basic criterion. Indeed, as soon as the input m causes the clock c to take the suitable values, the activation conditions are satisfied, since the latter depend only on the clock. In particular, in case that the value of m at the first cycle is an integer different to zero (for sake of simplicity, let us consider $m = 2$), the BC is satisfied in two steps since the corresponding values for c are c=*true*, c=*false*. On the contrary, if at the first execution cycle m equals to zero, the basic criterion is satisfied after three steps with the corresponding values for c: c=*true*, c=*true*, c=*false*. These two samples of input test sequences and the corresponding outputs are shown in Table 3.

Table 3. Test cases samples for the input m

	c_1	c_2	c_3	c_4	...		c_1	c_2	c_3	c_4
m	$i_1\,(\neq 0)$	i_2	i_3	i_4	...	m	$i_1\,(= 0)$	i_2	i_3	...
c	*true*	*false*	*false*	*true*	...	c	*true*	*true*	*false*	...
y	i_1	$i_1 - 1$	0	i_4	...	y	0	i_2	$i_2 - 1$	...

4.2   An Illustrative Example

Let us consider a LUSTRE node that receives at the input a boolean signal *set* and returns at the output a boolean signal *level*. The latter must be true during *delay* cycles after each reception of *set*. Now, suppose that we want the *level* to be high during *delay* seconds, instead of *delay* cycles. Taking advantage of the use of the `when` and `current` operators, we could call the above node on a

```
node TIME_STABLE(set, second: bool; delay: int) returns
(level: bool);
var ck: bool;
let
  level = current(STABLE((set, delay) when ck));
  ck = true -> set or second;
tel;

node STABLE(set: bool; delay: int) returns (level: bool);
var count: int;
let
  level = (count>0);
  count = if set then delay
          else if false->pre(level) then pre(count)-1
          else 0;
tel;
```

Fig. 6. The node `TIME_STABLE`: a simple example with the `when` and `current` operators

suitable clock by filtering its inputs. The second must be provided as a boolean input *second*, which would be *true* whenever a second elapses. The node must be activated only when either a *set* signal or a *second* signal occurs and in addition at the initial cycle, for initialization purposes. The LUSTRE code is quite simple and it is shown in Figure 6, followed by the associated operator network[4].

Similarly to the previous example, the paths to be covered are:

$p_1 = \langle set, T_2, T_3, T_9, level \rangle$
$p_2 = \langle delay, T_8, T_3, T_9, level \rangle$
$p_3 = \langle set, T_1, ck, T_2, T_3, T_9, level \rangle$
$p_4 = \langle second, T_1, ck, T_2, T_3, T_9, level \rangle$
$p_5 = \langle set, T_1, ck, T_8, T_3, T_9, level \rangle$
$p_6 = \langle second, T_1, ck, T_8, T_3, T_9, level \rangle$

To cover all these paths, one has to select a test set satisfying the following activation conditions, calculated as it is described above:

$AC(p_1) = ck$, where $ck = true$ -> $set\,or\,second$
$AC(p_2) = ck\,and\,set$
$AC(p_3) = ck\,and\,false$ -> $set\,or\,not\,(second)$
$AC(p_4) = ck\,and\,false$ -> $second\,or\,not\,(set)$
$AC(p_5) = ck\,and\,set\,and\,false$ -> $set\,or\,not\,(second)$
$AC(p_6) = ck\,and\,set\,and\,false$ -> $second\,or\,not\,(set)$

Since the code ensures the correct initialization of the clock, hence its activation at the first cycle, the above paths are always activated at the first execution cycle. For the rest of the execution, the basic criterion is satisfied with the

---

[4] The nested node `STABLE` is used unfolded, since with this criteria definition, the dependencies between a called node inputs and outputs cannot be determined.
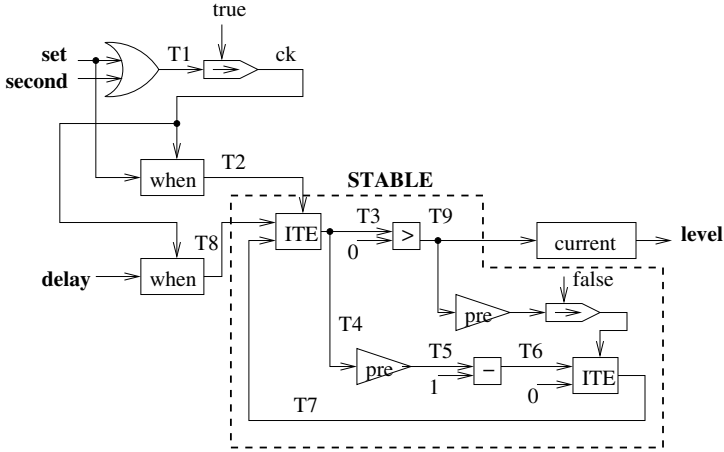
Fig. 7. The operator network for the node `TIME_STABLE`

following test sequence for the inputs $(set, second)$: $(1, 0),(0, 1)$, $(1, 1)$. This test
set, which contains almost every possible combination of the inputs, satisfies
also the elementary conditions criterion (ECC), since the activation of the paths
depends on both boolean inputs.

Admittedly, the difficulty to meet the criteria is strongly related to the complexity of the system under test as well as to the test case generation effort.
Moreover, activation conditions covered with short input sequences are easy to
be satisfied, as opposed to long test sets that correspond to complex instance
executions of the system under test. Experimental evaluation on more complex
case studies, including industrial software components, is necessary and part of
our future work in order to address these problems. Nonetheless, the enhanced
definitions of the structural criteria presented in this paper complete the coverage assessment issue for LUSTRE programs, as all the language operators are
supported. In addition, the complexity of the criteria is not further affected,
because, in substance, we use nothing but `if-then-else` operators.

5   Conclusion

We presented the extension of the LUSTRE structural coverage criteria to support
the use of multiple clocks. We defined the activation conditions for the temporal
operators `when` and `current`, which are used to affect the clock of a LUSTRE
expression. We applied the results on suitable examples and we described how
the criteria could be employed. Yet, the research work presented in this paper
needs to be implemented and incorporated in LUSTRUCTU, a tool which measures
the structural coverage of LUSTRE programs.

In SCADE, coverage is measured through the Model Test Coverage (MTC)
module, in which the user can define his own criteria by defining the conditions to
be activated during testing. Thus, our work could be easily integrated in SCADE

in the sense that activation conditions corresponding to the defined criteria (BC, ECC, MCC) could be assessed once they are transformed into suitable MTC expressions. These issues are currently investigated within the framework of a collaborative research project[5].

Future work includes the evaluation of the proposed criteria involving industrial case studies. Furthermore, it is necessary to analyze the test sets to determine their ability to satisfy the criteria and observe what happens with the paths that the tests cannot cover.

Integration testing issues are also under study. In case of long paths to be covered, the total path number highly increases causing the coverage measures to be non applicable. As a result, integration testing requires extending the definition of the activation conditions to internal nodes, that is to the operators that the user can define. Such an extension should make it possible to apply the code coverage criteria on LUSTRE nodes that call other nodes (compound operators) without having to unfold the latter ones and reducing the overall complexity.

References

1. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Le Guernic, P., de Simone, R.: The synchronous languages 12 years later. Proceedings of the IEEE 91(1), 64–83 (2003)
2. Boussinot, F., De Simone, R.: The Esterel language. Proceedings of the IEEE 79(9), 1293–1304 (1991)
3. Clarke, L.A., Podgurski, A., Richardson, D.J., Zeil, S.J.: A formal evaluation of data flow path selection criteria. IEEE Trans. Software Eng. 15(11), 1318–1332 (1989)
4. Girault, A., Nicollin, X.: Clock-driven automatic distribution of lustre programs. In: Alur, R., Lee, I. (eds.) EMSOFT 2003. LNCS, vol. 2855, pp. 206–222. Springer, Heidelberg (2003)
5. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language lustre. Proceedings of the IEEE 79(9), 1305–1320 (1991)
6. Halbwachs, N., Lagnier, F., Ratel, C.: Programming and verifying real-time systems by means of the synchronous data-flow language lustre. IEEE Trans. Software Eng. 18(9), 785–793 (1992)
7. Lakehal, A., Parissis, I.: Structural test coverage criteria for lustre programs. In: The 10th International Workshop on Formal Methods for Industrial Critical Systems (FMICS), a joint event of ESEC/FSE 2005, Lisbon, Portugal, September 2005, pp. 35–43 (2005)
8. Le Guernic, P., Gautier, T., Le Borgne, M., Le Maire, C.: Programming Real-Time Applications with SIGNAL. Proceedings of the IEEE 79(9), 1321–1336 (1991)
9. Rapps, S., Weyuker, E.J.: Selecting software test data using data flow information. IEEE Trans. Software Eng. 11(4), 367–375 (1985)
10. Woodward, M.R., Hedley, D., Hennell, M.A.: Experience with path analysis and testing of programs. IEEE Trans. Softw. Eng. 6(3), 278–286 (1980)

Fighting State Space Explosion: Review and Evaluation

Radek Pelánek⋆

Department of Information Technology, Faculty of Informatics
Masaryk University Brno, Czech Republic
`xpelanek@fi.muni.cz`

Abstract. In order to apply formal methods in practice, the practitioner has to comprehend a vast amount of research literature and realistically evaluate practical merits of different approaches. In this paper we focus on explicit finite state model checking and study this area from practitioner's point of view. We provide a systematic overview of techniques for fighting state space explosion and we analyse trends in the research. We also report on our own experience with practical performance of techniques. Our main conclusion and recommendation for practitioner is the following: be critical to claims of dramatic improvement brought by a single sophisticated technique, rather use many different simple techniques and combine them.

1 Introduction

If you are a practitioner who wants to apply formal methods in industrial critical systems, you have to address following problems: Which of the many approaches should I use? If I want to improve the performance of my tool, which of the techniques described by researchers should I use? Which techniques are worth the implementation effort? These are important questions which are not easy to answer, nevertheless they are seldom addressed in research papers.

Research papers rather propose a steady flow of novel techniques, improvements, and optimizations. However, the experimental work reported in research papers has often poor quality [59] and it is difficult to judge the practical merit of proposed techniques. The goal of this paper is to provide an overview of research and realistic assessment of practical merits of techniques. The paper should serve as a guide for a practitioner who is trying to answer the above given questions.

It is not feasible to realize this goal for the whole field of formal verification at once. Therefore, we focus on one particular area (explicit model checking) and give an overview of research in this area and report on practical experience. Even though our discussion is focused on one specific area, we believe that our main recommendation — that it is better to combine several simple techniques rather than to focus on one sophisticated one — is applicable to many other areas of formal methods.

Explicit model checking is principally very simple — a brute force traversal of all possible model states. Despite the simplicity of the basic idea, explicit model checking is still the best approach for many practically important areas of application, e.g., verification of communication protocols and software. The popularity of the approach is illustrated by large number of available tools (e.g., Spin, CADP, mCRL2, Uppaal, Divine, Java PathFinder, Helena) and widespread availability of courses and textbooks on the topic (e.g., [10]).

The main obstacle in applying explicit model checking in practice is the state space explosion problem. Hence, the research focuses mainly on techniques for fighting state space explosions — during the last 15 years more than 100 papers have been published on the topic, proposing various techniques for fighting state space explosion. What are these techniques and how can we classify them? What is the real improvement brought by these techniques? Which techniques are practically useful? Which techniques should a practitioner study and use?

We try to answer these questions, particularly we provide the following:

- We overview techniques for fighting state space explosion in explicit model checking and divide them into four main areas (Section 2).
- We review and analyse research on fighting state space explosion, and discuss main trends in this research (Section 3).
- We report on our own practical experience with application and evaluation of techniques for fighting state space explosion (Section 4).
- Based on the review of literature and our experience, we provide specific recommendation for practitioner in industry (Section 5).

The main aim of this paper is to present and support the following message: Rather than optimizing the performance of a single sophisticated technique, we should use many different simple techniques, study how to combine them, and how to run them effectively in parallel.

2   Overview of Techniques for Fighting State Space Explosion

Fig. 1. gives an algorithm EXPLORE which explores the reachable part of the state space. This basic algorithm can be directly used for verification of simple safety properties; for more complex properties, we have to use more sophisticated algorithms (e.g., cycle detection [72]). Nevertheless, the basic ideas of techniques for fighting state space explosion are similar. For clarity, we discuss these techniques mainly with respect to the basic EXPLORE algorithm.

The main problem of explicit state space exploration is state space explosion problem and consequently memory and time requirements of the algorithm EXPLORE. Techniques for fighting state space explosion can be divided into four main groups:

```
proc EXPLORE(M)
   Wait = {s₀};  Visited = ∅
   while Wait ≠ ∅ do
           get s from Wait
           explore state s
           foreach s′ ∈ successors of s do
              if s′ ∉ Visited then
                                    add s′ to Wait
                                    add s to Visited fi od
       od
   end
```

Fig. 1. The basic algorithm which explores all reachable states. Data structure *Wait* (also called open list) holds states to be visited, data structure *Visited* (also called visited list, closed list, transposition table, or just hash table) stores already explored states.

1. Reduce the number of states that need to be explored.
2. Reduce the memory requirements needed for storing explored states.
3. Use parallelism or distributed environment.
4. Give up the requirement on completeness and explore only part of the state space.

In the following we discuss these four types of approaches and for each of them we list examples of specific techniques.

2.1   State Space Reductions

When we inspect some simple models and their state spaces, we quickly notice significant redundancy in these state spaces. So the straightforward idea is to try to exploit this redundancy and reduce the number of states visited during the search. In order to exploit this idea in practice, we have to specify which states are omitted from the search and we have to show the correctness of the approach, i.e., prove that the visited part of the state space is equivalent to the whole state space with respect to some equivalence (typically bisimulation or stutter equivalence).

State based reductions. State based reductions exploit observation that if two states are bisimilar then it is sufficient to explore successors of only one of them. The reduction can be performed either on-the-fly during the exploration or by a static modification of the model before the exploration. Examples of such reductions are symmetry reduction [12,20,43,44,70,74], live variable reduction [21,69], cone of influence reductions, and slicing [19,35].

Path based reductions. Path based reductions exploit observation that sometimes it is sufficient to explore only one of two sequences of actions because they are just different linearizations of "independent" actions and therefore have the

same effect. These reductions try to reduce the number of equivalent interleavings. Examples of such reductions are transition merging [18,48], partial order reduction [27,33,40,63,64], τ-confluence [11], and simultaneous reachability analysis [55].

Compositional methods. Systems are often specified as a composition of several components. This structure can be exploited in two ways: compositional generation of the state space [46] and assume-guarantee approach [22,32,66].

2.2   Storage Size Reductions

The main bottleneck of model checking are usually memory requirements. Therefore, we can save some memory at the cost of using more time, i.e., by employing some kind of time-memory trade-off. The main source of memory requirements of the algorithm EXPLORE is the structure *Visited* which stores previously visited states. Hence, techniques, which try to lower memory requirements, focus mainly on this structure.

State compression. During the search, each state is represented as a byte vector which can be quite large (e.g., 100 bytes). In order to save space, this vector can be compressed [25,26,30,39,49,56,73] or common components can be shared [38]. Instead of compressing individual states, we can also represent the whole structure *Visited* implicitly as a minimized deterministic automaton [41].

Caching and selective storing. Instead of storing all states in the structure *Visited*, we can store only some of these states — this approach can lead to revisits of some states and hence can increase runtime, but it saves memory. Techniques of this type are for example:

- caching [24,28,65], which deletes some currently stored states when the memory is full,
- selective storing [9,49], which stores only some states according to given heuristics,
- sweep line method [15,54,68], which uses so called progress function; this function guarantees that some states will not be revisited in the future and hence these states can be deleted from the memory.

Use of magnetic disk. Simple use of magnetic disk leads to an extensive swapping and slows down the computation extremely. So the magnetic disk have to be used in a sophisticated way [7,8,71] in order to minimize disk operations.

2.3   Parallel and Distributed Computation

Another approach to manage a large number of states is to use even more brute force — more processors.

Networks of workstations. Distributed computation can be realized most easily by network of workstations connected by fast communication medium

(i.e., workstations communicate by message passing). In this setting the state space is partitioned among workstations (i.e., each workstation stores part of the data structure *Visited*) and workstations exchange messages about states to be visited (*Wait* structure), see e.g., [23,50,51]. The application of distributed environment for verification of liveness properties is more complicated, because classical algorithms are based on depth-first search, which cannot be easily adapted for distributed environment. Hence, for verification of liveness properties we have to use more sophisticated algorithms, see e.g., [1,2,3,5,13,14].

Multi-core processors. Recently, multi-core processors become widely available. Multi-core processors provide parallelism with shared memory, i.e., the possibility to reduce run-time of the verification by parallel exploration of several states, see e.g., [4,42].

2.4 Randomized Techniques and Heuristics

If the memory requirements of the search are too large even after the application of above given techniques, we can use randomized techniques and heuristics. These techniques explore only part of the state space. Therefore, they can help only in the detection of an error; they cannot assist us in proving correctness.

Heuristic search (also called directed or guided search). States are visited in an order given by some heuristics, i.e., *Wait* list is implemented as priority queue [31,47,67]. Different heuristic approach is to use genetic algorithm which tries to 'evolve' a path to a goal state [29].

Random walk and partial search. Random walk does not store any information and always visits just one successor of a current state [34,60]. This basic strategy can be extended in several ways, e.g., by visiting a subset of all successors (instead of just one state), storing some states in the *Visited* structure, or combining random walk with local breadth-first search, see e.g., [36,45,52,53,60].

Bitstate hashing. The algorithm does not store whole states but only one bit per state in a large hash table [37]. In a case of collision some states are omitted by the search. A more involved version of this technique is based on Bloom filters [16,17].

3 Research Analysis

What are the trends in the research literature about techniques for fighting state space explosion? Is the quality of experimental evidence improving? How significant is the improvement reported in research papers? How is this improvement changing over time?

3.1 Research Papers

In order to answer the above given questions, we have collected and analyzed large set of research papers. More specifically, we collected research papers that
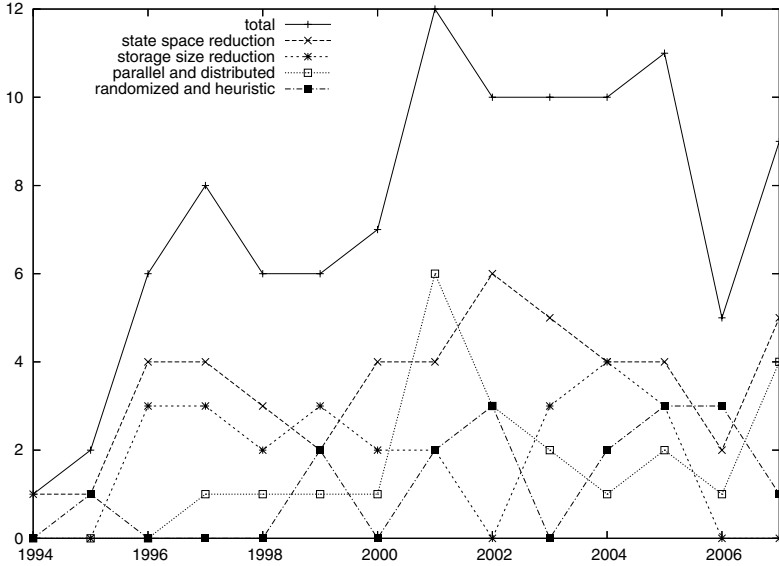
Fig. 2. Numbers of publications; note that some papers can be counted in two categories

describe techniques for fighting state space explosion in explicit model checking of finite state systems[1].

The collection contains more than 100 papers – these papers were obtained by systematically collecting papers from the most relevant conferences and by citation tracking. The full list of reviewed papers, which includes all papers referenced above, is freely available[2]. The collection is certainly not complete, but we believe that it is a good sample of research in the area.

Fig. 2. shows the number of publications in each year during the last 13 years. Although there are rises and downfalls, the overall flow of publications on the topic is rather steady. The figure also shows that all four areas described in the previous section are pursued concurrently.

3.2   Quality of Experiments

Although some of the considered research is rather theoretically oriented (e.g., partial order reduction), all considered techniques are in fact heuristics which aim at improving performance of model checking tools. So what really matters is the practical improvement brought by each technique. To assess the improvement

---

[1] In few cases we also include techniques which are not purely explicit, but target the similar application domain (i.e., the experiments are done on same models as for other included papers).

[2] http://www.fi.muni.cz/~xpelanek/amase/reductions.bib
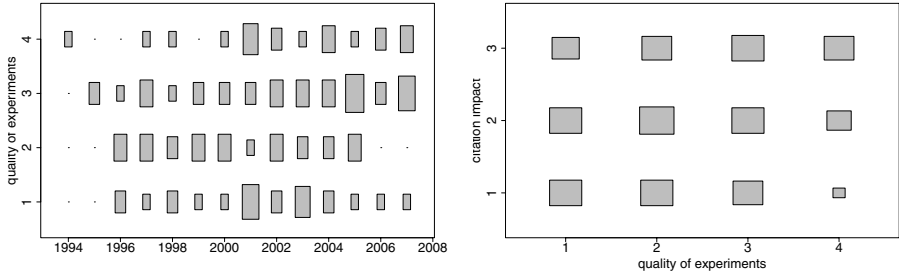
Fig. 3. The first graph shows the quality of experiments reported in model checking papers during time. The size of a box corresponds to a number of published papers in a given year and quality category. The second graph shows the relation between experiment quality and citation impact; citation impact is divided into three categories: less than 10 citations, 10–30 citations, more than 30 citations; only publication before 2004 are used.

it is necessary to perform experimental evaluation. Only good experiments can provide realistic evaluation of practical merits of proposed techniques.

In order to study the quality of experiments, we classify experiments in each paper into one of four classes, depending on the number and type of used models[3]:

1. Random inputs or few toy models.
2. Several toy models (possibly parametrized) or few simple models.
3. Several simple models (possibly parametrized) or one large case study.
4. Exhaustive study of parametrized simple models or several case studies.

Fig 3. presents the quality of experiments in papers from our sample. The figure shows that the quality on average is not very good and, what is even more disappointing, that there is slow progress in time, although many realistic case studies are available (see [59] for more detailed discussion of these issues).

Since there is a large number of techniques, it is important to compare performance of novel techniques with previously studied one. However, analysis of our research sample shows that only about 40% papers contain some comparison with similar techniques; this ratio is improving with time, but only slowly. Moreover, the comparison is usually only shallow.

Our analysis also shows one encouraging trend. Fig. 3. shows that there is a relation between quality of experiments and citation impact of a paper — research with better experiments is more cited.

3.3 Reported Improvement

Before the discussion of improvements reported in research papers, we clarify the terminology that we use to measure this improvement. We use the notion

---

[3] The classification is clearly slightly subjective. Nevertheless, we believe that the main conclusions of our analysis do not depend on the subjective factor.
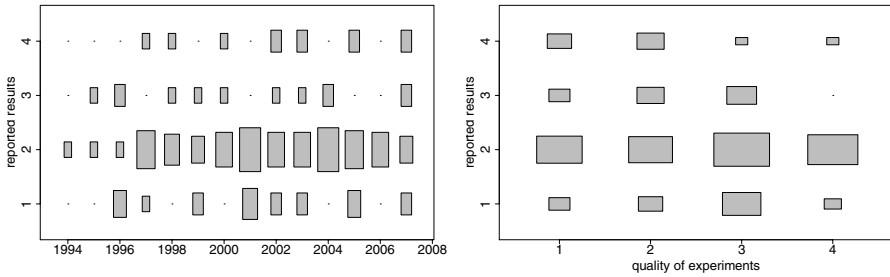
Fig. 4. Reported improvement with respect to time and quality of experiments

'reduction ratio' to denote the ratio between the memory consumption of the technique for fighting state space explosion and the memory consumption of the standard reachability (exploration of full reachable state space). Some authors report 'reduced by' factor, i.e., if we report 'reduction ratio' 80%, it means that the memory consumption was 'reduced by' 20%. Note that in this section we analyze reduction ratios as reported by authors, not what we consider realistic reduction ratios of techniques.

For clarity of presentation, we again divide the reported reduction ratios into four classes:

1. Reported reduction ratio is 50% or worse (or sometimes good but sometimes worse than 100%).
2. Reported reduction ratio is in most cases 10%-50%.
3. Reported reduction ratio is in most cases 1%-10%.
4. Reported reduction ratio is better than 1% (or exponential improvement is reported or only out-of memory for full search is reported, i.e., reduction ratio is impossible to assess).

Fig. 4. shows that in most cases the reported reduction is in the second category (reduction between 10% and 50%). The relation with the quality of experiments clearly demonstrates that this is also the most realistic evaluation — better results are often caused by poor experiments, not by special features of techniques.

There is no clear trend with respect to time, i.e., it seems that novel techniques do not significantly improve on performance of previous techniques. This does not automatically mean that the recent research is misguided. In some cases novel technique provides principally different way how to obtain the reduction and can be combined with previously proposed techniques in orthogonal way. Novel technique can also extend the application domain of previously studied techniques.

As we already mentioned, the research in this domain is purely practically motivated. However, the amount of research into certain topic is not really related to its practical merit. For example, our experience (described in next section) shows that the dead variable reduction brings similar improvement as partial order reduction. Nevertheless, there are significantly more research papers about

partial order reduction than about dead variable reduction. This is probably not due to the practical merits of partial order reduction, but because it can be extensively studied theoretically.

4   Practical Experience

In this section we report on our experience with techniques for fighting state space explosion. Our experience is based on large-scale research studies, which are described in stand-alone publications. Here we provide only a brief description of these studies and present their main conclusions. Technical details can be found in cited papers.

Our experience report obviously does not cover all techniques for fighting state space explosion. However, we cover all four areas described in Section 2 and the main conclusions are in all cases similar, so we believe that it is reasonable to generalize our experience.

4.1   On-the-fly State Space Reductions

We evaluated several techniques for on-the-fly state space reductions. Setting of this study (see [57] for details):

- Implementation: publicly available implementations of explicit model checkers (Spin, Murphy, DiVinE).
- Models: models included in tool distributions plus few more publicly available case studies.
- Techniques: dead variable reduction, partial order reduction, transition merging, symmetry reduction.

When we measured the performance of techniques over realistic models, we found that the reduction ratio is usually worse than what is reported in research papers — research papers often use simple models with artificially high values of model parameters. More specifically, the main results of our evaluation are the following: dead variable reduction works on nearly all models, reduction ratio is usually between 10% and 90%; partial order reduction works only in some cases, reduction ratio is between 5% and 90%; transition merging works in similar cases as partial order reduction, it is weaker but easier to realize, reduction ratio is usually between 50% and 95%; symmetry reduction works only for few models (symmetric ones), reduction ratio is usually between 8% and 50%.

Our main conclusion from this study are the following:

- Each technique is applicable only to some types of models. No technique works really universally; more specialized techniques yield better reduction.
- On real models, no single technique is able to achieve reduction ratio significantly under 5%. Claims about drastic reduction, which occur in some papers, are not really appropriate.
- Since there are many techniques and many of them are orthogonal, most models can be reduced quite significantly.

4.2   Caching and Compression

From the area of 'storage size reduction' techniques we evaluated two techniques for reducing memory consumption of the data structure *Visited*. Setting of this study (see [62] for details):

- Implementation: all techniques are implemented in uniform way using the DiVinE environment [6] (source codes are publicly available).
- Models: 120 models from BEEM (BEnchmarks for Explicit Model checkers) [59].
- Techniques: state caching with 7 different caching strategies, state compression with Huffman coding (two variants: static code and code computed by training runs).

In the study we also reviewed previous research on storage size reduction techniques. We found that using proper parameter values with our simple and easy-to-implement techniques, we were able to achieve very similar results to those reported in other works which use far more sophisticated approaches. Concrete results of the evaluation are the following:

- Caching strategies are to a certain degree complementary. Using an appropriate state caching strategy, the reduction ratio is in most cases 10% to 30%.
- Using state compression, the reduction ratio is usually around 60%.
- The two techniques combine well.

4.3   Distributed Exploration

From the area of parallel and distributed techniques we report on the basic distributed approach to explicit model checking: we have a network of workstations connected by fast Ethernet, workstations communicate via message passing (MPI library), state space is partitioned among workstations. In this setting the reduction ratio is clearly bounded by $1/n$, where n is the number of workstations. In practice the reduction ratio is worse because of communication overhead. Here we report on results of our evaluation, however the results are rather typical in this area.

Setting of this study (see [58] for details):

- Implementation: the DiVinE tool (public version).
- Models: 120 models from BEEM (BEnchmarks for Explicit Model checkers) [59].
- Techniques: distributed reachability on 20 workstations.

In this study the speedup varies from 2 to 12, typical value of the speedup is between 4 and 6 (i.e., reduction ratio around 20%). We also found that the speedup is negatively correlated with the speed of successor generation by the tool.

4.4 Error Detection Techniques

From the area of 'randomized techniques and heuristics' we have chosen 9 techniques and evaluated their performance. In this case we do not study the reduction ratio, because it is not known — the experiments are done on models for which the standard reachability is not feasible. Therefore, we focus on relative performance of techniques and on the issue of complementarity.

Setting of this study (see [61] for details):

– Implementation: all techniques are implemented in uniform way using the DiVinE environment [6] (source codes are publicly available),
– Models: 54 models (with very large state space) from BEEM [59].
– Techniques: breadth-first search, depth-first search, randomized DFS, two variants of random walk, bitstate hashing with repetition, two variants of directed search, and under-approximation refinement based on partial order reduction.

For the evaluation we used several performance measures: number of steps needed to find an error, length of reported counterexample, and coverage metrics. The main results of this study are the following:

– There is no single best technique. Results depend on used performance metrics, even for a given metric, the most successful technique is the best one only over 25% of models.
– It is important to focus on complementarity of techniques, not just on their overall (average) performance. For example, in our study the random walk technique had rather poor overall performance, but it was successful on models where other techniques fall, i.e., it is a useful technique which we should not discard.

5 Conclusions

This paper is concerned with techniques for fighting state space explosion problem in explicit model checking. We review the research in the area during the last 15 years (more than 100 research papers) and report on our practical experience. As a result of our review we identify four main groups of techniques: state space reductions, storage size reductions, parallel and distributed computation, randomization and heuristics. These four groups are rather orthogonal and can be combined; within each group techniques are often based on similar ideas and their combination can be difficult.

The review of research shows that despite a steady flow of publications on the topic, the progress is not very significant — in fact the reduction ratio reported in research papers stays practically the same over the last 15 years. This analysis stresses the need for good practical evaluation. However, realistic evaluation of research progress is complicated by rather poor experimental standards and by unjustified claims by researchers.

Results reported in research papers often make an impression of dramatic improvements. Our practical experience suggests that it is not realistic to get better reduction ratio than 5% with a single technique, in fact in most cases the obtained reduction ratio is between 20% and 80%. Nevertheless, this does not mean that techniques for fighting state space explosion are not useful. Techniques of different types can be combined, and together they might be able to bring a significant improvement.

Our experience also suggest that simple techniques are often sufficient. The performance obtained by sophisticated techniques is often similar to performance of basic techniques from each area. Complicated techniques often achieve better results only for specialized application domains. This observation can be also supported by analysis of techniques implemented in model checking tools. Tools usually implement basic versions of many techniques, sophisticated techniques are often implemented only in a tool used by authors of the technique.

To summarise, we propose following recommendations for those who want to apply model checking in practice:

- Use large number of simple techniques of different types.
- Do not try to find 'the best' technique of a specific type. Try to find a set of simple complementary techniques and run all of them (preferably in parallel).
- Be critical to claims in research papers, particularly if the experimental evidence is poor.
- Use sophisticated techniques only if they are specifically targeted at your domain of application.
- Focus on combination of orthogonal techniques.

Researchers, we believe, should focus not just on the development of novel techniques, but also on issues of techniques combination, selection, and efficient scheduling: How to select right technique for a given model? In what order we should try available techniques? Can information gathered by one technique be used by another techniques?

Acknowledgment

References

1. Barnat, J., Brim, L., Černá, I.: Property driven distribution of nested DFS. In: Proc. of Workshop on Verification and Computational Logic, number DSSE-TR-2002-5 in DSSE Technical Report, pp. 1–10. University of Southampton, UK (2002)
2. Barnat, J., Brim, L., Chaloupka, J.: Parallel breadth-first search LTL model-checking. In: Proc. of Automated Software Engineering (ASE 2003), pp. 106–115. IEEE Computer Society, Los Alamitos (2003)

3. Barnat, J., Brim, L., Chaloupka, J.: From distributed memory cycle detection to parallel LTL model checking. ENTCS 133(1), 21–39 (2005)
4. Barnat, J., Brim, L., Rockai, P.: Scalable multi-core LTL model-checking. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 187–203. Springer, Heidelberg (2007)
5. Barnat, J., Brim, L., Stříbrná, J.: Distributed LTL model-checking in SPIN. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 200–216. Springer, Heidelberg (2001)
6. Barnat, J., Brim, L., Černá, I., Moravec, P., Rockai, P., Šimeček, P.: DiVinE - a tool for distributed verification. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 278–281. Springer, Heidelberg (2006), `http://anna.fi.muni.cz/divine`
7. Barnat, J., Brim, L., Šimeček, P.: I/o efficient accepting cycle detection i/o efficient accepting cycle detection. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 281–293. Springer, Heidelberg (2007)
8. Barnat, J., Brim, L., Šimeček, P., Weber, M.: Revisiting resistance speeds up i/o-efficient ltl model checking. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 48–62. Springer, Heidelberg (2008)
9. Behrmann, G., Larsen, K.G., Pelánek, R.: To store or not to store. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 433–445. Springer, Heidelberg (2003)
10. Ben-Ari, M.: Principles of the SPIN Model Checker. Springer, Heidelberg (2008)
11. Blom, S., van de Pol, J.: State space reduction by proving confluence. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 596–609. Springer, Heidelberg (2002)
12. Bosnacki, D.: A light-weight algorithm for model checking with symmetry reduction and weak fairness. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 89–103. Springer, Heidelberg (2003)
13. Brim, L., Černá, I., Krčál, P., Pelánek, R.: Distributed LTL model checking based on negative cycle detection. In: Hariharan, R., Mukund, M., Vinay, V. (eds.) FSTTCS 2001. LNCS, vol. 2245, pp. 96–107. Springer, Heidelberg (2001)
14. Černá, I., Pelánek, R.: Distributed explicit fair cycle detection. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 49–73. Springer, Heidelberg (2003)
15. Christensen, S., Kristensen, L.M., Mailund, T.: A sweep-line method for state space exploration. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 450–464. Springer, Heidelberg (2001)
16. Dillinger, P.C., Manolios, P.: Bloom filters in probabilistic verification. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 367–381. Springer, Heidelberg (2004)
17. Dillinger, P.C., Manolios, P.: Fast and accurate bitstate verification for SPIN. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 57–75. Springer, Heidelberg (2004)
18. Dong, Y., Ramakrishnan, C.R.: An optimizing compiler for efficient model checking. In: Proc. of Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX), pp. 241–256. Kluwer, Dordrecht (1999)
19. Dwyer, M.B., Hatcliff, J., Hoosier, M., Ranganath, V.P.: Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 73–89. Springer, Heidelberg (2006)

20. Emerson, E.A., Wahl, T.: Dynamic symmetry reduction. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 382–396. Springer, Heidelberg (2005)
21. Fernandez, J.C., Bozga, M., Ghirvu, L.: State space reduction based on live variables analysis. Journal of Science of Computer Programming (SCP) 47(2-3), 203–220 (2003)
22. Flanagan, C., Qadeer, S.: Thread-modular model checking. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 213–224. Springer, Heidelberg (2003)
23. Garavel, H., Mateescu, R., Smarandache, I.: Parallel state space construction for model-checking. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 217–234. Springer, Heidelberg (2001)
24. Geldenhuys, J.: State caching reconsidered. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 23–38. Springer, Heidelberg (2004)
25. Geldenhuys, J., de Villiers, P.J.A.: Runtime efficient state compaction in SPIN. In: Dams, D.R., Gerth, R., Leue, S., Massink, M. (eds.) SPIN 1999. LNCS, vol. 1680, pp. 12–21. Springer, Heidelberg (1999)
26. Geldenhuys, J., Valmari, A.: A nearly memory-optimal data structure for sets and mappings. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 136–150. Springer, Heidelberg (2003)
27. Godefroid, P.: Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem. In: Godefroid, P. (ed.) Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032, p. 142. Springer, Heidelberg (1996)
28. Godefroid, P., Holzmann, G.J., Pirottin, D.: State space caching revisited. In: Probst, D.K., von Bochmann, G. (eds.) CAV 1992. LNCS, vol. 663, pp. 178–191. Springer, Heidelberg (1993)
29. Godefroid, P., Khurshid, S.: Exploring very large state spaces using genetic algorithms. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 266–280. Springer, Heidelberg (2002)
30. Gregoire, J.: State space compression in spin with GETSs. In: Proc. Second SPIN Workshop, Rutgers University, New Brunswick, New Jersey (1996)
31. Groce, A., Visser, W.: Heuristics for model checking java programs. Software Tools for Technology Transfer (STTT) 6(4), 260–276 (2004)
32. Grumberg, O., Long, D.E.: Model checking and modular verification. ACM Transactions on Programming Languages and Systems 16(3), 843–871 (1994)
33. Gueta, G., Flanagan, C., Yahav, E., Sagiv, M.: Cartesian partial-order reduction. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 95–112. Springer, Heidelberg (2007)
34. Haslum, P.: Model checking by random walk. In: Proc. of ECSEL Workshop (1999)
35. Hatcliff, J., Dwyer, M.B., Zheng, H.: Slicing software for model construction. Higher Order Symbol. Comput. 13(4), 315–353 (2000)
36. Holzmann, G.J.: Algorithms for automated protocol verification. AT&T Technical Journal 69(2), 32–44 (1990)
37. Holzmann, G.J.: An analysis of bitstate hashing. In: Proc. of Protocol Specification, Testing, and Verification, pp. 301–314. Chapman & Hall, Boca Raton (1995)
38. Holzmann, G.J.: State compression in SPIN: Recursive indexing and compression training runs. In: Proc. of SPIN Workshop (1997)
39. Holzmann, G.J., Godefroid, P., Pirottin, D.: Coverage preserving reduction strategies for reachability analysis. In: Proc. of Protocol Specification, Testing, and Verification (1992)

40. Holzmann, G.J., Peled, D.: An improvement in formal verification. In: Proc. of Formal Description Techniques VII, pp. 197–211. Chapman & Hall, Ltd., Boca Raton (1995)
41. Holzmann, G.J., Puri, A.: A minimized automaton representation of reachable states. Software Tools for Technology Transfer (STTT) 3(1), 270–278 (1998)
42. Holzmann, G.J., Bosnacki, D.: The design of a multicore extension of the spin model checker. IEEE Transactions on Software Engineering 33(10), 659–674 (2007)
43. Iosif, R.: Symmetry reduction criteria for software model checking. In: Bošnački, D., Leue, S. (eds.) SPIN 2002. LNCS, vol. 2318, pp. 22–41. Springer, Heidelberg (2002)
44. Ip, C.N., Dill, D.L.: Better verification through symmetry. Formal Methods in System Design 9(1–2), 41–75 (1996)
45. Jones, M.D., Sorber, J.: Parallel search for LTL violations. Software Tools for Technology Transfer (STTT) 7(1), 31–42 (2005)
46. Krimm, J.P., Mounier, L.: Compositional state space generation from Lotos programs. In: Brinksma, E. (ed.) TACAS 1997. LNCS, vol. 1217, pp. 239–258. Springer, Heidelberg (1997)
47. Kuehlmann, A., McMillan, K.L., Brayton, R.K.: Probabilistic state space search. In: Proc. of Computer-Aided Design (CAD 1999), pp. 574–579. IEEE Press, Los Alamitos (1999)
48. Kurshan, R.P., Levin, V., Yenigün, H.: Compressing transitions for model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 569–581. Springer, Heidelberg (2002)
49. Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: Efficient verification of real-time systems: Compact data structure and state-space reduction. In: Proc. of Real-Time Systems Symposium (RTSS 1997), pp. 14–24. IEEE Computer Society Press, Los Alamitos (1997)
50. Lerda, F., Sisto, R.: Distributed-memory model checking with SPIN. In: Dams, D.R., Gerth, R., Leue, S., Massink, M. (eds.) SPIN 1999. LNCS, vol. 1680, p. 22. Springer, Heidelberg (1999)
51. Lerda, F., Visser, W.: Addressing dynamic issues of program model checking. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 80–102. Springer, Heidelberg (2001)
52. Lin, F., Chu, P., Liu, M.: Protocol verification using reachability analysis: the state space explosion problem and relief strategies. Computer Communication Review 17(5), 126–134 (1987)
53. Mihail, M., Papadimitriou, C.H.: On the random walk method for protocol testing. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 132–141. Springer, Heidelberg (1994)
54. Mailund, T., Westergaard, W.: Obtaining memory-efficient reachability graph representations using the sweep-line method. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 177–191. Springer, Heidelberg (2004)
55. Ozdemir, K., Ural, H.: Protocol validation by simultaneous reachability analysis. Computer Communications 20, 772–788 (1997)
56. Parreaux, B.: Difference compression in SPIN. In: Proc. of Workshop on automata theoric verification with the SPIN model checker (SPIN 1998) (1998)
57. Pelánek, R.: Evaluation of on-the-fly state space reductions. In: Proc. of Mathematical and Engineering Methods in Computer Science (MEMICS 2005), pp. 121–127 (2005)
58. Pelánek, R.: Web portal for benchmarking explicit model checkers. Technical Report FIMU-RS-2006-03, Masaryk University Brno (2006)

59. Pelánek, R.: BEEM: Benchmarks for explicit model checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007)

60. Pelánek, R., Hanžl, T., Černá, I., Brim, L.: Enhancing random walk state space exploration. In: Proc. of Formal Methods for Industrial Critical Systems (FMICS 2005), pp. 98–105. ACM Press, New York (2005)

61. Pelánek, R., Rosecký, V., Moravec, P.: Complementarity of error detection techniques. In: Proc. of Parallel and Distributed Methods in verifiCation (PDMC) (2008)

62. Pelánek, R., Rosecký, V., Šeděnka, J.: Evaluation of state caching and state compression techniques. Technical Report FIMU-RS-2008-02, Masaryk University Brno (2008)

63. Peled, D.: Combining partial order reductions with on-the-fly model-checking. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 377–390. Springer, Heidelberg (1994)

64. Penczek, W., Szreter, M., Gerth, R., Kuiper, R.: Improving partial order reductions for universal branching time properties. Fundamenta Informaticae 43(1-4), 245–267 (2000)

65. Penna, G.D., Intrigila, B., Melatti, I., Tronci, E., Zilli, M.V.: Exploiting transition locality in automatic verification of finite state concurrent systems. Software Tools for Technology Transfer (STTT) 6(4), 320–341 (2004)

66. Pnueli, A.: In transition from global to modular temporal reasoning about programs. Logics and models of concurrent systems, 123–144 (1985)

67. Qian, K., Nymeyer, A.: Guided invariant model checking based on abstraction and symbolic pattern databases. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 497–511. Springer, Heidelberg (2004)

68. Schmidt, K.: Automated generation of a progress measure for the sweep-line method. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 192–204. Springer, Heidelberg (2004)

69. Self, J.P., Mercer, E.G.: On-the-fly dynamic dead variable analysis. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 113–130. Springer, Heidelberg (2007)

70. Sistla, A.P., Godefroid, P.: Symmetry and reduced symmetry in model checking. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 91–103. Springer, Heidelberg (2001)

71. Stern, U., Dill, D.L.: Using magnetic disk instead of main memory in the murphi verifier. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 172–183. Springer, Heidelberg (1998)

72. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Kozen, D. (ed.) Proceedings of the First Annual IEEE Symposium on Logic in Computer Science (LICS 1986), pp. 332–344. IEEE Computer Society Press, Los Alamitos (1986)

73. Visser, W.: Memory efficient state storage in SPIN. In: Proc. of SPIN Workshop, pp. 21–35 (1996)

74. Wahl, T.: Adaptive symmetry reduction. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 393–405. Springer, Heidelberg (2007)

Local Quantitative LTL Model Checking[⋆]

Jiří Barnat, Luboš Brim, Ivana Černá, Milan Češka, and Jana Tůmová

Faculty of Informatics, Masaryk University
Brno, Czech Republic
{barnat,brim,cerna,xceska,xtumova}@fi.muni.cz

Abstract. Quantitative analysis of probabilistic systems has been studied mainly from the global model checking point of view. In the global model-checking, the goal of verification is to decide the probability of satisfaction of a given property for all reachable states in the state space of the system under investigation. On the other hand, in local model checking approach the probability of satisfaction is computed only for the set of initial states. In theory, it is possible to solve the local model checking problem using the global model checking approach. However, the global model checking procedure can be significantly outperformed by a dedicated local model checking one. In this paper we present several particular local model checking techniques that if applied to global model checking procedure reduce the runtime needed from days to minutes.

1 Introduction

System design techniques employing probability are becoming widely used. They provide designers with reasonably efficient means to break symmetry in the system or to implement randomized algorithms. Probabilistic actions are also used for modeling various nondeterministic aspects such as human unpredictable decisions, occurrence of external stimuli, or simply the presence of hardware errors. As the interest in the probabilistic systems is growing, supported mainly by their potential practical use, there is also increased interest in formal techniques for their analysis and verification, model checking in particular.

There are two different tasks related to model checking over probabilistic systems. Given a formula and probabilistic system, the so called *qualitative* analysis refers to the problem of deciding whether the probabilistic system satisfies the formula with the probability one. On the other hand, the so called *quantitative* model checking refers to the problem of deciding the maximal and minimal probability the given formula is satisfied for the probabilistic system. For model checking linear time properties, the qualitative problem can be solved similarly to the nondeterministic case, i.e. using automata-based approach. The problem reduces to the problem of the detection of an *Accepting End Component* (AEC) in the graph of the underlying product of the probabilistic system and the ω-regular automaton expressing the (negation of) the verified property [22,11].

For the quantitative case the model checking procedure is a little bit more complex [3,12]. Similarly to the qualitative case, the probabilistic system is multiplied with the semi-deterministic ω-regular property automaton and all the AECs are identified in its underlying graph. After that, the graph is transformed into a linear programming problem (set of inequalities over states of the probabilistic system and an objective function to be maximized). Every variable in the linear programming instance corresponds to a state in the system in the sense that the value computed for the variable is exactly the maximal probability of satisfaction of the examined property, if the property is evaluated from the particular state. States in an AEC satisfy the examined property with the probability one.

Both qualitative and quantitative analysis of probabilistic systems has been studied mainly from the *global model checking* point of view. In the global model-checking, the goal of verification is to decide the probability of satisfaction of a given property for all reachable states in the state space of the system under investigation. On the other hand, in *local model checking* approach the probability of satisfaction is computed only for the set of initial states. In theory, it is possible to solve the local model checking problem using the global model checking approach. However, the global model checking procedure can be significantly outperformed by a dedicated local model checking one. It is a well-known fact that from practical point of view, the system designers are often interested in the probability of satisfaction of the property for some particular states only (initial state most typically). This is not taken into account in the general global model checking scheme as suggested in [3,12].

There are several software tools performing qualitative and/or quantitative probabilistic model checking. Probably the most established probabilistic model checker is the *symbolic* model checker PRISM [16]. It provides support for automated analysis of a wide range of quantitative properties for three types of probabilistic models: discrete-time Markov chains, continuous-time Markov chains and Markov decision processes (MDPs). The property specification language of PRISM incorporates the temporal logics PCTL [15] and CSL [1] as well as extensions for quantitative specifications and costs/rewards. As for *enumerative* approach to model checking, the model checker to be mentioned is LiQuor [10]. LiQuor is capable of verifying probabilistic systems modeled as ProbMeLa programs. ProbMeLa is a probabilistic guarded command language with an operational semantics based on finite MDPs. LiQuor allows qualitative and/or quantitative analysis for ω-regular linear time properties. The tool follows the standard automata-based model checking approach and involves partial order reduction technique for MDPs [4] to fight the state explosion problem. Recently, a parallel enumerative probabilistic model checker – ProbDiVinE, has been released [5]. Likewise LiQuor, ProbDiVinE provides means for verification of quantitative and qualitative linear time properties of MDPs. The unique feature of ProbDiVinE is its capability of employing combined power of multiple CPU cores available on latest hardware systems to solve large verification problems. All the techniques presented in this paper have been implemented and experimentally evaluated using the ProbDiVinE model checker. Yet another tool for

verification of MDPs is the model-checker RAPTURE [8]. It employs an automatic abstraction refinement and essential state reduction techniques to fight the state explosion problem [8].

As the main contribution of this paper we introduce several techniques that allow to improve the general *quantitative* verification procedure using the locality of the model checking goal. These local model checking techniques can be applied to the global model checking procedure resulting in a significant speed-up as indicated by our experimental evaluation. In addition, the locality of the techniques supports their natural integration into a parallel tool, giving thus further advantages in terms of speed and scalability.

The rest of the paper is organized as follows. Section 2 states the necessary definitions and recalls the general scheme of quantitative model checking procedure. Section 3 introduces our new techniques to improve the general verification scheme, Section 4 reports on experimental evaluation of these techniques, and Section 5 concludes the paper.

2   Preliminaries

In this subsection, we briefly review fundamentals of LTL model checking over finite state probabilistic systems and fix some notation.

2.1   Probabilistic Model Checking

Markov decision processes (MDPs) are used as the standard modeling formalism for asynchronous probabilistic systems, supporting both nondeterminism and probability. A Markov decision process [13,21,22], is a tuple $M = (S, Act, P, init, AP, L)$, where S is a finite set of states, Act is a finite set of actions, $P : (S \times Act \times S) \to [0, 1]$ is a probability matrix, $init \in S$ is the initial state, AP is a finite set of atomic propositions, and $L : S \to 2^{AP}$ is a labeling function. $Act(s)$ denotes the set of actions that are enabled in the state s, i.e. the set of actions $\alpha \in Act$ such that $P(s, \alpha, t) > 0$ for some state $t \in S$. For any state $s \in S$, we require that $Act(s) \neq \emptyset$ and $\forall \alpha \in Act(s). \sum_{s' \in S} P(s, \alpha, s') = 1$.

An infinite run of an MDP is a sequence $\tau = s_0, \alpha_1, s_1, \alpha_2, \ldots \in (S \times Act)^\omega$ such that $\alpha_i \in Act(s_{i-1})$. A trajectory of τ is the word $L(s_0), L(s_1), L(s_2), \ldots$ over the alphabet 2^{AP} obtained by the projection of τ to the state labels.

The intuitive operational semantics of an MDP is as follows. If s is the current state then an action $\alpha \in Act(s)$ is chosen nondeterministically and is executed leading to a state t with probability $P(s, \alpha, t)$. We refer to t as an α-*successor* of s if $P(s, \alpha, t) > 0$. State s is called *deterministic* if exactly one action is enabled in s. If all states of an MDP are deterministic, the MDP is called *Markov chain*. To resolve the nondeterminism of an MDP a *scheduler* function is used. We consider deterministic history dependent schedulers which are given by a function D assigning an action $D(\sigma) \in Act(s_n)$ to every finite run $\sigma = s_0, \alpha_1, \ldots, \alpha_n, s_n$. Given a scheduler D, the behavior of M under D can be formalized as a Markov chain.

Let M be a Markov Chain, $s \in S$ be a state of M, and X be a set of runs of M originating at s. We define *the probability of the set X* as a measure of the set X in the set of all runs of M originating at s. A set X of runs of a Markov Chain M is called *basic cylinder set* if there is a prefix $s_0, \alpha_1, \ldots \alpha_n, s_n$ such that X contains exactly all runs of M with that prefix. The probability measaure of a basic cylinder set with prefix $s_0, \alpha_1, \ldots \alpha_n, s_n$ is then

$$\prod_{i=0}^{n-1} P(s_i, \alpha_{i+1}, s_{i+1}).$$

If the set X of runs of M is not a basic cylinder set, its measure is determined as a sum of measures of maximal (w.r.t. inclusion) basic cylinder sets fully contained in X [11].

In this paper we focus on the *quantitative model checking* of MDPs against properties specified in Linear temporal logic (LTL). Formulas of LTL are built over a set AP of atomic propositions and are closed under the application of Boolean connectives, the unary temporal connective X (next), and the binary temporal connective U (until). LTL is interpreted over computations. A *computation* is a function $\pi : \omega \to 2^{AP}$, which assigns truth values to the elements of AP at each time instant and as such it can be viewed as an infinite word over the alphabet 2^{AP}. For an LTL formula φ, we denote by $\mathcal{L}(\varphi)$ the set of all computations satisfying φ.

A run of a Markov chain satisfies the formula φ, if the trajectory of the run is in $\mathcal{L}(\varphi)$. A Markov Chain M satisfies the formula φ with probability p, if the set of runs of M satisfying the formula has the probability p. An MDP M satisfies the formula φ with the probability at least p (at most p) if for every scheduler D, M under D satisfies the formula with the probability at least p (at most p). The problem of quantitative model checking is to determine the minimal and/or maximal probability that an MDP satisfies a given property. Note that for the computation of the minimal and/or maximal probability that an MDP satisfies an ω-regular property, it is sufficient to consider only history independent schedulers [12].

The goal of the *global* quantitative model checking is to calculate the minimal and/or maximal probability of the satisfaction of the property for every state s of an MDP. The goal of the *local* quantitative model checking is, however, to determine the minimal and/or maximal probability of satisfaction of the property for the initial state only.

A Büchi automaton is a tuple $A = (\Sigma, Q, q_{init}, \delta, F)$, where Σ is a finite alphabet, Q is a finite set of states, $q_{init} \in Q$ is an initial state, $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation, and $F \subseteq Q$ is a set of *accepting states*. A run of A over an infinite word $w = a_1 a_2 \ldots \in \Sigma^\omega$ is a sequence $q_0, q_1, \ldots$, where $q_0 = q_{init}$ and $(q_{i-1}, a_i, q_i) \in \delta$ for all $i \geq 1$. Let $\inf(\rho)$ denote the set of states that appear in the run ρ infinitely often. A run ρ is accepting iff $\inf(\rho) \cap F \neq \emptyset$. A state $s \in Q$ of a Büchi automaton A is called *deterministic* if and only if for all $a \in \Sigma$ there is at most one $s' \in A$ such that $(s, a, s') \in \delta$. A Büchi automaton is *deterministic*

in the limit if and only if all the accepting states and their descendants are deterministic [11].

We use the automata based approach to probabilistic LTL model checking. Given an LTL formula φ, it is possible to build a Büchi automaton A with $2^{\mathcal{O}(|\varphi|)}$ states such that $L(A) = \mathcal{L}(\varphi)$ [23]. Moreover, for any Büchi automaton A with n states a Büchi automaton B with $2^{\mathcal{O}(n)}$ state such that B is *deterministic in the limit* and $L(A) = L(B)$ can be built [11]. Similarly to model checking non-probabilistic systems, the model is synchronized with the automaton corresponding to the negation of the formula in the case we are interested in the minimal probability or with the automaton corresponding to the formula in the case we are interested in the maximal probability. However, unlike the non-probabilistic case, automata which are deterministic in the limit have to be used instead of non-deterministic Büchi automata.

Let $M = (S, Act, P, s_0, AP, L)$ be an MDP and let $A = (Q, 2^{AP}, q_0, \Delta, F)$ be a Büchi automaton. The synchronized product of M and A is an extended MDP $M \times A = (S \times Q, Act_{M \times A}, P_{M \times A}, init, AP, L_{M \times A}, Acc)$, where $Act_{M \times A}((s, p)) = Act(s)$, $P_{M \times A}((s, p), \alpha, (t, q)) = P(s, \alpha, t)$ if $(p, L(s), q) \in \delta$ or 0 otherwise, $init = (s_0, q_0)$, $L_{M \times A}((s, t)) = L(s)$, and $Acc = S \times F$ is the set of accepting states. Note that the synchronized product is not a regular MDP as it distinguishes between accepting and non-accepting states and may contain states without enabled actions.

In order to describe the algorithmic solution to the quantitative LTL model checking we often view an MDP or MDP synchronized with a Büchi automaton as a graph. Therefore, we recall some basic notions from the graph theory. A state s' is *reachable* from a state s in a set of states $R \subseteq S$, denoted as $s \leadsto_R^+ s'$ iff there is a sequence of states $s_0, s_1, \ldots, s_k \in R$ such that $s = s_0, s' = s_k$ and for all $0 \leq i < k$ there is an action $\alpha \in Act(s_i)$ such that $P(s_i, \alpha, s_{i+1}) > 0$. A set of states R is strongly connected if for all $r, r' \in R : r \leadsto_R^+ r'$ or $|R| = 1$. A *strongly connected component* (SCC) is a maximal strongly connected set of states. The graph of strongly connected components of G is called the *quotient graph* of G. An SCC C is *trivial* if $|C| = 1$. An SCC is *terminal* if it has no successors in the quotient graph. For every component C let $Input(C) = \{c \in C \mid$ there is an SCC $C' : \exists c' \in C' : \exists \alpha \in Act(c') : P(c', \alpha, c) > 0\}$ if $init \notin C$ otherwise $Input(C) = \{init\}$. Furthermore, for each nonterminal component C we define $Output(C) = \{s \in S \setminus C \mid \exists c \in C : \exists \alpha \in Act(c) : P(c, \alpha, s) > 0\}$.

Given an MDP graph G, an *accepting end component* (AEC) is a maximal set C of states of G that forms (not necessary maximal) strongly connected component in G such that $C \cap Acc \neq \emptyset$ and if there is an enabled action α in a state of the component, the component contains either all the α-successors or none of them [12]. From [13,14] it follows that for any state s of an MDP graph of $M \times A$ that belongs to an AEC, there exists a scheduler D such that the probability measure of runs originating in s and remaining in the AEC is 1 in M under D.

Let s be a state in the MDP product graph. We define the maximal probability x_s of reaching an AEC from s as follows. If s belongs to an AEC, $x_s = 1$, if no
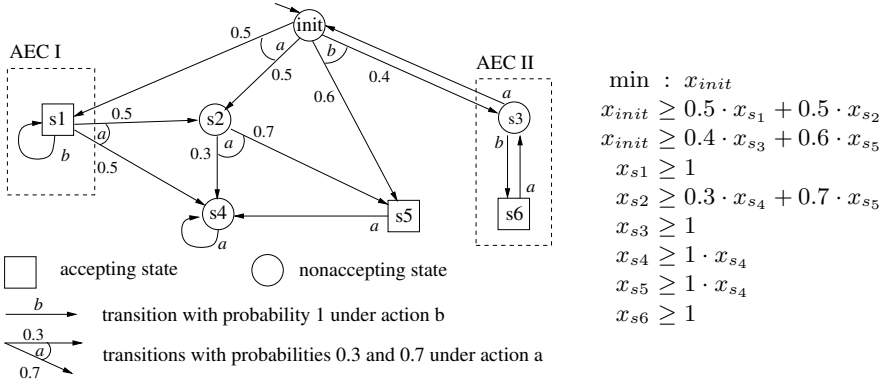
Fig. 1. MDP and its corresponding local linear programming problem

AEC is reachable from s, $x_s = 0$. For remaining cases the value of x_s can be calculated by solving the linear programming problem with inequalities

$$x_s \geq \sum_{v \in S \times Q} P_{M \times A}(s, \alpha, v) \cdot x_v \quad \forall \alpha \in Act_{M \times A}(s)$$

minimizing the objective function $f = \sum_{u \in S \times Q} x_u$. For more details see [3,12].
Note that in the context of local model checking the objective function can be simplified to $f = x_{init}$. An example is given in Figure 1.

After the solution of the linear programming problem is found, x_{init} contains the value of maximal probability an AEC is reached from the state $init$. If the MDP was synchronized with the automaton corresponding to the negation of a formula φ, the minimal probability the MDP satisfies the formula φ is $1 - x_{init}$. If the MDP was synchronized with the automaton corresponding to a formula ψ (without negation), the maximal probability the MDP satisfies ψ is x_{init}.

2.2   Algorithm

The algorithm for finding all AECs was introduced in [11,12] and it was based on recursive decomposition of MDP graph into SCCs. Our approach employs a parallel adaptation of the algorithm of Bianco and de Alfaro (BdA) [7] that computes a set of states for which there exists a scheduler such that the maximal probability of reaching an AEC from the set is equal to 1. Clearly, this set can be used instead of the set of all AECs. Henceforward, the set is refered to as AS.

The algorithm maintains an *approximation set* of states that may belong to an AEC. The algorithm repeatedly refines the approximation set by locating and removing states that cannot belong to an AEC, we call this a *pruning step*. The algorithm for quantitative verification is obtained by a modification of BdA. As the final approximation set is AS, the linear programming problem is extended with inequalities $x_u \geq 1$ for all $x_u \in AS$. The overall scheme of how the algorithm proceeds is given as Algorithm 1.

Algorithm 1. Scheme of algorithm for local quantitative analysis

---

1: compute the set AS using BdA algorithm
2: create the linear programming problem LP
3: compute the solution of LP
4: **return** x_{init}

---

3   Local Model Checking Techniques

In this section we introduce three optimization techniques that can significantly speed-up the verification process. We also propose a way how these techniques can be employed in a parallel environment, shared-memory multi-core architectures in our case. This is in particular very important in handling very large real-life systems in practice.

3.1   Minimal Subgraph Identification

The first of the proposed algorithmic modifications helps to reduce the size of the linear programming problem by pruning the MDP product $M \times A$ into the so called minimal subgraph.

The probability of a state depends on the probabilities of its successors. However, once we know that the probability of a state is 1, we do not need to know the exact probabilities of its successors. Also, the probability of a state is 0 if no state with probability 1 can be reached from it. Henceforward, we say that a state is *relevant* if it is on a path from an initial state to a state with probability 1 such that the path does not contain any other state with probability 1. The last state on the path is referred to as a *seed*. Relevant states define in a natural way a slice in the original MDP (see the example in Figure 2). We call this slice a *minimal subgraph*. The probability of the initial state is fully determined by the states in the minimal subgraph only.

With the minimal subgraph we associate the linear programming problem mLP to be minimized in the following way. Let $init, s_0, s_1, \ldots, s_{r-1}, s_r$ be a path in the minimal subgraph from the initial state $init$ to a seed s_r. We add to mLP the inequalities of form:

$$x_{init} \geq \ldots + p_{s_0} x_{s_0} + \ldots$$
$$x_{s_0} \geq \ldots + p_{s_1} x_{s_1} + \ldots$$
$$\vdots$$
$$x_{s_{r-1}} \geq \ldots + p_{s_r} x_{s_r} + \ldots$$

It is not difficult to prove that pruning the original MDP graph into the minimal one does not have any influence on the solution of the linear programming problem.
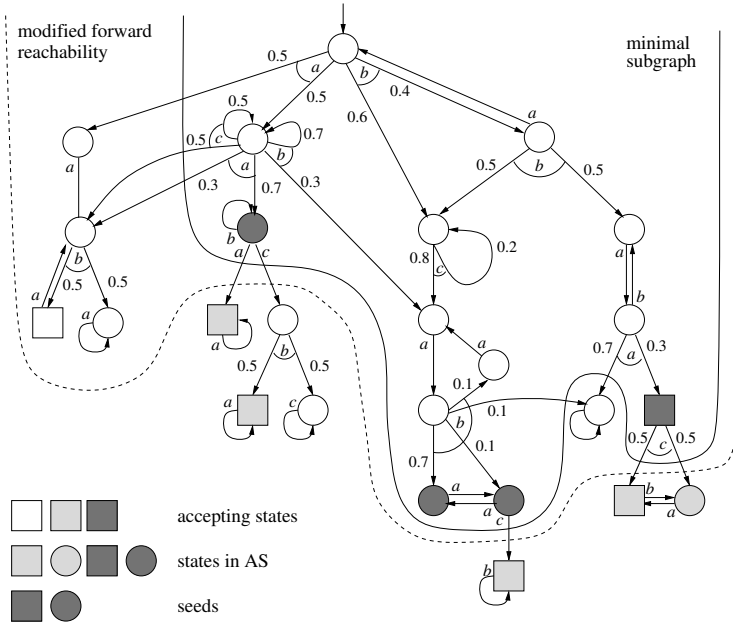
Fig. 2. Minimal subgraph identification

Lemma 1. *Let M be a synchronous product of MDP and Büchi automaton, and MG its minimal subgraph. The solution of linear programming problem LP is equal to the solution of linear programming problem mLP.*

Having computed AS we can identify relevant states, i.e. the minimal subgraph, as follows. First, we run a forward reachability from the initial state that does not explore states beyond a state from AS. States from AS visited in this reachability are the seeds. Second, we run backward reachability from seeds to identify the minimal subgraph. All states visited by the backward reachability are relevant states. In this manner we omit states whose probability of reaching an AEC equals to zero. The result of applying both forward and backward reachability to obtain the minimal subgraph is illustrated in Figure 2.

3.2 Iterative Computation

Another practical technique is to decompose the given problem into simpler subtasks. In this way we have a good chance to end-up with a set of smaller linear programming problems that can be solved much faster.

The core idea is to decompose the minimal subgraph into SCCs, create the appropriate quotient graph, and then iteratively solve the linear programming problem by solving the subproblems given by the individual components in a bottom-up manner.

Some subtasks can be solved independently, which provides a basis for an effective parallel procedure as described in Subsection 3.4. Furthermore, we show in Subsection 3.3 that some subtasks can be solved without employing an external LP solver. Iterative computation can lead to a significant speed-up as compared to the computation of the entire LP problem (see Section 4).

Let us consider a minimal subgraph MG, its strongly connected components component C is formed by all inequalities $x_s \geq \ldots$ from mLP such that $s \in C$. The objective function of LP_C minimizes the sum of variables x_s, $s \in Input(C)$, i.e. states with a predecessor outside the component, or the value x_{init} in case $init \in C$.

The solution of LP_C for each component C depends only on C itself and on the states in $Input(C_t)$ for each immediate successor component C_t of C, as only variables corresponding to these states appear in the inequalities. This means, that for a terminal SCC T we can find the solution of LP_T directly. Once we have solutions for all successor components C_t of C, we can substitute all the variables x_s, such that $s \notin C$, with already computed values. LP_C does not depend on components C_t any more and the solution of LP_C can thus be computed. We call the SCC C *solved* if LP_C has been solved, i.e. the values x_s for all $s \in Input(C)$ have been computed. An unsolved SCC C is called *prepared* if for all $t \in Output(C)$ the state t is in a solved SCC.

Lemma 2. *For each $s \in Input(C)$, the solution of LP_C assigns to x_s the value equal to the maximal probability that the set AS is reachable from s.*

Corollary 1. *For the component C containing the initial state* init*, the solution of LP_C assigns to x_{init} the value equal to the maximal probability AEC is reachable from init.*

The pseudo-code of the iterative computation is described in Algorithm 2.

Algorithm 2. Iterative computation

Require: minimal subgraph MG
 1: decompose MG into SCCs
 2: build the quotient graph of MG
 3: **while** there is an unsolved SCC **do**
 4:     compute the set P of prepared SCCs
 5:     **for all** $C \in P$ **do**
 6:         create the linear programming problem LP_C
 7:         substitute for x_s such that $s \in Output(C)$ in LP_C
 8:         compute the solution of LP_C
 9:         mark C as solved
10:     **end for**
11: **end while**
12: **return** x_{init}
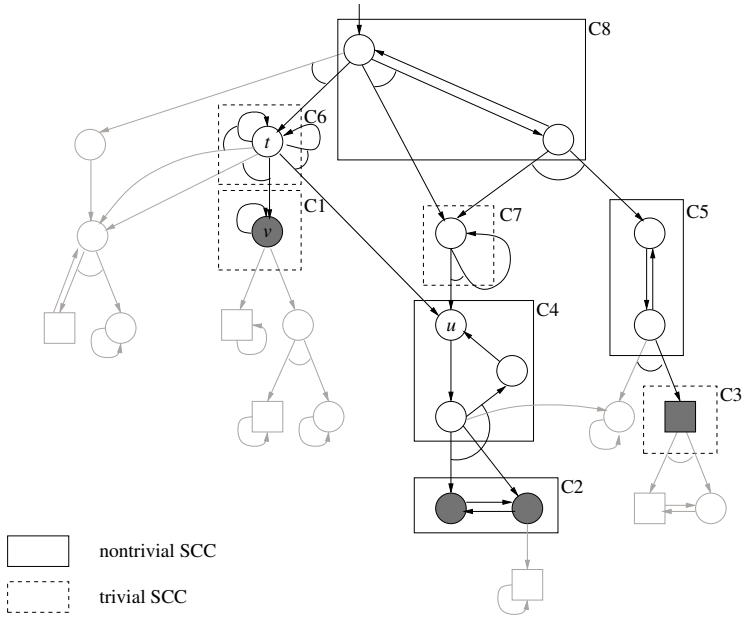
Fig. 3. SCC decomposition

Figure 3 shows the decomposition into strongly connected components. Components $C1, C2$ and $C3$ are terminal and thus prepared. After they are solved, components $C4$ and $C5$ become prepared. In the next iteration of the algorithm, components $C6$ and $C7$ are prepared and finally, after their solution, the component $C8$ containing the initial state is ready to be solved.

3.3   Trivial SCC

Let us suppose we perform the iterative computation on the minimal subgraph MG as introduced in the previous Subsection, and let the next *prepared* SCC to be solved is a *trivial* strongly connected component $T = \{t\}$ with the corresponding linear programming subtask LP_T. In the following we show, that the linear programming subtask LP_T can be solved without employing an external LP solver.

Let us firstly recall that due to deAlfaro [12] there is a *history independent* scheduler that yields the maximum value for the state t. We denote by x_t^{α} the probability of the state t under the history independent scheduler choosing the action $\alpha \in Act(t)$ whenever the state t is visited. Since it is sufficient to consider only history independent schedulers for the computation of the probability x_t of the state t, it follows directly that

$$x_t = \max_{\alpha \in Act(t)} x_t^{\alpha}.$$

Lemma 3 says how to compute the value of x_t^α. Before we state the lemma, we introduce the necessary notation. Suppose an action α to be executed. Let $u_0, u_1, \ldots, u_n$ be the α-successors of t that are outside the component T. Each u_i is reached with the probability $P(t, \alpha, u_i)$ for $0 \leq i \leq n$. Let us denote these probabilities $p_{u_0}^\alpha, p_{u_1}^\alpha, \ldots, p_{u_n}^\alpha$, respectively. Since the states are outside the component T, the probability values for these states are already known and are refered to as $v_{u_0}, v_{u_1}, \ldots, v_{u_n}$. Futhermore, we denote the probability $P(t, \alpha, t)$ by p_t^α.

Lemma 3. *Let* $pv_u^\alpha = p_{u_0}^\alpha v_{u_0} + p_{u_1}^\alpha v_{u_1} + \ldots + p_{u_n}^\alpha v_{u_n}$. *Then,*

$$x_t^\alpha = \begin{cases} \frac{pv_u^\alpha}{1 - p_t^\alpha} & \text{if } p_t^\alpha \neq 1 \\ 0 & \text{otherwise.} \end{cases}$$

Proof. For a Markov chain the following holds:

$$\sum_{i=0}^{n} p_{u_i}^\alpha + p_t^\alpha \leq 1$$

Therefore, if $p_t^\alpha = 1$ then $p_{u_i}^\alpha = 0$ for all $0 \leq i \leq n$ and thus $x_t^\alpha = 0$. Otherwise

$$x_t^\alpha = pv_u^\alpha + p_t^\alpha (pv_u^\alpha) + (p_t^\alpha)^2 (pv_u^\alpha) + (p_t^\alpha)^3 (pv_u^\alpha) + (p_t^\alpha)^4 (pv_u^\alpha) + \ldots =$$
$$pv_u^\alpha (1 + p_t^\alpha + (p_t^\alpha)^2 + (p_t^\alpha)^3 + (p_t^\alpha)^4 + \ldots) = pv_u^\alpha \frac{1}{1 - p_t^\alpha} \qquad \square$$

To compute the value of x_t we enumerate the values x_t^α according to the previous Lemma and compute their maximum. For an example, we refer to Figures 2 and 3. The component $C6$ containing the state t is a trivial one. After the components $C1$ and $C4$ are solved, we have $v_u = 0.9$, $u \in Input(C4)$ and $v_v = 1$, $v \in Input(C1)$. The value of x_t can be now computed without employing the external LP solver. Altogether, there are three actions a, b, c enabled in t resulting in the following three cases:

$$x_t^a = \frac{pv_v^a}{1 - p_t^a} = \frac{0.7 \cdot 1}{1} = 0.7 \quad x_t^b = \frac{pv_u^b}{1 - p_t^b} = \frac{0.3 \cdot 0.9}{1 - 0.7} = 0.9 \quad x_t^c = \frac{0}{1 - 0.5} = 0$$

Finally, $x_t = \max(x_t^a, x_t^b, x_t^c) = 0.9$.

3.4   Parallelization

The improved algorithm, described as Algorithm 3, consists of several consecutive phases, each of them parallelized to a certain level.

Parallel version of BdA algorithm performs qualitative model checking and computes AS as a basis for quantitative verification. The main idea builds on the topological sort for cycle detection – an algorithm that does not depend on DFS postorder and can be thus parallelized reasonably well. Minimal Subgraph Identification employs only one forward and one backward reachability and thus

---

Algorithm 3. Improved algorithm for local quantitative analysis

---

1: compute the set AS using **parallel** version of BdA algorithm
2: compute the minimal subgraph MG using **parallel** reachability
3: decompose MG into SCCs using **parallel** OBF algorithm
4: build the quotient graph of MG in **parallel**
5: **while** there is an unsolved SCC **do**
6:    compute the set P of prepared SCCs
7:    **for all** $C \in P$ **do**
8:       **in parallel do**
9:          **if** C is trivial **then**
10:             compute the solution of C
11:          **else**
12:             create the linear programming problem LP_C
13:             substitute for x_s such that $s \in Output(C)$ in LP_C
14:             compute the solution of LP_C
15:          **end if**
16:          mark C as solved
17:       **end in parallel**
18:    **end for**
19: **end while**
20: **return** x_{init}

---

this phase is parallelized effectively. In order to parallelize SCC Decomposition, the implementation is based on recursive variant of OBF algorithm as described in [6]. Iterative Computation allows to solve prepared SCCs independently by parallel running threads. However, each component has to be solved by calling the external serial LP solver `lpsolve`. This last limitation could be eventually relaxed by a parallel LP solver (we were unfortunately not able to get access to a suitable free parallel solver).

4   Experimental Evaluation

We have implemented all the described algorithms and techniques in the tool called PROBDIVINE. The tool uses DIVINE LIBRARY [20] and a generally available LP solver `lpsolve`. We ran a set of experiments on machines equipped with Intel Xeon 5130 and AMD Opteron 885 processors allowing us to measure the performance of the tool when using 1 to 8 threads.

We have used five different experimental models of randomized protocols with properties yielding minimal probability other than 0 or 1:

- Cons – randomized consensus protocol [2]
- Crypts – randomized dining cryptographers [9]
- Leads – asynchronous leader election protocol [18]
- Phils – randomized dining philosophers [19]
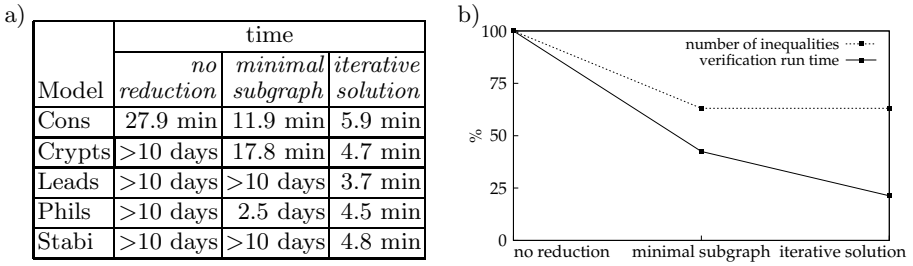- Stabi – randomized self-stabilizing protocol [17]

a)

	time		
Model	*no reduction*	*minimal subgraph*	*iterative solution*
Cons	27.9 min	11.9 min	5.9 min
Crypts	>10 days	17.8 min	4.7 min
Leads	>10 days	>10 days	3.7 min
Phils	>10 days	2.5 days	4.5 min
Stabi	>10 days	>10 days	4.8 min

b)

Fig. 4. a) Runtimes for various models when no reduction is used, when only the minimal subgraph is considered, and when both the minimal subgraph and iterative processing is involved. **b)** Correlation between the number of inequalities and runtime.

Table 1 captures the size of the linear programming problem (the number of inequalities to be solved by an external LP solver). The column *whole graph* gives the size before applying any reduction, the column *reduced graph* gives the size when redundant inequalities were removed by pruning the graph into the minimal subgraph. The column *largest problem* gives the maximal size of a problem to be solved by an LP solver, when the original problem was decomposed into subproblems that were processed independently. Three of the models contain only trivial SCCs, thus the LP solver is not called at all and the size of the largest problem solved by LP solver is thus 0.

The table in Figure 4 demonstrates that the size and the structure of the problem plays a crucial role in the performance of the tool. The table gives overall run times corresponding to the used reduction techniques. The first column gives run time when no reduction technique is used (*no reduction*), the second one when redundant inequalities are removed (*minimal subgraph*), and the third one when the technique of iterative computation and trivial SC solving are applied on the minimal subgraph (*iterative solution*). A correlation between times in Figure 4 and sizes in Table 1 is observable. With decreasing number of inequalities the runtimes tend to speed-up dramatically. As for the speed-up, the most inconvenient case is when the graph is made of one large component. In such a case, the pruning and parallel processing cannot be done and the verification runtime is dominated by the single call to the LP solver.

Table 1. The size of LP problem with respect to used reduction techniques

		# inequalities for LP solver			% of the whole graph	
Model	# states	*whole graph*	*reduced graph*	*largest problem*	*reduced graph*	*largest problem*
Cons	48 669	132 243	83 395	20 368	63.06	15.40
Crypts	2 951 903	8 954 217	108 045	0	1.21	0
Leads	2 995 379	8 800 096	5 678 656	0	64.5	0
Phils	5 967 065	14 740 726	1 623 722	246	11.0	Almost 0
Stabi	4 061 570	6 897 480	5 983 080	0	86.7	0
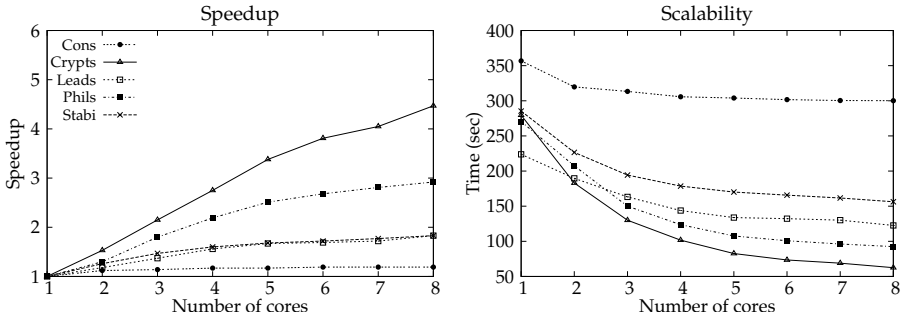
Fig. 5. Overall speed-up and scalability

In the case of *Cons* model the reduction techniques help less than in the other cases. However, run times decrease still significantly. Figure 4.a) gives runtimes of verification when various degree of improvement is used. Figure 4.b) depicts the relative decrease in the runtime and number of inequalities when various reduction techniques are involved. In particular, the number of inequalitites decreases to 63% of the original number, while the runtime decreases to 21% of the original time needed to perform the verification task.

Figure 5 reports on the overall speed-up and scalability of the verification process we achieved using our tool on various number of CPU cores. Poor scalability in case of *randomized consensus protocol* can be explained, because the time consumed by the sequential LP solver takes the major part of the runtime of the whole verification process.

Figure 6 aims on the qualitative analysis as a part of the whole verification process. The table in Figure 6 shows the ratio between runtimes of the qualitative analysis and the whole verification process. The graph in Figure 6 presents speed-up of qualitative analysis (the first phase of the algorithm). In comparison to the quantitative verification, the speed-up is much better due to the fact, that the whole verification process contains phases where parallelization does not help

Model	qualitative analysis	whole analysis	% qualit of whole
Cons	30.53	358.15	8.52
Crypts	275.96	279.47	98.75
Leads	68.28	223.76	30.51
Phils	180.6	270.46	66.77
Stabi	67.36	285.34	23.61

Fig. 6. Ratio between runtimes of the qualitative analysis and the whole verification process (table on the left). Speed-up of qualitative analysis only (on the right).

much as they require frequent synchronization. We can observe, that the bigger the part the qualitative analysis forms in the whole verification process, the better the overall scalability is. In case of *dining cryptographers protocol* model, the qualitative verification forms 98.75 % of 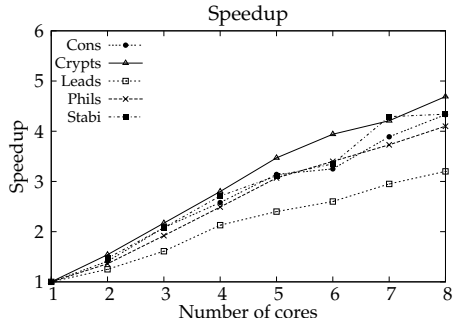the whole verification process and all the LPs are solved without using external LP solver. Therefore the scalability and speed-up are the best over all the examples.

All in all, we claim that our approach is quite successful as overall runtimes tend to decrease as more CPU cores are used.

The structure of a graph is a crucial aspect affecting the runtime of the verification process. For instance the *Crypts* model contains approximately the same number of states as *Leads*, but runtimes of qualitative verification differ a lot. On the other hand, runtime of *Leads* is comparable to *Stabi*, but their number of states and speedup differ.

5 Conclusion

As probabilistic systems gain popularity and are coming into wider use, the need for formal verification and analysis methods, techniques and tools capable of handling these systems become more critical. The theory and algorithms for formal verification of probabilistic systems have been around for some time. However, it is the existence of a good and efficient formal verification tool that makes the theory valid from the practitioner's point of view.

In this paper we presented several techniques that allow to build competitive enumerative model checking tool for quantitative analysis of linear temporal properties over finite state probabilistic systems. In particular, we showed how to involve parallelism and employ locality to increase the performance of such a tool. We also showed that the costly call to the linear programming solver can be either replaced with multiple successive calls for smaller problems, or avoided at all. Using this approach we achieved order-of-magnitude reduction in runtime of verification in many cases.

References

1. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.K.: Verifying continuous-time Markov Chains. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 269–276. Springer, Heidelberg (1996)
2. Aspnes, J., Herlihy, M.: Fast randomized consensus using shared memory. Journal of Algorithms 15(1), 441–460 (1990)
3. Baier, C.: On the Algorithmic Verification of Probabilistic Systems. Habilitation Thesis, Universität Mannheim (1998)
4. Baier, C., Größer, M., Ciesinski, F.: Partial Order Reduction for Probabilistic Systems. In: 1st International Conference on Quantitative Evaluation of Systems (QEST 2004), pp. 230–239. IEEE Computer Society, Los Alamitos (2004)
5. Barnat, J., Brim, L., Černá, I., Češka, M., Tůmová, J.: ProbDiVinE-MC: Multi-Core LTL Model Checker for Probabilistic Systems. In: Proceedings of QEST 2008, Tool Paper. IEEE, Los Alamitos (2008) (to appear), `http://anna.fi.muni.cz/probdivine`

6. Barnat, J., Chaloupka, J., van de Pol, J.: Improved Distributed Algorithms for SCC Decomposition. Electron. Notes Theor. Comput. Sci. 198(1), 63–77 (2008)
7. Bianco, A., de Alfaro, L.: Model checking of probabilistic and nondeterministic systems. In: Thiagarajan, P.S. (ed.) FSTTCS 1995. LNCS, vol. 1026, pp. 499–513. Springer, Heidelberg (1995)
8. Jeannet, B., de Argenio, P., Larsen, K.G.: RAPTURE: A tool for verifying Markov Decision Processes. In: Proc. Tools Day / CONCUR 2002. Tech. Rep. FIMU-RS-2002-05. MU Brno, pp. 84–98 (2002)
9. Chaum, D.: The dining cryptographers problem: Unconditional sender and recipient untraceability. Journal of Cryptology 1, 65–75 (1988)
10. Ciesinski, F., Baier, C.: LiQuor: A tool for Qualitative and Quantitative Linear Time analysis of Reactive Systems. In: Proc. of QEST 2006, pp. 131–132. IEEE Computer Society, Los Alamitos (2006)
11. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. Journal of the ACM 42(4), 857–907 (1995)
12. de Alfaro, L.: Formal Verification of Stochastic Systems. PhD thesis, Stanford University, Department of Computer Science (1997)
13. Derman, C.: Finite State Markovian Decision Processes. Academic Press, Inc., Orlando (1970)
14. Doob, J.L.: Measure theory. Springer, Heidelberg (1994)
15. Hansson, H., Jonsson, B.: A Framework for Reasoning about Time and Reliability. In: IEEE Real-Time Systems Symposium, pp. 102–111 (1989)
16. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
17. Israeli, A., Jalfon, M.: Token management schemes and random walks yield self-stabilizing mutual exclusion. In: Proc. ACM Symposium on Principles of Distributed Computing, pp. 119–131 (1990)
18. Itai, A., Rodeh, M.: Symmetry breaking in distributed networks. Information and Computation 88(1) (1990)
19. Lehmann, D., Rabin, M.: On the advantage of free choice: A symmetric and fully distributed solution to the dining philosophers problem (extended abstract). In: Proc. 8th Annual ACM Symposium on Principles of Programming Languages (POPL 1981), pp. 133–138 (1981)
20. ProbDiVinE homepage (2008), `http://anna.fi.muni.cz/probdivine`
21. Puterman, M.L.: Markov Decision Processes-Discrete Stochastic Dynamic Programming. John Wiley &Sons, New York (1994)
22. Vardi, M.Y.: Probabilistic linear-time model checking: an overview of the automata-theoretic approach. In: Katoen, J.-P. (ed.) AMAST-ARTS 1999, ARTS 1999, and AMAST-WS 1999. LNCS, vol. 1601, pp. 265–276. Springer, Heidelberg (1999)
23. Vardi, M.Y., Wolper, P.: Reasoning about infinite computation paths. In: Proceedings of 24th IEEE Symposium on Foundation of Computer Science, Tuscan, pp. 185–194 (1983)

Efficient Symbolic Model Checking
for Process Algebras[*]

José Vander Meulen[1] and Charles Pecheur[2]

[1] Université catholique de Louvain
`jose.vandermeulen@uclouvain.be`
[2] Université catholique de Louvain
`charles.pecheur@uclouvain.be`

Abstract. Different approaches have been developed to mitigate the
state space explosion of model checking techniques. Among them, sym-
bolic verification techniques use efficient representations such as BDDs
to reason over sets of states rather than over individual states. Unfor-
tunately, past experience has shown that these techniques do not work
well for loosely-synchronized models. This paper presents a new algo-
rithm and a new tool that combines BDD-based model checking with
partial order reduction (POR) to allow the verification of models featur-
ing asynchronous processes, with significant performance improvements
over currently available tools. We start from the ImProviso algorithm
(Lerda et al.) for computing reachable states, which combines POR and
symbolic verification. We merge it with the FwdUntil method (Iwashita
et al.) that supports verification of a subset of CTL. Our algorithm has
been implemented in a prototype that is applicable to action-based mod-
els and logics such as process algebras and ACTL. Experimental results
on a model of an industrial application show that our method can ver-
ify properties of a large industrial model which cannot be handled by
conventional model checkers.

1   Introduction

Model checking is a technique used to verify concurrent systems such as sequen-
tial circuit designs and communication protocols, by exhaustively exploring the
state space of a finite-space description of the processes involved. The properties
to be verified on such systems are typically expressed in (linear or branching)
temporal logics such as LTL, CTL or CTL*, with different algorithms and com-
plexities depending on the logic used. In particular, McMillan achieved a break-
through with the use of symbolic representations based on the use of Ordered
Binary Decision Diagrams (BDD) [1] to perform model checking of CTL, making
it possible to verify systems with a very large number of states [2].

Unfortunately, the size of the state space to be explored is frequently pro-
hibitive due, among other causes, to the modeling of concurrency by interleaving.
The aim of partial order reduction (POR) techniques is to reduce the number of
interleaving sequences that must be be considered. When a specification cannot

---

distinguish between two interleaving sequences that differ only by the order in which concurrently executed events are taken, it is sufficient to analyse one of them [3].

This paper presents a new algorithm and tool that combine BDD-based model checking with partial order reduction (POR) to allow the verification of models featuring asynchronous processes, with significant performance improvements over currently available tools. We start from the ImProviso algorithm of Lerda et al. [4] for computing reachable states, which combines POR and symbolic verification. We merge it with the FwdUntil method of Iwashita et al. [5] that supports verification of a subset of CTL. Our algorithm has been implemented in a prototype that is applicable to action-based models and logics such as LOTOS [6] and ACTL [7].

The main contributions of this paper are the FwdUntilPOR algorithm that combines POR and forward CTL model checking, a new symbolic model checker which implements this algorithm and is applicable to action-based models and logics, and an experimental evaluation of this algorithm and tool on a realistic-sized model.

The remainder of the paper is structured as follows. In Section 2, we introduce some background concepts, definitions and notations that are used throughout the paper. In Section 3, we review the Two-Phase algorithm for POR, and its symbolic incarnation in *ImProviso*. In Section 4, we discuss a forward approach to CTL symbolic model-checking, built around the *FwdUntil* operator. In Section 5, we present our own combination of ImProviso and FwdUntil, leading to the *ForwardUntil-POR* algorithm. In Section 6, we present a prototype of a new symbolic model checker implementing the FowardUntilPOR method. In Section 7, we present the results obtained by applying our method on a case study. Section 8 reviews related works. In Section 9, we give conclusions as well as directions for future work.

2   Background

2.1   Transitions Systems

We model the behaviour of a system as a set T of transitions over some state space S, where each transition is a binary relation over states. Formally, let AP be a set of atomic propositions. A *state transition system* is a four tuple $M = (S, T, S_0, L)$ where S is a finite set of states, $S_0 \subseteq S$ is the set of initial states, T is a set of transitions such that for each $\alpha \in T$, $\alpha \subseteq S \times S$ and $L : S \to 2^{AP}$ is a function that labels each state with the set of atomic propositions that are true in that state.

We write $s \xrightarrow{\alpha} s'$ for $(s, s') \in \alpha$. A transition α is *enabled* in a state s iff there is a state s' such that $s \xrightarrow{\alpha} s'$. We write $enabled(s)$ for the set of enabled transitions in s. A transition α is *deterministic* in a state s iff there is at most one s' such that $s \xrightarrow{\alpha} s'$.

We define the classical *pre-* and *post-image* of a set of states X over a set of transitions R, used in backward and forward state-space traversal respectively:

$$pre(R, X) = \{s' \in S | \exists s \in X, \alpha \in R \cdot s' \xrightarrow{\alpha} s\}$$
$$post(R, X) = \{s' \in S | \exists s \in X, \alpha \in R \cdot s \xrightarrow{\alpha} s'\}$$

2.2   Partial-Order Reduction

The goal of the partial-order reduction methods (POR) is to reduce the number of states explored by model-checking, by not exploring different equivalent interleavings of concurrent events. Naturally, these methods are best suited for strongly asynchronous programs. Interleavings which are required to be preserved may depend on the property to be checked.

Partial-order reduction is based on the notions of *independence* between transitions and *invisibility* of a transition with respect to a property. Two transitions are *independent* if they do not disable one another and executing them in either order results in the same state. A transition is *invisible* with respect to a property f when its execution from any state does not change the value of the atomic propositions in f. Intuitively, if two independent transitions α and β are invisible w.r.t. the property f that one wants to verify, then it does not matter whether α is executed before or after β, because they lead to the same state and do not affect the truth of f.

Partial-order reduction consists in identifying such situations and restricting the exploration to either of these two alternatives. In effect, POR amounts to exploring a reduced model $M' = (S', T', S_0, L)$ with $S' \subseteq S$ and for each $\alpha' \in T'$ there is an $\alpha \in T$ such that $\alpha' \subseteq \alpha$. In practice, classical POR algorithms [3,8] execute a modified depth-first search (DFS). At each state s, an adequate subset $ample(s)$ of the transitions enabled in s are explored. To ensure that this reduction is adequate, that is, that verification results on the reduced model hold for the full model, $ample(s)$ has to respect a set of conditions, based on the independence and invisibility notions previously defined. In some cases, all enabled transitions have to be explored. The following conditions are set forth in [9]:

C1. Only operations in $T - ample(s)$ that are independent from operations in $ample(s)$ can be executed before an operation from $ample(s)$ is executed.[1]

C2. For every cycle in the state graph, there is at least one state s in the cycle that is fully expanded, i.e. $ample(s) = enabled(s)$.

C3. If $ample(s) \neq enabled(s)$, then all transitions in $ample(s)$ are invisible.

C4. If $ample(s) \neq enabled(s)$, then $ample(s)$ contains only one transition that is deterministic in s.[2]

Conditions C1, C2 and C3 are sufficient to guarantee that the reduced model preserves properties expressed in LTL_X, i.e. linear temporal logic without the *next* operator (see e.g. [8]). Condition C4 is significantly more restrictive but is necessary to ensure preservation of branching temporal logics. In that case, [9] shows that there is a so-called *visible bisimulation* between the complete and reduced models, which ensures preservation of both state-based logics such as CTL_X^* (and thus CTL_X). This fits our purpose, since we address action-based models and logics.

Conditions C1 and C2 depend on the whole state graph and are not directly exploitable in a verification algorithm. Instead, one uses sufficient conditions,

---

[1] or equivalently in [9], No operation in $T - ample(s)$ that is dependent on an operation in $ample(s)$ can be executed before an operation from $ample(s)$ is executed.

[2] This is more general than [9], which assumes that all transitions are deterministic.

typically derived from the structure of the model description, to safely decide where reduction can be performed. In our case, we refine the model in terms of processes, with local and global variables and transitions and base the reduction on the notion of *safe local transition*, as developed in next section.

2.3   Process Models

We assume a process-oriented modeling language, where a concurrent system consists of a finite set of *processes* P accessing a set of *variables* V. Each process $p \in P$ maintains a set of local variables $V(p)$ that only it can access. All the processes also share a set of global variables, given by $V - \bigcup_{p \in P} V(p)$. The *global state* $s \in S$ consists of the values of all the variables; the *local state* of p, written $s(p)$, consists of the values of the local variables $V(p)$ in s. More generally, we write $s(W)$ for the values of a set of variables $W \subseteq V$ in s. The set of all possible states of the system is thus the cartesian product of the range of all variables, which we assume to be finite.

Transitions $\alpha \in T$ are characterized by the variables $V(\alpha)$ that they may modify or depend on: α is reducible to a relation α_0 on the range of $V(\alpha)$ such that $s \xrightarrow{\alpha} s'$ iff $s(V(\alpha)) \xrightarrow{\alpha_0} s'(V(\alpha))$ and $s(V - V(\alpha)) = s'(V - V(\alpha))$.

We consider that processes participate in all the transitions that affect their variables, and define the set of transitions of a process p as $T(p) = \{\alpha \in T | V(p) \cap V(\alpha) \neq \emptyset\}$, which divides into *local* transitions $T_l(p) = \{\alpha \in T(p) | V(\alpha) \subseteq V(p)\}$ and *shared* transitions $T_s(p) = T(p) - T_l(p)$. The *locally offered transitions* of p in s are the transitions of p that are allowed by $s(p)$, that is, $trans(p, s) = \{\alpha \in T(p) | \exists s' \cdot s'(p) = s(p) \wedge \alpha \in enabled(s')\}$. Note that a locally offered transition is not necessarily (globally) enabled, i.e. $trans(p, s) \not\subseteq enabled(p)$. A transition is *safe* iff it is deterministic and invisible (w.r.t. the property f being verified) in all states. We write $safe(p)$ for the set of safe local transitions of p, where $safe(p) \subseteq T_l(p) \subseteq T(p)$. It is easily seen that if $V(\alpha) \cap V(\beta) = \emptyset$, then α an β are independent. In particular, if $\alpha \in T_l(p)$ and $\alpha' \notin T(p)$ then α and α' are independent.

With this in hand, we can define sufficient conditions to apply partial-order reduction to our refined process-based models: essentially, as long as some process offers only safe local transitions, among which only one is enabled, then that transition can be selected as the ample set while meeting conditions C1 to C4. More formally, a process p is defined as *deterministic* in state s if the following conditions are met: (1) only safe local transitions are locally offered by p in s, that is, $trans(p, s) \subseteq safe(p)$, and (2) only one transition of p is enabled in s, that is, $|T(p) \cap enabled(s)| = 1$.

If p is deterministic in s, then s can be partially expanded by $ample(s) = T(p) \cap enabled(s) = \{\alpha\}$, where α is a single safe local transition of p, while maintaining the validity of verification (cf. Section 5).

3   The Two-Phase Approach to Partial Order Reduction

3.1   The Two-Phase Algorithm

The Two-Phase algorithm (presented in [10]) is a variant of the classical DFS algorithm with POR of [3,8]. It alternates between two distinct phases:

- Phase 1 expands only deterministic states considering each process at a time, in a fixed order. As long as a process is deterministic, the single transition that is enabled for that process is executed. Otherwise, the algorithm moves on to the next process. After expanding all processes, the last reached state is passed on to phase 2.
- Phase 2 is simple. It performs a full expansion of the states resulting from the phase 1, then applies phase 1 to the reached states.

In order not to postpone a transition indefinitely, for each cycle in the reduced state space, at least one state in this cycle must be fully expanded. Such an indefinite postponing can only arise within phase 1, and is handled by detecting cycles within the current phase 1 expansion and switching to a phase 2 expansion when they occur.

As shown in [10], the Two-Phase algorithm produces a reduced state space which is stuttering equivalent to the whole one, and therefore preserves CTL_X properties [9].

3.2 ImProviso

[4] proposes `ImProviso`, a symbolic version for computing the reachable states of the Two-Phase algorithm. It efficiently combines the advantages of POR and symbolic methods. Classical symbolic model checking algorithms use a single transition relation (partitioned or not) to carry out the required computation on the state space. On the other hand, the `ImProviso` method defines $n + 1$ transition relations, where n is the number of processes in the system. One is the full transition relation T used in phase 2, and the others contain only the safe local transitions from deterministic states, denoted as $T_1(p)$ of each process p, used in phase 1.

Contrary to the nested DFS preferred by classical POR methods, the symbolic methods amount to a *breadth-first* search (BFS). It is thus harder to detect cycles within phase 1 in the symbolic case, and that detection is required to maintain the validity of the algorithm. ImProviso adopts a pessimistic approach: at each step during phase 1, it is assumed pessimistically that any previously expanded state that is reached again might close a cycle, although these occurrences might actually be on different execution paths. This over-approximation guarantees that all cycles are correctly identified, but possibly needlessly reduces the number of states where phase 1 can be applied. This is the key justification for basing ImProviso on the Two-Phase algorithm, as this limits the need for cycle detection to each single execution of phase 1, as opposed to the whole exploration for more traditional POR approaches.

The original ImProviso algorithm is not detailed here but is very similar to the `FwdUntilPOR` algorithm of Section 5, which is based on ImProviso.

4 Forward Symbolic Model-Checking of CTL

In [5], Iwashita et al. present a model checking algorithm based on forward state traversal, which is shown to be more effective than backward state traversal in

many situations. Forward traversal is applicable only to a subset of CTL, but can be combined with backward traversal for the rest of the formulæ. In the following sections we combine this algorithm with ImProviso in order to extend the advantages of partial-order reduction in symbolic methods from reachability properties to CTL properties.

The semantics of a CTL formula f is defined as a relation $s \models f$ over states $s \in S$. In this paper we define the *language* of f as $\mathcal{L}(f) = \{s \in S | s \models f\}$. In the sequel we assimilate temporal logic formulæ f to the set of states $\mathcal{L}(f)$ that they denote, for the sake of simplifying the notations. In particular, we denote set-based computations as the formula-based fixpoints that they compute.

Given a formula f and initial conditions h_0, conventional BDD-based symbolic model-checking can be described as evaluating $\mathcal{L}(f)$ over the sub-formulas of f in a bottom-up manner, and checking whether $\mathcal{L}(h_0) \subseteq \mathcal{L}(f)$. This can be expressed as checking whether $h_0 \Rightarrow f$, or equivalently, whether $h_0 \wedge \neg f = false$. The evaluation of (future) CTL operators in f results in a backward state-space traversal of the model.

[5] introduces forward exploration by transforming a property $h \wedge op(f) = false$ into equi-satisfiable one $op'(h) \wedge f = false$, where a future, backward-traversal CTL operator op in the right term is transformed into a past, forward-traversal operator op' in the left term. In general, h is then a past-CTL formula. The following (past-temporal) operations over formulæ are defined

$$FwdUntil(h, f) = \mu Z.[h \vee post(Z \wedge f)]$$
$$EH(h) = \nu Z.[h \wedge post(Z)]$$
$$FwdGlobal(h, f) = EH(FwdUntil(h, f) \wedge f)$$

$FwdUntil(h, f)$ computes states s that can be reached from h within f (except for s itself), and $EH(h)$ computes states reachable from a cycle, all within h. On this basis, the following equivalences are established:

$$h \wedge EXf = false \iff post(h) \wedge f = false$$
$$h \wedge E[g\,Uf] = false \iff FwdUntil(h, g) \wedge f = false$$
$$h \wedge EGf = false \iff FwdGlobal(h, f) = false$$

The transformation process starts from $h_0 \wedge \neg f = false$, where h_0 is the initial conditions and $\neg f$ is the (suitably re-written) negation of the property to be verified. The equivalences above are applied recurrently until the right part cannot be reduced further, either because all temporal operators have been eliminated or because no rule applies to those remaining. Disjunctions in f can also be handled by case-splitting. Given the final $h \wedge f = false$, $\mathcal{L}(h)$ is evaluated using forward traversal, and the resulting set of states is used as the new initial conditions for a classical, backward model-checking of the remaining f.

By using these equivalences, it is possible to replace an outermost EX, EU or EG operator in f with a forward traversal operator in h. For instance, one can derive the following equivalence:

$$h_0 \Rightarrow AG(req \rightarrow AFack) \iff$$
$$FwdGlobal((FwdUntil(h_0, true) \wedge req), \neg ack) = false$$

Unfortunately, it is not possible to achieve the complete conversion of all CTL properties by applying this method. For instance, the following property is not fully transformable, because the negation in the right term cannot be eliminated:

$$p_0 \Rightarrow AG\ EFa \iff FwdUntil(p_0, true) \wedge \neg EFa = false$$

Informally, reduction is possible for a restricted fragment of *universal CTL*, where temporal operators do not appear in the context of disjunctions nor on the left side of *Until* operators.[3]

5   Forward Model Checking with Partial Order Reduction

In this section we bring together the POR approach of `ImProviso`, presented in Section 3.2, and the forward model checking approach of Section 4. The key element is to define a new algorithm *FwdUntilPOR*, which applies ImProviso's principles to perform POR during the forward exploration of *FwdUntil*. The algorithm for $FwdUntilPOR(h, f)$ is given in Listing 1.1, and is based on ImProviso's algorithm in [5].

Given two visible constraints h and f, the `FwdUntilPOR` algorithm computes the set of states of the reduced state space belonging to a path of the form $s_0 \to s_1 \to \ldots \to s_{n-1} \to s_n$ where $s_0 \models h$ and $\forall i \in \{0, \ldots, n-1\} : s_i \models f$. T is the global transition relation of the model, and `T1[p]` contains safe local transitions of process p, and only from such states where p is deterministic. The `deadState(T, X)` function computes the states of X that have no enabled transition from T, i.e. $deadStates(T, X) = \{s \in X | \neg \exists s' \in S, \alpha \in T \cdot s \xrightarrow{\alpha} s'\}$.

The `FwdUntilPOR` procedure initializes the global variables and performs the two phases alternatively until no states to visit remain. The global variable `frontier` contains the current frontier, that is, the set of states which have been reached but not expanded yet. The global variable `visited` contains all the reached states.

The `phase1` procedure performs the first phase, consisting of partial expansion of deterministic transitions. It is composed of two nested loops. The outer one (lines 22 – 33) expands all processes in a given order. The inner one (lines 25 – 31) expands all deterministic transitions of the current process, from states satisfying f, until no more new states can be found.

The `stack` variable contains all the states which have already been reached during the current run of `phase1`. `cycleApprox` over-approximates the set of states closing a cycle. It contains all the states which have been reached twice during the current run of phase 1. The `dead` variable gathers all the states with no outgoing deterministic transition for the current process; those states are added back to the current frontier when moving to the next process (line 32).

The original ImProviso algorithm defines an additional outermost loop in phase 1, which guarantees that a state is passed from phase 1 to phase 2 only if

---

[3] *Universal CTL* is the fragment of CTL such that negated normal forms contain only universal path quantifiers (AX, AU, AG, AF). As detailed in [5], if the model has a single initial state $(\mathcal{L}(h_0) = \{s_0\})$, then the validity of f can also be phrased as $h_0 \wedge f \neq false$, and existential formulæ can be handled as well.

it has no enabled deterministic transition for any process. This is useful in the
case where a local transition from one process can activate another process, as
for example posting a message to a channel that can be subsequently received.
In our case, this fixpoint calculation is not needed because by construction our
notions of local transition and deterministic process do not allow this kind of
situation. It would easily be added back if it were to become useful.

The `phase2` procedure performs a full expansion of the states of the current
frontier satisfying the constraint f.

```
1   global T          //total transition relation
2   global T1[1..n] // safe local transitions of each process
3
4   global f          // constraints f
5   global frontier // current frontier
6   global visited  // visited states
7
8   procedure FwdUntilPOR(inH, inF)
9     frontier := inH
10    f := inF
11    visited := inF
12    while (frontier != {}) {
13      phase1()
14      phase2()
15    }
16  }
17
18  procedure phase1() {
19    local cycleApprox := {}
20    local stack := frontier
21
22    foreach(p : Processes) {
23      local dead := {}
24      local image := post(T1[p], frontier and f)
25      while ((image - stack) != {}) {
26        dead := dead or deadStates(T1[p], frontier)
27        cycleApprox := cycleApprox or (image and stack)
28        stack := stack or image
29        frontier := image - stack
30        image := post(T1[p],frontier and f)
31      }
32      frontier := frontier or dead
33    }
34    frontier := frontier or cycleApprox
35    visited := visited or stack
36  }
37
38  procedure phase2() {
39    local image := post(T, frontier and f)
40    frontier := image - visited
41    visited :=  visited or image
42  }
```

Listing 1.1. FwdUntilPOR algorithm

Correctness. A full formal proof of correctness of the proposed approach is
beyond the scope of this paper. The validity of the overall verification technique
depends on a number of components, a number of which are inherited from
existing techniques and tools. One important point is the validity of the ample
sets used for POR, which we address in more details first, based on the definitions
of Section 2. The following lemma will be useful in the main proof:

Lemma 1. *If a process p is deterministic in a state s with $enabled(s) \cap T(p) =$ $\{\alpha\}$ and there is a transition $\alpha' \neq \alpha$ such that $s \xrightarrow{\alpha'} s'$ then $s'(p) = s(p)$ and p is deterministic in s' with $enabled(s') \cap T(p) = \{\alpha\}$.*

Proof. We have that $\alpha' \in enabled(s)$. Since $enabled(s) \cap T(p) = \{\alpha\}$, $\alpha' \notin T(p)$ and $s(p) = s'(p)$. $trans(p, s)$ only depends on $s(p)$ so $trans(p, s) = trans(p, s')$ and $enabled(s') \cap T(p) \subseteq trans(p, s') \subseteq safe(p)$ so $enabled(s') \cap T(p) = enabled(s) \cap T(p)$. □

Then we come to the main result:

Theorem 1 (Correctness of Ample Sets). *Given a state s, if $trans(p, s) \subseteq safe(p)$ and $\big| T(p) \cap enabled(s) \big| = 1$, then $ample(s) = T(p) \cap enabled(s) = \{\alpha\}$ is a valid ample set for s.*

Proof. This requires checking that $ample(s)$ meets conditions C1 to C4 of Section 2.2.

C1 is proved by contradiction. Suppose that there is a path $s = s_0 \xrightarrow{\alpha_1} s_1 \ldots \xrightarrow{\alpha_m} s_m \xrightarrow{\alpha'} s'$, where all α_i are independent from α and α' is dependent on (but different from) α. By applying lemma 1 inductively in $s, s_1, \ldots, s_{m-1}$, we get that $s(p) = s_1(p) = \cdots = s_m(p)$, and p is deterministic in s_m with $T(p) \cap enabled(s_m) = \{\alpha\}$, and therefore $\alpha' \notin T(p)$. If $\alpha' \notin T(p)$, then since $\alpha \in safe(p)$, α and α' are independent, a contradiction.

C2 is satisfied because in every cycle in the reduced graph, at least one state is fully expanded in phase 2 of the algorithm. If the cycle contains at least one states s where no process is deterministic, then that state will not be expanded in phase 1. If the cycle is composed exclusively of states where a process is deterministic, then the algorithm guarantees conservatively that the loop is detected in phase 1 and one state is deferred to phase 2 (see the `cycleApprox` variable, line 27).

C3 is satisfied because either $ample(s) = enabled(s)$ or $ample(s) \subseteq safe(s)$ by construction.

C4 is satisfied by construction of $ample(s)$. □

The validity of the overall technique follows, based on the following arguments:

1. The equivalence relations between backward and forward operators of section 4 are valid, in the sense that the transformed formulas, introducing forward traversal where feasible, are satisfiable if and only if the original formulas are satisfiable. This result is assumed from [5].
2. Classical backward BDD-based model-checking, and in particular the reduction of CTL to EX, EU and EG operators, is valid. This is a well-established result, see e.g. [8].
3. The single enabled transition of a deterministic process p in a state s, as defined in Section 2.3, is indeed a valid ample set for s (Theorem 1).
4. Assuming § 3, `FwdUntilPOR` performs a valid exploration of a subset of the behaviours explored by $FwdUntil$, reduced through POR. This is verified by checking that `FwdUntilPOR` is a valid symbolic implementation of the Two-Phase algorithm based on deterministic processes as defined in Secton 2.3 in the same way as the original ImProviso, but adapted for adapted for restricting the exploration to paths of the form $s_0 \to s_1 \to \ldots \to s_{n-1} \to s_n$ where $s_0 \models h$ and $\forall i \in \{0, \ldots, n-1\} : s_i \models f$.

5. The overall, combined forward and backward exploration is a valid model-checking technique for CTL_X. This combines the validity of the transformation (§ 1), of classic CTL model-checking (§ 2) and of the POR-reduced exploration by `FwdUntilPOR` (§ 4), combined with the observation in Section 2.2 that POR reduction respecting conditions C1 to C4 preserves CTL_X properties.

6   Implementation

We have developed the `FwdUntilPOR` method in a new symbolic model checker. It allows to describe concurrent systems and to verify CTL properties, and action-based extension thereof, on these models.

Our prototype has been implemented with the Scala language [11]. Scala is a multi-paradigm programming language, fully interoperable with Java, designed to integrate features of object-oriented programming and functional programming. Scala is a pure object-oriented language in the sense that every value is an object. Scala is also a functional language in the sense that every function is a value. To obtain better performance, our model checker uses a BDD package, named BuDDy[12], written in C.

The model checker defines a language for describing transitions systems. The design of the language has been influenced on the one hand by process algebras and on the other hand by the NuSMV language [13]. A model of a concurrent system declares a set of global variables, a set of shared actions and a set of processes. A process p declares a set of local variables, a set of local actions and the set of shared actions which p is synchronized on. Each process has a distinguished local program counter variable pc. For each value of pc, the behavior of a process is defined by means of a list of *action-labelled guarded commands* of the form $[\alpha]$ `c` $\rightarrow$ `u`, where α is an action, `c` is a condition on variables and `u` is an assignment updating some variables. *Shared* actions are used to define synchronization between the processes. A *shared* action occurs simultaneously in all the processes that share it, and only when all enable it.

Properties are expressed in an action-based extension of CTL similar to ACTL [7] [4]. These properties can be checked with three techniques: the backward traversal method, the FwdUntil method (Section 4) and the FwdUntilPOR method (Section 5). If the FwdUntilPOR is applied, some syntactic restrictions are imposed in order to satisfy the conditions allowing the POR. For instance, the propositions allowed in the CTL_X properties can only concern the global variables so as to satisfy the visibility condition. For each p, safe commands are determined at compile time and combined into $T1(p)$. A guarded command is considered as safe if it contains only local variables and actions. For each pc, a list of guarded command gcs is considered as safe, if all elements of gcs are safe and all of them are mutually exclusive .

Ordering of the Variables. BDDs require a fixed ordering among the boolean variables used to represent the system. The size of BDDs, and therefore the

---

[4] Specifically, we use Action-Restricted CTL (ARCTL) [14], which associates actions to path quantifiers rather than temporal operators.

performance of BDD-based model-checking, strongly depends on this ordering. For instance, the size of the BDD representing a n-bit comparator ($x_1 = x'_1 \wedge \ldots \wedge x_n = x'_n$) can go from $3*n+2$ nodes with the order $x_1 \prec x'_1 \prec \ldots \prec x_n \prec x'_n$ to $3*2^n - 1$ nodes with the order $x_1 \prec \ldots \prec x_n \prec x'_1 \prec \ldots \prec x'_n$. In general, finding the best variable ordering is a NP-complete problem. The topic has been intensively studied and several heuristics have been developed for finding a good ordering between variables.

This research has mostly focused on ordering variables within a state, but there is also an opportunity for optimizing the order of variables used for the transitions relation $T(s, \alpha, s')$, which ranges over sets of boolean variables, s, α, s' respectively. If $s \xrightarrow{\alpha} s'$, s is named the *source state* and s' is named the *target state*. In order to represent the relation T as a boolean function $T(s, \alpha, s')$, three sets of boolean variables are used: $s = s_1, s_2, \ldots, s_m$, $\alpha = \alpha_1, \alpha_2, \ldots, \alpha_n$ and $s' = s'_1, s'_2, \ldots, s'_m$. An intuitive approach would be to start with α, followed by s, then s'. In the case of strongly asynchronous systems, this approach leads to an explosion of the BDD size [15]. A better solution is proposed in [15] for asynchronous models such as those obtained from process algebra specifications. The action variables are encoded first, followed by an "interlacing" between the source variables and the target variables: $a_1 \prec a_2 \prec \ldots \prec a_n \prec s_1 \prec s'_1 \prec s_2 \prec s'_2 \prec \ldots \prec s_m \prec s'_m$.

Experimental results show that the resulting BDDs only grow linearly in the number of asynchronous components. Intuitively, the ordering works well due to the fact that, in the case of asynchronous processes, most of the time a small number of processes proceed, so only the variables of those processes change while most variables remains the same (i.e. $s_i = s'_i$). These constraints are more efficiently encoded in the BDD, if s_i and s'_i are next to each other in the ordering, similarly to the n-bit comparator example above.

Table 1 compares the transition relation BDD size and the time between the intuitive and the interlaced ordering, based on the case study of Section 7. The size of the model is driven by the parameter #drill, and the time corresponds to verifying property $p6$. It confirms the much reduced growth rate of the interlaced ordering, allowing a much larger number of components to be added.

Table 1. Size of the transition relation BDD (in # nodes) and verification time (in seconds) for property p6 of the Turntable case study, using interlaced vs. non-interlaced orderings, — correspond to memory exhausted (2 GB)

# drills	# vars	interlaced		non-interlaced	
		size	time	size	time
1	24	1 543	.041	153 056	6.222
2	31	1 913	.070	4 051 081	409.078
3	38	2 307	.114	—	—
20	157	12 184	4.436	—	—
40	297	31 572	30.884	—	—

7   Case Study

In order to assess the effectiveness of our method, we applied it to a turntable model which is described in [16,17]. For initial experiments, we modelled the system in the NuSMV language. We then converted the language of our prototype. We compared performance of verification using three methods: classical backward, FwdUntil and FwdUntil with POR, as well as with the NuSMV tool and with the non-symbolic tool from the CADP toolset: Evaluator [18]. This section presents the system and the results we obtained. All the test have been run on a 2,16 GHz Intel Core 2 Duo with 2 GB of RAM memory.

The turntable system consists of a round turntable, n drills and a testing device, as illustrated in Figure 1. The turntable transports products between the drills, the testing device and input and output positions. The drills bore holes in the products. After being drilled, the products are delivered to the tester, where the depth of the holes is measured, since it is possible that drilling went wrong. The turntable has $n + 3$ slots that each can hold a single product. The original model had only one drill; we extended it to represent an arbitrary number of drills. Although a turntable with 40 drills is a bit artificial, it gives a model of a fairly large realistic size.
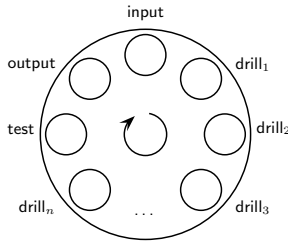
Fig. 1. Turntable System

The original model was described in LOTOS, a formal specification technique based on process algebras [6]. First we translated the LOTOS model into a NuSMV model. The difficult part of this task comes from the fact that LOTOS and NuSMV do not have the same concurrency model. LOTOS has a more expressive synchronization mechanism. The conversion in the prototype language was easier because our language is inspired by languages like LOTOS.

We have verified 13 properties from [17] expressed as a regular alternation-free μ-calculus formulae [19], here labelled $p1$ to $p13$. $p1$ to $p6$ are safety properties and $p7$ to $p13$ are liveness properties. For instance, the safety property $p6$ states that if a piece is well drilled, no alarm will be raised during the next cycle. The liveness property $p11$ states that each piece will be removed from the turntable after it is tested.

```
P6 :[true*.INF !TESTED !TRUE.(not INF !TURNED)*.INF !TURNED.(not INF !TURNED)*.ERR] false
P11:[true*.INF !TESTED.*]inev(not CMD !TURN,CMD !TURN,inev(not CMD !TURN,REQ !REMOVE.*,true))
```

For 11 of the 13 properties, the FwdUntilPOR method outperforms the classical backward CTL algorithm. However for $p1$ and $p2$, the classical method is approximately 30 times faster than the FwdUntilPOR algorithm. Currently, we do not

Table 2. Verification times (in seconds) for properties p6 and p11 of the Turntable model, using NuSMV, CADP and our prototype using standard backward exploration (Bwd), FwdUntil (Fwd) and FwdUntilPOR (Fwd+POR). — indicates that the computation did not end within 5 hours.

# drill	property p6					property p11			
	NuSMV	CADP	Bwd	Fwd	Fwd+POR	CADP	Bwd	Fwd	Fwd+POR
1	2.001	2.770	.041	.022	.055	3.640	.037	.076	.097
2	36.400	5.480	.070	.043	.082	19.350	.062	.132	.139
3	578.500	81.260	.114	.074	.106	335.130	.094	.167	.185
4	6617.000	13393.630	.157	.104	.144	19031.340	.132	.244	.245
10	—	—	.721	.875	.335	—	.663	1.064	.589
20	—	—	4.436	9.698	.838	—	6.112	9.187	1.496
30	—	—	13.842	31.295	1.475	—	19.412	25.520	2.780
40	—	—	30.884	80.519	2.499	—	40.304	67.761	4.355

have an explanation for such a difference which is left for further investigation. On this model, the FwdUntil method is less efficient than the classical method, taking exception from the general observation reported in [5].

Table 2 shows the time for the verification of the properties p6 and p11. If the turntable comprises 40 drills, p6 properties is checked approximately 12 times faster and p11 is checked approximately 9 times faster with the FwdUntilPOR method than with the backward method. The causes for the huge increase for CADP between 3 and 4 drills remain to be investigated.

Table 3 compares the time needed for computing the reachable state space between NuSMV and our prototype. We notice that NuSMV cannot handle a model beyond 4 drills, while our prototype can still easily handle up to 40 drills. It is quite interesting to note that while POR increases the number of BDD nodes for the reduced state space (likely due to breaking some symmetry in the full state space), it results in substantial speed improvements. One possible explanation for the huge difference between NuSMV and our prototype is that the modeling language of NuSMV, as opposed to that of our prototype, does not support synchronization through shared actions, and so the translation of the original LOTOS model is more convoluted and less straightforward. This issue deserves further investigation.

Table 3. BDD size (in # nodes), state space size (in # states) and computation time (in seconds) for the reachable state space of the Turntable model in NuSMV vs. our prototype, both with full and POR exploration (i.e. using the ImProviso algorithm). — indicates that the computation did not end within 5 hours.

# drills	# nodes			# states			time		
	NuSMV	proto full	proto POR	NuSMV	proto full	proto POR	NuSMV	proto full	proto POR
1	2660	131	196	10 068	9 084	5572	1.009	.289	.149
2	12888	274	488	170 058	146 784	7948	32.000	.297	.153
3	64616	428	869	$\approx 10^6$	$\approx 10^6$	10324	553.400	.401	.181
4	244967	582	1286	$\approx 10^7$	$\approx 10^7$	12700	4 784.600	.545	.228
10	—	1506	4709	—	$\approx 10^{11}$	26956	—	1.892	.528
20	—	3046	14039	—	$\approx 10^{20}$	50716	—	7.984	1.147
40	—	6126	49649	—	$\approx 10^{37}$	98236	—	63.721	3.434

8   Related Work

In [20], Alur et al. transform an explicit model checking algorithm performing partial order reduction and able to check invariance of local properties. They start from a DFS algorithm to obtain a modified BFS algorithm. Both expand an ample set of transitions in each step. In order to detect the cycles, they assume pessimistically that each previous expanded state might close a cycle. By contrast, `ImProviso` makes a smaller over-approximation of such states because it only needs to consider cycles formed exclusively by deterministic transitions. Consequently it looks for possible cycles only with respect to states visited during phase 1.

In [21], Abdulla et al. present a general method for combining POR and symbolic model checking. Their method can check safety properties either by backward or forward reachability analysis. So as to perform the reduction, they employ the notion of commutativity in one direction, a weakening of the dependency relation which is usually used to perform POR. It can be applied either to finite or infinite state spaces. One difference between this approach and ours is the checked properties. This approach deals both with backward and forward reachability analysis, while we are able to check a subset of CTL_X properties using only forward analysis.

In [22], Kurshan et al. introduce a partial order reduction algorithm based on static analysis. They notice that each cycle in the state space is composed of some local cycles. The method performs a static analysis of the checked model so as to discover local cycles and set up all the reductions at compile time. The reduced state space can be handled with symbolic techniques. Their approach differs from ours in that it performs the reduction at compile time. On the contrary our approach performs the reduction at run time. Lerda et al. suggest that the Improviso method is more efficient than the one introduced by Kurshan et al.[4]. However, we think that it still would be interesting to see how both approaches can benefit from each other.

In [23], Fantechi et al. present SAM, a symbolic model checker based on BSP (Boolean symbolic programming), a programming language aimed at defining computations on boolean functions. SAM takes as input an LTS s and a (possibly recursive) μ-ACTL formula p, and transforms both into BSP programs, which are then compiled into a sequence of calls to BDD primitives. Checking that s verifies p reduces to checking whether the boolean function "$tr(s) \Rightarrow tr(p)$" is a tautology. SAM is able to check μ-ACTL formulae which is a richer language that the one of our prototype, but does not address performance optimizations such as partial-order reduction.

9   Conclusion and Perspectives

In this paper, we introduced the FwdUntilPOR algorithm that combines two existing techniques to provide efficient symbolic model checker of CTL on asynchronous models. The first technique is the ImProviso algorithm which efficiently merges POR and symbolic methods. The second technique is the forward symbolic model checking approach applicable to a subset of CTL.

We also implemented the FwdUntilPOR algorithm in a new symbolic model checker. Its input syntax supports actions-based models and logics. We show on a realistic-sized case study that our method achieves a strong improvement in comparison to the classical backward algorithm, in the majority of cases.

Although it is usually considered that symbolic model checking is inadequate for loosely-synchronized models, our results show that with appropriate optimization this approach might in fact be quite effective to tackle the state space explosion problem. On this basis, we plan to develop our approach and our prototype in a number of ways:

- We plan to extend the FwdUntilPOR method for applying POR to a larger subset of CTL. We will investigate how the approach of [21] can be extended for combining the classical backward symbolic model checking algorithms and POR.
- We need to explore how it is possible to compute a better approximation of the deterministic states. There exists a large body of literature on this subject.
- We need to extend our prototype by adding generation of counter-examples for failed properties Another source of improvement can come from applying traditional partitioning techniques to BDDs representing the transition relations. Besides, it would be convenient to accept or translate, as input, a popular language such as LOTOS in order to exploit the numerous case studies available in this formalism.
- As observed in our case study, for some properties the FwdUntilPOR method performs much worse than the standard backward model checking. We will investigate this issue, in order to try to characterize the classes of properties where this happens and to investigate whether our algorithm can be improved to better handle those cases.

References

1. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers 35(8), 677–691 (1986)
2. Burch, J., Clarke Jr., E., McMillan, K., Dill, D., Hwang, L.: Symbolic Model Checking: 10^{20} States and Beyond. In: Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science, Washington, D.C., pp. 1–33. IEEE Computer Society Press, Los Alamitos (1990)
3. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032. Springer, Heidelberg (1996)
4. Lerda, F., Sinha, N., Theobald, M.: Symbolic model checking of software. In: Cook, B., Stoller, S., Visser, W. (eds.) Electronic Notes in Theoretical Computer Science, vol. 89. Elsevier, Amsterdam (2003)
5. Iwashita, H., Nakata, T., Hirose, F.: CTL model checking based on forward state traversal. In: ICCAD 1996: Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design, Washington, DC, USA, pp. 82–87. IEEE Computer Society, Los Alamitos (1996)
6. ISO/IEC: LOTOS — a formal description technique based on the temporal ordering of observational behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève (1988)

7. De Nicola, R., Vaandrager, F.: Action versus state based logics for transition systems. In: Guessarian, I. (ed.) LITP 1990. LNCS, vol. 469, pp. 407–419. Springer, Heidelberg (1990)
8. Clarke Jr., E., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (1999)
9. Gerth, R., Kuiper, R., Peled, D., Penczek, W.: A partial order approach to branching time logic model checking. Inf. Comput. 150(2), 132–152 (1999)
10. Nalumasu, R., Gopalakrishnan, G.: A new partial order reduction algorithm for concurrent systems. In: Delgado Kloos, C., Cerny, E. (eds.) Hardware Description Languages and their Applications (CHDL 1997), Toledo, Spain. Chapman and Hall, Boca Raton (1997)
11. Odersky, M., Spoon, L., Venners, B.: Programming in Scala, a comprehensive step-by-step guide. PrePrint^{TM} Edition, Version 3 (May 2008)
12. Lind-Nielsen, J.: Buddy - a binary decision diagram package (June 10, 2008), http://vlsicad.eecs.umich.edu/BK/Slots/cache/www.itu.dk/research/buddy/index.html
13. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: a new symbolic model verifier. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 495–499. Springer, Heidelberg (1999)
14. Pecheur, C., Raimondi, F.: Symbolic model checking of logics with actions. In: Edelkamp, S., Lomuscio, A. (eds.) MoChArt IV. LNCS, vol. 4428, pp. 113–128. Springer, Heidelberg (2007)
15. Enders, R., Filkorn, T., Taubner, D.: Generating BDDs for symbolic model checking in CCS. Distrib. Comput. 6(3), 155–164 (1993)
16. Bortnik, E., Trčka, N., Wijs, A.J., Luttik, B., van de Mortel-Fronczak, J., Baeten, J.C.M., Fokkink, W.J., Rooda, J.: Analyzing a χ model of a turntable system using SPIN, CADP and UPPAAL. Journal of Logic and Algebraic Programming 65(2), 51–104 (2005)
17. Mateescu, R.: 5. IC2 treatise. Systèmes temps réel 1 - techniques de description et de vérification. Lavoisier, 151–180 (2006)
18. Garavel, H.: Open/Cæsar: An open software architecture for verification, simulation, and testing. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 68–84. Springer, Heidelberg (1998)
19. Mateescu, R., Sighireanu, M.: Efficient on-the-fly model-checking for regular alternation-free mu-calculus. Science of Computer Programming 46(3), 255–281 (2003)
20. Alur, R., Brayton, R.K., Henzinger, T.A., Qadeer, S., Rajamani, S.K.: Partial-order reduction in symbolic state space exploration. In: Computer Aided Verification, pp. 340–351 (1997)
21. Abdulla, P.A., Jonsson, B., Kindahl, M., Peled, D.: A general approach to partial order reductions in symbolic verification (extended abstract). In: Computer Aided Verification, pp. 379–390 (1998)
22. Kurshan, R.P., Levin, V., Minea, M., Peled, D., Yenigün, H.: Static partial order reduction. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 345–357. Springer, Heidelberg (1998)
23. Fantechi, A., Gnesi, S., Mazzanti, F., Pugliese, R., Tronci, E.: A symbolic model checker for ACTL. In: Hutter, D., Traverso, P. (eds.) FM-Trends 1998. LNCS, vol. 1641, pp. 228–242. Springer, Heidelberg (1999)

Reentrant Readers-Writers: A Case Study Combining Model Checking with Theorem Proving

Bernard van Gastel, Leonard Lensink, Sjaak Smetsers, and Marko van Eekelen

Institute for Computing and Information Sciences, Radboud University Nijmegen
Heyendaalseweg 135, 6525 AJ, Nijmegen, The Netherlands
`b.vangastel@student.science.ru.nl`,
`{l.lensink,s.smetsers,m.vaneekelen}@cs.ru.nl`

Abstract. The classic readers-writers problem has been extensively studied. This holds to a lesser degree for the reentrant version, where it is allowed to nest locking actions. Such nesting is useful when a library is created with various procedures that each start and end with a lock. Allowing nesting makes it possible for these procedures to call each other. We considered an existing widely used industrial implementation of the reentrant readers-writers problem. We modeled it using a model checker revealing a serious error: a possible deadlock situation. The model was improved and checked satisfactorily for a fixed number of processes. To achieve a correctness result for an arbitrary number of processes the model was converted to a theorem prover with which it was proven.

1 Introduction

It is generally acknowledged that the growth in processor speed is reaching a hard physical limitation. This has led to a revival of interest in concurrent processing. Also in industrial software, concurrency is increasingly used to improve efficiency [26]. It is notoriously hard to write correct concurrent software. Finding bugs in concurrent software and proving the correctness of (parts of) this software is therefore attracting more and more attention, in particular where the software is in the core of safety critical or industrial critical applications.

However, it can be incredibly difficult to track concurrent software bugs down. In concurrent software bugs typically are caused by infrequent 'race conditions' that are hard to reproduce. In such cases, it is necessary to thoroughly investigate 'suspicious' parts of the system in order to improve these components in such a way that correctness is guaranteed.

Two commonly used techniques for checking correctness of such system are *formal verification* and *testing*. In practice, testing is widely and successfully used to discover faulty behavior, but it cannot assure the absence of bugs. In particular, for concurrent software testing is less suited due to the typical characteristics of the bugs (infrequent and hard to reproduce). There are roughly two approaches to formal verification: *model checking* and *theorem proving*. Model checking [6,23] has the advantage that it can be performed automatically, provided that a suitable model of the software (or hardware) component has been

created. Furthermore, in the case a bug is found model checking yields a counterexample scenario. A drawback of model checking is that it suffers from the state-space explosion and typically requires a closed system. In principle, theorem proving can handle any system. However, creating a proof may be hard and it generally requires a large investment of time. It is only partially automated and mainly driven by the user's understanding of the system. Besides, when theorem proving fails this does not necessarily imply that a bug is present. It may also be that the proof could not be found by the user.

In this paper we consider the *reentrant readers-writers* problem as a formal verification case study. The classic readers-writers problem [8] considers multiple processes that want to have read and/or write access to a common resource (a global variable or a shared object). The problem is to set up an access protocol such that no two writers are writing at the same time and no reader is accessing the common resource while a writer is accessing it. The classic problem is studied extensively[22]; the reentrant variant (in which locking can be nested) has received less attention so far although it is used in Java, C# and C++ libraries.

We have chosen a widely used industrial library (Trolltech's Qt) that provides methods for reentrant readers-writers. For this library a serious bug is revealed and removed. This case study is performed in a structured manner combining the use of a model checker with the use of a theorem prover exploiting the advantages of these methods and avoiding their weaknesses.

In Section 2 we will introduce the case study. Its model will be defined, improved and checked for a fixed number of processes in Section 3. Using a theorem prover the model will be fully verified in Section 4. Finally, related work, future work and concluding remarks are found in Sections 5 and 6.

2   The Readers-Writers Problem

If in a concurrent setting two threads are working on the same resource, synchronization of operations is often necessary to avoid errors. A *test-and-set* operation is an important primitive for protecting common resources. This atomic (i.e. non-interruptible) instruction is used to both test and (conditionally) write to a memory location. To ensure that only one thread is able to access a resource at a given time, these processes usually share a global boolean variable that is controlled via *test-and-set* operations, and if a process is currently performing a test-and-set, it is guaranteed that no other process may begin another test-and-set until the first process is done. This primitive operation can be used to implement *locks*. A lock has two operations: lock and unlock. The lock operation is done before the critical section is entered, and the unlock operation is performed after the critical section is left. The most basic lock can only be locked one time by a given thread. However, for more sophisticated solutions, just an atomic test-and-set operation is insufficient. This will require support of the underlying OS: threads acquiring a lock already occupied by some thread should be de-scheduled until the lock is released. A variant of this way of locking is called *condition locking*: a thread can wait until a certain condition is satisfied, and will automatically continue when notified (*signalled*) that the condition has

been changed. An extension for both basic and condition locking is *reentrancy*, i.e. allowing nested lock operations by the same thread.

A so-called *read-write* lock functions differently from a normal lock: it either allows multiple threads to access the resource in a read-only way, or it allows one, and only one, thread at any given time to have full access (both read and write) to the resource ([10]). These locks are the standard solution to the producer/consumer problem in which a buffer has to be shared.

Several kinds of solutions to the classical readers-writers problem exist. Here, we will consider a *read-write* locking mechanism with the following properties.

writers preference. Read-write locks suffer from two kinds of starvation, one with each kind of lock operation. Write lock priority results in the possibility of reader starvation: when constantly there is a thread waiting to acquire a write lock, threads waiting for a read lock will never be able to proceed. Most solutions give priority to write locks over read locks because write locks are assumed to be more important, smaller, exclusive, and to occur less.

reentrant. A thread can acquire the lock multiple times, even when the thread has not fully released the lock. Note that this property is important for modular programming: a function holding a lock can use other functions which possibly acquire the same lock. We distinguish two variants of reentrancy:

1. *Weakly reentrant*: only permit sequences of either read or write locks;
2. *Strongly reentrant*: permit a thread holding a write lock to acquire a read lock. This will allow the following sequence of lock operations: write_lock, read_lock, unlock, unlock. Note that the same function is called to unlock both a write lock and a read lock. The sequence of a read lock followed by a write lock is not admitted because of the evident risk of a deadlock (e.g. when two threads both want to perform the locking sequence read_lock, write_lock they can both read but none of them can write).

2.1   Implementation of Read-Write Locks

In this section we show the C++ implementation of weakly reentrant read/write locks being part of the multi-threading library of the Qt development framework, version 4.3. The code is not complete; parts that are not relevant to this presentation are omitted. This implementation uses other parts of the library: threads, mutexes and conditions. Like e.g. in Java, a `condition` object allows a thread that owns the lock but that cannot proceed, to wait until some condition is satisfied. When a running thread completes a task and determines that a waiting thread can now continue, it can call a signal on the corresponding condition. This mechanism is used in the C++ code listed in Figure 1.

The structure `QReadWriteLockPrivate` contains the attributes of the class. These attributes are accessible via an indirection named `d`. The attributes `mutex`, `readerWait` and `writerWait` are used to synchronize access to the other administrative attributes, of which `accessCount` keeps track of the number of locks (including reentrant locks) acquired for this lock. A negative value is used for write

```
struct QReadWriteLockPrivate
{
  QReadWriteLockPrivate()
  : accessCount(0),
    currentWriter(0),
    waitingReaders(0),
    waitingWriters(0)
  { }

  QMutex mutex;
  QWaitCondition readerWait,
                 writerWait;

  Qt::HANDLE currentWriter;
  int accessCount,waitingReaders,
      waitingWriters;
};

void QReadWriteLock::lockForRead()
{
  QMutexLocker lock(&d->mutex);
  while (d->accessCount < 0 ||
         d->waitingWriters) {
    ++d->waitingReaders;
    d->readerWait.wait(&d->mutex);
    --d->waitingReaders;
   }
  ++d->accessCount;
  Q_ASSERT_X(d->accessCount>0,
        "...","...");
}

void QReadWriteLock::lockForWrite()
{
  QMutexLocker lock(&d->mutex);
```

```
  Qt::HANDLE self =
      QThread::currentThreadId();
  while (d->accessCount != 0) {
    if (d->accessCount < 0 &&
        self == d->currentWriter) {
      break; // recursive write lock
    }
    ++d->waitingWriters;
    d->writerWait.wait(&d->mutex);
    --d->waitingWriters;
  }
  d->currentWriter = self;
  --d->accessCount;
  Q_ASSERT_X(d->accessCount<0,
        "...","...");

}

void QReadWriteLock::unlock()
{
  QMutexLocker lock(&d->mutex);
  Q_ASSERT_X(d->accessCount!=0,
        "...","...");
  if ((d->accessCount > 0 &&
       --d->accessCount == 0) ||
      (d->accessCount < 0 &&
       ++d->accessCount == 0)) {
    d->currentWriter = 0;
    if (d->waitingWriters) {
      d->writerWait.wakeOne();
    } else if (d->waitingReaders) {
      d->readerWait.wakeAll();
    }
  }
}
```

Fig. 1. QReadWriteLock class of Qt

access and a positive value for read access. The attributes waitingReaders and waitingWriters indicate the number of threads requesting a read respectively write permission, that are currently pending. If some thread owns the write lock, currentWriter contains a HANDLE to this thread; otherwise currentWriter is a null pointer.

The code itself is fairly straightforward. The locking of the mutex is done via the constructor of the wrapper class QMutexLocker. Unlocking this mutex happens implicitly in the destructor of this wrapper. Observe that a write lock can only be obtained when the lock is completely released (d->accessCount == 0), or the thread already has obtained a write lock (a reentrant write lock request, d->currentWriter == self).

The code could be polished a bit. E.g. one of the administrative attributes can be expressed in terms of the others. However, we have chosen not to deviate from the original code, except for the messages in the assertions which were, of course, more informative.

3   Model Checking Readers/Writers with Uppaal

Uppaal [17] is a tool for modeling and verification of real-time systems. The idea is to model a system using timed automata. Timed automata are finite state machines with time. A system consists of a collection of such automata. An automaton is composed of locations and transitions between these locations defining how the system behaves. To control when to fire a transition one can use guarded transitions and synchronized transitions. Guards are just boolean expressions whereas the synchronization mechanism is based on hand-shakes: two processes (automata) can take a simultaneous transition, if one does a send, `ch!`, and the other a receive, `ch?`, on the same channel `ch`. For administration purposes, but also for communication between processes, one can use global variables. Moreover, each process can have its own local variables. Assignments to local or global variables can be attached to transitions as so-called *updates*.

In this paper we will not make use of time. In Uppaal terminology: we don't have `clock` variables. Despite the absence of this most distinctive feature of Uppaal, we have still chosen to use Uppaal here because of our local expertise and the intuitive and easy to use graphical interface which supports understanding and improving the model in a elegant way. The choice of model checker is however not essential for the case study. It could also have been performed with any other model checker such as e.g. SMV [19], mCRL2 [11] or SPIN [14].

Constructing the Uppaal Model

Our intention is to model the code from Figure 1 as an abstract Uppaal model, preferably in a way that the distance between code and model is kept as small as possible. However, instead of trying to model Qt-threads in Uppaal we will directly use the built-in Uppaal processes to represent these threads. Thread handles are made explicit by numbering the processes, and using these numbers as identifications. `NT` is the total number of processes. The identification numbers are denoted by `tid` in the model, ranging 0 to `NT` - 1. The `NT` value is also used to represent the null pointer for the variable `currentWriter` in the C++ code. Mutexes and conditions directly depend on the thread implementation, so we cannot model these objects by means of code abstraction. Instead we created an abstract model in Uppaal that essentially simulates the behavior of these objects. The result is shown in Figure 2. In this basic locking model, method calls are simulated via synchronization messages. The conditions are represented by two integer variables, `sleepingReaders` and `sleepingWriters`, that maintain the number of waiting readers and waiting writers, respectively. A running process can signal such a process which will result in a wake up message. A process receiving such a message should always immediately try to acquire the lock, otherwise mutual exclusion is not guaranteed anymore.
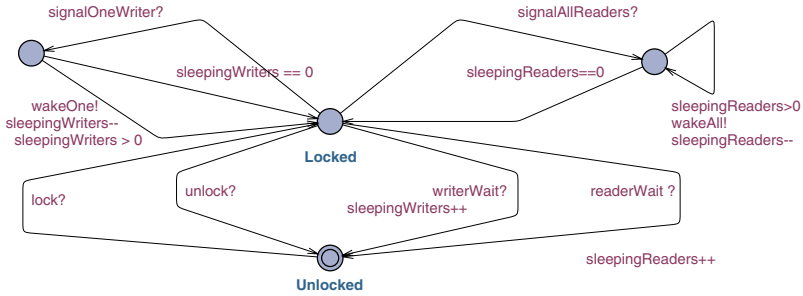
Fig. 2. Mutex and condition model

The `RWLock` implementation is model checked using the combination of this basic locking process with a collection of concurrent processes, each continuously performing either a `lockForRead`, `lockForWrite`, or `unlock` step. The abstract model (see Figure 3) is obtained basically by translating C++ statements into transitions.

For convenience of comparison, we have split the model into three parts, corresponding to `lockForRead`, `writeLock` and `unlock` respectively. These parts can be easily combined into a single model by collapsing the `Start` states, and, but not necessarily, the `Abort` states. The auxiliary functions `testRLock`, `testWLock`, and `testReentrantWLock` are defined as:

```
bool testRLock(ThreadId tid)
{ return waitingWriters>0 ||(currentWriter!=NT && currentWriter!=tid);}

bool testWLock (ThreadId tid)        bool testReentrantWLock (ThreadId tid)
{  return accessCount != 0 &&       {  return accessCount != 0 &&
         currentWriter != tid;             tid == currentWriter;
}                                    }
```

If a process performs a lock operation it will enter a location that is labeled with `EnterXX`. Here, `XX` corresponds to the called operation. The call is left via a `LeaveXX` location. For example, if a thread invokes `lockForRead` it will enter the location `EnterRL`. Hereafter, the possible state transitions directly reflect the corresponding flow of control in the original code for this method. The call ends at `LeaveRL`. These special locations are introduced to have a kind of separation between definition and usage of methods. If the thread was suspended (due to a call to the `wait` method on the `readerWait` condition) the process in the abstract model will be waiting in the location `RWait`. The wrapper `QMutexLocker` has been replaced by a call to `lock`. To take the effect of the destructor into account, we added a call to unlock at the end of the scope of the wrapper object. Furthermore, observe that assertions are modeled as a 'black hole': a state, labeled `Abort`, from which there is no escape possible.
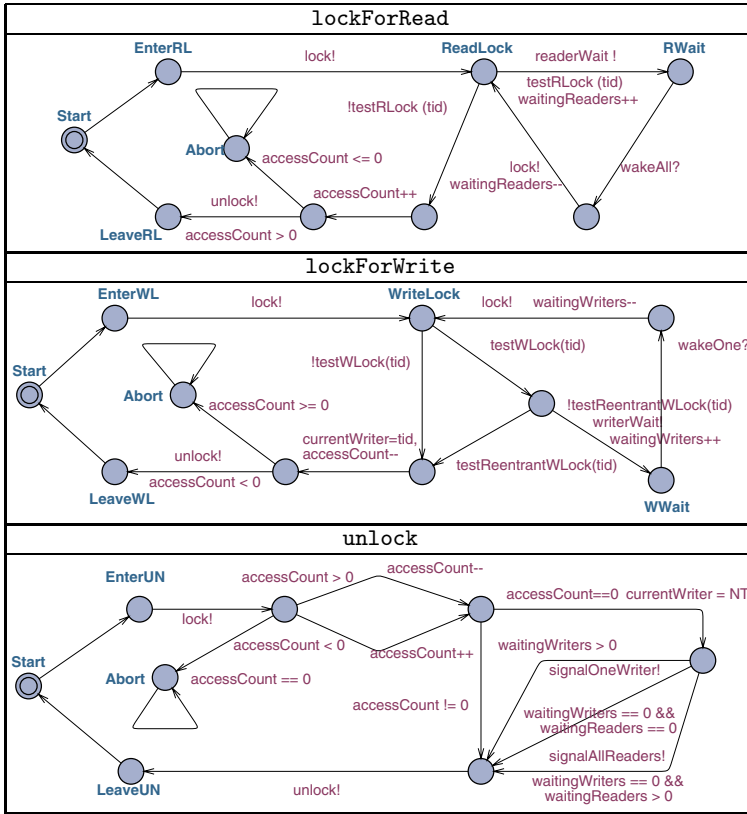
Fig. 3. Uppaal models of the locking primitives

Checking the Model

The main purpose of a model checker is to verify the model w.r.t. a requirement specification. In Uppaal, requirements are specified as *queries* consisting of path and state formulae. The latter describe individual states whereas the former range over execution paths or traces of the model. In Uppaal, the (state) formula `A[]` φ expresses that φ should be true in all reachable states. `deadlock` is a built-in formula which is true if the state has no outgoing edges.

In our example we want to verify that the model is deadlock-free, which is a state property. This can easily be expressed by means of the following query:

`A[]` **not deadlock**

When running Uppaal on this model consisting of 2 threads, the verifier will almost instantly respond with: `Property is not satisfied`. The trace generated by Uppaal shows a counter example of the property, in this case a scenario leading to a deadlock. The problem is that if a thread, which is already holding a read lock, does a (reentrant) request for another read lock, it will be suspended

if another thread is pending for a write lock (which is the case if the write lock was requested after the first thread obtained the lock for the first time). Now both threads are waiting for each other.

3.1   Correcting the Implementation/Model

The solution is to let a reentrant lock attempt always succeed. To avoid writers starvation, new read lock requests should be accepted only if there are no writers waiting for the lock. To distinguish non-reentrant and reentrant uses, we maintain, per thread, the current number of nested locks making no distinction between read and write locks. Additionally, this solution allows strongly reentrant use. In the implementation this is achieved by adding a *hash map* (named `current` of type `QHash`) to the attributes of the class that maps each thread handle to a counter. To illustrate our adjustments, we show the implementation of `lockForRead` [1].

```
void QReadWriteLock::lockForRead() {
    QMutexLocker lock(&d->mutex);

    Qt::HANDLE self = QThread::currentThreadId();

    QHash<Qt::HANDLE, int>::iterator it = d->current.find(self);
    if (it != d->current.end()) {
         ++it.value();
         Q_ASSERT_X(d->numberOfThreads > 0, "...", "...");
         return;
    }
    while (d->currentWriter != 0 || d->waitingWriters > 0) {
         ++d->waitingReaders;
         d->readerWait.wait(&d->mutex);
         --d->waitingReaders;
    }
    d->current.insert(self, 1);
    ++d->numberOfThreads;
    Q_ASSERT_X(d->numberOfThreads> 0, "...", "...");
}
```

To verify this implementation we again converted the code to Uppaal. Since handles where represented by integers ranging from 0 to `NT - 1` (where `NT` denotes the number of threads), we can use a simple integer array to maintain the number of nested locks per thread, instead of a hash map. In this array, the process id is used as an index. Figure 4 shows the part of the Uppaal model that corresponds to the improved `lockForRead`. For the full Uppaal model, see www.cs.ru.nl/$\sim$sjakie/papers/readerswriters/.

To limit the state space we have added an upper bound `maxNest` to the nesting level and a counter `readNest` indicating the current nesting level. This variable is decremented in the unlock part of the full model. Running Uppaal on

---

[1] For the complete code, see www.cs.ru.nl/~sjakie/papers/readerswriters/.
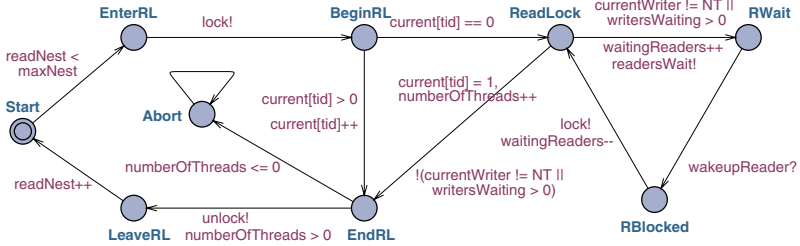
Fig. 4. Uppaal model of the correct version of `lockForRead`

the improved model will, not surprisingly, result in the message: `Property is satisfied`. In this experiment we have limited the number of processes to 4, and the maximum number of reentrant calls to 5. If we increase these values slightly, the execution time worsens drastically. So, for a complete correctness result, we have to proceed differently.

4   General Reentrant Readers-Writers Model

In this section we will formalize the Uppaal model in PVS [21].

We prove that the reentrant algorithm is free from deadlock when we generalize to *any* number of processes. While explaining the formalization we will briefly introduce PVS. For the complete PVS specification, see `www.cs.ru.nl/~sjakie/papers/readerswriters/`.

4.1   Readers-Writers Model in PVS

PVS offers an interactive environment for the development and analysis of formal specifications. The system consists of a specification language and a theorem prover. The specification language of PVS is based on classic, typed higher-order logic. It resembles common functional programming languages, such as Haskell, LISP or ML. The choice of PVS as the theorem prover to model the readers writers locking algorithm is purely based upon the presence of local expertise. The proof can be reconstructed in any reasonably modern theorem prover, for instance Isabelle [20] or Coq[5]. There is no implicit notion of state in PVS specifications. So, we explicitly keep track of a system state that basically consists of the system variables used in the Uppaal model.

In the Uppaal model a critical section starts with a `lock!` and ends with either a `unlock!`, `readersWait!` or `writersWait!` synchronization. Not all the state transitions are modelled individually in the PVS model. All actions occuring inside a critical section are modeled as a single transition. This makes the locking mechanism protecting the critical sections superfluous in the PVS model and enables us to reduce the number of different locations. Only these locations in

the Uppaal model that are outside a critical section are needed and are tracked by the `ThreadLocation` variable. Furthermore, the `EnterXX` and `LeaveXX` locations are ignored, because they are only used as a label for a function call and have no influence on the behavior of the modeled processes.

With `NT` denoting the total number of processes, we get the following representation:

```
ThreadID       : TYPE = below(NT)²
ThreadLocation : TYPE = { START, RWAIT, RBLOCKED, WWAIT, WBLOCKED }
ThreadInfo     : TYPE = [# status : ThreadLocation, current : nat #]³

System         : TYPE = [# waitingWriters, waitingReaders,
                             numberOfThreads : nat,
                             currentWriter   : below(NT+1),
                             threads : ARRAY [ThreadID → ThreadInfo] #]⁴
```

The auxiliary variables `readNest`, `writeNest` and `maxNest` restrict the Uppaal model to a maximum number of nested reads and writes. They also prevent unwanted sequences of lock/unlock operations, e.g. when a write lock request occurs after a read lock has already been obtained. In the PVS model we allow for any amount of nesting, so the variables `writeNest` and `maxNest` introduced to limit nesting can be discarded. The `readNest` variable is used to check whether there already is a read lock present when a write lock is requested. In the PVS model we have implemented this check by testing whether the lock counter for this particular thread is 0 before it starts waiting for a (non-reentrant) write lock. The logic behind it is that if, previously, a read lock had been obtained by this thread, the counter would have been unequal to 0.

Because none of the variable updates in the Uppaal model occur outside of a critical section, we can model the concurrent execution of the different processes obtaining writelocks, readlocks and releasing them by treating them as interleaved functions.

We first define a step function that executes one of the possible actions for a single process. The step function is restricted to operate on a subset of the `System` data type, signified by the `validState?` predicate, further explained in Section 4.3. The actions themselves do not deliver just a new state but a *lifted* state. In PVS, the predefined `lift` datatype, consisting of two constructors `up` and `bottom`, adds a bottom element to a given base type, in our case `validState?` incorporating the state of the model. This is useful for defining partial functions, particularly to indicate the cases that certain actions are not permitted.

In essence the step function corresponds to the center of the Uppaal model consisting of the `Start` and the `EnterXX`/`LeaveXX` states.

```
step(tid:ThreadID, s1, s2: (validState?) ):  bool =
    writelock(s1,tid) = up(s2) ∨ readlock(s1,tid) = up(s2) ∨
        unlock(s1,tid) = up(s2)
```

---

[2] Denotes the set of natural numbers between 0 and NT, exclusive of NT.

[3] Recordtypes in PVS are surrounded by [# and #].

[4] Arrays in PVS are denoted as functions.

The predicate `interleave` simulates parallel execution of threads.

```
interleave (s1,s2:System): bool =
    ∃ (tid:ThreadID): step(tid,s1,s2) ∧
        ∀ (other_tid: ThreadID): other_tid ≠ tid ⇒
            s1'threads(other_tid) = s2'threads(other_tid)  5
```

4.2 Translation from Uppaal to PVS

The functions that perform the readlock, writelock and unlock respectively are essentially the same as in the original code. It is very well possible to derive the code automatically from the Uppaal model by identifying all paths that start with a `lock!` action on its edge and lead to the first edge with an `unlock!`, `readersWait!` or `writersWait!` action. The `readlock` function is provided as an example of this translation. For instance, the round trip in Figure 4 from the `Start` location, through `BeginRL` directly going to `EndRL`, has guard `current[tid] > 0`, and action `current[tid]++;` associated with it. It starts and ends in the `START` location of the PVS model. This can be recognized as a part of the code of the `readlock` function below.

```
readlock(s1:(validState?), tid:ThreadID) : lift[(validState?)]  =
LET thread = s1'threads(tid) IN
    CASES thread'status OF
    START:
        IF thread'current > 0
        THEN up(s1 WITH [threads := s1'threads WITH
                [tid := thread WITH [current := thread'current+1]]])
        ELSIF s1'currentWriter ≠ NT ∨ s1'waitingWriters > 0
        THEN up(s1 WITH [waitingReaders := s1'waitingReaders + 1,
                threads := s1'threads WITH
                [tid := thread WITH [status := RWAIT]]])
        ELSE up(s1 WITH [ numberOfThreads := s1'numberOfThreads + 1,
                threads := s1'threads WITH
                [tid := thread WITH [current := 1]]])
        ENDIF,
    RBLOCKED:
        IF s1'currentWriter ≠ NT ∨ s1'waitingWriters > 0
        THEN up(s1)
        ELSE up(s1 WITH [ numberOfThreads := s1'numberOfThreads + 1,
                waitingReaders := s1'waitingReaders - 1,
                threads := s1'threads WITH
                [tid := thread WITH [current := 1, status := START]]])
        ENDIF
    ELSE:
        up(s1)
    ENDCASES
```

---

[5] The ' operator denotes record selection.

4.3 System Invariants

Not every combination of variables will be reached during normal execution of
the program. Auxiliary variables are maintained that keep track of the total
amount of processes that are in their critical section and of the number of pro-
cesses that are waiting for a lock. We express the consistency of the values of
those variables by using a `validState?` predicate. This is an invariant on the
global state of all the processes and essential in proving that the algorithm is
deadlock free. We want to express in this invariant that the global state is sane
and safe. Sanity is defined as:

- The value of the `waitingReaders` should be equal to the total number of
  processes with a status of `RWAIT` or `RBLOCKED`.
- The value of the `waitingWriters` should be equal to the total number of
  processes with a status of `WWAIT` or `WBLOCKED`.
- The value of the `numberOfThreads` variable should be equal to the number
  of processes with a lock count of 1 or higher.

Besides the redundant variables having sane values, we also prove that the in-
variant satisfies that any waiting process has a count of zero current readlocks,
stored in the `current` field of `ThreadInfo`. Furthermore, if a process has ob-
tained a write lock, then only that process can be in its critical section:

```
s: VAR System countInv(s): bool = s'numberOfThreads = count(s'threads)

waitingWritersInv(s): bool = s'waitingWriters = waitingWriters(s)
waitingReadersInv(s): bool = s'waitingReaders = waitingReaders(s)

statusInv(s): bool = ∀(tid:ThreadID):
  LET thr = s'threads(tid) IN
     thr'status = WWAIT ∨ thr'status = WBLOCKED ∨
     thr'status = RWAIT ∨ thr'status = RBLOCKED ⇒ thr'current = 0

writeLockedByInv(s) : bool = LET twlb = s'currentWriter IN
  twlb ≠ NT ⇒ s'numberOfThreads = 1 ∧
     s'threads(twlb)'status = START ∧  s'threads(twlb)'current > 0 ∧
     ∀(tid:ThreadID): tid ≠ twlb ⇒ s'threads(tid)'current = 0))

validState?(s) : bool =  countInv(s) ∧ waitingWritersInv(s) ∧
          statusInv(s) ∧ writeLockedByInv(s) ∧ waitingReadersInv(s)
```

Before trying to prove the invariant with PVS, we have first tested the above
properties (except for `waitingWritersInv`) and `waitingReadersInv`) in the Up-
paal model to see if they hold in the fixed size model (see Figure 5). The proper-
ties `waitingWritersInv` and `waitingReadersInv` cannot be expressed in Uppaal
because one cannot count the number of processes residing in a specific location.
The inspection of the above properties in Uppaal enables us to detect any mis-
takes in the invariant before spending precious time on trying to prove them in
PVS.

- A[] countCurrents() = numberOfThreads   (COUNT INV.) [6]
- A[] ∀t ∈ ThreadId : Thread(t).WWait ∨ Thread(t).RWait∨
  Thread(t).WBlocked ∨ Thread(t).RBlocked ⇒ current[t] = 0   (STATUS INV.)
- A[] currentWriter ≠ NT ⇒   (WRITELOCKEDBY INV.)
  numberOfThreads = 1 ∧
  ¬Thread(currentWriter).writeLockEnd ⇒ current[currentWriter] > 0 ∧
  ∀t ∈ ThreadId : t ≠ currentWriter ⇒ current[t] = 0

Fig. 5. The invariants checked in Uppaal

The definition of the `readlock` function over the dependent type `validState?` implies that automatically type checking conditions are generated. They oblige us to prove that, if we are in a valid state, the transition to another state will yield a state for which the invariant still holds. The proof itself is a straightforward, albeit large (about 400 proof commands), case distinction with the help of some auxiliary lemmas.

4.4   No Deadlock

The theorem-prover PVS does not have an innate notion of deadlock. If, however, we consider the state-transition model as a directed graph, in which the edges are determined by the `interleave` function, deadlock can be detected in this state transition graph by identifying a state for which there are no outgoing edges. This interpretation of deadlock can be too limited. If, for example, there is a situation where a process alters one of the state variables in a non terminating loop, the state-transition model will yield an infinite graph and a deadlock will not be detected, because each state has an outgoing edge. Still, all the other processes will not be able to make progress. To obtain a more refined notion of deadlock, we define a well founded ordering on the system state and show that for each state reachable from the starting state (except for the starting state itself), there exists a transition to a smaller state according to that ordering. The smallest element within the order is the starting state. This means that each reachable state has a path back to the starting state and consequently it is impossible for any process to remain in a such a loop indefinitely. Moreover, this also covers the situation in which we would have a *local deadlock* (i.e. several but not all processes are waiting for each other).

```
t : VAR ThreadInfo
starting? : PRED[ThreadInfo] = { t | t'status = START ∧ t'current = 0}

startingState(s: (validState?)): bool =
  ∀(tid:ThreadID): starting?(s'threads(tid))
```

In the starting state all processes are running and there are no locks.

We create a well founded ordering by defining a state to become smaller if the number of waiting processes decreases or alternatively, if the number of waiting

---

[6] `countCurrents` determines the number of threads having a `current` greater than 0.

processes remains the same and the total count of the number of processes that have obtained a lock is decreasing. Well foundedness follows directly from the well foundedness of the lexicographical ordering on pairs of natural numbers.

```
smallerState(s2, s1 : (validState?)) : bool =
    numberWaiting(s2) < numberWaiting(s1) ∨
        numberWaiting(s2) = numberWaiting(s1) ∧
        totalCount(s2) < totalCount(s1)
```

The `numberWaiting` function as well as the `totalCount` function are recursive functions on the array with thread information yielding the number of processes that have either a `RBLOCKED`, `RWAIT`, `WBLOCKED` or `WWAIT` status, and sum of all `current` fields respectively.

Once we have established that each state transition maintains the invariant, all we have to prove is that each transition, except for the starting state will possibly result in a state that is smaller. This is the `noDeadlock` theorem. Proving this theorem is mainly a case distinction with a couple of inductive proofs thrown in for good measure. The induction is needed to establish that the increase and decrease in the variables can only happen if certain preconditions are met. The proof takes about 300 proof commands.

```
noDeadlock: THEOREM
    ∀(s1: (validState?)) : ¬startingState(s1) ⇒
        ∃(s2: (validState?)) : interleave(s1, s2) ∧ smallerState(s2, s1)
```

5   Related and Future Work

Several studies investigated *either* the conversion of code to state transition models, as is done e.g. in [28] with mcrl2 *or* the transformation of a state transition model specified in a model checker to a state transition model specified in a theorem prover, as is done e.g. in [16] using VeriTech. With the tool TAME one can specify a time automaton directly in the theorem prover PVS [3]. For the purpose of developing consistent requirement specifications, the transformation of specifications in Uppaal [17] to specifications in PVS has been studied in [9].

In [22] model checking and theorem proving are combined to analyze the classic (non-reentrant) Readers/Writers problem. The authors do not start with actual source code but with a tabular specification that can be translated straightforwardly into SPIN and PVS. Safety and clean completion properties are derived semi-automatically. Model checking is used to validate potential invariants.

[13] reports on experiments in combing theorem proving with model checking for verifying transition systems. The complexity of systems is reduced abstracting out sources for unboundedness using theorem proving, resulting in an bounded system suited for being model checked. One of the main difficulties is that formal proof techniques are usually not scalable to real sized systems without an extra effort to abstract the system manually to a suitable model.

The verification framework SAL (See [25]) combines different analysis tools and techniques for analyzing transition systems. Besides model checking and theorem proving it provides program slicing, abstraction and invariant generation.

In [12] part of an aircraft control system is analyzed, using a theorem prover. This experiment was previously performed on a single configuration with a model checker. A technique called *feature-based decomposition* is proposed to determine inductive invariants. It appears that this approach admits incremental extension of an initially simple base model making it better scalable than traditional techniques.

Java Pathfinder (JPF) [29] operates directly on Java making a transformation of source code superfluous. However, this tool works on a complete program, such that it is much more difficult to create abstractions. The extension of JPF with symbolic execution as discussed by [1] might be a solution to this problem.

An alternative for JPF is Bandera [7], which translates Java programs to the input languages of SMV and SPIN. Like in JPF, it is difficult to analyse separate pieces of code in Bandera. There is an interesting connection between Bandera and PVS. To express that properties do not depend on specific values, Bandera provides a dedicated language for specifying abstractions, i.e. concrete values are automatically replaced by abstract values, thus reducing the state space. The introduction of these abstract values may lead to prove obligations which can be expressed and proven in PVS.

In [24] a model checking method is given which uses an extension of JML [18] to check properties of multi-threaded Java programs.

With Zing [2] on the one hand models can be created from source code and on the other hand executable versions of the transition relation of a model can be generated from the model. This has been used successfully by Microsoft to model check parts of their concurrency libraries.

Future Work

The methodology used (creating in a structured way a model close to the code, model checking it first and proving it afterwards) proved to be very valuable. We found a bug, improved the code, extended the capabilities of the code and proved it correct. One can say that the model checker was used to develop the formal model which was proven with the theorem prover. This decreased significantly the time investment of the use of a theorem prover to enhance reliability. However, every model was created manually. We identified several opportunities for tool support and further research.

Model checked related to source code. Tool support could be helpful here: not only to 'translate' the code from the source language to the model checker's language. It could also be used to record the abstractions that are made. In this case that were: basic locks → lock process model, hash tables → arrays, threads → processes and some name changes. A tool that recorded these abstractions, could assist in creating trusted source code from the model checked model.

Model checked related to model proven. It would be interesting to prove that the model in the theorem prover is equivalent with the model checked. Interesting methods to do this would be using a semantic compiler, as was done in the European Robin project [27], or employing a specially designed formal library for models created with a model checker, like e.g. TAME [3].

Model proven related to source code. Another interesting future research option is to investigate generating code from the fully proven model. This could be code generated from code-carrying theories [15] or it could be proof-carrying code through the use of refinement techniques [4].

6   Concluding Remarks

We have investigated Trolltech's widely used industrial implementation of the reentrant readers-writers problem. Model checking revealed an error in the implementation. Trolltech was informed about the bug. Recently, Trolltech released a new version of the thread library (version 4.4) in which the error was repaired. However, the new version of the Qt library is still only weakly reentrant, not admitting threads that have write access to do a read lock. This limitation unnecessarily hampers modular programming.

The improved Readers-Writers model described in this paper is *deadlock free* and *strongly reentrant*. The model was first developed and checked for a limited number of processes using a model checker. Then, the properties were proven for any number of processes using a theorem prover.

Acknowledgements

References

1. Anand, S., Pasareanu, C.S., Visser, W.: Jpf-se: A symbolic execution extension to java pathfinder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 134–138. Springer, Heidelberg (2007)
2. Andrews, T., Qadeer, S., Rajamani, S.K., Rehof, J., Xie, Y.: Zing: A model checker for concurrent software. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 484–487. Springer, Heidelberg (2004)
3. Archer, M., Heitmeyer, C., Sims, S.: TAME: A PVS interface to simplify proofs for automata models. In: User Interfaces for Theorem Provers, Eindhoven, The Netherlands (1998)
4. Barbosa, M.A.: A refinement calculus for software components and architectures. SIGSOFT Softw. Eng. Notes 30(5), 377–380 (2005)
5. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. In: Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer, Heidelberg (2004)

6. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In: POPL, pp. 117–126 (1983)
7. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Pasareanu, C.S., Robby, Zheng, H.: Bandera: extracting finite-state models from java source code. In: Proceedings of the 2000 International Conference on Software Engineering, pp. 439–448 (2000)
8. Courtois, P.J., Heymans, F., Parnas, D.L.: Concurrent control with "readers" and "writers". Commun. ACM 14(10), 667–668 (1971)
9. de Groot, A.: Practical Automaton Proofs in PVS. PhD thesis, Radboud University Nijmegen (2008)
10. Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., Lea, D.: Java Concurrency in Practice. Addison Wesley Professional, Reading (2006)
11. Groote, J.F., Mathijssen, A.H.J., Reniers, M.A., Usenko, Y.S., van Weerdenburg, M.J.: The formal specification language mCRL2. In: Proc. Methods for Modelling Software Systems, number 06351 in Dagstuhl Seminar Proceedings (2007)
12. Ha, V., Rangarajan, M., Cofer, D., Rues, H., Dutertre, B.: Feature-based decomposition of inductive proofs applied to real-time avionics software: An experience report. In: ICSE 2004: Proceedings of the 26th International Conference on Software Engineering, Washington, DC, USA, pp. 304–313. IEEE Computer Society, Los Alamitos (2004)
13. Havelund, K., Shankar, N.: Experiments in Theorem Proving and Model Checking for Protocol Verification. In: Gaudel, M.-C., Woodcock, J.C.P. (eds.) FME 1996. LNCS, vol. 1051, pp. 662–681. Springer, Heidelberg (1996)
14. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering 23(5), 279–295 (1997)
15. Jacobs, B., Smetsers, S., Wichers Schreur, R.: Code-carrying theories. Formal Asp. Comput. 19(2), 191–203 (2007)
16. Katz, S.: Faithful translations among models and specifications. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 419–434. Springer, Heidelberg (2001)
17. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. International Journal on Software Tools for Technology Transfer 1(1-2), 134–152 (1997)
18. Leavens, G.T., Kiniry, J.R., Poll, E.: A jml tutorial: Modular specification and verification of functional behavior for java. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, p. 37. Springer, Heidelberg (2007)
19. McMillan, K.L.: The SMV System. Carnegie Mellon University (1998-2001), `http://www.cs.cmu.edu/~modelcheck/smv.html`
20. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
21. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992)
22. Pantelic, V., Jin, X.-H., Lawford, M., Parnas, D.L.: Inspection of concurrent systems: Combining tables, theorem proving and model checking. In: Arabnia, H.R., Reza, H. (eds.) Software Engineering Research and Practice, pp. 629–635. CSREA Press (2006)
23. Queille, J.-P., Sifakis, J.: Specification and verification of concurrent systems in cesar. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Programming 1982. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982)
24. Robby, E.R., Dwyer, M.B., Hatcliff, J.: Checking jml specifications using an extensible software model checking framework. STTT 8(3), 280–299 (2006)

25. Shankar, N.: Combining theorem proving and model checking through symbolic analysis. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 1–16. Springer, Heidelberg (2000)
26. Sutter, H.: The free lunch is over: A fundamental turn toward concurrency in software. Dr. Dobb's Journal 30(3) (March 2005)
27. Tews, H., Weber, T., Völp, M., Poll, E., van Eekelen, M., van Rossum, P.: Nova Micro–Hypervisor Verification. Technical Report ICIS–R08012, Radboud University Nijmegen, Robin deliverable D13 (May 2008)
28. van Eekelen, M., ten Hoedt, S., Schreurs, R., Usenko, Y.S.: Analysis of a session-layer protocol in mcrl2. verification of a real-life industrial implementation. In: Leue, S., Merino, P. (eds.) FMICS 2007. LNCS, vol. 4916, pp. 182–199. Springer, Heidelberg (2008)
29. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. 10(2), 203–232 (2003)

Using CSP‖B Components: Application to a Platoon of Vehicles[⋆]

Samuel Colin[1], Arnaud Lanoix[1], Olga Kouchnarenko[2], and Jeanine Souquières[1]

[1] LORIA – DEDALE Team – Nancy Université
Campus scientifique
F-54506 Vandoeuvre-Lès-Nancy, France
{firstname.lastname}@loria.fr
[2] LIFC – TFC Team – University of Franche-Comté
16 route de Gray
F-25030 Besançon, France
{firstname.lastname}@lifc.univ-fcomte.fr

Abstract. This paper presents an experience report on the specification and the validation of a real case study in the context of the industrial CRISTAL project. The case study concerns a platoon of a new type of urban vehicles with new functionalities and services. It is specified using the combination, named CSP‖B, of two well-known formal methods, and validated using the corresponding support tools. This large – both distributed and embedded – system typically corresponds to a multi-level composition of components that have to cooperate. We identify some lessons learned, showing how to develop and verify the specification and check some properties in a compositional way using theoretical results and support tools to validate this complex system.

Keywords: formal methods, CSP‖B, compositional modelling, specification, verification, case study.

1 Introduction

This paper is dedicated to an experience report on the specification and the validation of a real case study in the land transportation domain. It takes place in the context of the industrial CRISTAL project which concerns the development of a new type of urban vehicles with new functionalities and services. One of its major cornerstones is the development, the validation and the certification of platoon of vehicles. A platoon is a set of autonomous vehicles which have to move in a convoy – i.e. following the path of the leader – through an intangible hooking.

Through the CRISTAL project's collaboration, we have decided to consider each vehicle, named Cristal in the following, as an agent of a Multi-Agent System (MAS). The Cristal driving system perceives information about its environment before producing an instantaneous acceleration passed to its engine. In this context, we consider the
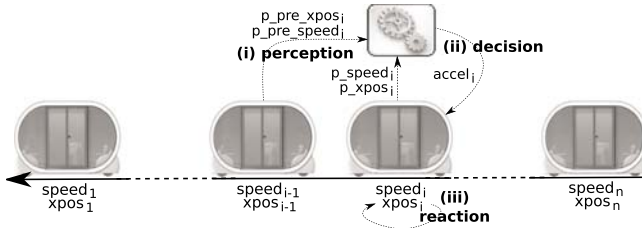
Fig. 1. A platoon of Cristals

platooning problem as a situated MAS which evolves following the Influence/Reaction model (I/R) [1] in which agents are described separately from the environment. The driving control concerns both a longitudinal control, i.e. maintaining an *ideal* distance between each vehicle, and a lateral control, i.e. each vehicle should follow the track of its predecessor, see Fig. 1. Both controls can be studied independently [2]. At this time, we focus solely on the longitudinal control.

The platoon of Cristal vehicles is a mix of distributed and embedded systems. The former are usually hard to understand and to debug as they can exhibit obscure behaviours. The latter require the satisfaction of safety/security/confidence requirements, alone and when composed together. To address these problems, we reuse the CSP‖B framework proposed by Schneider and Treharne [3] of well-established formal methods, B, an environment for the development of provably correct software [4], and CSP (for Communicating Sequential Processes), a process algebra introduced by Hoare [5] for modelling patterns of interactions. We motivate the use of CSP‖B by the existence of pure B models describing the agents and vehicles behaviours [6]. By using CSP for coordinating B machines, we aim at giving these B models the architectural, compositional description they lack.

Our approach can be described as a mix between a "bottom-up" and a component-based development. On the one hand, B machines are seen as the smallest abstract components representing various parts of a Cristal vehicle. On the other hand, CSP is used to put these components together, to describe higher-level compounds such as a vehicle or a whole convoy and to make them communicate.

Our first experience with the CSP‖B platoon model is presented in a short paper [7]. Here the description of the case study involves detailing two architectural levels. We first consider a single Cristal, then we show how to reuse it to constitute a platoon. Later on we make the model evolve by replacing one component with several others to separate functionalities and refine them[1]. This can be achieved for instance by adapters to connect these new components within the initial architecture [8]. We follow a similar approach, only CSP-oriented. Moreover we use previous theoretical results on CSP‖B in an unintended way in this context.

On both the model description and its evolution, we illustrate the relevance of CSP‖B for eliminating errors and ambiguities in an assembly and its communication

---

[1] CSP‖B specifications discussed in this paper are available at
http://tacos.loria.fr/platoon-fmics08.zip

protocols. We are convinced that writing formal specifications can aid in the process of designing autonomous vehicles.

This paper is organised as follows. Section 2 briefly introduces the basic concepts and existing tools on CSP‖B. Section 3 presents the specification and the verification process of a single Cristal vehicle whereas Sect. 4 is dedicated to a platoon of vehicles. Section 5 details a vehicle introducing new components, the engine and the location ones. Section 6 presents related works, and Sect. 7 ends with lessons learned from this industrial experience and some perspectives of this development.

2   Basic concepts and Tools on CSP‖B

The B machines specifying components are open modules which interact by the authorised operation invocations. CSP describes processes, i.e. objects or entities which exist independently, but may communicate with each other. When combining CSP and B to develop distributed and concurrent systems, CSP is used to describe execution orders for invoking the B machines operations and communications between the CSP processes.

2.1   B Machines

B is a formal software development method used to model and reason about systems [4]. The B method has proved its strength in industry with the development of complex real-life applications such as the Roissy VAL [9]. The principle behind building a B model is the expression of system properties which are always true after each evolution step of the model. The verification of a model correctness is thus akin to verifying the preservation of these properties, no matter which step of evolution the system takes.

The B method is based on first-order logic, set theory and relations. Properties are specified in the **INVARIANT** clause of the model, and its evolution is specified by the operations in the **OPERATIONS** clause (see Fig. 3 for an example). The verification of a B model consists in verifying that each operation – assuming its precondition and the invariant hold – satisfies the **INVARIANT**, i.e. the model is *consistent*.

Support tools such as B4free (`http://www.b4free.com`) or AtelierB (`http://www.atelierb.eu`) automatically generate Proof Obligations (POs) to ensure the consistency. In our case study in Sect. 3 we use the B4free proof tool for ensuring this consistency: this tool generates so-called "obvious" POs automatically discharged and normal POs which have to be proved interactively if it was not done automatically.

A strength of the B method is its stepwise refinement feature: the **REFINEMENT** of a model makes it less indeterministic and more precise with the introduction of more programming language-like features. Refinement can be done until the code of the operations can actually be implemented in a programming language. The consistency of a refinement must also be checked, this time by ensuring that the newly introduced behaviour and/or data do not contradict the model they refine.

2.2   Communicating Sequential Processes (CSP)

CSP allows the description of entities, called processes, which exist independently but may communicate with each other. Thanks to dedicated operators it is possible to describe a set of processes as a single process, making CSP an ideal formalism for building a hierarchical composition of components. CSP is supported by the FDR2 model checker (http://www.fsel.com). This tool is based on the generation of all the possible states of a model and the verification of these states against a desired property. We used it for our case study in Sect. 3 and 4.

The denotational semantics of CSP is based on the observation of process behaviours. Three kinds of behaviours [10] are observed and well suited for the expression of properties:

- traces, i.e. finite sequences of events, for safety properties;
- stable failures, i.e. traces augmented with a set of unperformable events at the end thereof, for liveness properties and deadlock-freedom;
- failures/divergences, i.e. stable failures augmented with traces ending in an infinite loop of internal events, for livelock-freedom.

Each semantics is associated with a notion of process refinement denoted:

- $\sqsubseteq_T$ for traces refinement. This refinement is based on the equality of execution traces.
- $\sqsubseteq_{SF}$ for stable failures refinement. It is based on traces equality and failures equality, i.e. traces ending in a deadlock must be the same in the abstract process and its refinement.
- $\sqsubseteq_{FD}$ for failures/divergences refinement. It is based on traces, failures and divergences equality, i.e. traces ending in an infinite loop must also be equal in the abstract process and its refinement. It is the strongest form of refinement.

2.3   CSP∥B Components

In this section, we sum up the works by Schneider and Treharne on CSP∥B. The reader interested in theoretical results is referred to [3,11,12]; for case studies, see for example [13,14].

Specifying CSP controllers. In CSP∥B, the B part is specified as a standard B machine without any restriction, while a controller for a B machine is a particular kind of CSP process, called a CSP controller, defined by the following (subset of the) CSP grammar:

$$P ::= c \ ? \ x \ ! \ v \ \rightarrow P \ | \ ope \ ! \ v \ ? \ x \rightarrow P \ | \ b \ \& \ P$$
$$| \ P1 \ \Box \ P2 \ | \ \textbf{if} \ b \ \textbf{then} \ P1 \ \textbf{else} \ P2 \ | \ S(p)$$

The process $c \ ? \ x \ ! \ v \rightarrow P$ can accept input x and output v along a communication channel c. Having accepted x, it behaves as P.

A controller makes use of *machine channels* which provide the means for controllers to synchronise with the B machine. For each operation $x \leftarrow ope(v)$ of a controlled machine, there is a channel $ope \ ! \ v \ ? \ x$ in the controller corresponding to the operation

call: the output value v from the CSP description corresponds to the input parameter of the B operation, and the input value x corresponds to the output of the operation. A controlled B machine can only communicate on the machine channels of its controller.

The behaviour of a guarded process b & P depends on the evaluation of the boolean condition b: if it is true, it behaves as P, otherwise it is unable to perform any events. In some works (e.g. [3]), the notion of *blocking assertion* is defined by using a guarded process on the inputs of a channel to restrict these inputs: c ? x & E(x) → P.

The external choice P1 □ P2 is initially prepared to behave either as P1 or as P2, with the choice made on the occurrence of the first event. The conditional choice **if** b **then** P1 **else** P2 behaves as P1 or P2 depending on b. Finally, S(p) expresses a recursive call.

Assembling CSP∥B components. In addition to the expression of simple processes, CSP provides operators to combine them. The sharing operator P1 $\|_E$ P2 executes P1 and P2 concurrently, requiring that P1 and P2 synchronise on the events into the sharing alphabet E and allowing independent executions for other events. When combining a CSP controller P and a B machine M associated with P, the sharing alphabet can be dropped ($(P \|_{\alpha(M)} M) \equiv P \| M$) as there is no ambiguity.

We also consider an indexed form of the sharing operator $\|_{E_i}^i P(i)$ which executes the processes P(i) in a sharing manner. It is used to build up a collection of similar controlled machines which exchange together.

Verifying CSP∥B components. The verification process to ensure the consistency of a controlled machine (P∥M) in CSP∥B consists in verifying the following conditions:

1. the M machine *consistency* is checked using the B4Free proof tool;
2. the P controller *deadlock-freedom* in the stable-failures model is checked with the FDR2 model-checking tool;
3. the P controller *divergence-freedom* is checked with FDR2;
4. the *divergence-freedom* of (P∥M) can be deduced by using a technique based on *Control Loop Invariants* (CLI):
    – P is translated into a B machine BBODY$_P$ using the rewriting rules of [11];
    – a CLI is added to BBODY$_P$;
    – the BBODY$_P$ machine consistency checking is performed with B4Free;
    – by way of [12, Theorem 1], we deduce the *divergence-freedom* of (P∥M);
5. by way of [3, Theorem 5.9] and the fact that P is deadlock-free, we deduce the *deadlock-freedom* of (P∥M) in the stable failures model.

This verification process can be generalised to achieve the consistency checking of a collection of controlled machines $\|_{E_i}^i (Pi \| Mi)$:

1. we check the divergence-freedom of each (Pi∥Mi) as previously;
2. by way of [3, Theorem 8.1], we deduce the divergence-freedom of $\|_{E_i}^i (Pi \| Mi)$;
3. we check the deadlock-freedom of $\|_{E_i}^i (Pi)$ with FDR2;
4. by way of [3, Theorem 8.6], we deduce the deadlock-freedom of $\|_{E_i}^i (Pi \| Mi)$.
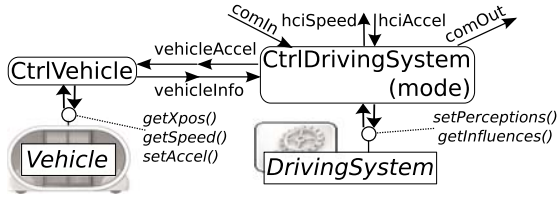
Fig. 2. Architectural view of a Cristal

```
MODEL Vehicle
VARIABLES
  speed, xpos
OPERATIONS
  speed0 ← getSpeed = BEGIN speed0 := speed END ;
  xpos0 ← getXpos = BEGIN xpos0 := xpos END ;
  setAccel(accel) =
   PRE accel ∈ MIN_ACCEL..MAX_ACCEL
   THEN
      ANY new_speed
      WHERE new_speed = speed + accel
      THEN
        IF (new_speed > MAX_SPEED)
        THEN
          xpos := xpos + MAX_SPEED ‖ speed := MAX_SPEED
        ELSE
          IF (new_speed < 0)
          THEN
            xpos := xpos − (speed × speed) / (2 × accel)
            ‖ speed := 0
          ELSE
            xpos := xpos + speed + accel / 2 ‖ speed := new_speed
          END
        END
      END
    END
END
```

```
REFINEMENT CtrlVehicle_ref
VARIABLES
  xpos_csp, speed_csp, cb
INVARIANT
  xpos_csp ∈ Positions_csp
∧ speed_csp ∈ Speeds_csp
∧ cb = 0
OPERATIONS
CtrlVehicle =
  BEGIN
    CHOICE
      BEGIN
        xpos_csp ← getXpos ;
        speed_csp ← getSpeed ;
        ANY accel_csp WHERE
          accel_csp ∈ Accels_csp
        THEN
          setAccel(accel_csp); cb := 0
        END
      END
    OR
      BEGIN
        speed_csp ← getSpeed ;
        xpos_csp ← getXpos;
        ANY accel_csp WHERE
          accel_csp ∈ Accels_csp
        THEN
          setAccel(accel_csp); cb := 0
        END
      END
    END
  END
END
```

Fig. 3. The Vehicle B model

Fig. 4. B rewriting of CtrlVehicle

```
CtrlVehicle =
  ( getXpos ? xpos → getSpeed ? speed → vehicleInfo ! xpos ! speed →
    vehicleAccel ? accel → setAccel ! accel → CtrlVehicle )
  □
  ( getSpeed ? speed → getXpos ? xpos → vehicleInfo ! xpos ! speed →
    vehicleAccel ? accel → setAccel ! accel → CtrlVehicle )
```

Fig. 5. The CtrlVehicle CSP controller

3   Specifying a Single Cristal

As depicted in Fig. 2, in a first approximation, a Cristal vehicle is composed of two parts: the vehicle and its driving system which controls the vehicle. Each part is itself built upon a B machine controlled by an associated CSP process.

3.1   The Vehicle

Specifying the vehicle. The vehicle is a behavioural component reacting to a given acceleration for speeding up or slowing down. It is built upon a Vehicle B machine that describes its inner workings, i.e. its knowledge of speed and location as well as how it updates them w.r.t. a given acceleration, as illustrated in Fig. 3. The speed ← getSpeed() and xpos ← getXpos() methods capture data from the vehicle. The setAccel(accel) method models how the vehicle behaves when passed on a new instantaneous acceleration.

The B machine is made able to communicate by adding a CSP controller, CtrlVehicle, depicted in Fig. 5. It schedules the calls to its various methods. The speed and the location are passed on to the controller through getSpeed ? speed and getXpos ? xpos channels corresponding to invocations of the homonymous methods of the B machine to retrieve the speed and the location of the vehicle. Then, information about speed and location is sent to requesting components through vehicleInfo ! xpos ! speed. Similarly, the controller receives new instantaneous acceleration orders through vehicleAccel ? accel and passes them on through setAccel ! accel to the B machine.

The whole vehicle component with communication facilities is then defined as a parallel composition of the Vehicle machine and its CtrlVehicle controller.

Verifying the vehicle. We follow the verification process given Sect. 2.3 to ensure the consistency of (CtrlVehicle ‖ Vehicle):

- the Vehicle B machine consistency is successfully checked using B4Free (11 obvious POs + 10 normal POs, 2 of them have been proved interactively)
- the CtrlVehicle controller deadlock-freedom and its divergence-freedom are successfully checked with FDR2 (6 states and 7 and transitions[2] have to be checked);
- Figure 4 illustrates the B rewriting of CtrlVehicle. Its CLI is actually as simple as the ⊤ predicate modulo the typing predicates. This rewriting is shown consistent with B4Free (11 obvious POs + 7 normal POs), then (CtrlVehicle ‖ Vehicle) is divergence-free;
- we automatically deduce the deadlock-freedom of (CtrlVehicle ‖ Vehicle).

3.2   The Driving System

Specifying the driving system. The driving system (CtrlDrivingSystem(mode)‖Driving System) is built up in a similar way. A DrivingSystem B machine models the decision system: it updates its perceptions and decides for an acceleration passed on to the physical vehicle later on.

---

[2] Verifications with FDR2 took place on a Macbook Core 2 Duo 2GHz with 1 GB of RAM.

```
DrivingSys_percept(mode) =
  (  (mode == SOLO) &
  vehicleInfo ? myXpos ? mySpeed → hciSpeed ! mySpeed →  DrivingSys_act(mode) )
  □
  (  (mode == LEADER) &
  vehicleInfo ? myXpos ? mySpeed → hciSpeed ! mySpeed  → comOut ! mySpeed ! myXpos →
  DrivingSys_act(mode) )
  □
  (  (mode == FOLLOWER) &
  vehicleInfo ? myXpos ? mySpeed → comIn ? preSpeed ? preXpos → hciSpeed ! mySpeed →
  setPerceptions! myXpos ! mySpeed ! preXpos ! preSpeed → comOut ! mySpeed ! myXpos →
  DrivingSys_act(mode) )
  □
  (  (mode == LAST) &
  vehicleInfo ? myXpos ? mySpeed → comIn ? preSpeed ? preXpos → hciSpeed ! mySpeed →
  setPerceptions! myXpos ! mySpeed ! preXpos ! preSpeed → DrivingSys_act(mode) )

DrivingSys_act(mode) =
  (  (mode == SOLO) ∨ (mode == LEADER) &
  hciAccel ? accel → vehicleAccel ! accel → DrivingSys_percept(mode) )
  □
  (  (mode == FOLLOWER) ∨ (mode == LAST) &
  getInfluences ? accel → vehicleAccel ! accel → DrivingSys_percept(mode) )

CtrlDrivingSystem(mode) = DrivingSys_percept(mode)
```

Fig. 6. The CtrlDrivingSystem(mode) CSP Controller

Communications are managed by a CtrlDrivingSystem CSP controller shown Fig. 6. It has four running modes corresponding to different uses of a Cristal: SOLO, LEADER of a platoon of Cristals, FOLLOWER of another Cristal into a platoon, and LAST vehicle of a platoon.

In the SOLO mode, the controller requests Cristal speed from the vehicle via vehicle Info ? myXpos ? mySpeed so as to make the HCI displays it (hciSpeed ! mySpeed). It also receives an acceleration from the human driver passed on through hciAccel ? accel and sends this desired acceleration to the vehicle through vehicleAccel ! accel.

The LEADER mode is very similar to the SOLO mode. The only difference consists in additional sending of the Cristal information to the following Cristal via comOut ! mySpeed ! myXpos.

The FOLLOWER mode uses the DrivingSystem B machine: information required by the machine to compute an accurate speed are obtained from the vehicle (vehicleInfo ? myXpos ? mySpeed) and from the leading Cristal (comIn ? preSpeed ? preXpos). Once data are obtained, they are passed on to the B machine through the setPerceptions() method and sent to the following Cristal via comOut ! mySpeed ! myXpos. Otherwise, the acceleration is obtained by a call to the getInfluences() method, and the result is passed on to the vehicle via vehicleAccel ! accel.

The LAST mode is very similar to the FOLLOWER mode. The only difference is that the last vehicle does not send its data to another one.

Verifying the driving system. Using the verification process given Sect. 2.3, the CtrlDrivingSystem(mode)‖DrivingSystem driving system is shown divergence-free and deadlock-free:

- the DrivingSystem B machine is consistent (24 obvious POs + 1 normal PO);
- for each mode, the CtrlDrivingSystem(mode) CSP controller is deadlock-free and divergence-free (4-4 states-transitions for the SOLO mode, 5-5 for the LEADER mode, 7-7 for the FOLLOWER mode and 6-6 for the LAST mode);
- the B rewriting of CtrlDrivingSystem(mode) is consistent (23 obvious POs + 30 normal POs with 2 POs proved interactively).

3.3   The Cristal(mode) Assembly

Specifying the assembly. As illustrated Fig. 2, a Cristal is defined as the parallel composition of a vehicle and its associated driving system, expressed in CSP by:

$$\text{Cristal(mode)} = \big(\text{CtrlVehicle} \parallel \text{Vehicle}\big) \underset{\left\{\left|\begin{smallmatrix}\text{vehicleInfo,}\\\text{vehicleAccel}\end{smallmatrix}\right|\right\}}{\parallel} \big(\text{CtrlDrivingSystem(mode)} \parallel \text{DrivingSystem}\big)$$

Verifying the assembly. Cristal (mode) is shown consistent following the verification process given in Sect. 2.3:

- (CtrlVehicle ‖ Vehicle) and (CtrlDrivingSystem(mode)‖DrivingSystem) are divergence-free, hence Cristal (mode) is also divergence-free;
- Cristal (mode) is deadlock-free as a consequence of the deadlock-freedom of (CtrlVehicle ‖ CtrlDrivingSystem(mode)) checked with FDR2 (8-9 states-transitions for the SOLO mode, 9-10 for LEADER, 11-12 for FOLLOWER, 10-11 for LAST).

Checking a safety property. A safety property we are interested in, states that perception and reaction should alternate while the Cristal runs, i.e. the data are always updated (vehicleInfo) before applying an instantaneous acceleration to the vehicle (vehicleAccel). This property is captured by the following CSP process:

$$\text{Property} = \text{vehicleInfo ? xpos ? speed} \longrightarrow \text{vehicleAccel ? accel} \longrightarrow \text{Property}$$

We need to show that the Cristal meets this property. For that, we first successfully check with FDR2 that there is a trace refinement between the CSP part of Cristal (mode) and Property, i.e. Property $\sqsubseteq_T$ CtrlVehicle ‖ CtrlDrivingSystem(mode). Then, by applying [3, Corollary 7.2], we obtain that Property $\sqsubseteq_T$ Cristal (mode), i.e. the property is satisfied by the Cristal (mode). The verification with FDR2 involved the same figures for states-transitions as for the assembly verification above.

4   Specifying a Platoon of Cristals

Once we dispose of a correct model for a single Cristal (mode), we can focus on the specification of a platoon as presented Fig.7. We want the various Cristals to avoid going stale when they move in a platoon. This might happen because a Cristal waits for information from its leading one, i.e. we do not want the communications in the convoy to deadlock.
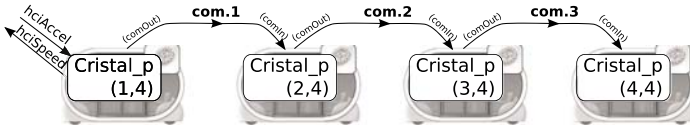
Fig. 7. A Platoon of four Cristals

```
Cristal_p (pos,max) =
  if  (pos == 1)
  then ( Cristal (LEADER) [[ comOut ← com.pos ]] )
  else if  (pos == max)
          then ( Cristal (LAST) [[comIn ← com.(pos−1) ]] )
          else ( Cristal (FOLLOWER) [[comIn ← com.(pos−1), comOut ← com.pos ]] )
```

Fig. 8. Cristal_p (pos,max)

Specifying the assembly. From the CSP‖B specification of a generic Cristal (mode) given in the previous section, we first define a Cristal occupying the position pos into a platoon of max vehicles, as presented Fig. 8: if the Cristal is at the first position, it runs on the LEADER mode, if it is at the last position, it runs on the LAST mode, otherwise, it runs on the FOLLOWER mode. The communication channels are renamed by com.pos/com.pos−1, so that the comOut channel of one Cristal matches with the comIn channel of the following Cristal.

A platoon of max Cristals is defined as an assembly of max Cristal_p(pos,max) synchronised on {|com.pos|}, as illustrated Fig. 7 for four vehicles:

$$\text{Platoon(max)} = \underset{\{|com.pos|\}}{\overset{pos \in \{1..max\}}{\big\|\big\|}} \Big(\text{Cristal\_p(pos,max)} \Big)$$

Verifying the assembly. To check the consistency of Platoon(max), we follow the verification process presented in Sect. 2.3. Since each Cristal is proved divergence-free, Platoon(max) is divergence-free.

We have to consider the parallel composition of the CSP parts of all the Cristals. Table 1 shows results for the considered number of vehicles into the checked platoon. The verification becomes more time-consuming starting from about 11 vehicles. However, starting from four vehicles, the number of vehicles does not change the communication modes because it is all what we need to check all kinds of intercommunications: between a leader and a follower, between two following vehicles and between a follower and the last vehicles.

Table 1. Checks of the CSP parts of Platoon(max)

	states	transitions	time
Platoon(2)	45	95	0
Platoon(3)	225	700	0
Platoon(4)	1,125	4,625	0
Platoon(5)	5,625	28,750	0
Platoon(6)	28,125	171,875	0
Platoon(7)	140,625	1,000,000	2s
Platoon(8)	703,125	5,703,125	14s
Platoon(9)	3,515,625	32,031,250	1m27s
Platoon(10)	17,578,125	177,734,375	8m09s
Platoon(11)	87,890,625	976,562,500	3h01m56s

FDR2 checks that this assembly is deadlock-free, hence Platoon(max) is deadlock-free. Consequently, this verification process validates the safety property introduced at the beginning of Sect. 4 saying that the communications, expressed through renaming, should not deadlock.
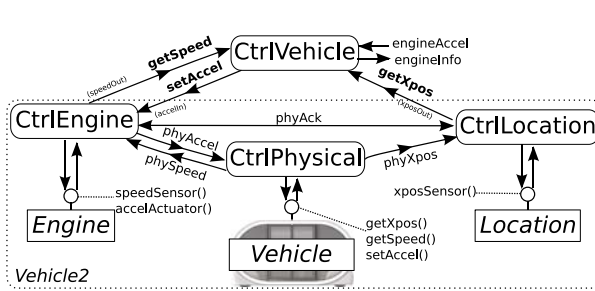
Fig. 9. The Vehicle2 component
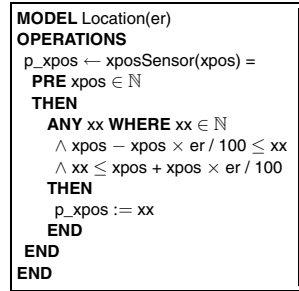
Fig. 10. The Location B model

5   Detailing (CtrlVehicle(mode)‖Vehicle)

The definition of the vehicle part presented in Sect. 3.1 is very general. In order to detail information about the vehicle engine and its location, reflecting separation of concerns inside the (CtrlVehicle (mode)‖Vehicle) component, we make the model presented in Fig. 2 evolve. This evolution introduces new components as illustrated in Fig. 9. They correspond to the following design choices:

1. Now the Vehicle B machine represents the "real" physical vehicle.
2. For compatibility purpose with the rest of the system, the CtrlVehicle is preserved without any modifications.
3. Two new B components are added, modelling two sensors and an actuator, introducing a loss of precision to represent the sensor and actuator effects:
   - The B Location machine show Fig. 10 represents an abstract location system able to determine the geographic location of the physical vehicle. It perceives the "real" location and returns an approximated value through
     p_xpos ← xposSensor(xpos) (with an error of er%). It might be implemented later on by a GPS system, for instance.
   - The B Engine machine is introduced to model a speed sensor on the physical vehicle and an acceleration actuator. It senses the "real" speed, returns an approximated value through p_speed ← speedSensor(speed) and applies a decided acceleration order through accel ← accelActuator(d_accel).
4. Three new CSP controllers must be introduced to control the new B machines and to manage communications, i.e. perceptions on the physical world and exchanges between the machines.

5.1   Three New CSP controllers

Specifying CtrlPhysical. This controller manages the perceptions on the real vehicle. It calls the speed ← getSpeed() and xpos ← getXpos() B methods – to accurate the "real" speed and xpos – and sends these data on phyXpos ! xpos and phySpeed ! speed.

It receives a decided acceleration through phyAccel ? accel, then it calls the method setAccel(accel) .

```
CtrlPhysical  =
  ( getSpeed ? speed → phySpeed ! speed → getXpos ? xpos →
    phyXpos ! xpos  → phyAccel ? accel → setAccel ! accel → CtrlPhysical  )
  □
  ( getXpos ? xpos → phyXpos ! xpos → getSpeed ? speed →
    phySpeed ! speed  → phyAccel ? accel → setAccel ! accel → CtrlPhysical )
```

Specifying CtrlLocation. This controller manages the B Location machine. It perceives the "real" location on phyXpos ? xpos and calls p_xpos ← xposSensor(xpos) to pass them on to the Location component. It sends the *perceived* location through xposOut ! p_xpos.

```
CtrlLocation  =
  phyXpos ? xpos → xposSensor ! xpos ? p_xpos → xposOut ! p_xpos → phyAck → phyAck→ CtrlLocation
```

Specifying CtrlEngine. This controller is in charge of the Engine B machine, i.e. the speed sensor and the acceleration actuator. A speed perception consists in receiving the "real" speed on phySpeed, passing it on to the B machine by calling the p_speed ← speedSensor(speed) method, and sending the *perceived* speed through speedOut ! p_speed. An acceleration setting consists in receiving the decided acceleration on accelIn ? d_accel, passing them on to Engine by calling accel ← accelActuator(d_accel) and sending it to the real vehicle through phyAccel ! accel.

```
CtrlEngine =
  phySpeed ? speed → speedSensor ! speed ? p_speed → speedOut ! p_speed → phyAck →
  accelIn ? d_accel → accelActuator ! d_accel ? accel → phyAccel ! accel → phyAck→ CtrlEngine
```

In our first model, speed and location perceptions are done before acceleration is applied. Now, with the separation of concerns introduced by the two components Location and Engine, it would be possible for location perception to be realised *after* an acceleration setting, for instance. In order to ensure this, CtrlEngine and CtrlLocation are synchronised through phyAck.

Verifying the new components. We successfully establish the consistency of (CtrlPhysical ‖ Vehicle), (CtrlEngine‖Engine) and (CtrlLocation ‖ Location) using B4Free and FDR2 by following the verification process presented in Sect. 2.3.

5.2   The Vehicle2 Assembly

Vehicle2 is defined as an assembly of the previously detailed components, synchronised on their common channels:

$$
\text{Vehicle2} = \left(\left\{ \begin{smallmatrix} \text{phyAccel,} \\ \text{phySpeed,} \\ \text{phyXpos} \end{smallmatrix} \right\} \Big\| \left(\begin{array}{c} (\text{CtrlEngine} \| \text{Engine}) \underset{\{|\text{phyAck}|\}}{\|} (\text{CtrlLocation} \| \text{Location}) \\ \\ (\text{CtrlPhysical}\|\text{Vehicle}) \end{array} \right) \right) \left[\begin{smallmatrix} \text{accelIn} \leftarrow \text{setAccel,} \\ \text{xposOut} \leftarrow \text{getXpos,} \\ \text{speedOut} \leftarrow \text{getSpeed} \end{smallmatrix} \right]
$$

Some channels have to be renamed to match those of the CtrlVehicle controller.

Verifying that Vehicle2 refines Vehicle. The goal of the Vehicle component evolution is to retain the initial architecture, i.e. we want to replace Vehicle into Cristal (mode) by Vehicle2 and prove that the already established properties are still valid, among which:

- the deadlock-freedom of the whole vehicle (Sect. 3.1);
- the fact that perceptions and actions alternate (Sect. 3.3);
- the deadlock-freedom of the whole convoy (Sect. 4).

Hence Vehicle2 must externally show the same traces as Vehicle and should not introduce new deadlocks. Proving that Vehicle2 refines Vehicle in the stable failures semantics suffices for ensuring that. Indeed, the stable failures refinement preserves safety properties (because it implies trace refinement), liveness properties and deadlock-freedom [10].

We unfortunately face a problem. Vehicle is a B model and Vehicle2 is an assembly of CSP controllers and B machines: there is no manner to check this kind of refinement. To solve this problem, our proposal consists in lifting the refinement checking to an upper level, where refinement is well-defined. In a nutshell, we thus have to prove that the (CtrlVehicle ∥ Vehicle) component is refined by the (CtrlVehicle ∥ Vehicle2) component in the stable failures model which is denoted by:

$$(\text{CtrlVehicle} \| \text{Vehicle}) \setminus \alpha(\text{Vehicle}) \sqsubseteq_{SF} (\text{CtrlVehicle} \| \text{Vehicle2}) \setminus \alpha(\text{Vehicle})$$

where $\alpha(\text{Vehicle}) \equiv \{|\text{getXpos}, \text{getSpeed}, \text{setAccel}|\}$.

PROOF:
ASSUME:

$$\text{CtrlVehicle2} = \left(\underset{\left\{\left|\begin{array}{c}\text{phyAccel,}\\\text{phySpeed,}\\\text{phyXpos}\end{array}\right|\right\}}{\|} \left(\underset{\{|\text{phyAckl}|\}}{\left(\text{CtrlEngine} \| \text{CtrlLocation}\right)} \atop \text{CtrlPhysical} \right) \right) \left[\left[\begin{array}{c}\text{accelIn} \leftarrow \text{setAccel,}\\\text{xposOut} \leftarrow \text{getXpos,}\\\text{speedOut} \leftarrow \text{getSpeed}\end{array}\right]\right]$$

(CtrlVehicle2 is the CSP part of Vehicle2)

1. (CtrlVehicle ∥ Vehicle) \ α(Vehicle) $\sqsubseteq_{SF}$ (CtrlVehicle ∥ CtrlVehicle2) \ α(Vehicle)
PROOF:
    1.1. CtrlVehicle \ α(Vehicle) $\sqsubseteq_{SF}$ (CtrlVehicle ∥ CtrlVehicle2) \ α(Vehicle)
                     (verification carried out by FDR2 – 6 states and 7 transitions)
    1.2. (CtrlVehicle ∥ Vehicle) \ α(Vehicle) $\sqsubseteq_{SF}$ CtrlVehicle \ α(Vehicle)
    PROOF:
        1.2.1. traces(((CtrlVehicle ∥ Vehicle) \ α(Vehicle) = traces(CtrlVehicle \ α(Vehicle))
                       (definition of traces, hiding of internal channels)
        1.2.2. failures((CtrlVehicle ∥ Vehicle) \ α(Vehicle)) = failures(CtrlVehicle \ α(Vehicle)) = ∅
             (deadlock-freedom verified by FDR2 – 32 states and 48 transitions, [3, theorem 5.9])
        1.2.3. (CtrlVehicle ∥ Vehicle) \ α(Vehicle) $\sqsubseteq_{SF}$ CtrlVehicle \ α(Vehicle)
                       (1.2.1, 1.2.2, definition of $\sqsubseteq_{SF}$)
    ☐
    1.3. (CtrlVehicle ∥ Vehicle) \ α(Vehicle) $\sqsubseteq_{SF}$ (CtrlVehicle ∥ CtrlVehicle2) \ α(Vehicle)
                       (1.1, 1.2, transitivity of $\sqsubseteq_{SF}$)
    ☐
2. (CtrlVehicle ∥ CtrlVehicle2) \ α(Vehicle) $\sqsubseteq_{SF}$ (CtrlVehicle ∥ Vehicle2) \ α(Vehicle)
PROOF:
    2.1. CtrlVehicle2 \ α(Vehicle) $\sqsubseteq_{SF}$ Vehicle2 \ α(Vehicle)
                       ([3, corollary 8.7] applied to controllers of Vehicle2)
    2.2. (CtrlVehicle ∥ CtrlVehicle2) \ α(Vehicle) $\sqsubseteq_{SF}$ (CtrlVehicle ∥ Vehicle2) \ α(Vehicle)
                       (2.1, monotonicity of $\sqsubseteq_{SF}$ w.r.t. ∥ and hiding)
    ☐
3. (CtrlVehicle ∥ Vehicle) \ α(Vehicle) $\sqsubseteq_{SF}$ (CtrlVehicle ∥ Vehicle2) \ α(Vehicle)
                       (1, 2, transitivity of $\sqsubseteq_{SF}$)

As (CtrlVehicle ‖ Vehicle) \ α(Vehicle) $\sqsubseteq_{SF}$ (CtrlVehicle ‖ Vehicle2) \ α(Vehicle) is true, all the properties we wanted to preserve from Vehicle to Vehicle2 are still true: the deadlock-freedom of a vehicle, the deadlock-freedom of the whole convoy as well as the alternation of perceptions and actions. In conclusion, we can replace Vehicle by Vehicle2 without having to check the properties again.

6   Related Works

In addition to works on CSP‖B mentioned in Sect. 2, we would like to cite [15], where the authors present a formal framework for verifying distributed embedded systems. An embedded system is described as a set of concurrent real time functions which communicate through a network of interconnected switches involving messages queues and routing services. It presents an abstraction-based verification method which consists in abstracting the communication network by end-to-end timed channels. Proving a given safety property "requires then (1) to prove a set of proof obligations ensuring the correctness of the abstraction step (i.e. the end-to-end channels correctly abstract the network), and (2) to prove [this property] at the abstract level". The expected advantage of such a method lies on the ability to overcome the combinatorial explosion frequently met when verifying complex systems. This method is illustrated by an avionic case study.

As a comparison point, in [3] Schneider & Treharne illustrate their use of CSP‖B with a multi-lift system that can be seen as a distributed system using several instances of a lift, minus the fact that the interactions of the lifts are actually centralised in a dedicated dispatcher. Our goal is very similar, but in contrast to [3], we want to avoid relying on a centralised, or orchestrating, controller.

Similar works exist on structured development with the B method using decomposition, hence in a more "top-down" approach, and refinement. For instance, Bontron & Potet [16] propose a methodology for extracting components out of the enrichments brought by refinement. The extracted components can then be handled to reason about them so as to validate new properties or to detail them more. The interesting point is that their approach stays within the B method framework: this means that the modelling of component communication and its properties has to be done by using the B notation, which can quickly get more cumbersome than an ad-hoc formalism like CSP. Abrial [17] introduces the notion of decomposition of an event system: components are obtained by splitting the specification in the chain of refinements into several specifications expressing different views or concerns about the model. Attiogbé [18] presents an approach dual to the one of Abrial: event systems can be composed with a new asynchronous parallel composition operator, which corresponds to bringing "bottom-up" construction to event systems. In [19], Bellegarde & al. [19] propose a "bottom-up" approach based on synchronisation conditions expressed on the guards of the events. The spirit of the resulting formalism is close to that of CSP‖B. Unfortunately, it does not seem to support message passing for communication modelling.

As stated in the introduction, this paper is an evolution of [7]. More precisely, in addition to a more detailed explanation of the specification process we followed with our model, we exploited the renamings of channels so as to give a fitter way for instanciating

and assembling several Cristals. We also illustrated a novel use of CSP‖B theoretical results: Indeed, theorems about refinement or equivalences of CSP‖B components are usually used for easing verification by allowing one to re-express a CSP controller into a simpler one. We used these results to show how to insert new behaviours by splitting up a controller/machine compound without breaking previously verified properties.

7   Conclusion

With the development of a real case study, a platoon of a new type of urban vehicles in the context of the industrial CRISTAL project, we address the importance of formal methods and their utility for highly practical applications. Our contribution mainly concerns methodological aspects for applying known results and tool supports (FDR2 and B4Free). We show how to use the CSP‖B framework to compositionally validate the specifications and prove properties of component-based systems, with a precise verification process to ensure the consistency of a controlled machine ($P\|M$) and its generalisation to a collection of controlled machines $\|_{E_i}^i (Pi \| Mi)$.

These formal specifications form another contribution of this work. Indeed, having formal CSP‖B specifications help – by establishing refinement relations – to prevent incompatibility among various implementations. Moreover, writing formal specifications help in designing a way to manage the multi-level assembly.

This work points out the main drawback of the CSP‖B approach: at the interface between the both models, CLIs and augmented B machines corresponding to CSP controllers are not automatically generated. However, this task requires a high expertise level. In our opinion, the user should be able to conduct all the verification steps automatically. Automation of these verification steps could be a direction for future work.

On the case-study side, to go further, we are currently studying new properties such as the non-collision, the non-unhooking and the non-oscillation: which ones are expressible with CSP‖B, which ones are tractable and verifiable? This particular perspective is related to a similar work by the authors of CSP‖B dealing with another kind of multi-agent system in [14]. So far our use of CSP‖B for the platooning model reaches similar conclusions. This nonetheless raises the question of which impact the expression of more complex emerging properties does have on the model.

Further model development requires checking other refinement relations. It also includes evolutions in order to study what happens when a Cristal joins or leaves the platoon, and which communication protocols must be obeyed to do so in a safe manner. We also plan to take into account the lateral control and/or perturbations such as pedestrians or other vehicles.

References

1. Ferber, J., Muller, J.P.: Influences and reaction: a model of situated multiagent systems. In: 2nd Int. Conf. on Multi-agent Systems, pp. 72–79 (1996)
2. Daviet, P., Parent, M.: Longitudinal and lateral servoing of vehicles in a platoon. In: Proceeding of the IEEE Intelligent Vehicles Symposium, pp. 41–46 (1996)
3. Schneider, S.A., Treharne, H.E.: CSP theorems for communicating B machines. In: Formal Aspects of Computing, Special issue of IFM 2004 (2005)

4. Abrial, J.R.: The B Book. Cambridge University Press, Cambridge (1996)
5. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall, Englewood Cliffs (1985)
6. Simonin, O., Lanoix, A., Colin, S., Scheuer, A., Charpillet, F.: Generic Expression in B of the Influence/Reaction Model: Specifying and Verifying Situated Multi-Agent Systems. INRIA Research Report 6304, INRIA (2007)
7. Colin, S., Lanoix, A., Kouchnarenko, O., Souquières, J.: Towards Validating a Platoon of Cristal Vehicles using CSP‖B. In: Meseguer, J., Roşu, G. (eds.) AMAST 2008. LNCS, vol. 5140, pp. 139–144. Springer, Heidelberg (2008)
8. Lanoix, A., Hatebur, D., Heisel, M., Souquières, J.: Enhancing dependability of component-based systems. In: Abdennahder, N., Kordon, F. (eds.) Ada-Europe 2007. LNCS, vol. 4498, pp. 41–54. Springer, Heidelberg (2007)
9. Badeau, F., Amelot, A.: Using B as a high level programming language in an industrial project: Roissy VAL. In: Treharne, H., King, S., Henson, M.C., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 334–354. Springer, Heidelberg (2005)
10. Roscoe, A.W.: The theory and Practice of Concurrency. Prentice Hall, Englewood Cliffs (1997)
11. Treharne, H., Schneider, S.: Using a Process Algebra to Control B OPERATIONS. In: 1st International Conference on Integrated Formal Methods (IFM 1999), pp. 437–457. Springer, New York (1999)
12. Schneider, S., Treharne, H.: Communicating B Machines. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) B 2002 and ZB 2002. LNCS, vol. 2272, pp. 416–435. Springer, Heidelberg (2002)
13. Evans, N., Treharne, H.E.: Investigating a file transfer protocol using CSP and B. Software and Systems Modelling Journal 4, 258–276 (2005)
14. Schneider, S., Cavalcanti, A., Treharne, H., Woodcock, J.: A layered behavioural model of platelets. In: 11th IEEE International Conference on Engieerging of Complex Computer Systems, ICECCS (2006)
15. Carcenac, F., Boniol, F.: A formal framework for verifying distributed embedded systems based on abstraction methods. Int. J. Softw. Tools Technol. Transf. 8(6), 471–484 (2006)
16. Bontron, P., Potet, M.-L.: Automatic Construction of Validated B Components from Structured Developments. In: Bowen, J.P., Dunne, S., Galloway, A., King, S. (eds.) B 2000, ZUM 2000, and ZB 2000. LNCS, vol. 1878, pp. 127–147. Springer, Heidelberg (2000)
17. Abrial, J.R.: Discrete System Models, Version 1.1 (2002)
18. Attiogbé, C.: Communicating B Abstract Systems, Research Report RR-IRIN 02.08 (2002) (updated July 2003)
19. Bellegarde, F., Julliand, J., Kouchnarenko, O.: Synchronized parallel composition of event systems in B. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) B 2002 and ZB 2002. LNCS, vol. 2272, pp. 436–457. Springer, Heidelberg (2002)

Formal Verification of the Implementability of Timing Requirements

Xiayong Hu, Mark Lawford⋆, and Alan Wassyng⋆

Software Quality Research Laboratory,
Department of Computing and Software McMaster University,
Hamilton, Canada L8S 4K1
huxy@mcmaster.ca, lawford@mcmaster.ca, wassyng@mcmaster.ca

Abstract. There has been relatively little work on the implementability of timing requirements. We have previously provided definitions of fundamental timing operators that explicitly considered tolerances on property durations and intersample jitter. In this work we identify three environmental assumptions and compare the implementability of a *Held_For* operator in each of them, formalizing this analysis in PVS. We show how to design a software component that implements the *Held_For* operator and then verify it in PVS. This pre-verified component is then used to guide the design of more complex components and to decompose their design verification into simple inductive proofs as demonstrated through the implementation of a timing requirement for an example application.

1 Introduction

Specifying, implementing and verifying real-time requirements for embedded software systems can be a difficult and time consuming task. Hence real-time systems have become an active area of research in the formal methods community. The extensive survey of formal methods for the specification and verification of real-time systems in [1] contains references to over 200 publications. The overwhelming majority of the cited works are dedicated to the specification and validation of real-time requirements. Despite this intensity of research, relatively little work has been done on formally modeling timing tolerances.

Implicit in many of the formal models of timing requirements is the assumption that the real-time system implementing the timing requirements continuously monitors its inputs and can instantaneously react to the occurrence of an "event" (a significant change in the inputs). Due to their clock driven nature, computer control systems must typically sample some set of inputs and then update a set of outputs. Models that consider the sampling required for a computer controlled implementation of system requirements will often make the simplifying assumption that all samples are uniformly spaced and sufficiently fast to guarantee system response.

⋆ Supported by the Natural Sciences and Engineering Research Council of Canada.

Practical implementations have to worry about sampling rates, schedulability, computation time, latency, and jitter, all of which involve tolerances in some form when interfacing a physical plant and a software control system.

Motivated by our work on the Darlington Nuclear Generating Station Shutdown Systems software redesign project [2] and the difficulties and effort involved with the verification of timing requirements on that project, we began studying timing requirements with tolerances. In [3] we justified the use of several different types of tolerances that must be fully specified at the requirements level in order to properly deal with the timing tolerances that are inherent in the system implementation. These included tolerances on *functional timing requirements* (FTRs), and tolerances on *performance timing requirements* (PTRs) that allow for deviation from the idealized behaviour specified by the requirements models. By modeling these requirements, we presented *Implementability Results* which allow some timing requirements to be verifiably implemented at a significantly lower CPU bandwidth than normally assumed.

In this paper we investigate different environmental timing assumptions and present the implementability results for each of them. This can provide detailed answers to questions that are of interest to real-time system engineers. For example, nowadays, with cheap, high performance chips, more and more industry implementations take the "easy" approach, which is to use chips with high sampling rates to achieve the PTR. An obvious question to ask is: *"Is it always necessary to sample at fast sampling rates and is it safe to assume that sampling faster is the best way to implement the system?"* Another important question is: *"The timing environment has been changed, how do I know my implementation will still work for the new timing environmental assumption?"*

In order to formally provide answers to the above questions, we refined the model and formalized the analysis of the *Held_For* operator of [3] in the PVS[1] theorem prover. *Held_For* is an operator that describes "sustained behaviour"' with tolerances on the timing duration. For example, $(signal \geq setpoint)$ *Held_For* $(300\pm50ms)$ specifies that the result shall be *true* if $(signal \geq setpoint)$ is *true* for $(300\pm50ms)$. Now consider an environmental timing assumption that the software "knows" the exact timing of the sample instances. For this assumption, we provide a full formal proof of necessary and sufficient conditions for when it is possible to construct a discrete implementation of such a requirement, with duration d and tolerances of $[d - \delta L, d + \delta R]$. The implementation may use nonuniformly spaced samples, as long as the intersample spacing is bounded. As a result of the formalization in PVS, we discovered a missing boundary case in the original theorem statement of [3]. The implementability results under two new environments are also formally verified and presented. Also, by comparing the results in different environments we can predict the implementability of real-time timing requirements under new environmental assumptions.

We provide an intermediate representation of the *Held_For* requirement on the implementation's sampled signals, that we use to verify an implementation model of the *Held_For* requirement via a two step process. Once the implementation has

---

[1] Files available at http://www.cas.mcmaster.ca/~lawford/papers/FMICS08.html

been verified in PVS, we can verify the implementation of any specific requirement by simply instantiating the PVS theorems with appropriate values. Thus the verification is reduced to a standard untimed verification on the remainder of the requirement's functionality. We demonstrate this process with a simple *Delayed Trip System (DTS)* example in Section 5.

1.1   Related Work

Recent work addresses the issue of timing tolerances required to verify implementations of requirements modeled as timed automata with ASAP semantics [4,5]. De Wulf, *et al*, consider the case of implementing a continuous-time controller with a discrete-time system, assuming that there is a delay Δ associated with the controller's reaction to the environment. The implementation (e.g., C code on a BrinkOS platform) can be generated from the controller's automata.

The assumption of zero-time for computational action in the model language is impossible to ensure on the target platform in the implementation language [9]. Thus the predictable design approach introduced an ϵ-hypothesis to fill the gap between the physical domain and the software domain [10]. This ϵ-hypothesis requires the model and its realization to have the same observable execution sequence. Also, time deviations between activations of corresponding actions in the model and realization should be less than ϵ seconds.

The approaches with global tolerances (e.g, reaction delay parameter Δ in [4] and ϵ-hypothesis in [10]) define a global constraint as the constant upper bound of the delay during implementation. However, in most industrial requirements, it is typical that different timing requirements need different tolerances. Our approach replaces a very conservative global tolerance by including tolerances on each individual timing requirement. We have found that this may significantly reduce unnecessary load on the target platform. This is illustrated by the Delayed Trip System example in Section 5.

Most research based on the platform-independent idea will plug in another layer between the high level requirements and coding implementation, e.g, "program generation" in the Giotto approach [8] and in the POOSL model [9,10]. These approaches cannot determine the feasibility of an implementation on a target platform until the scheduling stage is finalized. In the case of the generation of an unimplementable result, the designer has to improve the hardware performance or relax the timing requirements, both of which are problematic. In our approach, the implementability of the system is predictable in the first stages of analysis, avoiding unnecessarily complex implementation and verification.

The remainder of this paper is organized as follows: Section 2 presents the preliminary work in [3] and a two step Systematic Design Verification (SDV) procedure. Section 3 introduces four different environmental assumptions and shows the relationship between the implementability results under each of them. An estimation approach is also provided, which allows one to estimate or even precisely predict the implementability of the timing properties in a new environment. Sections 4 and 5 present the approach to refine and implement the high level timing requirements (e.g., *Held_For*), through an *Implementation Template*

(e.g., a pre-verified timer design). An example is provided to demonstrate how the implementation and verification work has been efficiently reduced. Conclusions and future work are discussed in Section 6.

2   Preliminaries

2.1   Functional and Performance Timing Requirements

We differentiate between *Functional Timing Requirements (FTRs)* and *Performance Timing Requirements (PTRs)*. *FTRs* are timing requirements that are directly related to the required behavior of the application. *PTRs* are really timing tolerances that we specify so that the application does not have to adhere to the idealized behavior described by the requirements model. For more background on PTRs and FTRs, readers are referred to [3].

Definition of the *Held_For* operator with tolerance. *Held_For* is a common FTR which specifies a condition must be sustained over a particular time duration. A formal definition of the *Held_For* operator was specified in [3] as shown in Fig. 1.

(Condition) *Held_For* (d: $\mathbb{R}^{>0}$, δL, $\delta R : \mathbb{R}^{\geq 0}$) :bool
Initially: duration = any value in $[d - \delta L, d + \delta R]$, Event_start_time$_{-1}$ = 0,
Condition$_{-1}$ = *FALSE*

	duration	Event_start_time
(Condition = *TRUE*) & (Condition$_{-1}$ = *FALSE*)	Any value in $[d - \delta L, d + \delta R]$	t_{now}
(Condition = *FALSE*) OR (Condition$_{-1}$ = *TRUE*)	No Change	No Change

		Held_For
Condition = *TRUE*	$t_{now}-$ Event_start_time$\geq$ duration	*TRUE*
	$t_{now}-$ Event_start_time$<$ duration	*FALSE*
Condition = *FALSE*		*FALSE*

Fig. 1. Formal Definition of "(Condition) *Held_For* (d, δL, δR)"

There are a number of important points to emphasize. i) Duration is measured from when the event started in the physical domain. It does not make sense to define timing requirements with reference to when events are detected. ii) Many different implementations are valid. The behavior between $[d - \delta L, d + \delta R]$ is not deterministic. iii) Even though we have introduced tolerances into the requirement, *Held_For* is a FTR and still describes idealized behavior understood within the constraints of the requirements model. For instance, it does not take into account that processing time is not infinitely small, and it makes no reference to how often the application samples the values of the sensor.

As one of the PTRs introduced in [3], the Response Allowance (RA) for a controlled-monitored variable pair specifies an allowable processing delay. The RA is measured from the time the event actually occurred in the physical domain, until the time the value of the controlled variable is generated and crosses the application boundary into the physical domain.

The Timing Resolution (TR) for a monitored-controlled variable pair, can be thought of as the minimum duration event involving those variables that must be detected by the software [3].

2.2   Requirements Refinement and SDV Procedure Overview

In this section we provide an overview of our verification process in a two step approach based on the Systematic Design Verification (SDV) procedure introduced in [11]. In the first step, a pseudo-SRS[2] is created and verified as a refinement of the high level requirements. In the second step, we verify that the Software Design Description (SDD) is in compliance with the requirements for the behavior as specified in the pseudo-SRS.

To ensure the pseudo-SRS is a correct refinement, we must verify the pseudo-SRS based on all the timing requirements that are specified in the high level requirements. Let *pseudo-REQ* and *REQ* denote the pseudo-SRS state transition function and the high level Software Requirements Specification, respectively. The proof obligation of our first verification step can be formalized as:

$$pseudo\text{-}REQ \subseteq REQ$$

We restrict the pseudo-SRS to be a functional refinement of the high level requirements. The second step of the verification process is the SDV procedure based on a modified 4 variable model [11,12].
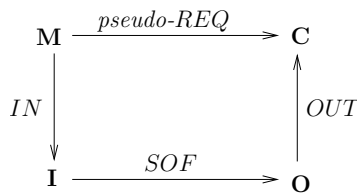

Fig. 2. Modified Commutative Diagram for 4 Variable Model

In Fig. 2, *pseudo-REQ* represents the pseudo-SRS state transition function mapping the monitored variables **M** to the controlled variables represented by **C**. *SOF* represents the SDD state transition function mapping the behavior of

---

the implementation input variables (represented by **I**) to the behavior of the software output variables (represented by **O**). The mapping IN models hardware functionality and relates the specification's monitored variables to the implementation's input variables. Similarly, the mapping OUT also models hardware functionality, and relates the implementation's output variables to the specification's controlled variables. The resulting proof obligation:

$$pseudo\text{-}REQ = OUT \circ SOF \circ IN \tag{1}$$

is illustrated by the solid lines in the commutative diagram of Fig. 2, which verifies the functional equivalence of the pseudo-SRS and SDD by comparing their respective one step transition functions [13]. Here $\circ$ is used to denote *functional composition*.

Through this two step SDV procedure, the high level requirements are connected with the low level implementation. In each of the steps, the verification can be formally conducted (e.g., by PVS). In later sections, we demonstrate this approach and provide the reader with an example.

2.3   Sample Instances

Let *Sample* be a possible sequence of sample times and $Sample(n)$ be the time of the $(n + 1)$-th sample ($n \in \mathbb{N}$). *Sample* is assumed to satisfy the bounded jitter constraint, where T_{min} and T_{max} are the minimum and maximum sample intervals over the complete range of sample intervals, respectively.

$$Sample(0) = 0 \wedge \forall n : Sample(n + 1) - Sample(n) \in [T_{min}, T_{max}].$$

We then also assume that the first sample point happens when $t = 0$, which is $Sample(0) = 0$. Note that when $T_{max} = T_{min}$, the problem is simplified to a fixed sample interval scenario, which is discussed in [14] for *Held_For* without tolerances.

In the example shown in Fig. 3, we assume $T_{min} = 10$ and $T_{max} = 20$. The first sample $Sample(0)$ occurs when $t = 0$, and the interval between any two consecutive sample points is in the range $[10, 20]$, e.g., $Sample(6) - Sample(5) = 85 - 70 = 15$. Further details on the example in Fig. 3 can be found in Section 3.1.
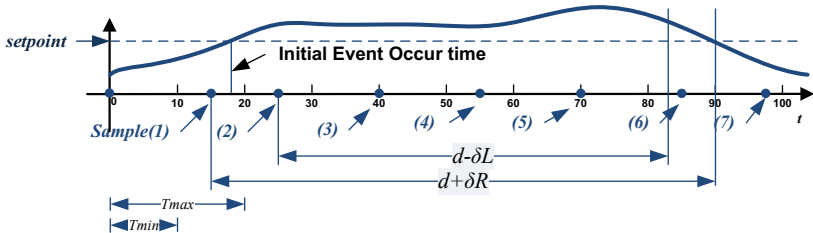

Fig. 3. An Example of Decision Points

3   Environmental Assumptions and Their Impact on Implementability

We have shown in [3] that the implementability results of the *Held_For* operator *with tolerance* are determined by the interaction between the FTRs (e.g., duration tolerances of the *Held_For* operator, δL and δR) and PTRs (the upper and lower bound of sample intervals: T_{min} and T_{max}).

In this section, we show feasibility analyses to answer the questions listed in Section 1. We first formally analyze the implementability results under three different environmental assumptions. By comparing the results across the environments, we develop an estimation approach based on the relationship between the environments. Finally we show that it is possible to estimate the range or even precisely predict the implementability results for a new environment.

3.1   Environmental Assumptions

We consider four different implementation environments which govern how we recognize a sustained event like the *Held_For* operator. They are the *Omniscient*, the *Perfect Clock*, the *Imperfect Clock* and the *No Clock* environments. We limit the scope of the analysis by assuming the implemented system will refresh the output at each sample point, which is a polling based rather than interrupt driven setting.

Perfect Clock: This environment provides the value of the condition only at sample instances and we know the exact timing of samples by using a perfect real valued clock. We can take actions (e.g., produce outputs) on the events only at sample times.

To properly state the environment conditions for implementation, we define the predicate *Feasible(d)* as a function of the sustained condition's nominal duration d and assume that the other parameters, $\delta L, \delta R, T_{min}$ and T_{max}, are fixed. The feasibility function of the *Perfect Clock* environment is defined as follows.

Definition 1. *Feasible_PerfectClock(d)* : $bool = \forall Sample : \forall n : \exists n_d :$
  $\forall (t | Sample(n) < t \leq Sample(n+1)) : d - \delta L \leq Sample(n_d) - t \leq d + \delta R$

In the function above t represents the event start time and n_d is the index of the sample where we will make our decision. It is known from earlier work [3] that if the system behavior is specified in the form of $(Condition)Held\_For$ $(d, \delta L, \delta R)$, the final decision as to whether *Held_For* generates *TRUE* or *FALSE* based on the sampled values, cannot be made until we are sure that a time period with length $d - \delta L$ has elapsed since the event occurred in the physical domain (i.e. $d - \delta L \leq Sample(n_d) - t$). The decision also must be made before $d + \delta R$ has elapsed since the event occurred.

To explain this, we introduce an input signal and the duration with tolerances to the example in Fig. 3. The initial event (when the *signal* goes above the *setpoint*) occurs between $Sample(1)$ and $Sample(2)$. It is not hard to find that all the sample points up to and including $Sample(5)$ are too early for us to determine

the value of the *Held_For* operator, and all the sample points from $Sample(7)$ onwards are too late for us to make the decision. Only when $t = Sample(6)$, is it the right sample for us to make the decision.

Omniscient: This environment provides full read access to the timing of the events that happen in the physical domain. In this environment, we know the exact time of each event when the condition becomes *TRUE* or *FALSE*. However, we can only take actions on these events at sample times. The difference in comparison to the *Perfect Clock* environment is the relaxation of the existence requirement for the decision point. For any t between $Sample(n)$ and $Sample(n+1)$, a different decision sample point $Sample(n_d)$ is acceptable. Putting this all together we can find the feasibility function in the *Omniscient* environment.

Definition 2. $Feasible\_Omniscient(d) : bool = \forall Sample : \forall n :$
$\forall(t | Sample(n) < t \le Sample(n+1)) : \exists n_d : d - \delta L \le Sample(n_d) - t \le d + \delta R$

Imperfect Clock: This environment is the same as in the *Perfect Clock* environment but with access to an imperfect clock (e.g. finite precision, bounded drift, etc). We leave as future work, the formalization of possible subcases that are associated with different imperfect clock assumptions. At the end of this section we will apply our estimation approach to this environment, which allows us to predict the implementability without having to perform complicated feasibility analyses and verification.

No Clock: Under this environmental assumption, our access to the timing of the events becomes very limited. The exact time of samples is not exposed even in the software domain. Our knowledge is only that each sample interval is between T_{min} and T_{max} and we also know the number of samples since the condition became *TRUE*. In this case we have no recourse in our implementation but to simply count the number of samples since we first detected the event. In this case we need a "count" value n_d that will work under any possible bounded sample spacing and actual time of occurrence of the event. Let $Sample(n+n_d)$ be the decision sample point, which is the n_dth sample point since $Sample(n)$. Then we have the definition of the feasibility function in the *No Clock* environment as follows:

Definition 3. $Feasible\_NoClock(d) : bool = \exists n_d : \forall Sample : \forall n :$
$\forall(t | Sample(n) < t \le Sample(n+1)) : d - \delta L \le Sample(n + n_d) - t \le d + \delta R$

3.2   Latest Environment Based Feasibility Analyses

Manual analysis in [3] shows that the only way that we can ensure the feasibility is to make sure that we have at least two sample points inside that interval $[d - \delta L, d + \delta R]$. After the recent PVS formal verification work, it turns out this is a necessary condition, but it is not sufficient. In this section, we will demonstrate how manual analysis with Fig. 4 provides a neat roadmap to guide us to the major results and how PVS formal verification captured a missing case in the manual analysis.
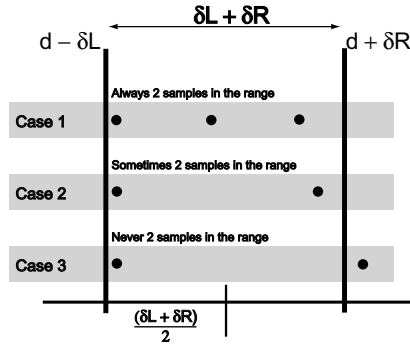
Fig. 4. Sample Points in the Duration Interval

Case 1: $0 < T_{max} \leq (\delta L + \delta R)/2$. In this case, we can guarantee there are always at least two sample points in the time interval $[d - \delta L, d + \delta R]$ in Fig. 4, based on which we can ensure the implementability of *Held_For*. Theorem 1 is proved for both the *Perfect Clock* and *Omniscient* environments.

Theorem 1. *Assume $T_{max} \leq (\delta L + \delta R)/2$. Then*

$$Feasible\_PerfectClock(d) \wedge Feasible\_Omniscient(d)$$

In the *No Clock* environment, implementability cannot be assumed. To understand this new result, shown in Theorem 2, we consider the two extreme cases, when the sample intervals are always T_{min} or T_{max}. For the T_{min} case, the first sample that is guaranteed to be on the right side of $d - \delta L$ is $\left\lceil \frac{d-\delta L}{T_{min}} \right\rceil + 1$. If $k = \left\lceil \frac{d-\delta L}{T_{min}} \right\rceil + 1$, then it is obvious that for feasibility in the T_{max} case, we must have that $k \times T_{max}$ cannot be to the right of $d + \delta R$.

Theorem 2. *Assume $T_{max} < (\delta L + \delta R)/2$. Then*

$$\left(\left(\left\lceil \frac{d - \delta L}{T_{min}} \right\rceil + 1 \right) \times T_{max} \leq d + \delta R \Leftrightarrow Feasible\_NoClock(d) \right)$$

Case 2: $(\delta L + \delta R)/2 < T_{max} \leq (\delta L + \delta R)$. It may happen that the hardware platform is not fast enough for us to arrange a sample interval that always works as defined in *Case 1*. Alternatively, we might be interested in operating at a slower sample rate in order to conserve power. In *Case 2*, two sample points will be in the time interval $[d - \delta L, d + \delta R]$ under certain conditions, which guides us to identify the necessary and sufficient conditions to implement *Held_For*.

Let $K_{min} = \left\lfloor \frac{d-\delta L}{T_{max}} \right\rfloor$ and $K_{max} = \left\lfloor \frac{d-\delta L}{T_{min}} \right\rfloor$, then the feasibility result for *Case 2* is given by the following theorem:

Theorem 3. *Assume $(\delta L + \delta R)/2 < T_{max} \leq \delta L + \delta R \wedge T_{min} \neq T_{max}$. Then*

$$T_{min} \geq \frac{d - \delta L}{K_{min} + 1} \wedge (K_{min} + 2) \times T_{max} \leq d + \delta R \Leftrightarrow Feasible\_PerfectClock(d)$$

We note that in [3] the conjunct $T_{min} \geq \frac{d-\delta L}{K_{min}+1}$ was incorrectly stated as $K_{min} = K_{max}$. During the course of formalizing the results of [3] in PVS, we identified a missing boundary condition which is also feasible under *Case 2*. The missing boundary case resulting in the new statement shown in Theorem 3 is when $K_{max} = K_{min}+1$ and $K_{max} \times T_{min} = d - \delta L$. This is when equality holds in the new conjunct (i.e. $T_{min} = \frac{d-\delta L}{K_{min}+1}$). While the latter condition restricts the application of this boundary case in practice, for the completeness of our results, this scenario needs to be considered to obtain the correct necessary and sufficient conditions for *Case 2* under the *Perfect Clock* and *No Clock* environmental assumptions.

Case 3: $T_{max} > (\delta L + \delta R)$. In *Case 3*, there is at most one sample point in the range of $[d - \delta L, d + \delta R]$ (shown in Fig. 4). Therefore, it is not possible to implement the *Held_For* operator.

Theorem 4. *Assume $T_{max} > \delta L + \delta R$. Then*

$$\neg Feasible\_PerfectClock(d) \wedge \neg Feasible\_NoClock(d) \wedge \neg Feasible\_Omniscient(d)$$

3.3   Comparing the Feasibility Results in Different Environments

In this section, we provide an overview and comparison of the feasibility results in the three environments: *Perfect Clock*, *No Clock* and *Omniscient*.

To compare the results we introduce the following two feasibility conditions.

$$Condition\ 1 : \left(\left\lceil \frac{d - \delta L}{T_{min}} \right\rceil + 1\right) \times T_{max} \leq d + \delta R$$

$$Condition\ 2 : T_{min} \geq \frac{d - \delta L}{K_{min} + 1} \wedge (K_{min} + 2) \times T_{max} \leq d + \delta R$$

Table 1 provides the comparison of the feasibility results and other important facts in each environment. The column headings of the table are *Environments*, *Case 1*, *Case 2*, *Case 3*, *Event Visibility* and *Clock Readable*. The *Environments* column lists the environments. The *Imperfect Clock* case will be discussed at the end of this section. Columns *Case 1-3* list the necessary and sufficient conditions of the feasibility function for that case in the different environments. *Event Visibility* specifies in which domain we can access the timing of any physical event. The final column of the table is *Clock Readable*, indicating whether the clock is accessible in the environment. Taking the *Perfect Clock* environment as an example, here is the approach we used to fill in the values in this comparison table.

Table 1. Comparison of Implementability Results

Environments	Case 1	Case 2	Case 3	Event Visibility	Clock Readable
Omniscient	TRUE	TRUE	FALSE	Physical Domain	YES
Perfect Clock	TRUE	Condition 2	FALSE	Software Domain	YES
Imperfect Clock	???	Condition 2	FALSE	Software Domain	YES
No Clock	Condition 1	Condition 2	FALSE	Software Domain	NO

We filled in *TRUE* for *Case 1* because we do not require any additional condition to attain feasibility for *Case 1*. For *Case 2* and *Case 3*, the values are *Condition 2* and *FALSE* respectively, based on Theorem 3 and Theorem 4. We set *Event Visibility* to "Software Domain" because we will not be able to observe any event in the physical domain until the next sample point occurs in the software domain. Based on our discussion in Section 3.1, the value for *Clock Readable* should be *YES*.

We can now discuss the comparisons contained in Table 1. At one extreme, the *Omniscient* environment assumes that the time of the event is instantaneously reported to the software domain and the controller can calculate and produce the output simultaneously. The idealization embodied by this assumption allows us to design the implementation of the *Held_For* operator in a simpler way than any practical capability will allow. In *Case 2*, this environment does not require any feasibility condition. On the other hand, the *No Clock* assumption completely forbids access to the clock during the implementation process, which increases the difficulty of the implementation. Therefore, even in *Case 1*, an implementation is not always feasible. In *Case 3*, the *Held_For* operator is not implementable under any of the three environmental assumptions.

Note that the difficulty of the implementation of the *Held_For* operator increases as we progress down Table 1. The following states this more formally.

Theorem 5. *Feasible_NoClock(d)* $\implies$ *Feasible_PerfectClock(d)* $\wedge$
$\qquad\qquad$ *Feasible_PerfectClock(d)* $\implies$ *Feasible_Omniscient(d)*

The relationship between the feasibility functions under different environmental assumptions determines the difficulty of implementing *Held_For* in those environments. For example, *Feasible_NoClock(d)* $\implies$ *Feasible_PerfectClock(d)*, so we find that for any of the *Cases 1-3*, the condition to implement *Held_For* is always equivalent or more restricted under the *No Clock* environmental assumption than under the *Perfect Clock* environmental assumption. Now we consider the possible *Imperfect Clock*s described in Section 3.1. The information available to the implementation in this environment falls in between the *Perfect Clock* and *No Clock* cases. Since the latter two scenarios have the same necessary and sufficient conditions in *Case 2*, we can conclude that *Condition 2* is necessary and sufficient for any imperfect clock environment! For *Case 1* the *Imperfect Clock* case shows *???* since the precise feasibility function may depend upon what clock imperfections are considered, but it should be no stronger than *Condition 1*.

4   Implementation of the Held_For Operator

In this section, we briefly describe an implementation of the *Held_For* operator *with tolerance* in the *Perfect Clock* environment and refer the reader to [17] for a more detailed account. We refine our model of time to a discrete time model that assumes arbitrarily small clock ticks, which allows us to apply a straightforward inductive proving approach to verify the implementation of the *Held_For* operator.

The *Held_For* operator defined in Fig. 1 specifies the tolerance on the duration of the sustained window, and it leads to indeterminism in the implementation of the system. In particular, if the *Condition* has been sustained for an interval that is in the range $[d - \delta L, d + \delta R)$, then the current value of the *Held_For* operator can be either *TRUE* or *FALSE*. Based on the first step in Section 2.2, we can refine the requirements of the *Held_For* operator to a deterministic subset of the high level requirements that matches our implementation's behaviour at the sampling points. This is done by defining the `Held_For_S` operator as follows:

```
Held_For_S(P,duration,Sample)(ne):bool=
  EXISTS(n0|Sample(ne)-Sample(n0) >= duration):
    FORALL (n: nat | n0 <= n AND n <= ne): P(Sample(n))
```

When this operator defined on sample indexes is lifted to the arbitrarily fast clock tick level of the requirements in the natural way, it can be shown to be a refinement of the original *Held_For* operator under an appropriate PTR assumption [17].

4.1   Timer Implementation of Held_For_S

Timer_S and TimerUpdate Functions. To implement the `Held_For_S` operator, we can design a timer that updates its value at every sample instance. In Fig. 5, the `Timer_S` PVS function updates its value through a `TimerUpdate` function, by passing the following information: the condition at both the current and last sample instances, the pre-set timeout value, the current value of the timer and the elapsed time since the last update of the timer.

Then the `TimerUpdate` function will update the timer by returning the latest value.

In our design, we pass the values of the current and last sample instances, `P(Sample(ne))` and `P(Sample(ne-1))`, to `TimerUpdate` as the first and second parameters, `CurrentPP` and `PreviousPP`. The `TimerUpdate` function will reset the `Timer` to 0 when any of them is *FALSE*. When both of them are *TRUE*, `TimerUpdate` will update the `Timer` function by adding the elapsed time (step) to the previous `Timer` value. If the previous value has exceed the `TimeOut` value, the `TimerUpdate` function will do nothing but return the previous value to avoid an eventual overflow error.

We can then verify that an appropriately defined predicate on the current input value and the PVS function `Timer_S` is an implementation of the `Held_For_S` function.

```
TimerUpdate(CurrentPP, PreviousPP, TimeOut, PreviousTimerValue, step): tick =
TABLE
                              %+---------------------------+----------------------------++
                              |[ PreviousTimerValue < TimeOut| PreviousTimerValue >= TimeOut]|
%---------------------------+---------------------------+----------------------------++
| CurrentPP AND PreviousPP  | PreviousTimerValue + step | PreviousTimerValue          ||
%---------------------------+---------------------------+----------------------------++
|NOT(CurrentPP AND PreviousPP)| 0                       | 0                           ||
%---------------------------+---------------------------+----------------------------++
ENDTABLE

Timer_S(P, Sample, TimeOut)(ne): RECURSIVE tick =
 TABLE
 %--------+-------------------------------------------------------------------------++
 | ne = 0 | TimerUpdate(P(Sample(ne)), FALSE, TimeOut, 0, 0)
 %--------+-------------------------------------------------------------------------++
 | ne > 0 | TimerUpdate(P(Sample(ne)), P(Sample(ne - 1)),
           TimeOut, Timer_S(P, Sample, TimeOut)(ne - 1), Sample(ne) - Sample(ne - 1)) ||
 %--------+-------------------------------------------------------------------------++
 ENDTABLE
MEASURE ne
```

Fig. 5. TimerUpdate and Timer_S Functions

```
TimerGeneral_S1: THEOREM Held_For_S(P, timeout - delta_L, Sample)(n+1)
   IFF (P(Sample(n + 1)) AND Timer_S(P,Sample,timeout-delta_L)(n) +
                        Sample(n+1)-Sample(n)>=timeout-delta_L)
```

The `Timer_S` design yields a relatively easy implementation of the `Held_For_S` operator. Other equivalent implementations can be defined, and, in practice, there could be many similar implementations using the same design pattern. Our objective here is not to create a strict formula for software designers to follow, but to provide a generic design pattern like `Timer_S`, so that designers can customize the `Timer_S` design based on different situations. In the next section, we present the DTS example. By utilizing the general theorem `TimerGeneral_S1`, a large amount of the verification work is saved by proving the equivalence of the customized timer implementation to the original `Timer_S` implementation.

5   Example: Delayed Trip System with Tolerances

We now revisit the Delayed Trip System (DTS) [16] which was implemented and verified in [14] - but without explicitly considering timing tolerances. A modified Software Requirement Specification (SRS) for the DTS with explicit tolerances is shown at the top of Fig. 6. In this version, the requirements are specified with the *Held_For* operator *with tolerances*. If the condition *PP* has held for *timeout1*, the relay must be open. When the power drops below *PT* or the pressure becomes lower than *DSP*, the relay must not close until after another time period of *timeout2*. The bottom portion of Fig. 6 presents part of the PVS for the Software Design Description (SDD) of the DTS, the function `RelayUpdate`. With the help of the pre-verified `TimerGeneral_S1` theorem, we

Condition	Result
	relay
(PP) *Held_For*$(timeout1, \delta L1, \delta R1)$	*TRUE*
$(\neg \, [(PP) \; Held\_For \; (timeout1, \, \delta L1, \, \delta R1)]) \; Held\_For \; (timeout2, \, \delta L2, \, \delta R2)$	*FALSE*
$\neg \, (PP) \; Held\_For(timeout1, \delta L1, \delta R1) \, \wedge$ $\neg \, (\neg \, [(PP) \; Held\_For \; (timeout1, \, \delta L1, \, \delta R1)]) \; Held\_For \; (timeout2, \, \delta L2, \, \delta R2)$	No Change

$$\text{where } PP(t) = Power(t) \geq PT \wedge Pressure(t) \geq DSP$$

```
SDD_State: TYPE =
     [# Relay: Relay_State, Timer1: tick, Timer2: tick,
        PreviousInput1: bool, PreviousInput2: bool #]


RelayUpdate(timeout1, timeout2, CurrentPP, S, step): Relay_State =
TABLE
%------------------------------------------------------------------+-------++
| CurrentPP&(Timer1(S)+step>=timeout1)                             |OPEN   ||
%------------------------------------------------------------------+-------++
| NOT(CurrentPP&Timer1(S)+step>=timeout1)&Timer2(S)+step>=timeout2|CLOSED||
%------------------------------------------------------------------+-------++
| NOT(CurrentPP&Timer1(S)+step>=timeout1)&
        NOT (Timer2(S)+step>=timeout2)                           |Relay(S)||
%------------------------------------------------------------------+-------++
ENDTABLE
```

Fig. 6. The SRS (top) and SDD (bottom) for the DTS with Tolerances

can clearly identify the functional behaviour of the DTS mapping from SRS to SDD tables. For example, the *(PP)Held_For(timeout1,...)* in the first row is implemented by `Timer1` and current condition `CurrentPP` (as shown in the first lines of both tables). Similarly, `Timer2` (with its current condition) implements the *Held_For* (with *timeout2*), as shown in the seconds line of both tables.

Both of the `Timer` functions call the `TimerUpdate` function to update themselves. Function `RelayUpdate` updates the current output of the relay, based on the *timeout1* and *timeout2*, the current condition `PP` and `step` (as introduced for `TimerUpdate`). Here the variable `S` is of record type `SDD_State`. It stores the system state - the status of the `Relay`, values of `Timer1` and `Timer2` and the input conditions of each timer at the previous sample time, `PreviousInput1` and `PreviousInput2`. These last two fields are passed to the `TimerUpdate` function applications as `PreviousPP` parameters, in order to determine whether the timer should add a step increment or perform a reset.

Note that the *Held_For* operator with duration *timeout1* has tolerance settings $\delta L1$ and $\delta R1$ and another *Held_For* operator with duration *timeout2* has its own tolerance settings $\delta L2$ and $\delta R2$. This may better fit a real-world engineering specification, where *timeout1* and *timeout2* may differ by more than an order of magnitude. In this case it typically would not make sense for the timing requirements to share a single global tolerance. For example, we may want *timeout1*=300±2 seconds and *timeout2*=2±0.1 seconds. This provides an example

in which the requirements of the system do not fit into a global tolerance model (e.g, the reaction delay parameter Δ of the Almost ASAP semantics in [4] and ϵ-hypothesis in [10]). Instead of performing a scheduling check in the final stage [8,10], our approach can also determine whether further work on an implementation is worthwhile as soon as the timing requirements have been specified (based on the *feasibility analyses* discussed in Section 3.2).

We have shown (in the complete PVS source code) how to reuse this result to reduce the verification work of the customized timer components, through the `TimerGeneral_S1` theorem. This general theorem requires more than 16 lemmas and 600 PVS commands to complete. This one time effort can benefit other going forward. In the DTS example, 51% of the total PVS prover commands used in the verification are eliminated by repeated instantiations of this theorem. More details on the DTS example are available in [17].

6   Summary

In this paper, we expand the scope of our *feasibility analyses* to explicitly include environmental assumptions. Our latest results show that implementability of timing requirements (e.g., *Held_For* operator) is determined by both the implementation environment and the interaction of the timing requirements. Now we are in a position to answer the questions we proposed in Section 1.

"Is it always necessary to sample at fast sampling rates and is it safe to assume that sampling faster is the best way to implement the system?" The latest feasibility analyses show that sampling faster is not always the only option and also not always the correct choice in implementing real-time systems. The feasibility analyses show that it is still possible to implement the *Held_For* operator when $T_{max} > (\delta L + \delta R)/2$, which provides an alternative solution to the designer of real-time systems, when coping with hardware limitations. On the other hand, the results of *Case 1* in the *No Clock* environment show that it is not always safe to assume implementability when $T_{max} \leq (\delta L + \delta R)/2$.

"The timing environment has been changed, how do I know my implementation will still work for the new timing environmental assumption?" This could be easily determined since we have the *implementability results* for different environments. Further, if the target environment is altered for a particular timing requirement, the relationships between feasibility functions under different environmental assumptions can help us estimate the implementability results for the new environment.

We have introduced a pre-verified *Implementation Template*, which benefits real-time software in two respects. First, it allows domain experts to specify different tolerances for each functional timing requirement, instead of a global tolerance for the timing behavior on the target system. Second, it helps to simplify and reduce the effort required in both the implementation and verification stages.

References

1. Wang, F.: Formal verification of timed systems: A survey and perspective. Proceedings of the IEEE 92(8), 1283–1307 (2004)
2. Wassyng, A., Lawford, M.: Lessons learned from a successful implementation of formal methods in an industrial project. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 133–153. Springer, Heidelberg (2003)
3. Wassyng, A., Lawford, M., Hu, X.: Timing tolerances in safety-critical software. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 157–172. Springer, Heidelberg (2005)
4. De Wulf, M., Doyen, L., Raskin, J.F.: Almost asap semantics: From timed models to timed implementations. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 296–310. Springer, Heidelberg (2004)
5. De Wulf, M., Doyen, L., Markey, N., Raskin, J.F.: Robustness and implementability of timed automata. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS 2004 and FTRTFT 2004. LNCS, vol. 3253, pp. 118–133. Springer, Heidelberg (2004)
6. Abadi, M., Lamport, L.: An old-fashioned recipe for real time. ACM Transactions on Programming Languages and Systems 16(5), 1543–1571 (1994)
7. Shankar, N.: Verification of real-time systems using PVS. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 280–291. Springer, Heidelberg (1993)
8. Henzinger, T.A., Kirsch, C.M., Sanvido, M.A., Pree, W.: A Giotto-based helicopter control system. In: Sangiovanni-Vincentelli, A.L., Sifakis, J. (eds.) EMSOFT 2002. LNCS, vol. 2491, pp. 46–60. Springer, Heidelberg (2002)
9. Florescu, O., Voeten, J., Huang, J., Corporaal, H.: Error estimation in model-driven development for real-time software. In: Forum on specification and Design Languages, pp. 228–239 (2004)
10. Huang, J., Voeten, J., Florescu, O., van der Putten, P., Corporaal, H.: Predictability in real-time system development. In: Advances in Design and Specification Languages for SoCs, pp. 123–139. Kluwer Academic Publishers, Dordrecht (2005)
11. Lawford, M., Hu, X.: Right on time: Pre-verified software components for constructuion of real-time systems. Technical Report 8, Software Quality Research Lab, McMaster University, Hamilton, ON, Canada (2002)
12. Parnas, D.L., Madey, J.: Functional documents for computer systems. Science of Computer Programming 25(1), 41–61 (1995)
13. Lawford, M., McDougall, J., Froebel, P., Moum, G.: Practical application of functional and relational methods for the specification and verification of safety critical software. In: Rus, T. (ed.) AMAST 2000. LNCS, vol. 1816, pp. 73–88. Springer, Heidelberg (2000)
14. Hu, X.: Proving real-time properties of embedded software systems. M.Sc., Dept. of Computing and Software, McMaster University, Hamilton, ON, Canada (2002)
15. Website, N.L.P.L.O.:
    http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html
16. Lawford, M., Wonham, W.: Equivalence preserving transformations of timed transition models. IEEE Trans. Automatic Control 40(7), 1167–1179 (1995)
17. Hu, X.: Proving Implementability of Timing Properties with Tolerance. Ph.D., Dept. of Computing and Software, McMaster University, Hamilton, ON, Canada (2008)

Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties⋆

Christian Colombo[1], Gordon J. Pace[1], and Gerardo Schneider[2]

[1] Department of Computer Science, University of Malta, Malta
[2] Department of Informatics – University of Oslo, Oslo, Norway
{ccol002,gordon.pace}@um.edu.mt, gerardo@ifi.uio.no

Abstract. Given the intractability of exhaustively verifying software, the use of runtime-verification, to verify single execution paths at runtime, is becoming popular. Although the use of runtime verification is increasing in industrial settings, various challenges still are to be faced to enable it to spread further. We present dynamic communicating automata with timers and events to describe properties of systems, implemented in LARVA, an event-based runtime verification tool for monitoring temporal and contextual properties of Java programs. The combination of timers with dynamic automata enables the straightforward expression of various properties, including replication of properties, as illustrated in the use of LARVA for the runtime monitoring of a real life case study — an online transaction system for credit card. The features of LARVA are also benchmarked and compared to a number of other runtime verification tools, to assess their respective strengths in property expressivity and overheads induced through monitoring.

1 Introduction

As software systems grow bigger and more complex, and as they influence our lives in more frequent and direct ways, the need for their validation similarly grows. Over the past decades, program validation has become an increasingly important and active research area striving towards certified code with the ultimate holy grail of providing a guarantee of the absence of errors from a given system. Though testing and simulation have been the most widely-used techniques in industry, formal methods have started playing a more important role in program validation. Static analysis and model checking techniques have been improved, but still require expertise to apply on large complex systems, and usually fail to scale up to real-life systems. To address the problem of intractability, an alternative approach which has been developed is that of runtime verification, in which the desired properties are only checked at runtime on the active execution path. Properties are written in a formal logic and then transformed into a runtime monitor which is instrumented with the system to be monitored.

Subsequently, the runtime monitor observes the system while it is running, and triggers an appropriate response if a system property is violated. The main issue is a trade off between the expressivity of the logic and the overhead created on the monitored system.

Although the overhead induced through the monitors is undoubtedly crucial in certain application areas, indicating that the expressivity of the logic should be constrained so as to ensure effective monitors, the logic should be able to handle certain features to ensure its utility in a practical setting: (1) temporal aspects, including (i) consequentiality — e.g. 'authentication happens before data access', and (ii) real-time — e.g. 'a transaction takes no more than 30 seconds to execute'; (2) Contextual aspects — the possibility of monitoring objects either globally, grouped according to their container or individually, e.g. 'every account (in a given banking system) must belong to a registered user', or 'a bank transfer (corresponding to a given user) may only be performed without any charge if both accounts are registered (in European countries)'; (3) Exceptions — monitoring the exceptional cases in the execution of a program. Although the application domain clearly determines which type of properties one would need to monitor, ideally all the above should be expressible in the logic used for property specification.

In this paper we present dynamic communicating automata with timers and events to handle the above-mentioned aspects when describing properties of systems, and their implementation for runtime-verification in the tool LARVA. LARVA has been used in a real-life case study, consisting of a Java program which forms part of a software dealing with credit card transactions. Moreover, we compare our tool with a number of state-of-the-art runtime verification tools[1]: Java-MOP, Java-MaC, Hawk, ConSpec, and Lola.

The paper is organised as follows. In the next section we introduce dynamic automata with timers and events (DATEs) as the underlying property specification logic, and how monitors can be constructed directly from such structures, as implemented in the tool LARVA. In section 3, we then present the application of LARVA on a credit card transaction systems, and in section 4 we compare it to other tools.

2   Event-Based Runtime Monitoring

As argued in the introductory section, the expressivity of the logic in which to express properties is crucial. In this section we present a theory of communicating automata with events and timers used to express properties, and the construction of monitors from such properties — the basis of our tool LARVA.

2.1   Dynamic Automata with Events and Timers

The underlying logic we will use to define the specification properties of the system is based on communicating symbolic automata with timers, with

---

event-triggered transitions. Events can be visible system actions (such as method calls or exception handling), timer events, channel synchronisation (through which different automata may synchronise) or a combination thereof.

Definition 1. *Given a set* systemevent *of events which are generated by the underlying system and may be captured by the runtime monitor, a set of timer variables* timer, *and a set of* channels, *a composite* event *made up of system events, channel synchronisation, a timeout on a timer, a choice between two composite events (written $e_1 + e_2$) or the complement of a composite event (written $\overline{e}$), is syntactically defined as follows:*

$$\text{event} ::= \text{systemevent} \mid \text{channel}\,? \mid \text{timer } @\ \delta \mid \text{event} + \text{event} \mid \overline{\text{event}}$$

We say that a basic event x (which can be a system event, a channel synchronisation or a timeout event) will fire a composite event expression e (written $x \vDash e$) if either (i) x matches exactly event e; or (ii) $e = e_1 + e_2$ and either $x \vDash e_1$ or $x \vDash e_2$; or (iii) x is a system event and $e = \overline{e_1}$, and $x \nvDash e_1$.
The notion of firing of events can be extended to work on sets of events. Given a set of basic events X, a composite event e will fire (written $X \vDash e$) if either (i) e is a basic event expression, and for some event $x \in X$, $x \vDash e$; or (ii) $e = e_1 + e_2$ and either $X \vDash e_1$ or $X \vDash e_2$; or (iii) X contains at least one system event and $e = \overline{e_1}$, and for all $x \in X$, $x \nvDash e_1$.

The semantics of the complement of an event is constrained to fire when at least one system event fires, so as to avoid triggering whenever a timer event or channel communication happens, thus making such events to depend solely on the underlying system, hence increasing compositionality. This constraint can be relaxed without affecting the results in this paper.

Since we require real-time properties, we will introduce timers (ranging over non-negative real numbers), whose running may be paused or reset. The configuration of a finite set of timers determines the value and state of these timers.

Definition 2. *The* configuration of timers *(written $\mathcal{CT}$) is a function from timers to (i) the value time recorded on the timer; and (ii) the state of the timer (running or paused).* Timer resets, pauses *and* resumes *are functions from a timer's configuration to another changing only the value of one timer to zero (in the case of a reset), or the state of one timer (in the case of pause or resume). A* timer action *(written $\mathcal{TA}$) is the (functional) composition of a finite number of resets, pauses and resumes.*

Based on events and timers, we define *symbolic timed-automata* — similar to integration automata [1], but more expressive than Alur and Dill's Timed Automata [3] (since we enable reset, pause and resume actions on timers). Unlike integration automata, the automata we introduce have access to read and modify the underlying system state. In practice this can be used to access and modify variables making the transitions more symbolic and enumerative in nature.

Properties of a given system will be expressed as communicating timed-automata. These automata will have access to read and modify the state of the underlying system.

Definition 3. *A symbolic timed-automaton* running over a system with state *of type Θ is a quadruple $\langle Q, q_0, \rightarrow, B \rangle$ with set of states Q, initial state $q_0 \in Q$, transition relation $\rightarrow$, and bad states $B \subseteq Q$. Transitions will be labelled by (i) an event expression which triggers them; (ii) a condition on the system state and timer configuration which will enable the transition to be taken; (iii) a timer action to perform when taking the transition; (iv) a set of channels upon which to signal an event; and (v) code which may change the state of the underlying system:*

$$Q \times \text{event} \times (\Theta \times \mathcal{CT} \rightarrow \Delta = \Delta \times \mathcal{TA} \times 2^{\text{channel}} \times (\Theta \rightarrow \Theta) \times Q$$

We will assume that a total ordering $<$ exists on the transitions to ensure determinism.

The behaviour of an automaton M upon receiving a set of events consists of (i) choosing the highest priority transition which fires with the set of events and whose condition is satisfied; (ii) performing the transition (possibly triggering a new set of events); and (iii) repeating until no further events are generated, upon which the automaton waits for a system or timeout event.

Definition 4. *For a symbolic timed-automaton $M = \langle Q, q_0, \rightarrow, B \rangle$, we say that a set of system scheduled events X, system state $\theta \in \Theta$, timer configuration T and state q (in which M currently resides), performs a step to X', θ' and q', with timer update t' (written $(X, \theta, q) \Rightarrow_{t'}^{T} (X', \theta', q')$) if $q \notin B$ and $(q_1, e, c, t, O, f, q_2)$ be the largest (in terms of $<$) transition in $\rightarrow$ such that: (i) $q = q_1$; (ii) $X \vDash e$; (iii) $c(\theta, T)$, and the following hold: (i) $t' = t$; (ii) $q' = q_2$; (iii) $\theta' = f(\theta)$; (iv) $X' = O$. If no such transition exists, we write $(X, \theta, q) \Rightarrow_{id}^{T} (\emptyset, \theta, q)$.*
The notion of automata performing a step can be extended over to a vector of automata communicating via broadcast channels. Given a vector of n automata $\bar{M} = \langle M_1, M_2, \ldots M_n \rangle$, in states $\bar{q} = \langle q_1, q_2, \ldots q_n \rangle$ and with shared timers in state T, we write that $(X, \theta_0, \bar{q}) \Rightarrow_{t'}^{T} (X', \theta_n, \bar{q'})$ if (i) for each $1 \leq i \leq n$, $(X, \theta_{i-1}, q_i) \Rightarrow_{t_i}^{T} (X'_i, \theta_i, q'_i)$; (ii) $t' = t_{n-1} \circ t_{n-2} \circ \ldots \circ t_1$; (iii) $X' = X'_1 \cup X'_2 \cup \ldots \cup X'_n$.

Note that the order of execution is set by the order of the automata, once again to avoid non-determinism. Clearly, as in any programming with side-effects, the management of actions on the transitions must be carefully handled. Also note that the timer actions are accumulated so as to evaluate all conditions with the same initial timestamps.

The notions of symbolic timed-automata can be lifted to work on dynamic networks of symbolic timed-automata, in which we enable the creation of new automata during execution in a structured manner — referred to as Dynamic Automata with Events and Timers (DATE) in the rest of the paper.

Definition 5. *A DATE $\mathcal{M}$ is a pair $(\bar{M}_0, \nu)$ consisting of (i) an initial set of automata $\bar{M}_0$; and (ii) a set of automaton constructors ν of the form:*

$$\text{event} \times (\Theta \times \mathcal{CT} \rightarrow \Delta = \Delta \times (\Theta \times \mathcal{CT} \rightarrow Automaton)$$

Each triple $(e, c, a) \in \nu$ triggers upon the detection of event e, with the state and timer configurations satisfying condition c, and creating an automaton using function a. The triggered automata in time configuration T, with events X, in system state θ (written $tr(T, X, \theta)$) is defined to be:

$$tr(T, X, \theta) \stackrel{def}{=} \{a(\theta, T) \mid (e, c, a) \in \nu, \ X \vDash e, \ c(\theta, T)\}.$$

Finally, the events created by the transition can themselves trigger new transitions.

Definition 6. *The configuration of a DATE consists of (i) the state of the timers; (ii) the state of the underlying system; and (iii) the state of the currently running automata — a vector of a state for each automaton in the network.*
A DATE is said to perform a full-step *from configuration $(T, \theta, \bar{q})$ to configuration $(T', \theta', \bar{q}')$, upon receiving a set of system actions X, (written $(T, \theta, \bar{q}) \Mapsto_X (T', \theta', \bar{q}')$) if for some number n:*

$$(X_0, \theta_0, \bar{q}_0) \Rightarrow^T_{t_1} (X_1, \theta_1, \bar{q}_1) \Rightarrow^T_{t_2} \ldots (X_n, \theta_n, \bar{q}_n) \Rightarrow^T_{t_{n+1}} (\emptyset, \theta_{n+1}, q_{n+1}),$$

where: (i) $\bar{q}_0 = q$, $\theta = \theta_0$ and $\theta' = \theta_{n+1}$; (ii) the final state of the timer is updated according to the timer's accumulated actions: $T' = (t_{n+1} \circ t_n \circ \ldots t_1)(T)$; and (iii) the automata are updated as required by DATE triggers $q' = q_{n+1} \oplus \bigcup_i tr(T, X_i, \theta_i)$.
Such a step is called an accepting full-step, *if no bad states appear in the intermediate state vectors.*

Clearly, not all situations can perform a full-step — even a single automaton may create events on channels which trigger another transition indefinitely. To resolve the problem of livelock, we must ensure that there is no mutual dependency over the set of automata.

Definition 7. *The* output channels *of an automaton M, written $out(M)$, is the union of all output channels on the transitions in M. Similarly, the* input channels *of M, written $in(M)$, are the channels appearing on the event label of transitions in M. The* dependency relation *between channels for an automaton M, written $dep(M)$ is defined to be $in(M) \times out(M)$.*
A DATE structure $\bar{M}$ is said to be loop-free *if, for any channel c, $(c, c) \notin (\bigcup_i dep(M_i))^*$.*

The following result states that only loop-free automata can perform a full-step.

Proposition 1. *Given a loop-free collection of automata $\bar{M}$ in states $\bar{q}$, set of system events X and system state θ, there exist states $\bar{q}'$, system state θ' and timers T such that $(T, \theta, \bar{q}) \Mapsto_X (T', \theta', \bar{q}')$.*

Example 1. Consider a system where one needs to monitor the number of successive bad logins and the activity of a logged in user. By having access to *badlogin*, *goodlogin* and *interact* events, one can keep a successive bad-login counter and a clock to measure the time a user is inactive. Fig. 1(a) shows the property that
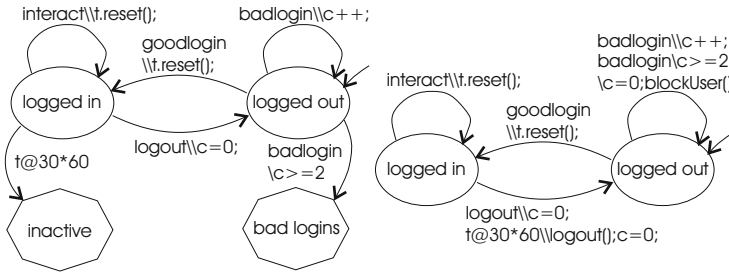
Fig. 1. (a) An automaton monitoring the bad logins occurring in a system; (b) The same automaton with recovery actions

allows for no more than two successive bad logins and 30 minutes of inactivity when logged in, expressed as a DATE. Upon the third bad login or 30 minutes of inactivity, the system reverts to a bad state. In the figure, transitions are labelled with events, conditions and actions, separated by a backslash. It is assumed that the bad login counter is initialised to zero.

Fig. 1(b) shows how actions can be used to remedy the situation when possible, instead of going to a bad state. For example, after too many bad logins, one can block the user from logging in for a period of time, and upon 30 minutes of inactivity when logged in, the user may be forced to logout.

2.2   Constructing Monitors from DATEs

These automata can be used to express properties, and can then be directly and automatically implemented as runtime-monitors for an underlying system. This transformation has been implemented in the system LARVA which embodies the implementation of these properties in an automatic manner.

System Events: As the underlying system events LARVA uses method calls, invocation of exception handlers, exception throws and object initialisations.

Actions: Actions which are performed on the system state upon each transition are essentially programs which can access code and data from the original program. Note that the model we have used performs the action without triggering any other transition directly. In LARVA this is emulated by ensuring that system events (eg. method calls) are masked from the automaton triggers when called as actions. Furthermore, as shown in the example from the previous section, unless mitigating a problem which arose, it is usually sufficient to constrain actions to access only data local to the automaton. For this purpose, LARVA provides the means to have code local to an automaton.

Dynamic triggers: Dynamic triggers are used in LARVA to enable multiple instances of a property. The property shown in Fig. 1 must be replicated for each user attempting to login to the system, in order to make it useful. For this purpose, LARVA enables properties to be replicated for multiple instances of an object — written `foreach object { property }`. Upon

capturing events in the property, the system checks whether it is a new object (using object equality, or a user provided mechanism) and if so creates a new automaton.

Context information: Various properties use nesting of the replicating mechanism — each bank client may have a number of accounts, upon which a number of transactions may take place. Properties about transactions must thus be created for each and every transaction created, but each must have access to its context — the account and client it belongs to. Giving replicated properties access to these inherited values, enables concise and clear properties to be expressed.

Invariants: Furthermore, various objects in the system are expected to satisfy invariants — once set, the ID of a transaction may not change throughout its lifetime. To enable this, LARVA also enables such properties to be expressed, and which are checked upon the arrival of each event. Internally, this is done by creating an implicit transition from each state which sends the automaton to a bad state should the condition not be satisfied.

Real-time: LARVA provides a clock/stopwatch construct that can trigger events after particular time intervals, implemented as Java threads using *wait* operations. The main drawback of this approach is that it may not be totally accurate due to the Java thread-scheduling mechanism.

LARVA uses aspect-oriented programming techniques [9] to capture events. Upon running the monitored system, the underlying automata are created and initialised. Through the use of aspect-oriented programming techniques, whenever an event is captured, control is passed back to the DATE , which performs a full-step, performing any timer actions and new timer events scheduled as necessary before returning control to the system to proceed. If the system reaches a bad state in any of the properties, appropriate action is taken to terminate or remedy the situation as specified by the user.

Example 2. To illustrate the use of LARVA, consider the monitoring of a simplified banking system, in which one might want to ensure that there should never be more than five users in the bank and that a deletion does not occur when there are no users. By identifying the system events, corresponding to the method calls in the target system, an automaton is constructed, to specify the properties desired. In Fig. 2 we show the automaton used for monitoring the adding and deleting of users, together with the equivalent LARVA code[2].

Furthermore, one may have properties which must hold for every user in a bank, or possibly properties which should hold for each account owned by each user. LARVA enables the specification of such properties using the `foreach` construct — instances of the properties (automata) appearing within the construct are created for each instance of the class. Furthermore, since when nesting the construct, the properties inside inner replicators have access to the contextual information (the instance of the outer iterator) under which they appear. In this

---

[2] The LARVA system, including further documentation and examples, is available from `http://www.cs.um.edu.mt/~svrg/Tools/LARVA`

```
GLOBAL {
 VARIABLES {
   int userCnt = 0;
 }
 EVENTS {
   addUser() = {*.addUser()}
   delUser() = {*.deleteUser()}
   allUsers() = {User u.*()}
 }
 PROPERTY users {
   STATES {
     BAD { toomany baddel }
     NORMAL { ok }
     STARTING { start }
   }
   TRANSITIONS {
     start -> ok [addUser()\\userCnt++;]
     start -> baddel [delUser()\\]
     ...
     ok -> ok [delUser()\\userCnt--;]
     ok -> ok [allUsers()]
   }
 }
}
```
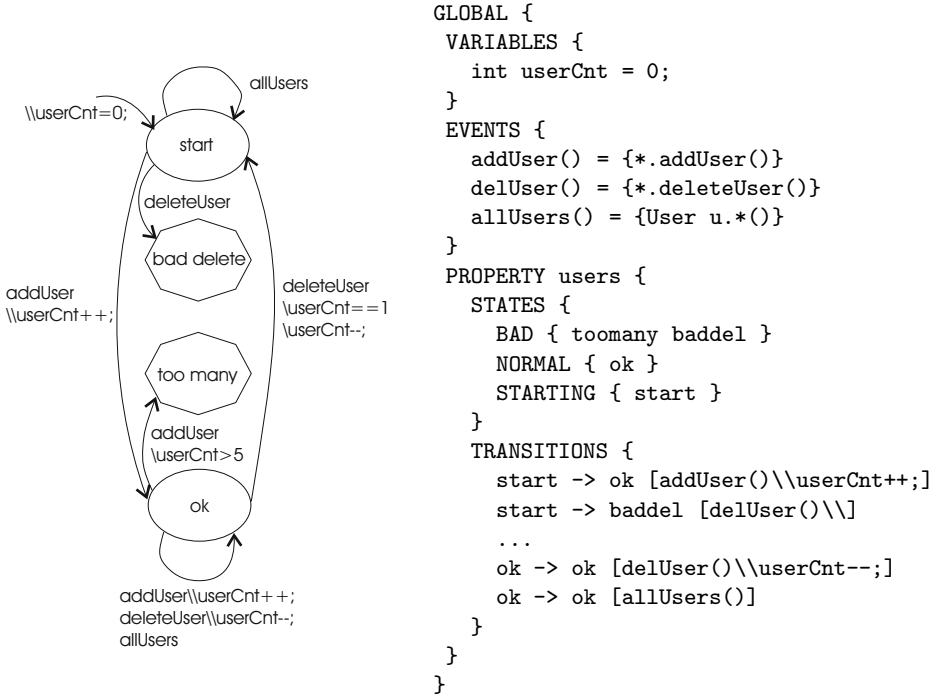
Fig. 2. The automaton and LARVA code of Example 2

case, not only the account, but also the user to whom the account belongs will
be known without explicitly invoking Java code each time. Invariants of the in-
stances of the class may also be specified directly, for example, to ensure that the
account number never changes. The following code illustrates these concepts.

```
FOREACH (User u) {
  ...
  FOREACH (Account a) {
    INVARIANTS
    { String accID = a.getID(); }

    PROPERTY
      { ... }
    }
}
```

3   Case Study

During its development, LARVA was used on an real-life system handling credit
card transactions. The complexity of this system lies not only in the size of the

underlying code, which although not exceptionally large, has over 26,000 lines of code, but also in the strong security implications and communication required among various components (including third party systems, such as banks). The system is designed to hold sensitive information of thousands of people — a single leak of sensitive information could undermine the confidence of the users in the system, leading to drastic financial losses. Furthermore, the system has real-time issues and is required to be able to handle over 1000 transactions per minute.

The system is composed of two parts, one handling the transactions and their database and the other handles the communication to the respective bank or entity which is involved in the transaction. These will be referred to as the *transaction handling system* and the *processor communication system* respectively. The whole system will be referred to as the *transaction system*.

A number of different classes of properties, as described below, were verified at runtime using LARVA on the system.

Logging of credit card numbers: During the development of the original transaction system, credit card numbers were logged for testing purposes. This is however, not in line with standard practice of secure handling of credit card numbers. These logging instances were manually removed from the code. However, to ensure that no instances escaped the developers' attention, a simple verification check to ensure that no data resembling a credit card number is ever logged.

Transaction execution: Transactions are processed by going through a number of stages, including authorisation, communication with the user interface, insertion of the transaction in the database and communication with the commercial entity involved in the transaction, the stages taken depend on which classification the transaction falls under. Designing properties to ensure that during its lifetime, all transactions go through the proper stages was straightforward, especially since the automata-based property language used in LARVA corresponds closely to the concept of stages, or modes in which a transaction resides.

Authorisation transactions: Authorisation transactions have to be checked to ensure that all the stages are processed in the correct order, keeping certain values unchanged — for instance, one must make sure that the ID of the transaction is never accidentally changed. Furthermore, other checks such as ensuring that transaction amounts are not changed after being set were also necessary.

Backlog: A particular feature of the system under scrutiny was a process called backlogging — if communication with a bank or a commercial entity fails, the request is retried a number of times after a given delay. The transaction handling system with backlogs can become rather complex, and properties were identified to ensure that the backlog process is performed for the expected number of times or until the transaction is approved.

Given the nature of the system, with different components and transactions communicating and synchronising their behaviour, it was difficult to measure the

overhead of the monitoring system for the case study. The case study, however, was essential to identify features necessary for the use of runtime verification on real-life case studies. The need for context-information and invariants arose directly from this experience.

4   Comparison of LARVA with Other Related Tools

In this section we compare LARVA with various other runtime-verification tools on a number of criteria, including both in terms of expressivity and overheads induced.

4.1   Related Tools

ConSpec [2] is inspired by PSLang, but restricted to mobile devices with limited resources. A contract is defined for each application and upon installation on a device, the contract is checked against the user's policies. If the application's contract does not comply with the user's policies, the application cannot be installed on the device. In other cases, where the application's contract cannot be definitively checked before installation, a runtime monitor is inlined to the application.

Java-MOP [5] is a monitoring-oriented development environment combining the specification and the implementation of a system. It goes further than runtime verification in that it not only specifies properties to detect violations and raise exceptions, but the violation handling mechanism is itself part of the design of the system's functionality. Hence, the monitoring is not simply an extra check on top of the system but an integral part of the system's design. An appealing feature of Java-MOP is that it can be extended with different logics including FTLTL, PTLTL, ERE and Jass.

The Monitoring and Checking (MaC) architecture [11] intends to bridge the gap between specification and implementation. It has two different specification languages: the Primitive Event Definition Language (PEDL) and Meta Event Definition Language (MEDL) allowing for a clear separation between the definition of the primitive events of a system and the system properties. An implementation of the Monitoring and Checking architecture for Java is Java-MaC [10], which enables automatic instrumentation to have access to the system events. Instrumented programs send an event stream to the event recogniser and to identify higher-level activities, which are in turn processed by the runtime checker to raise an alarm if any of the specified properties are violated.

Hawk [6] is programming-oriented extension of the rule-based Eagle logic. Eagle [4,8] is a runtime verification tool comprising a rule-based language and an interpreter for it, supporting future and past time logics, interval logics, extended regular expressions, state machines, real-time and data constraints and statistics. It is implemented as a Java library allowing rule parameterisation. Transitions carry a condition (on the input of the state machine but also on the variables which constitute the underlying system) and an action on the variables

of the state. Furthermore, the rules expressed in Eagle, can be either maximal or minimal fixpoint semantics, allowing for more flexibility in expressing weak and strong versions of the same operators.

Lola [7] is a synchronous language which allows the user to specify the properties of a program in past and future LTL. The advantage of Lola is that as a synchronous language it guarantees bounded memory to perform online monitoring, but differs from most other synchronous languages in that it is able to refer to future values in a stream. Lola allows the user to collect statistics at runtime and to express numerical queries.

4.2   The Benchmark

A Java program representing a bank processing a number of transactions for a number of users has been developed to experiment with the use of the different systems. The bank system has a database which is used to simulate communication and time delays in the system. When a transaction is executed there are three possible results: success, failure and exception. Upon a failure, the transaction is successively retried for another four times. No retries are performed in case of an exception. Note that the intention of the benchmark case study is primarily to compare property specification and monitoring.

We have identified a number of classes of properties, and concrete examples for the bank processing system, to compare and contrast the use of the different tools.

Scope: The type of scope which can be specified. Types of scope include object (obj.), session (sess.) — one run of the application, multisession — current and previous runs, global — all running applications of a system. This is used to specify on which level the property is verified. For example if the scope is 'object' then the property will be verified for each individual object.

Exceptions: Exception handling and throwing in an application usually represent important events in a system. This aspect represents whether or not the user can express properties which include exception throwing and handling.

Real-time: Real-time refers to whether or not the monitored properties can include real-time. This means that the verification system is able to trigger checks at particular time intervals and compare clock values upon particular system events.

Invariants: We use the term invariants to refer to inbuilt mechanisms in the verification system to monitor the changing of values of variables. The purpose is to be able to verify that certain variables only change when they are supposed to do so.

Feedback: It refers to the capability of the monitoring system to return feedback to the target system. This usually takes the form of a mitigation action in case a violation is found. In other cases this may be limited to stopping the program's execution (denoted by *Stop.* in Table 1).

Conditions: This refers to the ability to filter events by applying a condition on the parameters and/or monitoring variables.

Table 1. Expressivity features of various tools

Tool	LARVA	ConSpec	Java-MOP	Java-MaC	Hawk	Lola
Scope	*Sess./Obj.*	✓ [a]	*Sess./Obj.*	*Sess.*	*Sess.*	*Sess.*
Exceptions	✓	✓	×	×	×	×
Temporal Logics	×	×	✓	×	✓	✓
Real-Time	✓	×	×	✓ [b]	✓ [c]	×
Mobile Application Policies	×	✓	×	×	×	×
Invariants	✓	×	✓	✓	×	×
Feedback	✓	*Stop.*	✓	✓	×	×
Conditions	✓	✓	✓ [d]	✓	×	×
Numerical Queries	×	×	×	×	×	✓

[a] in specification it supports all the mentioned scopes but currently only *session* is supported
[b] restricted (cannot trigger clock events)
[c] can be extended to support real-time
[d] restricted to implementing conditions in violation/validation handling method

Temporal logics: It represents the fact that the tool supports specification written in temporal logics such as LTL.

Mobile application policies: We refer to the ability of defining a security policy which can be partially verified before runtime if the application also specifies its policy. Verifying applications for mobile devices require the monitoring system to be as lightweight as possible.

Numerical queries: This refers to explicit support to expressing numerical queries about statistics of the program being verified.

Table 1 shows which tool have *explicit* support for the aspect being considered. Note that the meaning of the scope object is sometimes referred to as class. In the case of LARVA, the same object need not necessarily be the same instance, but be equated through the (optional) use of a user-provided equality method. One advantage of this approach is that when monitoring objects which are serialised and de-serialised, the object before serialisation will still be considered the same as the object afterwards (even though they are not the same instance).

Although with its own limitations, LARVA can express a number of interesting classes of properties, not all of them expressible directly in the other tools. Two limitations of LARVA are that it cannot support different temporal logics (Hawk, Java-MOP and Lola do have this capability), and it is not suitable for security of mobile devices (in which ConSpec excels).

4.3   Performance of LARVA

Five tests have been built for the evaluation of the performance of LARVA in terms of overheads. *Test 0* executes a number of transactions but does not violate any of the given properties. Subsequently, *Test 1* violates the invariant property

Table 2. LARVA overheads when with the benchmark example

Test Reference Number	0	1	2	3	4
System without Monitoring					
time(ms)	4	4	6303	3	9
memory(Kb)	23	23	23	70	161
System with Monitoring					
time(ms)	123	120	6395	161	176
memory(Kb)	453	209	160	467	434
System with Monitoring without Clocks					
time(ms)	55	60	n/a	n/a	36
memory(Kb)	432	477	n/a	n/a	378

Table 3. Statistics obtained when trying *Test 0* on variations of the benchmark

Test Variation	Normal	Time Cons.	Big Obj.	Many Obj.
System without Monitoring				
time(ms)	4	4722	4874	53849
memory(Kb)	23	23	260	2384
System with Monitoring				
time(ms)	123	5321	5458	65153
memory(Kb)	453	418	458	3509

Table 4. The benchmark applied to various tools. (* — no logging, no clocks)

Test Ref. No.	0	1	4	0	1	4
	LARVA*			ConSpec		
time(ms)	27	23	30	7	n/a	n/a
memory(Kb)	91	136	208	54	n/a	n/a
	Java-MOP			Java-MaC		
time(ms)	23	23	52	7	n/a	n/a
memory(Kb)	174	173	312	26	n/a	n/a

by trying to change the transaction amount. *Test 2* violates the property that a retry should occur within two seconds. *Test 3* violates the property the a user cannot have more than five transactions. Finally, *Test 4* violates the property that upon an exception the transaction is not retried.

Table 2 shows statistics for the benchmark when the five tests were run under three different configurations: (i) without monitors; (ii) with monitors for all the properties; (iii) removing the monitors which include clocks with the purpose of investigating the impact of clocks on the monitoring system.

Although the resources required for the program to run without monitors as compared to when it was run with monitors seems huge, the overhead is linear in the size of the automaton used to describe the properties. In fact, increasing the size of the system with regards to the required memory and processing time preserves the complexity as shown in Table 3, in which *Test 0* is used.

The first experiment was to increase the processing time which the system without monitors require to complete the execution. One should notice how the increase from 4722 to 5321 milliseconds is relatively much smaller than that from 4 to 123 milliseconds. In the second experiment the memory required by each object was increased. In this case the total memory used was 458 kilobytes which is very close to the memory initially used for the initial experiment (453 kilobytes).

In order to investigate the real relationship between the size of the monitoring system and the monitored system, the number of monitored objects was increased by a factor of 10. The results obtained substantiate the intuition that the size of the monitor is linear with respect to the number of monitored objects.

It is difficult to have a true and fair comparison of the tools, since they do not all have the same expressive power and not all monitors were implementable on all tools. For instance, none of the other tools handle real-time properties. Another issue is that LARVA issues a report regarding the status of the monitoring system which is not done by other tools. Removing code and properties to enable common ground comparison between the tools, the results obtained are shown in Table 4. Note that Hawk and Lola are not in Table 4 since we did not have access to the tools. The results concerning their expressiveness were based on descriptions of the tools from papers and personal communication.

The most similar tool to LARVA is undoubtedly Java-MOP since it can implement the same properties in a very similar fashion and both LARVA and Java-MOP use AspectJ as the underlying framework. Compared together, the statistics show that there is little difference. ConSpec is restricted to security properties on mobile devices so the extent of the comparison is limited. To give an idea of resources used by ConSpec, we implemented the property which limits the number of users rather than the number of transactions per user. This explains why the time and memory required were much less than LARVA and Java-MOP. Finally, with Java-MaC, it is again difficult to compare the results since none of the properties could be implemented directly. Furthermore, Java-MaC uses a different technology — transmiting the event stream to other applications running simultaneously. These factors explain the difference in the amount of resources used.

5   Conclusions

Runtime verification has been widely used in various different contexts and for widely different systems. Automatically instrumenting code managing verification from properties gives various advantages. However, the need for a sufficiently expressive logic to be able to specify the system properties succinctly and clearly is essential for confidence in the overall monitoring process. In this paper, we have introduced dynamic communicating automata with timers and events (DATE) to describe properties of systems which need to be checked for different instances of a class. We have also presented LARVA, a runtime verification implementation of this logic. The combination of timers with dynamic automata enables the straightforward expression of various properties, as illustrated in the use of LARVA for the runtime monitoring of a real-time transaction system.

LARVA performs well in comparison to state-of-the-art runtime verification tools, and in terms of expressivity it comprises a set of features not presented as a whole in other tools.

So far the main limitation of LARVA is that it does not support the specification using different temporal logics. This is however not a drawback since the

underlying automata theory of LARVA are highly expressive, and we are currently implementing a translation from LTL and Duration Calculus to DATEs.

Further Work. We consider that LARVA to be mature enough to be used in the development phase to monitor programs were performance is not crucial, even if the overheads induced have been shown to be reasonable. Real-time properties are fragile under slowing down (by introducing monitors) or speeding up (by removing them), which makes runtime verification even more challenging. We are currently building a theoretical framework for the analysis of real-time properties to ensure they are invariant up to slowing down (or speeding up) the system. LARVA is being extended with this analysis to enable more confidence in the instrumentation and deinstrumentation of real-time properties.

References

1. Bouajjani, A., Lakhnech, Y., Robbana, R.: From duration calculus to linear hybrid automata. In: Wolper, P. (ed.) CAV 1995. LNCS, vol. 939. Springer, Heidelberg (1995)
2. Aktug, I., Naliuka, K.: Conspec: A formal language for policy specification. In: FLACOS 2007, Oslo, Norway, October 2007, pp. 107–109 (2007)
3. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
4. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 44–57. Springer, Heidelberg (2004)
5. Chen, F., Roşu, G.: Java-mop: A monitoring oriented programming environment for java. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 546–550. Springer, Heidelberg (2005)
6. D'Amorim, M., Havelund, K.: Event-based runtime verification of java programs. SIGSOFT Softw. Eng. Notes 30(4), 1–7 (2005)
7. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: Lola: Runtime monitoring of synchronous systems. In: TIME 2005, pp. 166–174. IEEE Computer Society Press, Los Alamitos (2005)
8. Goldberg, A., Havelund, K.: Automated runtime verification with eagle. In: MSVVEIS (2005)
9. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J.: Aspect-oriented programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
10. Kim, M., Viswanathan, M., Kannan, S., Lee, I., Sokolsky, O.: Java-mac: A run-time assurance approach for java programs. Formal Methods in System Design 24(2), 129–155 (2004)
11. Lee, I., Kannan, S., Kim, M., Sokolsky, O., Viswanathan, M.: Runtime assurance based on formal specifications. In: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (1999)

Can Flash Memory Help in Model Checking?[⋆]

Jiří Barnat[1], Luboš Brim[1], Stefan Edelkamp[2], Damian Sulewski[2],
and Pavel Šimeček[1]

[1] Masaryk University Brno, Czech Republic
[2] Technische Universität Dortmund, Germany

Abstract. As flash media become common and their capacities and
speed grow, they are becoming a practical alternative for standard me-
chanical drives. So far, external memory model checking algorithms have
been optimized for mechanical hard disks corresponding to the model of
Aggarwal and Vitter [1]. Since flash memories are essentially different,
the model of Aggarwal and Vitter no longer describes their typical behav-
ior. On such a different device, algorithms can have different complexity,
which may lead to the design of completely new flash-memory-efficient
algorithms. We provide a model for computation of I/O complexity on
the model of Aggarwal and Vitter modified for flash memories. We dis-
cuss verification algorithms optimized for this model and compare the
performance of these algorithms with approaches known from I/O effi-
cient model checking on mechanical hard disks. We also give an answer,
when the usage of flash devices pays off and whether their further evolu-
tion in speed and capacity could broaden a range, where new algorithms
outperform the old ones.

1   Introduction

There are numerous computational tasks that require to generate and process
that huge amount of data that cannot be simply kept in internal memory. Un-
fortunately, it is not acceptable in terms of performance to rely on the standard
memory virtualization techniques provided by the operating system, and special-
ized algorithms must be devised to efficiently manipulate data stored externally.
These are the so called I/O efficient or external-memory algorithms [2].

I/O efficient algorithms reflect physical properties of external memory devices,
i.e. they are designed to minimize expensive random accesses to data in favor
of their block processing. However, likewise all the PC components, also the
external memory devices are being continuously developed and their properties
are improving in time. Recently, flash memory based external memory devices
became widely used as the so called *solid state disks* (SSDs). Unlike its mag-
netic counterpart, SSD does not rely on physical movements of the head(s) to
access the data. Therefore, the access time is much smaller for a solid state disk

---

compared to the magnetic one. For example, the speed of random reads for a solid state disk build with NAND flash memory lies roughly at the geometric mean of the speeds of random access memory (RAM) and magnetic hard drive (HDD) [3]. The only factor limiting solid state disks from being massively spread is the cost of the device if expressed per stored bit. The cost per stored bit is still significantly higher for SSDs than for magnetic disks. However, the cost per bit is definitely subject to change in the future.

I/O efficient algorithms have been studied also in the context of formal verification, model checking [4] in particular, as one of the techniques to fight the well known state explosion problem. In this paper we focus on enumerative on-the-fly LTL model checking, which is the standard option for analyzing software systems. Our goal is to consider a simple question that comes up with the advent of solid state disks. Namely, if it is meaningful to design new I/O algorithms for LTL model checking that would take advantage of the fast random reads of a solid state disk, or if it is satisfactory to apply the existing I/O efficient LTL model checking algorithms even for SSDs whose characteristics differ significantly from the characteristics of the traditional magnetic disks.

To answer the question we design several techniques to implement an *SSD efficient* graph traversal procedure, namely we discuss several variants of hashing mechanism that is used by the Nested DFS algorithm to efficiently identify already generated states during the graph traversal. We also report on a preliminary experimental comparison of newly suggested SSD efficient and the standard I/O efficient techniques, and discuss the impact of possible technology improvements that may come in the future.

The paper is organized as follows. In Section 2 we briefly recall the standard I/O efficient techniques used for enumerative external memory LTL model-checking. In Section 3 we state the differences between the standard magnetic and new solid state disks. In Section 4 we describe several SSD efficient hashing techniques, and we show in Section 5 how these can be used to design SSD efficient Nested DFS algorithm. Section 6 report on our experimental evaluation of both the SSD and I/O efficient techniques. Finally, in Section 7 we conclude the paper and plot what impact may have possible future technological improvements.

2   I/O Efficient Model Checking with Mechanical Disks

LTL model checking problem can be reduced to the problem of *accepting cycle detection* in the graph [4]. In the context of enumerative LTL model checking, the graph to be searched for the presence of an accepting cycle is generated on-the-fly meaning that if a graph traversal algorithm needs to proceed to an immediate successors t of a state s, it computes state t from the state vector of s. To prevent re-visiting of already explored states, all states that have been processed are stored in memory, hence, if a state is generated it is first checked against the set of stored states to learn whether it is a new state or has been visited before. In the context of I/O efficient algorithms, this check is referred to as the *duplicate detection*.

Due to the huge number of states, their large size, and the speed of generating them, the memory demands while analyzing systems rise rapidly. In order to release memory, states stored in the set of visited states have to be fully or partially flushed to the external memory. Under this circumstances a check whether a state has been visited may involve I/O operation as not only the states stored in memory, but also the states stored on external memory device must be considered. This however renders a standard graph traversal algorithm inefficient as the I/O operation is in orders of magnitude slower than a single or several reads from the internal memory.

2.1   Graph Traversal

The core technique that gave birth to I/O efficient algorithms is the so called *delayed duplicate detection* [5,6,7,8] whose idea is to postpone the individual checks against the set of visited states and perform them together in a group amortizing thus the cost of I/O operations per a single check.

There are other techniques that have significant impact on the performance of an I/O efficient graph traversal algorithm. For example, it is possible to perform hash compaction or compression of states to be stored which results in less amount of data to be transferred between external and internal memory. Another quite successful improvement builds upon using a Bloom filter maintained in main memory in order to reduce unnecessary I/O operations. Also simple partitioning of states stored on external memory may have impact on the performance of an I/O efficient graph traversal procedure. For more details on these techniques we kindly refer the reader to [9].

As mentioned above, an important aspect of an I/O efficient algorithm is that the data stored on external memory is accessed in blocks. While the clever implementation techniques aim at reducing the number of I/O operations, or reducing the amount of data being transferred, there is also possibility to improve the performance of an I/O efficient algorithm by simple improving the performance of an I/O operation. For example, by connecting two identical external memory devices into a mirror RAID array we can achieve almost double bandwidth that the block of data may be read with from the external memory device. Note that this approach basically improves bandwidth only while does not influence the latency, i.e. the time needed to read the first bit.

Similarly, it is possible to reduce time needed for solving the problem if instead of the serial I/O efficient algorithm working over a single external device a parallel I/O efficient algorithm is used utilizing multiple external memory devices. This is, however, possible only if the algorithm involved allows parallel processing, which is, for example, the case of breadth-first search, but is not the case of depth-first search [10].

2.2   LTL Model Checking

For accepting cycle detection there is a space efficient optimal algorithm called *Nested Depth-First Search* [11]. Unfortunately, the algorithm becomes rather

Table 1. Characteristics of solid state and hard disk drives

	HDD	SSD
Read Bandwidth	65 MB/s	72 MB/s
Write Bandwidth	60 MB/s	70 MB/s
Random Read Access Time	11 ms	0.1 ms
Random Write Access Time	11 ms	5 ms

inefficient, as soon as states to be stored cannot be maintained in the main memory [10,12].

Recently, three different I/O efficient algorithms for solving the LTL model checking problem have been published [12,13,14]. In [12] the authors suggested to avoid the DFS-based accepting cycle detection by the reduction of the problem to the problem of testing reachability relation [15,16] whose I/O efficient solution was further improved by using the directed A* search and parallelism. Since the reduction to the reachability relation testing may result in up to quadratic increase in the space complexity, this algorithm should be rather viewed as a tool for bug hunting.

A new I/O efficient algorithm for LTL model checking was given in [13]. The algorithm avoids the expensive increase in the space complexity, but does not work on-the-fly, which means that the full state space must be generated and stored on external memory device before it is checked for the presence of an accepting cycle. This disadvantage makes the algorithm quite inefficient in the cases an error can be discovered quickly using some on-the-fly algorithm. Finally, the algorithm given in [14] is both on-the-fly and linear in the space requirements with the respect to the size of the state space.

3   From Mechanical to Solid State Disks

Mechanical hard disks have been around for quite a long time, and they have provided us with reliable service over these years. This is about to change with the advent of *Solid State Disks* (SSD). A solid state disk is electrically, mechanically and software compatible with a conventional (magnetic) hard disk drive. The difference is that the storage medium is not magnetic (like a hard disk) or optical (like a CD) but solid state semiconductor (NAND flash) such as battery backed RAM, EEPROM or other electrically erasable RAM-like chip. In last years, NAND flash memories outpaced DRAM in terms of bit-density [17] and the market with SSDs continues to grow. This provides faster access time than a disk, because the data can be randomly accessed and does not rely on a read/write interface head synchronising with a rotating disk. We list a typical data transfer bandwidth and access time for both magnetic and solid state disk in Table 1.

It became the standard to measure the analytical complexity of an I/O efficient algorithm using the complexity model by Aggarwal and Vitter [1]. However,

for solid state disk, the model is no more valid, since it does not cover the different access times for random read and write operations. For solid state disks, we propose to extend the model of Aggarwal and Vitter with a penalty factor p for random write operations.

4 I/O Efficient Graph Traversal with Solid State Disks

We observe that random read operations on SSDs are substantially faster than on mechanical disks, while other parameters are similar. Therefore, it appears natural to ask, whether it is necessary to employ *delayed duplicate detection* (DDD) known from the current I/O efficient graph algorithms, or it is possible to build an efficient SSD algorithm using the standard *immediate duplicate detection* (IDD), *hashing* in particular.

First, we study direct access to the solid state disk without exploiting RAM usage. This implies both random read and random write operations. The implementation serves as a reference, and can be scaled to any implicit search with a visited state space that fits on the solid state disk.

Next, we compress the state in internal memory to include the address on secondary memory only. For this case states are written sequentially to the background memory in the order of generation. For resolving hash synonyms, states lookup random reads are needed. Even though linear probing shows performance deficiencies for internal hashing, for block-wise strategies, it is the apparent candidates. Alternative hashing strategies can reduce the number of random reads.

The third option fosters flushing the internal hash table to the external device, once it becomes full. In this case, full state vectors are stored internally. For large amounts of background memory and small vector sizes, large state spaces can be looked at. Usually the exploration process is suspended while flushing the internal hash table. We observe different trade-offs for the amount of randomness for background readings and writing, which mainly depend on increasing the locality of the access.

4.1 Hashing

The general setting (see Fig. 1) is a background hash table H_b kept on the SSD, which can hold $m = 2^b$ entries. As said, SSDs prefer sequential writes and sequential read, but can cope with an acceptable number of random reads. We additionally assume a foreground hash table H_f with $m' = 2^f$ entries. The ratio between fore- and background is, therefore, $r = 2^k = 2^{b-f}$. Collisions especially on the background hash table can yield additional burden. As chaining requires overhead for storing and following links, we are left with open addressing and adequate probing strategies.

As linear probing finds elements through sequential scanning, it is I/O efficient. The efficiency analysis goes back to Knuth [18]. For a load factor of α a successful search requires about $1/2\left(1 + 1/(1-\alpha)\right)$ accesses on the average, while an unsuccessful search requires about $LP_\alpha = 1/2\left(1 + 1/(1-\alpha)^2\right)$ accesses
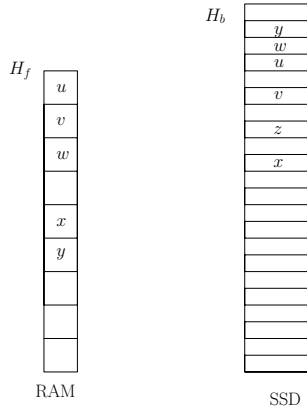
Fig. 1. Fore- and Background Memory, such as RAM and SSD

on the average. For a hash table that is filled up to $\alpha = 50\%$ we have less than three states to look at on the average, which easily fit into the I/O buffer. Given that random access is slower than sequential access, this implies that unless the hash table becomes filled, linear probing with one I/O per lookup per node is an appropriate option for SSD-based hashing.

4.2   Mapping

The simplest method to apply SSDs in graph search is to store each node at its background hash address in a file, and – if occupied – to apply conflict resolution strategy on disk. By their large seek times, this option is clearly infeasible for HDDs, but it does apply to some extent to SSDs. Nonetheless, besides extensive use of random writes that operate block-wise and are, thus, expected to be slow, one problem of the approach is the initialization time, incurred by erasing all existing data stored in background memory.

Hence, we apply a refinement to speed-up search. With one additional bit-vector array kept in RAM, we denote, whether or not a position is already occupied. This limits initialization time to reset all bits in main memory, which is much faster. Moreover, this saves lookup time in case of hashing a new state with an unused table entry. Viewed differently, one can think of a Bloom filter [19], with conflict resolution on disk. Figure 2 (left) illustrates the approach. The bit-vector *occupied* memorizes, whether the address on the SSD is in use or not.

The extra amount of RAM additionally limits the size of the search spaces to be processed. In search practice with a full state vector of several bytes to be stored in the background memory, however, investing one bit per state in RAM does not harm much, given that the ratio between main and external memory remains moderate. The only limit for the exploration is imposed by the number of states that can be stored on the solid state disk, which we assume to be sufficiently large.
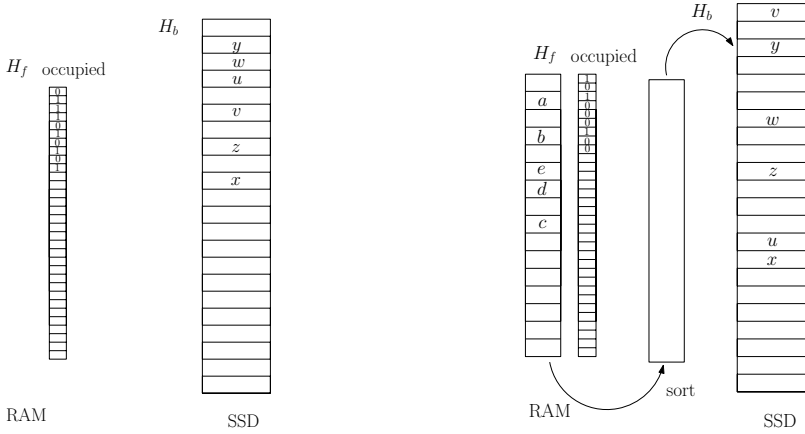
Fig. 2. External hashing without and with merging

For analyzing the approach, let n be the number of nodes and e be the number of edges in the state space graph that are looked at. Without occupied vector requires e lookup and n insert operations. Let B is the size of a block (amount of data retrieved, or written with one I/O operation) and $|s|$ be the length of a state. As long as $LP_\alpha \cdot |s| \leq B$, at most two[1] blocks are read for each lookup[2]. For $LP_\alpha \cdot |s| > B$ no additional random read access is necessary. After the lookup, an insert operation results in one random write. This results in a flash I/O complexity of $O(e + pn)$. Using the occupied vector, the number of read operations reduces from e to n, assuming that no collisions take place.

As the main bottleneck of the approach is random writing to the background memory, as another refinement we can additionally employ a foreground hash table as a write buffer. Due to numerous insert operations, the foreground hash table will once become filled, and then has to be flushed to the background, which incurs writes and subsequent reads. One option that we call *merging* is to sort the internal hash table wrt. to the external hash function before flushing. If the hash functions are correlated, the sequence is already presorted, by means that the number of inversions $inv(H_f) = |\{(i, j) \mid h_f(s_i) < h_f(s_j) \wedge h_b(s_i) > h_b(s_j)\}|$ is small. If $inv(H_f) = O(m')$ and given that we use an algorithm that exploits presorting[3], we obtain a linear time sorting algorithm. While flushing we now have a sequential write (due to the linear probing strategy), such that the total worst-case I/O time for flushing is bounded by the number of flushes times the efforts for sequential writes. Figure 2 (right) illustrates the approach. As we are able to exploit sequential data processing, updating the background hash table

---

[1] when linear probing arrives at the end of the table, an additional seek to the start of the file is needed.

[2] at our system $B = 4,096$ bytes, and $|s| \approx 40$ bytes.

[3] e.g. adaptive sort that runs in time $m' + m' \log \left(1 + \frac{inv(H_f)}{m'}\right)$.
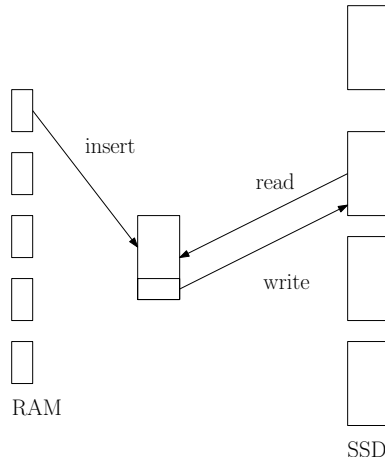
Fig. 3. Updating Tables in Hashing with Linear Probing while Merging

corresponds to a scan (Figure 3). Blocks are read into the RAM and merged with the internal information and then flushed back to SSD.

4.3   Compressing

State compression is a common option in LTL model checking. There are lossless compression strategies like FSM compaction [20], as well as lossy compression strategies like bit-state hashing [21] or hash compaction [22]. For the sake of completeness, in this paper we avoid lossy hash methods as they imply partial state space coverage.

Probably the best lossless compression ratio is obtained using practical perfect hash function [23,24]. Perfect hashing corresponds to an one-to-one mapping of some set S to $\{1, \ldots, |S|\}$. Different off-line algorithms [25] have been developed that include perfect hash functions for what has been coined to the term *semi-external* LTL model checking. We do not apply perfect hashing at all, as for the construction of perfect hash functions, set S has to be known, which contradicts the purpose of on-the-fly model checking.

Instead we store all state vectors in a file on the external storage device, and substitute the state vector by its relative file pointer position. For an external hash table of size m this requires $\lceil \log m \rceil$ bits per entry, that is $m \lceil \log m \rceil$ bits in total. Figure 4 illustrates the approach with arrows denoting the position on external memory. An additional bit-vector *occupied* is no longer needed.

This strategy also results in e lookups and n insert operations. Since the ordering of states on the SSD does not necessarily correlate with the order in main memory, the lookup of states due to linear probing induces multiple random reads. Hence, the amount of individual blocks which have to be read is bounded by $LP_\alpha \cdot e$. In contrast, all insert operations are performed sequentially, utilizing a cache of B bytes in memory. Subsequently this approach performs $O(LP_\alpha \cdot e)$
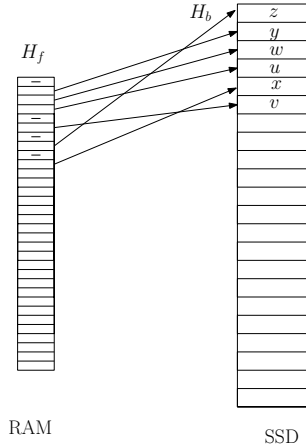
Fig. 4. State Compressing

random reads to the SSD. As long as $LP_\alpha < 2$ this approach performs less random read operations then mapping. By using another internal hashing strategy, e.g. cuckoo hashing [26] one reduces the number of lookups to at most 2. As sequential writing of n states of s bytes requires $n|s|/B$ I/Os, the total flash-memory I/O complexity is $O(LP_\alpha \cdot e + n|s|/B)$.

4.4   Flushing

The above approaches either require significant time to write data according to h_b, or request significant sizes of foreground memory. There are further trade-offs that we will consider next.

One first solution that we call *padding* is to append the entire foreground hash table as it is to the existing data on the background table. Hence, the background hash function can be roughly characterized as $h_b(s) = i \cdot m' + h_f(s)$, where i denotes the current number of flushes, and s the state to be hashed.

Writing is sequential, and conflict resolution strategy is inherited from the internal memory. For several flushing reading a state for answering membership queries becomes involved, as the search for one state incurs up to r many table lookups. Conflict resolution may lead to an even worse performance. For a moderate number of states that exceed RAM resources only by a very small factor, however, the average performance is expected to be good. As far as all states can reside in main memory no access to the background memory is needed.

We can safely assume that load factor α is small enough, so that the extra amount of work due to linear probing is transparent by using block accesses. Again e lookups and n insert operations are performed. Let e_i be the number of successors generated in stage i, $i \in \{0, \ldots, r - 1\}$. For stage 0 no access to the background table is needed. For stage i, $i > 0$, at most $O(i \cdot e_i)$ blocks have to be read. Together with the sequential write of n elements (in r rounds) this results in a flash memory complexity of $O(n|s|/B + rp + \sum_{0 \le i < r} i \cdot e_i)$ I/Os.
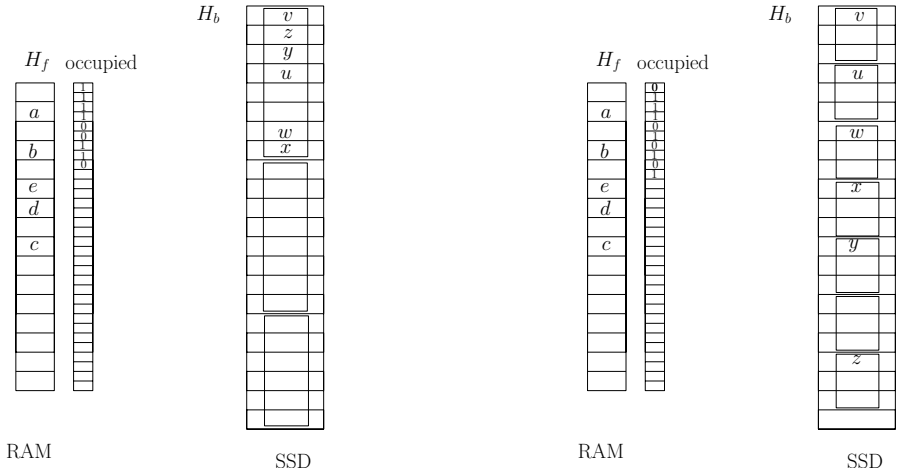
Fig. 5. Padding and slicing

An illustration is provided in Figure 5 (left). The entire foreground hash table has been flushed once, while the maximum number of flushes is set to 3.

The obvious alternative is to *slice* the background hash table such that $h_b(s)$ becomes $h_f(s) \cdot r + i$. An illustration is provided in Figure 5 (right); situation after one flush, and, again, at most 3 flushes are assumed.

The disadvantage of processing the entire external hash table during flushing is compensated by the fact that the probing sequences in the hash tables can now be searched concurrently. For the lookup we use a Boolean vector of size i that monitors if an individual probing sequence has terminated with an empty bucket. If all probing sequences fail, the query itself has failed.

5   I/O Efficient Model Checking with Solid State Disks

In Section 4 various implementations of graph traversal with SSD are shown. It is apparent that some of them are less I/O efficient, but have lower demands on the internal memory (mapping and flushing strategies), while others allocate more of RAM, but perform much less I/O operations in the ordinary case (compress strategy).

On the basis of these graph traversals, it is relatively easy to construct LTL model checking algorithms. Nested DFS, as introduced above, can be implemented with two independent hash tables. To save space it is, however, recommended to use one hash table for storing the states and one internal bit-vector array *flagged* to memorize if a state has been visited in the second depth-first search.

With the above hashing schemes, we arrive at full flexibility in applying immediate duplicate detection in Nested DFS. Table 2 summarizes the hash functions applied and the amount of memory required for the different hashing strategies in

Table 2. Trademarks for different hash strategies for on-the-fly LTL model checking algorithm. Upper two lines give an overview of hash functions, lower three lines show a space complexity in bits for different levels in memory hierarchy.

	Mapping	Compressing	Padding	Slicing
h_f	$-$	$h_d \bmod m$	$h_d \bmod m$	$h_d \bmod m$
h_b	$h_d \bmod m$	$h_d \bmod m$	$i \cdot m' + h_d \bmod m'$	$(h_d \bmod m) \cdot r + i$
RAM	$2m$	$m + m\lceil \log m \rceil$	$2m + m' \times |s|$	$2m + m' \times |s|$
SSD	$m \times |s|$	$m \times |s|$	$m \times |s|$	$m \times |s|$
HDD	$\max_i |Open_i| \times |s|$	$\max_i |Open_i| \times |s|$	$\max_i |Open_i| \times |s|$	$\max_i |Open_i| \times |s|$

$m = 2^b$, $m' = 2^f$, h_d is hash function in DiVinE [27], m is the size of background hash table (in the number of elements), $|s|$ is state vector size (measured in bits), $Open_i$ is the number of states in the search frontier in iteration i.

LTL model checking. Note that there are recent refinements to Nested DFS [28] that are faster, but need more bits.

6   Experimental Evaluation

We implemented our algorithms in DiVinE (DIstributed VerIficatioN Environment) [27], including only part of the library deployed with DiVinE, namely state generation and internal storage. For the implementation of external-memory container and for algorithms for efficient sorting and scanning we use STXXL (Standard Template Library for Extra Large Data Sets) [29]. Models are taken from the BEEM library [30].

For the first set of experiments we used a Desktop PC with AMD Athlon CPU (32 bit) a SATA HDD of 280 GB with 13.8 ms seek time and about 61.5 MB/s for sequential reading and a 32 GB 3.5" SATA high-speed flash memory solid state disk (HAMA), which has 0.14 ms seek time and scales to about 93 MB/s for sequential reading.

To confirm the theoretical results we check the Rether-4 protocol from the BEEM library (Fig. 6). The plot shows Nested DFS runs with different immediate duplicate detection strategies. All experiments, aside from the *mapping* strategy, were stopped after 40,000s (this strategy was stopped after 1,800s due to its obvious lack of performance). The mapping strategy is the worst one because of numerous random writes. We use padding as a flushing strategy. As linear probing is used to store the positions of the saved states, we observe an increased number of *read* operations as the internal hash table becomes filled. Compress strategy appears to perform the best, which corresponds to its I/O complexity without any penalties for random write operations. The difference between *compress* and *compress (stack on hdd)* is the location of the stack file. In the first case, it was located on the SSD, in the second it was on a separate HDD. We observe that having the stacks stored on a second hard disk gives another speed-up of about 30% for the state space traversal.

The motivation for use of SSDs was to exploit fast random access to them. Now, we compare new algorithms designed for SSDs to traditional I/O efficient
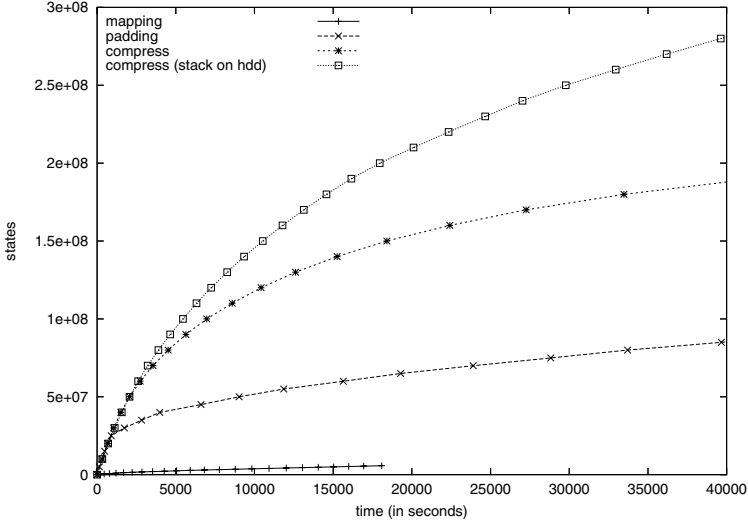
Fig. 6. Comparing the three strategies on the Rether-4 model

algorithms, which we run on SSDs too. To get a fair picture about both approaches, we perform a reachability analysis in breadth first order. As a novel approach we run BFS with immediate duplicate detection and compression strategy (Compress BFS). As a traditional approach we run a standard external BFS with delayed duplicate detection after each level (External BFS).

First, the state space of the Szymanski (5 prop4) model was generated using both approaches. The plot in Fig. 7 demonstrates the dependency in expanding speed between the Compress BFS and the BFS layer size, while the expanding time per layer remains almost the same for External BFS. This is due to the fact, that in delayed duplicate detection the time of level generation is mostly determined by the size of the visited states set, which is completely passed for each BFS layer. Thus, in large search depth, immediate duplicate detection saves much time, compared to delayed duplicate detection.

Therefore, it is apparent that results strongly depend on a structure of a state space. Provided that I/O complexity of External BFS is $O((e/m + \#layers)(n|s|/B)$ [13], it is clear that its I/O complexity is highly dependent on the number of BFS layers, while the I/O complexity of Compress BFS is not. This can be demonstrated on the model Rether-2, with 552 BFS layers (see Fig. 8). While External BFS performs poor on this model, Compress BFS finishes in several minutes. The new approach can also benefit from a small number of back edges and various heuristics helping to recognize duplicates with no reading from disk. This is a case of model Train-Gate, where the amount of random reads was only 30 million, even though the state space has 50 million states, due to the fact that duplicates were typically found in internal buffers (only 8 MB large) before flushing to disk. Model MCS is an example, where External BFS performs better – the state space has relatively low number of BFS levels (90).
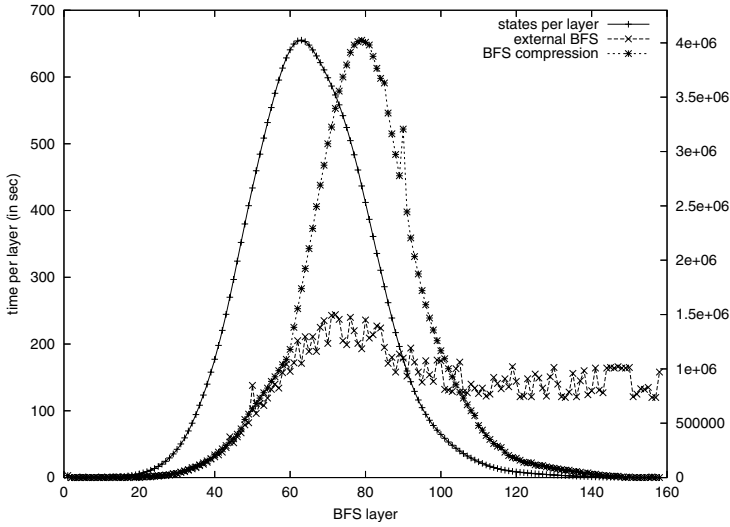
Fig. 7. Comparing Compress BFS to External BFS on the Szymanski 5 prop4 model. The right axis, together with the crossed plot shows the size of each layer. The remaining curves shows time per layer for the different approaches.
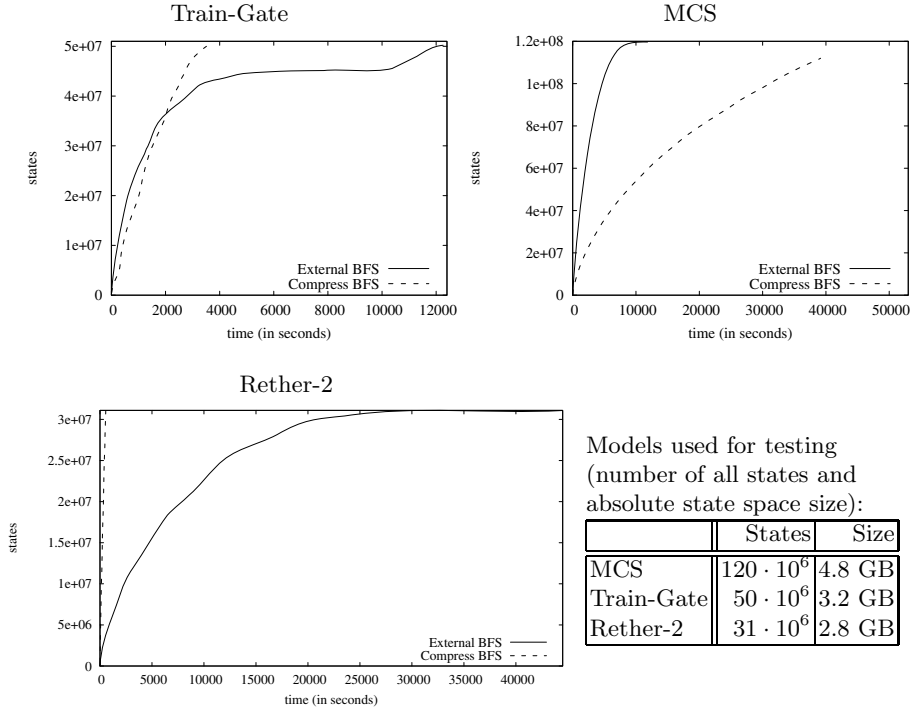

	States	Size
MCS	$120 \cdot 10^6$	4.8 GB
Train-Gate	$50 \cdot 10^6$	3.2 GB
Rether-2	$31 \cdot 10^6$	2.8 GB

Fig. 8. Comparison of External BFS and Compress BFS

From the I/O complexities of both algorithms and from our measurements it follows that External BFS has to slow down the exploration faster than Compress BFS with increasing portion of the state space explored. Thus, Compress BFS can often outperform it from some BFS level due to its linearity in I/O complexity. The moment, when Compress BFS outperforms External BFS depends to high extent on numerous platform and input specific factors: state space structure (number of BFS layers, portion of back edges), bandwidth, access time, file system, implementation (we did not implemented heuristics from [14] or [9]). Even though it is not easy to predict, whether or from which point of exploration Compress BFS outperforms External BFS, the main impact of behaviour of both algorithms is that there can be a threshold, from which Compress BFS outperforms External BFS on a given input and so algorithms for SSDs like Compress BFS are practical.

7   Conclusions

We have contributed several new approaches to hashing applied to SSDs. The most important observation is with the advent of SSD technology, immediate duplicate detection becomes tractable, offering much more flexibility for the choice of the exploration strategy. Monitoring CPU performance, we observed hashing strategies preserve ratios of 50% or more, suggesting that I/O waits are present, but not thrashing. With SSDs random access time decreasing, SSDs will likely become fast enough to rise the CPU usage to 100% making the SSD fully transparent to the user[4].

Compression, the best performing strategy, requires substantial main memory, which according to current ratios of space between RAM and SSDs is still no bottleneck. Although we have tested DFS and BFS, non heuristic algorithms, our SSD hashing strategies can also be applied to heuristic approaches, e.g. A* to rise the amount of states that can be visited. Using SSDs as a shared external storage device for cluster computers will result in an even higher throughput, even for random reads, giving a better possibility for parallel processing.

Directly compared to standard I/O algorithms, for a given model there can be a threshold in state space exploration, from which these new approaches pay off due to their linearity in size of state space – at least for the compress approach. Traditional I/O efficient algorithms are not linear, but they have good constant factors which allow them to outperform new approaches on many inputs. If the bandwidth of SSDs will grow faster, traditional I/O algorithms pay off. If the access time of SSDs will decrease faster than their bandwidth, the importance of new approaches will increase.

Due to easiness of parallel disk connection, large capacities of SSD are possible[5] . Nevertheless, prices for SDDs are still high. Fortunately, in last years they decrease reasonably as the market with flash memories grows.

---

[4] According to Dell, current prices for 32GB RAM are 6 times higher than for 32GB SSDs.

[5] E.g. StorgeSpire – 1 TB SDD array by Solid Data (`http://www.soliddata.com/products/storagespire`).

References

1. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Commun. ACM 31(9), 1116–1127 (1988)
2. Sanders, P., Meyer, U., Sibeyn, J.F.: Algorithms for Memory Hierarchies. Springer, Heidelberg (2002)
3. Min, S.L., Nam, E.H., Lee, Y.H.: Evolution of NAND flash memory interface. In: Choi, L., Paek, Y., Cho, S. (eds.) ACSAC 2007. LNCS, vol. 4697, pp. 75–79. Springer, Heidelberg (2007)
4. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (2000)
5. Korf, R.: Best-First Frontier Search with Delayed Duplicate Detection. In: AAAI 2004, pp. 650–657. AAAI Press / The MIT Press (2004)
6. Korf, R., Schultze, P.: Large-Scale Parallel Breadth-First Search. In: AAAI 2005, pp. 1380–1385. AAAI Press / The MIT Press (2005)
7. Munagala, K., Ranade, A.: I/O-Complexity of Graph Algorithms. In: SODA 1999, Philadelphia, PA, USA, pp. 687–694. Society for Industrial and Applied Mathematics (1999)
8. Stern, U., Dill, D.L.: Using Magnetic Disk Instead of Main Memory in the Murphi Verifier. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 172–183. Springer, Heidelberg (1998)
9. Hammer, M., Weber, M.: To Store Or Not To Store Reloaded: Reclaiming Memory On Demand. In: Brim, L., Haverkort, B.R., Leucker, M., van de Pol, J. (eds.) FMICS 2006 and PDMC 2006. LNCS, vol. 4346, pp. 51–66. Springer, Heidelberg (2007)
10. Barnat, J.: Distributed Memory LTL Model Checking. PhD thesis, Faculty of Informatics, Masaryk University Brno (2004)
11. Courcoubetis, C., Vardi, M., Wolper, P., Yannakakis, M.: Memory-efficient algorithms for the verification of temporal properties. Form. Methods Syst. Des. 1(2-3), 275–288 (1992)
12. Edelkamp, S., Jabbar, S.: Large-Scale Directed Model Checking LTL. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 1–18. Springer, Heidelberg (2006)
13. Barnat, J., Brim, L., Šimeček, P.: I/O Efficient Accepting Cycle Detection. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 281–293. Springer, Heidelberg (2007)
14. Barnat, J., Brim, L., Šimeček, P., Weber, M.: Revisiting Resistance Speeds Up I/O-Efficient LTL Model Checking. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 48–62. Springer, Heidelberg (2008)
15. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. Electr. Notes Theor. Comput. Sci. 66(2) (2002)
16. Schuppan, V., Biere, A.: Efficient Reduction of Finite State Model Checking to Reachability Analysis. International Journal on Software Tools for Technology Transfer (STTT) 5(2–3), 185–204 (2004)
17. Kim, K., Choi, J.H., Choi, J., Jeong, H.S.: The future prospect of nonvolatile memory. In: 2005 IEEE VLSI-TSA International Symposium on VLSI Technology (VLSI-TSA-Tech), pp. 88–94 (2005)
18. Knuth, D.E.: The Art of Computer Programming: Sorting and Searching, vol. 3. Addison Wesley, Reading (1973)
19. Bloom, B.: Space/time trade-offs in hashing coding with allowable errors. Communication of the ACM 13(7), 422–426 (1970)

20. Holzmann, G.J., Puri, A.: A minimized automaton representation of reachable states. International Journal on Software Tools for Technology Transfer 2(3), 270–278 (1999)
21. Holzmann, G.J.: An analysis of bitstate hashing. Formal Methods in System Design 13(3), 287–305 (1998)
22. Stern, U., Dill, D.L.: Combining state space caching and hash compaction. In: Methoden des Entwurfs und der Verifikation digitaler Systeme, 4. GI/ITG/GME Workshop, pp. 81–90. Shaker Verlag, Aachen (1996)
23. Botelho, F.C., Pagh, R., Ziviani, N.: Simple and space-efficient minimal perfect hash functions. In: Dehne, F., Sack, J.-R., Zeh, N. (eds.) WADS 2007. LNCS, vol. 4619, pp. 139–150. Springer, Heidelberg (2007)
24. Botelho, F.C., Ziviani, N.: External perfect hashing for very large key sets. In: CIKM 2007: Proceedings of the sixteenth ACM Conference on information and knowledge management, pp. 653–662 (2007)
25. Edelkamp, S., Sanders, P., Simecek, P.: Semi-external LTL model checking. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 530–542. Springer, Heidelberg (2008)
26. Pagh, R., Rodler, F.F.: Cuckoo hashing. In: Meyer auf der Heide, F. (ed.) ESA 2001. LNCS, vol. 2161, pp. 121–133. Springer, Heidelberg (2001)
27. Barnat, J., Brim, L., Černá, I., Moravec, P., Ročkai, P., Šimeček, P.: DiVinE – A Tool for Distributed Verification (Tool Paper). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 278–281. Springer, Heidelberg (2006)
28. Schwoon, S., Esparza, J.: A note on on-the-fly verification algorithms. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 174–190. Springer, Heidelberg (2005)
29. Dementiev, R., Kettner, L., Sanders, P.: STXXL: Standard template library for XXL data sets. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 640–651. Springer, Heidelberg (2005)
30. Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007)

From Informal Requirements to Property-Driven Formal Validation

Alessandro Cimatti, Marco Roveri, Angelo Susi, and Stefano Tonetta

Fondazione Bruno Kessler
Istituto per la Ricerca Scientifica e Tecnologica
{cimatti,roveri,susi,tonettas}@fbk.eu

Abstract. Flaws in requirements may have severe impacts on the subsequent phases of the development flow. However, an effective validation of requirements can be considered a largely open problem.

In this paper, we propose a new methodology for requirements validation, based on the use of formal methods. The methodology consists of three main phases: first, an informal analysis is carried out, resulting in a structured version of the requirements, where each fragment is classified according to a fixed taxonomy. In the second phase, each fragment is then mapped onto a subset of UML, with a precise semantics, and enriched with static and temporal constraints. The third phase consists of the application of specialized formal analysis techniques, optimized to deal with properties (rather than with models).

1 Introduction

Most of the efforts in formal methods have historically been devoted to comparing a design against a set of requirements. The validation of the requirements themselves, however, has often been disregarded, and it can be considered a largely open problem, which poses several challenges. First, requirements are often written in natural language, and may thus contain a high degree of ambiguity. Despite the progresses in Natural Language Processing techniques, the task of understanding a set of requirements cannot be automatized, and must be carried out by domain experts, who are typically not familiar with formal languages. Second, the informal requirements often express global constraints on the system-to-be (e.g. mutual exclusion), and, in order to retain a direct connection with the informal requirements, the formalization cannot follow standard model-based approaches, but must be complemented with more suitable formalisms such as temporal logics. Third, the formal validation of requirements suffers from the lack of a clear correctness criterion (which in the case of design verification is basically given by the availability of high-level properties). Finally, the expressiveness of the language used in the formalization may go beyond the theoretical and/or practical capacity of state-of-the-art formal verification.

In this paper, we present a new methodology for the validation of requirements, that is based on formal methods. The main phases are the following. In the first phase, the informal requirements are split into basic fragments, which are classified into categories, and the dependency relationships among them are identified. During this informal inspection analysis, the natural language is disambiguated, and easy-to-detect flaws are

discovered. In the second phase, each requirement fragment is formalized according to the categorization. The target formalism is a visual language such as UML, syntactically restricted in order to guarantee a formal semantics, and enriched with a highly-controlled natural language, to allow for expressing static and temporal constraints. In the third phase, an automatic formal analysis is carried out over the modeled requirements, using a number of advanced, complementary techniques. In particular, it is possible to carry out consistency checking, to verify whether some required properties are entailed, and whether the requirements are compatible with selected scenarios. Within this setting, diagnostic information is provided by means of traces, inconsistent cores, property vacuity, and scenario coverage.

The paper is structured as follows. In Section 2, we overview the methodology. In Section 3 and 4, we describe the informal analysis phase and the formalization phase. In Section 5, we present the procedures underlying the verification phase. In Section 6 we describe the support tools. In Section 7 and 8, we discuss the proposed methodology and the related work. Finally, in Section 9, we draw some conclusions and outline directions for future work.

2   Overview of the Methodology

We propose a novel methodology that addresses the issue of formalizing and validating a requirements specification written in an informal language.

Our approach builds on the use of the Unified Modeling Language (UML) to formalize the requirements, and on the use of a Controlled Natural Language (CNL) [30], based on a subset of the Property Specification Language (PSL) [21], to formalize the set of constraints on the requirements model. The set of UML concepts and artifacts we use in our approach represents a subset of the UML 2 concepts and diagrams described in the OMG UML 2 metamodel specification documents [1]. The subset of PSL our CNL builds on is the one that mixes Linear Time temporal Logic (LTL) [28] with Regular Expressions [4].

Our methodology consists of the the following three main steps:

M1 *Informal Analysis Phase.* It consists of the categorization and structuring of the informal requirement fragment described in the requirements document to produce categorized requirement fragments;
M2 *Formalization Phase.* The categorized requirement fragments are described trough the set of concepts and diagrams in UML, and additional constraints in the defined CNL to produce formalized requirement fragments;
M3 *Formal Validation Phase.* It consists of the identification of a subset of the formalized requirement fragments (together with the definition of a series of validation problems) for an automatic validation analysis.

Each step of the methodology is supported by a specific tool. In the following sections, we detail the methodology in terms of the associated sub-phases, artifacts, and modeling concepts. The categories of requirement fragments are detailed together with the steps for the analysis and structuring of the requirements. The methodology is presented using as running example a simple elevator control system specification.

Table 1. Requirement fragments categories

Requirement fragments conditions	Requirement fragment category
Does the requirement fragment define a particular concept in the domain?	Glossary
Does the requirement fragment introduce some system's modules and describe how they interact?	Architecture
Does the requirement fragment describe the steps a particular module performs or the states where the module can be?	Functional
Does the requirement fragment describe the messages some modules exchange?	Communication
Does the requirement fragment describe some constraints on the behaviors of system-to-be?	Behavioral
Does the requirement fragment describe some constraints on the environment?	Environmental
Does the requirement fragment describe a possible scenario of the domain?	Scenario
Does the requirement fragment describe an expected property of the domain?	Property
Is the requirement fragment a note in the specifications that does not add any information about the ontology or the behavior of the specified system?	Annotation

3   Informal Analysis Phase

The first activity in the methodology is the informal analysis of the set of requirements. In this phase the requirements are first categorized on the basis of their characteristics. Then, some dependencies among them are established to structure the categorized requirement fragments. The steps for this informal categorization and analysis are:

M1.1 Isolation of the fragments that identify a requirement unit of the domain requirements document.

M1.2 Categorization of the informal requirement fragments.

M1.3 Creation of the dependencies among the informal requirement fragments.

M1.4 Analysis of the informal requirement fragments based on standard inspection-based software engineering in order to identify flaws such as, e.g., recursive definitions.

The final result of the informal analysis phase is a database of categorized requirement fragments.

Requirement fragment categorization. We have identified nine possible categories: *Glossary*, *Architecture*, *Functional*, *Communication*, *Behavioral*, *Environmental*, *Scenario*, *Property*, *Annotation*. The conditions specified in Table 1 define the corresponding category to be assigned to each informal requirement fragment.

The categorization helps the analyst in understanding the domain and can also be used in the next steps of the methodology to guide the formalization by suggesting the use of particular UML or CNL constructs.

Dependencies definition. We have identified the following three kind of dependencies to describe the possible relationships among two requirement fragments A and B:

Table 2. Example of identified requirement fragments

R0	Elevator
R1.1	Call buttons to choose a floor
R1.2	Floor
R1.3	Key switches
R1.4	Call buttons could be key switches
R1.5	Certain floors are inaccessible unless using the key
R2.1	Elevator Door
R2.2	Door open buttons
R2.3	Door close buttons
R2.4	[Door close button] instructs the elevator [door] to close immediately
R2.5	[Door open button] instructs the elevator [door] to remain open longer

- *Strong Dependency* links: A cannot exist without B.
- *Weak Dependency* links: A can exist without B.
- *Refinement* links: A redefines some notions of B at a lower level of abstraction.

These dependencies are used in the formalization phase, to establish links among the formalized counterparts, and in the formal validation phase, to identify a well formed verification task.

3.1   Example of Informal Analysis

We start analyzing the specification of the elevator described at the URL `http://en.wikipedia.org/wiki/Elevator#Controlling_elevators`. We perform the step M1.1 of the methodology to get a set of requirement fragments. It is reported that: *A typical modern passenger elevator will have:*

1. *Call buttons to choose a floor. Some of these may be key switches: certain floors are inaccessible unless using the key.*
2. *Door open and door close buttons to instruct the elevator to close immediately or remain open longer.*

The first sentence introduces the concept *Elevator*. The first item introduces the concepts *Floor*, *Button*, *Call button* and *Key*. The functionality *choose a floor* is associated to concept *Call button*. Moreover, the concept *Key* is associated to the *Call button* by saying that *certain floors are inaccessible unless using the key*. The second item in the specification introduces three new concepts: the presence of an elevator *door*, the *door open and door close buttons* with their functionalities *to instruct the elevator to close immediately or remain open longer*. We can then define the requirement fragments as in Table 2.

In the step M1.2 we classify R1.{1-4} and R2.{1-3} as Glossary requirement fragments. The R1.5 and R2.{4-5} can be classified as a Behavioral requirement fragments (they describe constraints on the system-to-be). R1.5 can also be classified Scenario and Property.

In the step M1.3 we recognize a Strong Dependency of the requirement R1.5 to the requirements R1.2 and R1.3.

4   Formalization Phase

Our methodology requires proceeding with the formalization of the categorized requirement fragment version of the requirements produced as artifact of the informal analysis phase. The formalization phase consists of the following sub-activities:

M2.1 Formalize each requirement fragment identified in the informal analysis phase by specifying the corresponding UML concepts and diagrams, and/or the CNL constraints.

M2.2 Link the UML elements introduced in M2.1 to the textual requirements. The link is used for requirements traceability of the formalization against the informal textual requirements, and to select directly from the textual requirements document a categorized requirement fragment to validate.

We have adopted: *classes* and *class diagrams* to formalize the requirements that have been classified as Glossary; *state machines* to formalize requirements classified as Functional; *sequence diagrams* to represent those requirements classified as Scenarios that describe the interaction among a set of objects; *CNL* to specify the Behavioral, the Environmental, the Property requirements and the remaining Scenario requirements. The selection of UML diagrams and concepts has been performed on the basis of the expressive power of the UML concepts and on the need related to the formalization of the UML constructs in a formal language. (See [17] for details on the underlying object model extended with temporal constraints.)

Class diagrams. A *class* represents a concept in the domain. In our context, a class is associated with: a set of class attributes representing the set of characteristics of the concept; a set of class methods, representing actions/procedures the class can perform. A method accepts a set of parameters in input and has a return parameter. Each attribute and each method parameter has a type that can be primitive, e.g. $Integer$ or $Real$, or user defined, i.e., $enumerative$ or a class defined in the UML model.

   Relationships among classes represent the relations existing between domain concepts. In our context we allow only for the following relationships:

– *Association*: it is the basic relationship that can be established among two classes.
– *Aggregation*: it specifies that the class belongs to a collection (another class).
– *Generalization*: it indicates that one class is a "superclass" of the other.

Association and Aggregation relationships are characterized by their multiplicity. This multiplicity represents the range of the number of instances of the involved classes that exist in the domain (0..1 zero or one instance, 0..$*$ or $*$ no limit on the number of instances, 1 exactly one instance, 1..* at least one instance, and $n..m$ n to m instances).

State Machines. We use *state machines* to model the behavior of each method of a class in the domain. In our framework we restricted the syntax of the UML 2 state machines as follows. We only allow for the following kinds of UML 2 state machine states: *initial state*, that represents the entry point of the corresponding class method; *final state*, that represents the return point of the corresponding class method; "simple" *state*,

that represents a generic state of the corresponding class method; *conditional states*, that represent conditional branches in the execution of the corresponding method. A *transition* of a state machine represents the conditions that determine the change of state. Each transition is associated with a label of the form $event[guard]/activity$. The meaning of the label is that the transition is performed when the $event$ occurs and the $guard$ (that is a boolean predicate) is true. When the transition is fired the specified $activity$ is performed. In our framework, the events are restricted to be only class method calls, while the activity have to be a parallel combination of class method calls and class attribute assignments.

Sequence Diagrams. UML *sequence diagrams* model the evolution of the specified objects as a sequence of exchange of messages, focusing on the representation of their interactions. We use sequence diagrams to model Scenarios requirements that are requested/expected to happen in the domain in terms of exchange of messages. The UML 2 notation for a sequence diagram is restricted as follows: we use *objects* to represent the instances of the classes involved in a given interaction; *lifelines* to represent the lifetime of the objects involved in the interaction; *messages* are restricted to be only a method call performed by an object to a method of another object. We also allow for the specification of some *interaction operators* defined in UML 2 that specify particular configurations of messages, such as: the *negation*, the *alternative*, the *option*, the *parallel* and the *loop*.

Controlled Natural Language. In order to allow for the specification of constraints and temporal properties of the entities in the model, we extend the UML model with a constraint language. The constraint language is a Controlled Natural Language (CNL) [30], i.e., a well-defined subset of natural language whose grammar has been restricted in order to be automatically processable. The language includes temporal operators as in [25]. We proposed a CNL grammar, based on the subset of the PSL [21] logic that mixes Linear Temporal Logic (LTL) [28] operators with Regular Expressions [4]. This choice is motivated by the fact that this fragment, being linear time, is adapt to express constraints on the evolution of observable events of the system. The grammar has been defined to include enough syntactic sugar to be easily accepted and then used by non-experts in computer science or software engineering.

We have classified the CNL constraints we use to annotate the UML concepts and diagrams in the following five categories.

- *Initial*: it defines constraints that are valid initially.
- *Invariant*: it defines a constraint expected to be always valid over time.
- *Behavior*: it defines a constraint expressing admissible behaviors.
- *Scenario*: it describes behaviors that are expected to be admitted by the formalized requirement fragments.
- *Property*: it defines behaviors that every possible admissible behavior should satisfy (conversely, it defines a set of behaviors that are not admissible).

Example of formalization. Figure 1 represents the class diagram resulting from the formalization of the output of the informal analysis phase for the elevator. For requirements R1.{1-3} we defined the classes *Button*, *Call_Button*, *Floor*
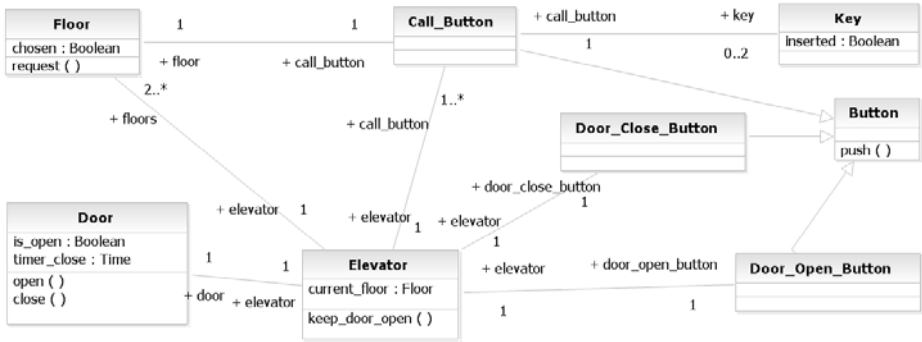
Fig. 1. The elevator class model

and *Key*, *Door_Open_Button*, *Door_Close_Button*, and *Elevator*. *Door_Open_Button*, *Door_Close_Button* and *Call_Button* are refinement of class *Button*. Requirement R1.4 is captured in the diagram by the association relationship between the two classes *Call_Button* and *Key* where is imposed, via the cardinalities at the two sides of the relationship, that for every *Call_Button* there could be 0 or at most 1 *Key* associated. Class *Floor* has an attribute *chosen* (from R1.1) of type Boolean to indicate whether the corresponding floor has been requested. Class *Key* has attribute *inserted*, of type Boolean, to indicate whether the key has been inserted. Class *Door* as attribute *is_open*, of type Boolean, to indicate whether the door is open or closed. Class *Door* has methods *close* and *open*, from requirement fragments R2.4 and R2.5 respectively, to model the activity of opening/closing the corresponding door. Class *Button* has method *push* to model the press of the button.

The analysis of the categorized requirement fragments for the elevator lead to the definition of some CNL constraints. Below we report an excerpt of the constraints classified Behavior (CB.#) necessary to formalize the elevator.

CB.1 *for all Call_Button b, whenever b.push(), b.floor.request()*
CB.2 *for all Call_Button b, if b has a key, whenever b.floor.request(), b.key.inserted*
CB.3 *for all Door_Close_Button b, whenever b.push(), b.elevator.door.close()*

Requirement R1.1 leads to the introduction of the constraint CB.1 to say that pushing the button related to a floor means to request that floor. Similarly, from R1.5 it is possible to extract the CNL constraint CB.2, that states that all call buttons that have a key are such that if the button has been pressed, then the key has been inserted. R2.4 states that when the door close button is pushed then the elevator door has to close. This is formalized with by the CNL constraint CB.3. From R1.5, it is possibile to obtain also the following constraints of type scenario (CS.#) and property (CP.#) respectively:

CS.1 *there exists a Call_Button b such that b has a key and in the future b.floor.request()*
CP.1 *for all Call_Button b, b has a key and never b.key.inserted implies never b.floor.request()*

Following the same process, other constraints have been imposed on the model on the basis of the specification. They have not been reported here for the sake of space. A more detailed description can be found at `http://es.fbk.eu/people/roveri/tests/fmics08`.

Phase M2.2 consists in linking all the elements in the model to the textual requirements fragments. For example, the requirement R1.1 will be directly related to the class *Call_Button* via a data structure that maintains this information (such as for instance the traceability structures provided by standard tools like IBM Rational RequisitPro and IBM Rational Software Architect).

5 Formal Validation Phase

The validation of the formalized requirement fragments aims at improving the quality of the requirements. This goal is achieved by performing several analysis steps, based on the use of formal techniques, that may help to pinpoint flaws that are not trivial to detect in an informal setting.

These steps include checks to identify inconsistencies, and to increase the confidence that the categorized requirement fragment and its corresponding formalized counterpart meet the design intent: for instance, a flaw may be in the fact that some desired behaviors have been ruled out by an over-constraining set of requirements; conversely, some undesired behavior may have not been ruled out by under-constraining requirements.

The formal validation phase of the methodology will be accomplished as follows:

M3.1 *Check the well-formedness of the formalized requirement fragments.* This initial activity aims at verifying that the formalized requirement fragments syntactically adhere to the formal language syntax, and that all the elements mentioned have been previously defined.

M3.2 *Narrowing of the formalized requirement fragments.* This phase aims at focusing the validation to a particular subset of interest of the formalized requirement fragments (e.g. to restrict the validation of the classes/functions of a specific module). In this phase the validation expert selects a set of objects per each class.

M3.3 *Formal validation of the identified formalized requirement fragments.* The subset of interest identified in M3.2 is formally analyzed to identify flaws if any.

Whenever a problem is identified in any of the above sub-phases, in order to try and solve the identified flaw, it may be required to go back to a previous phase. We remark that, in this phase, the domain expert responsible of the validation can specify additional desired and undesired behaviors w.r.t. the ones already formalized in previous phases, in order to guarantee that the design intents are captured, thus further enriching the formalized requirement fragment.

The phase M3.3 can be further decomposed depending on the scope and on the level of domain knowledge required to perform it. For this purpose we classify the validation checks in *Domain Independent* and *Domain Dependent* checks. There is a third kind of checks, aiming at further analyzing the *quality* of the results produced by the domain dependent checks e.g. by performing vacuity analysis, coverage analysis and safety analysis.

Domain Independent Checks. These checks aim at verifying properties of the formalized requirement fragment that do not require any domain knowledge, i.e. *logical consistency* and *realizability*.

Checking Logical Consistency. The formal notion of logical consistency can be intuitively explained as "freedom from contradictions". It is possible that two formalized requirement fragments mandate mutually incompatible behaviors. This check aims at formally verifying the absence of logical contradictions in the considered formalized requirement fragments. Consistency checking is carried out by dedicated formal verification algorithms [16].

Checking Realizability. Realizability [29,13] intuitively amounts to checking if there exists an open system implementing the considered formalized requirement fragments. The variables occurring in the considered formalized requirement fragments are classified as either controllable (by the specified system), or uncontrollable (depending on the environment). Moreover, the considered formalized requirement fragments are partitioned in two distinct sets, the formalized requirement fragments representing "assumptions" on the behavior of the uncontrollable variables, and the formalized requirement fragments representing the "guarantee", that must be enforced on the controlled variables. The check consists in verifying the existence of an open system whose controllable variables obey the guarantee for all possible behaviors of the uncontrollable signals obeying the assumptions. Realizability is substantially more informative than satisfiability, but also computationally more expensive [29]. Realizability checking is carried out by dedicated state of the art algorithms for checking and debugging realizability [15].

Providing Diagnostic Information. The checks for logical consistency and for realizability not only produce a yes/no answer, but they can also provide the validation expert with diagnostic information of different forms. For instance, when consistency checking succeeds, it is possible to produce a trace witnessing the consistency, i.e. satisfying all the constraints in the considered formalized requirement fragments. Similarly, as outcome of the realizability check, it is possible to generate a witness of realizability, that in this case has the form of a Finite State Machine satisfying the considered formalized requirement fragments. We notice that, if the specification is inconsistent, no behavior can be associated to the considered formalized requirement fragments; similarly, when it is not realizable, then no Finite State Machine can be associated. In these cases, the verification algorithms can also generate diagnostic information. For consistency check, this has the form of a small un-satisfiable subset of the considered formalized requirement fragment [16], while for unrealizability check, an un-realizable [15] subset is identified. This information can be given to the domain expert, to support the identification and the fix of the flaw. The formalized requirement fragments can be traced back to the corresponding categorized requirement fragment, and up to the original requirements in order to remove the identified flaw.

The fact that a given formalized requirement fragments is not consistent can be traced back to a misinterpretation in the formalization of the corresponding categorized requirement fragments. In this case, the subset of the considered formalized requirement fragments produced as diagnostic information needs to be revised to remove the ambiguity

that led to the misinterpretation of the original requirements. A possible explanation for the un-realizability can also be traced back to a missing assumption on the environment. In this case, the fix consists in revising the whole set of requirements to add the missing assumptions.

Domain Dependent Checks. These checks aim at verifying that the considered set of formalized requirement fragments really captures the design intent. In this case, the formalized requirement fragments are validated against descriptions of desired and undesired behaviors identified by the domain expert. Desired behaviors are used to ensure that the considered formalized requirement fragments are not too strict, and that they have not been ruled out (*scenario compatibility*). Dually, undesired behaviours are used to ensure that the considered formalized requirement fragments are not too weak, and that they have indeed been ruled out (*property checking*).

Scenario compatibility. This check aims at verifying whether a set of conditions (also called a scenario) is possible, given the constraints imposed by the considered formalized requirement fragments. Intuitively, the check for scenario compatibility can be seen as a form of simulation guided by a set of constraints. The behaviors used in this phase can be partial, in order to describe a wide class of compatible behaviors.

The check for scenario compatibility can be reduced to the problem of checking the consistency of the set of considered formalized requirement fragments with the constraint describing the scenario. Thus, if the scenario is compatible, we obtain a behavior trace compatible with both the considered formalized requirement fragments and with the constraint describing the scenarios. Otherwise, we obtain a subset of the considered formalized requirement fragments that prevents the scenario to happen.

Property checking. This check aims at verifying whether an expected property is implied by the considered formalized requirement fragments. This check is similar in spirit to Model Checking [19], where a property is checked against a model. Here the considered set of formalized requirement fragment plays the role of the model against which the property must be verified. When the property is not implied by the specification, a counterexample is produced. A counterexample is a behavior witnessing the violation of the property, i.e. a trace that is compatible with the considered formalized requirement fragment, but does not satisfy the property being analyzed.

Property checking can be reduced to the problem of checking the consistency of the considered formalized requirement fragments with the negation of the property. If this set is consistent, then a witness behavior compatible with the considered formalized requirement fragment and satisfying the negation of the property is produced. This behavior is a counterexample for the property. If such witness does not exist then the property holds.

If the verification of the property fails, two causes are possible: the first one is that the property is not correctly formalized; the second possibility is in a wrong formalization of the informal sentences in the categorized requirement fragment that need to be disambiguated and/or corrected. An inspection of the counterexample can be carried out in order to discriminate among the two possibilities. If the property is wrong, then it is corrected and the check is repeated. Otherwise, the formalized requirement fragments

has to be corrected, either by modifying the formalization or by adding additional constraints, until the satisfaction of the given property is achieved.

Quality of the results of Formal Validation. The previous analyses can produce diagnostic information in several forms (witness/counterexample behaviors). It is worth noticing that, the fact that the formalized requirement fragment is consistent or that a property holds can be due to some under-specification in the considered formalized requirement fragment or in the property itself. Moreover, if a property fails, there can be several reasons that can cause the failure. Thus, before starting to fix the formalized requirement fragment it would be useful to identify all the causes of the flaw.

The first problem is tackled by performing what we called *vacuity checking* and *coverage checking*, while the second is tackled by *safety analysis*.

Vacuity checking. Corresponds to checking whether a given property holds vacuosly [7]. For instance, consider the property "whenever the signal A is received, a corresponding signal B must be issued". If the formalized requirement fragment is such that the signal A can never be received, then the property trivially holds (the pre-condition of the implication is not satisfiable), and is thus not informative. Vacuity is typically considered to be a flaw in a specification [7] due to missing or redundant constraints.

Coverage checking. Corresponds to checking which elements of the considered formalized requirement fragment have been stimulated (covered) by a generated trace. This check plays for scenario checking and consistency the role that vacuity plays for properties. Suppose the validation generates a trace such that a certain signal A is never issued, and that the considered formalized requirement fragments (possibly together with a property scenario) for which this trace has been generated is mandating that "whenever a signal A is received, a corresponding signal B must be issued". The trace "trivially" satisfies the considered set of requirements, but it does not stimulate the consequence of the mandating property (which is what the domain expert is interested to see), thus the trace is not informative. The fact that a generated trace is not informative, is not a flaw per se, but it can indicate that for instance the assumptions on the environment are under-specified or that the scenario is under-specified.

Formal safety analysis. It aims to identify all the causes leading to the violation of an expected property. The domain expert can identify the variables of interest that are to be considered causes of a specific violation, and advanced algorithms [6,8,9] can then be used to gather a description of the causes, and to organize them in form of a fault tree.

Validation Loop. The above validation steps can be iterated arbitrarily, by correcting formalized requirement fragments and/or the corresponding categorized requirement fragments if necessary, creating new scenarios, new properties, and by analyzing different aspects of the requirements specification. The narrowing phase M3.2 allows the domain experts to focus only on a subset of the formalized requirement fragments by selecting specific modules and consider only some of the functions of the selected module thus enabling for a *modular validation* approach. It also allows performing several kinds of *what-if* analysis, in particular, it allows checking which properties and scenarios remain valid after adding/removing new formalized requirement fragments. Moreover, in

the narrowing phase we can ignore the requirements with low-level details and consider the requirements at a higher level of abstraction, thus enabling for a hierarchical verification approach. This process results in a validation loop where every check increases the confidence of the domain expert in the correctness of the formalized requirement fragments.

Example of validation. We applied the proposed validation loop to the Elevator example. We selected the formalized requirement fragments described in Section 4. In the narrowing phase (M3.2) we identified the following set of objects: one *Elevator*, four *Floors*, four *Call_Buttons*, one *Door*, one *Door_Open_Button*, one *Door_Close_Button*, and one *Key*.

We first checked for the consistency of the formalized requirement fragments. We automatically translated the class diagram and the constraints into the input language of the model checker NuSMV [14]. The tool provided us with a witness of the example's consistency consisting of a loop over the initial state. This trace described the case where the elevator is initially at the fourth floor with the door open and nothing happens.

We then verified if the model is compatible with a scenario where the elevator is initially at the first floor, there is a request to go to the third floor, and the elevator goes to the third floor. The tool provided us a trace witnessing the compatibility with such scenario: the produced trace is such that it loops over requesting both first and third floor at the same time, going to the third floor and then going back to the first floor.

Finally, we verified the scenario CS.1 and the property CP.1. The formalized requirement fragments results compatible also with CS.1 and produces a trace where all buttons have a key, and only the first and the fourth become requested. This trace seems to contradict the assumption that we have only one key, but the point is that we did not force the buttons not to share their keys. After adding this new assumption, we get a new trace where only the second button has a key, and all floors become requested. Finally, the model checker proved that there are no counterexamples with length less than 40 time steps for property CP.1.

6 Overview of the Support Tools

Our methodology is supported by a tool chain we developed on top of standard-de-facto industrial tools.

We used IBM Rational RequisitePro (RRP), interfaced with Microsoft Word, and IBM Rational Software Architect (RSA), to support the informal analysis phase and the tracebility of the link between the informal requirement fragments and their formal counterparts. We used RSA interfaced with RRP and with the validation tool to support the formalization phase. The developed interface allows mapping the formal model into the input language of the validation tools. Moreover, it maps back the verification results as to use them within RSA back to RRP to correct the possible flaws identified during the validation phase.

The validation tool has been built on top of an extended version of the state-of-the-art NuSMV [14] verification tool. This extension provides advanced techniques to compile LTL and PSL properties into automata [18], and advanced abstraction based

verification techniques [12], exploiting the MathSAT [10] SMT solver, to efficiently deal with infinite-state components.

7   Discussion of the Approach

We discuss how the proposed methodology addresses the challenges of requirement validation we consider most relevant for a successful adoption of formal methods in the design flow of complex safety-critical systems.

Choice of a formalization language. The proposed methodology provides a fully formal language. Every statement is associated with a formalized counterpart, that is given unambiguous semantics. Nevertheless, the language can be used by the domain experts because it exploits the usability of graphical languages such as UML and the closeness of CNL to Natural Language. This way, we try to maximize the usability and expressiveness of the language. At the same time, we provide the automatic techniques for the formal analysis.

Ambiguity of natural language. The proposed methodology addresses the key problem that the informal requirement fragments are ambiguous and unstructured as follows: the informal requirement fragments are structured and categorized by means of an informal requirement analysis; every informal requirement fragment is linked with a precise set of elements in the formal model; this way, if the validation phase detects some bugs, the domain expert can easily distinguish if they are due to a wrong formalization or to the ambiguity of the informal requirement fragment.

Incompleteness of requirements validation. New verification techniques and tools have been developed to overcome the inadequacy of traditional tools for model checking and design verification to validate requirements. The analysis is no longer directed on a design; rather, the properties themselves become the object of the analysis. It is possible to check whether the specification is strict enough, by checking whether undesired behaviors have been indeed eliminated. Technically, this problem is reduced to checking whether the expected property is a logical consequence of the set of requirements. Conversely, it is possible to check if the specification is not too strict, by checking whether desirable behaviors have not been eliminated. This approach to requirements analysis is described in [27] and in [23]; the RAT (Requirements Analysis Tool) has been developed [15] to this end.

Quality of the validation feedback. The formal validation phase of our methodology tries to maximize the information the validation tools can produce in order to help the domain expert to correct the specification or the formalization: it produces traces animating the requirements; it can enable the diagnosis of inconsistencies by identifying inconsistent cores; it can identify vacuous properties and uncovered requirements; it can enable the formal safety analysis by performing Fault-Tree Analysis (FTA) and Failure Modes and Effects Analysis (FMEA) using emerging techniques [6,9].

Complexity of the specification. The proposed methodology tackles the problem of the specification complexity by providing a mixed property/model-based approach. This allows handling and analyze complex system of specifications: each requirement is

encoded in a distinct piece of formalism; the methodology supports incremental and modular approaches to the validation; and, it enables for analysis at different level of abstraction. Most importantly, the property-based approach allows verifying the specification without the need of describing the model of the implementation. The approach is ideal for early validation when the requirements must be verified before the implementation of the system.

8   Related Work

The problem of formalizing and analyzing a requirement specification is one of the main challenges in Requirements Engineering.

Works such as [22] and [5] aim at extracting automatically from a natural language description a formal model to be analyzed. However, on one hand, their target formal languages cannot express temporal constraints over object models; on the other hand, they miss a methodology for an adequate formal analysis of the requirements. Nevertheless, our methodology can benefit from mature natural language processing techniques which are able to automatically dig out the ontology of the domain.

Several formal specification languages such as Z [31], Object-Z [11], VDM [2], B [3], and OCL [26] have been proposed for formal model-based specification. However, all of them are not adapt for the use by requirements analysts and domain experts. They are very expressive but require a deep background in order to write a correct formalization, they lack of completely automatic proof support tools, and the use of these tools requires deep knowledge of them in order to use them efficiently. Moreover, these languages have been designed for particular applications, and their usage for different purposes may become awkward and difficult. For instance, they are unable to express complex temporal constraints like, e.g., fairness.

Formal Tropos (FT) [32] and KAOS [20] are goal-oriented software development methodologies that provide a visual modelling language that can be used to define an informal specification. The visual modeling language is supported with annotations that characterize the valid behaviors of the model, expressed in a typed first-order linear time temporal logic (LTL). The main differences between the proposed approach and FT and KAOS are in the expressiveness of the formalization language: both FT and KAOS are limited to pure LTL and they are hardly committed to the goal representation of the requirements.

In [24], a framework is proposed for the automated checking of requirement specifications expressed in Software Cost Reduction tabular notation, which aims at detecting specification problems such as type errors, missing cases, circular definitions and non-determinism. Although this work has many related points to our approach, the proposed language is not adapt to formalize requirements that contain functional descriptions of the system at high level of abstraction with temporal assumptions on the environment.

9   Conclusions

In this paper we have presented a methodology for the validation of a requirements specification. The methodology first envisages an informal analysis of the requirements

document to categorize each requirement. In the second phase, each requirement fragment is formalized according to the categorization by means of UML diagrams and the use of a Controlled Natural Language as to facilitate the use by non experts in formal methods. In the third phase, automatic formal analysis is carried out to identify possible flaws in the formalized requirements. The methodology is supported by a chain of tools built on top of standard-de-facto industrial tools (like e.g. Rational RequisitePro and Software Architect), and on an extended version of the NuSMV model checker.

The methodology and the related tools are currently under evaluation in a real-world project that aims at formalizing and validating the European Train Control System (ETCS) specification. The project is in response to the European Railway Agency tender ERA/2007/ERTMS/OP/01 ("Feasibility study for the formal specification of ETCS functions"), awarded to a consortium composed by RINA SpA, Fondazione Bruno Kessler, and Dr. Graband and Partner GmbH (see `http://es.fbk.eu/events/formal-etcs/` for further information on the project). The documents under consideration contain a huge set of requirements, that are intended to guarantee the interoperability between trackside railway systems and trains throughout Europe. This consortium is currenlty applying the methodology, and carrying out a training activity for domain experts. A detailed reporting of the results of the project is the object of future activities.

References

1. UML Version 2.1.2., `http://www.omg.org/spec/UML/2.1.2/`
2. Bjorner, D., Jones, C.B. (eds.): The Vienna Development Method: The Meta-Language, vol. 61. Springer, Heidelberg (1978)
3. Abrial, J.R.: The B-book: Assigning Programs to Meanings. C.U. Press, Cambridge (1996)
4. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers - Principles, techniques and tools. Addison-Wesley, Reading (1986)
5. Ambriola, V., Gervasi, V.: On the Systematic Analysis of Natural Language Requirements with CIRCE. Autom. Softw. Eng. 13(1), 107–167 (2006)
6. Banach, R., Bozzano, M.: Retrenchment, and the generation of fault trees for static, dynamic and cyclic systems. In: Górski, J. (ed.) SAFECOMP 2006. LNCS, vol. 4166, pp. 127–141. Springer, Heidelberg (2006)
7. Beer, I., Ben-David, S., Eisner, C., Rodeh, Y.: Efficient detection of vacuity in temporal model checking. Formal Methods in System Design 18(2), 141–163 (2001)
8. Bertoli, P., Bozzano, M., Cimatti, A.: Symbolic model checking framework for safety analysis, diagnosis, and synthesis. In: Edelkamp, S., Lomuscio, A. (eds.) MoChArt IV. LNCS, vol. 4428, pp. 1–18. Springer, Heidelberg (2007)
9. Bozzano, M., Villafiorita, A.: The FSAP/NuSMV-SA Safety Analysis Platform. STTT 9(1), 5–24 (2007)

10. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: The MathSAT 4 SMT Solver. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 299–303. Springer, Heidelberg (2008)
11. Carrington, D.A., Duke, D.J., Duke, R., King, P., Rose, G.A., Smith, G.: Object-Z: An Object-Oriented Extension to Z. In: FORTE 1989, Amsterdam (NL), pp. 281–296 (1990)
12. Cavada, R., Cimatti, A., Franzén, A., Kalyanasundaram, K., Roveri, M., Shyamasundar, R.K.: Computing Predicate Abstractions by Integrating BDDs and SMT Solvers. In: FM-CAD, pp. 69–76 (2007)
13. Church, A.: Logic, arithmetic and automata. In: Proc. 1962 Int. Congr. Math., Upsala, pp. 23–25 (1963)
14. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NuSMV: A new symbolic model checker. STTT 2(4), 410–425 (2000)
15. Cimatti, A., Roveri, M., Schuppan, V., Tchaltsev, A.: Diagnostic information for realizability. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 52–67. Springer, Heidelberg (2008)
16. Cimatti, A., Roveri, M., Schuppan, V., Tonetta, S.: Boolean Abstraction for Temporal Logic Satisfiability. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 532–546. Springer, Heidelberg (2007)
17. Cimatti, A., Roveri, M., Susi, A., Tonetta, S.: Object models with temporal constraints. In: SEFM (2008) (to appear)
18. Cimatti, A., Roveri, M., Tonetta, S.: Syntactic Optimizations for PSL Verification. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 505–518. Springer, Heidelberg (2007)
19. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (1999)
20. Darimont, R., Delor, E., Massonet, P., van Lamsweerde, A.: GRAIL/KAOS: an environment for goal-driven requirements engineering. In: ICSE 1997, pp. 612–613. ACM, New York (1997)
21. Eisner, C., Fisman, D.: A Practical Introduction to PSL. Springer, Heidelberg (2006)
22. Fantechi, A., Gnesi, S., Ristori, G., Carenini, M., Vanocchi, M., Moreschini, P.: Assisting Requirement Formalization by Means of Natural Language Translation. Formal Methods in System Design 4(3), 243–263 (1994)
23. Fuxman, A., Liu, L., Mylopoulos, J., Pistore, M., Roveri, M., Traverso, P.: Specifying and analyzing early requirements in Tropos. Requirements Engineering 9(2), 132–150 (2004)
24. Heitmeyer, C.L., Jeffords, R.D., Labaw, B.G.: Automated consistency checking of requirements specifications. ACM Trans. Softw. Eng. Methodol. 5(3), 231–261 (1996)
25. Nelken, R., Francez, N.: Automatic Translation of Natural Language System Specifications. In: CAV, pp. 360–371 (1996)
26. OMG. Object Constraint Language: OMG available specification Version 2.0 (2006)
27. Pill, I., Semprini, S., Cavada, R., Roveri, M., Bloem, R., Cimatti, A.: Formal analysis of hardware requirements. In: DAC 2006, pp. 821–826 (2006)
28. Pnueli, A.: The temporal logic of programs. In: Proceedings of 18th IEEE Symp. on Foundation of Computer Science, pp. 46–57 (1977)
29. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: 16th Annual ACM Symposium on Principles of Programming Languages, pp. 179–190 (1989)
30. Schwitter, R.: Dynamic Semantics for a Controlled Natural Language. In: DEXA Workshops, pp. 43–47 (2004)
31. Spivey, J.M.: The Z Notation: a reference manual, 2nd edn. Prentice Hall, Englewood Cliffs (1992)
32. Susi, A., Perini, A., Giorgini, P., Mylopoulos, J.: The Tropos Metamodel and its Use. Informatica 29(4), 401–408 (2005)

Automated Certification of Non-Interference in Rewriting Logic⋆

Mauricio Alba-Castro[1,2], María Alpuente[1], and Santiago Escobar[1]

[1] Universidad Politécnica de Valencia, Spain
{alpuente,sescobar}@dsic.upv.es
[2] Universidad Autónoma de Manizales, Colombia
malba@autonoma.edu.co

Abstract. In this paper we propose a certification technique for non-interference of Java programs based on rewriting logic, a very general *logical* and *semantic framework* efficiently implemented in the high-level programming language Maude. Non–interference is a semantic program property that prevents illicit information flow to happen. Starting from a basic specification of the semantics of Java written in Maude, we develop an information–flow extension of this operational Java semantics which allows us to observe non-interference of Java programs. Then we develop in Maude an abstract, finite-state version of the information-flow operational semantics which supports finite program verification. As a by–product of the verification, a certificate of non-interference is delivered which consists of a set of (abstract) rewriting proofs that can be easily checked by the code consumer using a standard rewriting logic engine.

1 Introduction

In the last decade, we have observed an increasing interest in formal methods designed for trusting code coming from untrusted sources. Proof-carrying code (PCC), originated by Necula [26], is a mechanism for ensuring the secure behavior of programs that is useful for general software development, and particularly advantageous for the development of mobile code. In PCC, a program contains both the code and an encoding of an easy–to–check proof whose validity entails compliance with a predefined security policy supplied by the code consumer. The security certificate is automatically generated by the software producer.In [1] we proposed an abstract PCC methodology for certifying Java source code that is based on rewriting logic. *Rewriting logic* [22] is a flexible and expressive *logical framework* in which a wide range of logics and models of computation can be faithfully represented. The methodology of [1] is as follows. Consider a

---

(concurrent) Java program together with a specification of the Java semantics, given as a term rewriting system. Given a safety property (i.e. a system property defined in terms of certain events not happening), the unreachability of the system states denoting the situation that should never occur allows us to infer the desired safety property. Unreachability analysis is performed using the standard Maude (breadth-first) search command, which explores the entire (finite) state space of the program. In the case when the unreachability test succeeds, the corresponding rewriting proofs demonstrating that those states cannot be reached are delivered as the expected outcome certificate. Certificates are encoded as (abstract) rewriting sequences that, together with an encoding in Maude of the abstraction, can be checked by standard reduction. Our methodology extends to other mainstream conventional languages or lower level languages (e.g. Java bytecode) by simply replacing the concrete semantics by a semantics for the programming language at hand; for instance, a rewriting logic semantics for Java bytecode can be found in [15].

In this paper, we extend the methodology of [1] to certify *confidentiality* by analysing *non-interference*. Confidentiality is a property by which information related to an entity or party is not made available or disclosed to unauthorized individuals, entities, or processes. However, an authorized accessing program can, on purpose or not, leak secret data in some improper way. To ensure that the program does not disclose secret data and fulfills *data confidentiality policies*, it is necessary therefore to analyse and control how information flows within the program. In this paper we focus on data confidentiality certification of Java programs. In order to express the non–interference safety policies for ensuring confidentiality, we use standard JML [21], a property specification language for Java modules. Each variable in the Java code is annotated with a confidentiality label that represents the confidentiality level of the variable and its data values.

The contributions of this paper are as follows:

- Starting from a basic specification of the semantics of Java written in Maude [14], we develop an information–flow extension of such an operational Java semantics which allows us to observe non-interference of Java programs, and is also written in Maude. For the best of our knowledge, a clear-cut semantics for Java programs dealing with non-interference was lacking. Much of previous work on ensuring Java non–interference has focused on enforcing it by appropriate information flow type systems by certifying and type preserving compilers [24,25] or bytecode typechecking [6].
- We provide an abstract, finite-state version of the information-flow operational semantics which supports finite program verification. Thanks to the different handling of rules and equations in Maude we do not suffer the state–space explosion of more traditional approaches (see [23]).
- Our Java certification methodology allows us to deal with some Java features not considered in the related literature ([20,30]): object fields, local

variables and arrays. We deal with values delivered by a `return` statement, a case not considered in [12,2,20,19]. We also consider `return` and `break` statements within conditional and iteration statements. Finally, for method invocations, we propagate context labels as proposed in [20], whereas they did not implemented it.
- Regarding the confidentiality label inferred for assignment instructions, we improve the granularity of the analysis over previous proposals [2,20] by inferring the confidentiality label during the memory update.
- As a by–product of the verification, a certificate of non-interference is delivered which consists of a set of (abstract) rewriting proofs that can be easily checked by the code consumer using a standard rewriting logic engine.

Section 2 introduces the rewriting logic semantics of `Java` considered in this paper. In Section 3 we present the extended information-flow rewriting logic semantics of `Java`, and Section 4 formalizes its abstract version. In Section 5 we propose our certification methodology, which we illustrate in Section 6 with some encouraging experimental results that demonstrate the practicality of our approach. Finally, we discuss the related work in Section 7, and Section 8 concludes.

2   The Rewriting Logic Semantics of Java

We assume some basic knowledge of term rewriting [29] and rewriting logic [22]. In the following, we briefly describe the rewriting logic semantics of Java given in [14] and used by the JavaFAN verification tool [15,16]. Its novelty and interest are based on the following advantages: (i) formal specifications provide a rigorous semantic definition for a language that can be mathematically scrutinized; (ii) such formal specifications can be developed with relatively little effort[1], even for large languages like Java [15] and the JVM [16]; (iii) the Maude programming language [10], which implements rewriting logic, provides a formal analysis infrastructure, so that its formal analysis tools (such as state-space breadth-first search and LTL model checking) become available for free for each programming language that is specified in Maude; and (iv) in spite of their generality, those formal analyses can be performed with competitive performance; see [15].

In [14], a sufficiently large subset of full Java 1.4 language is specified in Maude, including multithreading, inheritance, polymorphism, object references, and dynamic object allocation. However, Java native methods and many of the Java built-in libraries available are not supported. The specification of Java operational semantics is a rewrite theory, that is, a triple $\mathcal{R}_{\text{Java}} = (\Sigma_{\text{Java}}, E_{\text{Java}}, R_{\text{Java}})$, with Σ_{Java} an order-sorted signature, $E_{\text{Java}} = \Delta_{\text{Java}} \uplus B_{\text{Java}}$ a set of Σ_{Java}-equational axioms where B_{Java} are axioms such as associativity, commutativity and unity and Δ_{Java} is a set of terminating and confluent (modulo B_{Java}) Σ_{Java}-rewrite rules.

---

[1] See the different programming languages available at
http://fsl.cs.uiuc.edu/index.php/Rewriting_Logic_Semantics

Finally, R_{Java} is a set of Σ_{Java}-rewrite rules that are not required to be confluent and terminating. Intuitively, the sorts and function symbols in Σ_{Java} describe the static structure of the Java program state space as an algebraic data type, the equations in Δ_{Java} describe the operational semantics of its deterministic features, and the rules in $\mathcal{R}_{\text{Java}}$ describe its concurrent features. Following the rewriting logic framework [29,22], we denote by $u \rightarrow^r_{\text{Java}} v$ the fact that concrete terms u, v, denoting Java program states, are rewritten (at the top position, see [14]) by using r, which is either a rule in R_{Java} or an equation in Δ_{Java} both applied modulo B_{Java}. We simply write $u \rightarrow_{\text{Java}} v$ when no confusion can arise. We denote by $\rightarrow^*_{\text{Java}}$ the extension of $\rightarrow_{\text{Java}}$ to multiple rewrite steps, i.e., $u \rightarrow^*_{\text{Java}} v$ if there exist $u_1, \ldots, u_k$ such that $u \rightarrow_{\text{Java}} u_1 \rightarrow_{\text{Java}} u_2 \cdots u_k \rightarrow_{\text{Java}} v$.

Associativity, commutativity and unity (written ACU) axioms of binary operations in B_{Java} allow us to elegantly and effectively define (and implicitly implement) the crucial infrastructure of the Java programming language, including environments, threads, memory, input/output, synchronization information, and stores as well as the lookup operations on them. All of them are implemented as a (multi)-set union operation that builds up a "soup" of elements.

The rewrite theory $\mathcal{R}_{\text{Java}}$ is defined as terms of a concrete sort `State`, with the main state attributes (i.e., constructor symbols of the algebraic type `State`) such as `in`, `out`, `mem`, or `store`. They define an algebraic structure which is parametric w.r.t. a generic sort `Value` that defines all the possible values returned by Java functions, or stored in the memory, etc. For instance, the `int` and `bool` constructor symbols describe Java, integer and boolean values and are defined in Maude as "`op int : Int -> Value .`" and "`op bool : Bool -> Value .`", where `Int` and `Bool` are the internal built–in Maude sorts that define integer and boolean values. Intuitively, equations in Δ_{Java} and rules in R_{Java} are used to specify the changes to the program state, i.e., the changes to the memory, threads, input/output, etc. The semantics of Java is defined modularly, i.e., different features of the language are defined in separate Maude modules so to ease extensions and maintenance [14].

The state space associated to a rewrite theory is determined in Maude only by the program rules, since equations are deterministic. That is, rules and equations are applied in the same way but Maude only keeps track of the rules applied and omits the information about the equations applied. Therefore, the number of rules and equations is relevant since the smaller the number of rules, the more efficient the verification analysis, because the search space is smaller. According to [14], the Java operational semantics contains about 424 equations and only 7 rules, which considerably saves memory and execution time.

The semantics of Java is defined in a *continuation-based style* [23]. Continuations maintain the control context of each thread, which explicitly specifies the next steps to be performed by the thread. Continuations are a typical technique to transform the uncontrollable control context into controllable data context, by stacking the sequence of actions that still need to be executed. Once the expression e on the top of a continuation ($e \rightarrow k$) is evaluated, its result will be passed to the remaining continuation k. For instance, the Java addition operation on

```
---First evaluate arguments
eq k((E + E') -> K) = k((E, E') -> (+ -> K)) .
---Then, compute addition
eq k((int(I), int(I')) -> (+ -> K)) = k(int(I + I') -> K) .
```

Fig. 1. Continuation-based equations for Java addition operator on integers

```
---First obtain location in store from variable name
eq k(Var -> K) env([Var, Loc] Env) obj(Obj)
    = k(#(Loc) -> K) env([Var, Loc] Env) obj(Obj) .
---Then obtain value stored in such location
rl t(k(#(Loc) -> K) id(I) TC) store(Loc, Value, -1] Store)
=> t(k(Value -> K) id(I) TC) store([Loc,Value,-1] Store) .
```

Fig. 2. Continuation-based equation and rule for variable content retrieval

```
---Obtain variable location while keeping expression in the continuation
eq k((Var = E) -> K) = k(getLocation(Var) -> (=(E) -> K)) .
---Once the location is obtained, evaluate expression keeping location
eq k(Loc -> (=(E) -> K)) = k(E -> (=(Loc) -> K)) .
---Once the expression is computed, assign to location
eq k(Value -> (=(L) -> K)) = k([Value -> L] -> (V -> K)) .
---General procedure to update the shared memory
rl t(k([Value -> Loc] -> K) id(I) TC) store([Loc,Value',-1]  ST)
=> t(k(K) id(I) TC) store([Loc, shared(Value), -1] ST) .
```

Fig. 3. Continuation-based equations and rules for Java assignment operator

Java integers is specified[2] in Figure 1 using continuations, where k is the con-
structor symbol used to denote a continuation in a thread, -> is the constructor
symbol used to concatenate continuations, int is the constructor symbol used
to denote a Java integer, and + with arity[3] 2 and inside the constructor int is
the Maude addition symbol, whereas + with arity 2 but outside the constructor
int is the Java addition symbol, and + with arity 0 is a continuation symbol
used to remember that the Java addition action is being stacked.

    Another important aspect of the semantics is the use of Java variables. In
Figure 2 we show how the contents of a Java variable is retrieved from the store
in the Java state. The assignment operator for Java variables is specified in
Figure 3. Note that the relative order among assignment and retrieval operations
is relevant since multiple threads can try to concurrently assign a value to a

---

[2] The Maude syntax is almost self-explanatory [10]. The general point is that each
syntactic element –e.g. a sort, an operation, an equation, a rule– is declared with an
obvious keyword: **sort**, **op**, **eq**, **rl**, etc., ended by a space and a period. We denote
variables with uppercase letters whereas lowercase letters denote Maude constructor
symbols.

[3] The Maude syntax allows overloading of operators, with different arities.

```
---Evaluates boolean expression keeping the then and else statements
eq k((if E S else S' fi) -> K) = k(E -> (if(S, S') -> K)) .
eq k(bool(true) -> (if(S, S') -> K)) = k(S -> K) .
eq k(bool(false) -> (if(S, S') -> K)) = k(S' -> K) .
```

Fig. 4. Continuation-based equations for if-then-else statement

```
eq t(k(V -> return -> K) holds(Ll') env(Env')
    fstack( fsi(K', (holds(Ll) env(Env) TC)) Fstack) TC')
= t(k(releaseEnv(Env') -> release(Ll, Ll') -> (V -> K')) holds(Ll)
    env(Env) fstack(Fstack) TC) .
```

Fig. 5. Continuation-based equation for return statement

variable or read its value from the store; hence a rule, instead of an equation, is used to represent the physical assignment as well as the physical retrieval from the store. In other words, the assignment operator and the retrieval of a variable value are non-deterministic due to the presence of different threads, and are specified with Maude rules instead of Maude equations.

A relevant aspect of the Java semantics for non-interference is the if-then-else statement, shown in Figure 4. Also important for non-interference is the semantic specification of the Java return statement, shown in Figure 5. The return statement restores the previous environment, the held locks and the local thread state from the function stack, and then updates the continuation to release the method local environment and locks, and to restore them from the stack.

3   An Information-Flow Rewriting Logic Semantics for Java

In this section, we develop an information-flow, extended version of the rewriting logic semantics of Java recalled in Section 2. In order to motivate the new semantics with appropriate Java examples, let us first briefly recall the Java modeling language JML [21].

JML is a behavioral interface specification language that accepts Java built-in operators in order to relieve Java programmers from the encumbrance of learning a language-independent formal specification language like OCL [9]. As an interface specification language, JML can describe the names and static information found in Java declarations of Java modules with preconditions (in `requires` clauses), normal postconditions (in `ensures` clauses), invariants (in `invariant` clauses) and assert statements (with the `assert` clauses), that express first-order logic statements. As a behavior specification language, JML can also describe how the module will behave when assertions are intermixed with the Java code.

The text of an annotation could be either in one line, after the marker //@, or in many lines enclosed between the markers /*@ and @*/. In this paper, we

consider lightweight specifications using the simplest JML clauses for Java methods and type specification of the simplest language level 0 (there are six levels of annotations). We use two method specification clauses, the `ensures` clause to indicate the required confidentiality label expected by the code consumer on the result of a function and the `requires` clause to indicate any precondition (`Low` or `High`) on the confidentiality label of a function input parameter. We use `assert` clauses to indicate the confidentiality label of local variables. The JML specifications written as code annotations are treated like Java comments that are ignored by traditional compilers whereas they are automatically handled by our certification methodology.

The problem of verification and certification of program non-interference using information flow analysis, was first considered in [12]. The flow policy is usually represented by a flow relation between security classes that specify the permissible flows between them. Each storage object (constant, scalar variable, array, or file) is assigned to a security class. This assignment is static and inferred from the declarations in the program. A non-interference policy means that variables have fixed confidentiality levels and that inputs with high confidentiality level do not influence outputs of lower confidentiality level [28,7,30,13]. This means that the values stored in the high confidentiality variables cannot flow to the lower confidentiality variables. It is implicitly assumed that constants appearing in a Java program always have the lowest confidentiality level, i.e., the considered Java program is authorized to access secret data but it does not contain secret data in its code.

A non-interference policy can be represented by a relation $\langle L, \leq \rangle$ and a labeling function $Lab : Var \rightarrow L$, where L is the finite set of confidentiality levels, $\leq$ is a partial order between confidentiality levels, and Var is the finite set of program variables. Usually there are two confidentiality levels, i.e., $Conf = \{\texttt{Low}, \texttt{High}\}$, representing respectively the public non-secret data (low confidentiality) and the secret private one (high confidentiality), so that $\texttt{Low} \leq \texttt{High}$. $\langle Conf, \leq \rangle$ forms a lattice where `Low` is the greatest lower bound or *bottom* ($\bot$), `High` is the least upper bound or *top* ($\top$), and the *join* operator ($\sqcup$) is defined as $\texttt{Low} \sqcup \texttt{Low} = \texttt{Low}$ and, otherwise, $X \sqcup Y = \texttt{High}$. This means that values of `Low` labeled variables can flow to `High` labeled variables, but also that values of `High` labeled variables cannot flow to `Low` labeled variables. The information that flows in a program is either explicit or implicit. An explicit illicit flow is caused by assignment statements in which the values of expressions with high variables are assigned to low variables [28,19], shown in the following.

Example 1. Consider the simple Java program borrowed from [20]. We use the `requires` and `ensures` clauses and the operator `\result`. This example has an illegal direct flow from the variable `high` with confidentiality label `High` to the variable `low` with label `Low` in the first assignment statement. Nevertheless, the final outcome is an integer constant value with the `Low` confidentiality label, so that the final output is legal.

```
public int mE1(int high,int low) { low = high; low = 2; return low;}
/*@ requires high == High && low == Low; @ ensures \result == Low; @*/
```

Another explicit illicit flow might occur in function and procedure invocations, shown in the following example.

Example 2. Consider the following `Java` program borrowed from [30], whose method `mE522` calls the method `decrementing` with two parameters. The explicit illicit flow occurs at the `decrementing` invocation, which passes the `High` variable `high` to the `Low` parameter `i`.

```
int decrementing(int high,int i)  { high = high - 1; return i; }
/*@ requires high == High && i == Low; @ ensures \result == Low;   @*/
int mE522(int high,int low){ low=decrementing(high,high); return low;}
/*@ requires high == High && low == Low; @ ensures \result == Low; @*/
```

The common source for implicit illicit flows, which can often go unnoticed [28,19], are control flow statements guarded by boolean expressions with variables of confidentiality `High`, shown in the following example.

Example 3. Consider a Java program, also borrowed from [30], with an `if` control flow statement. If the actual data passed to the `low` parameter is not `0` and the returned value is `0`, then we know that the secret variable `high` has a value greater than `2`. Note that the notion of a global confidentiality label (called context label) being updated after each conditional expression is necessary for proper verification of such an implicit leaking [12,20,19].

```
public int mE2(int high,int low) { if (high > 2) low = 0; return low;}
/*@ requires high == High && low == Low; @ ensures \result == Low; @*/
```

In order to avoid false positives, we will dynamically restore the previous global confidentiality label after each conditional construction, as shown in the following example.

Example 4. Consider a slight modification of Example 3 where the returned value does not actually depend on the value of the `High` variable `high`. That is, the variable `j` is affected by the value of the variable `high` but the variable `low` used in the return expression is not.

```
public int mE2*(int high,int low)
/*@ requires high == High && low == Low; @ ensures \result == Low; @*/
{int j=0;low = 0; /*@assert j==Low;@*/ if (high>2) j = 1; return low;}
```

We describe the information-flow extended version of the rewriting logic semantics of Java by the rewrite theory $\mathcal{R}_{\mathrm{Java^E}} = (\Sigma_{\mathrm{Java^E}}, E_{\mathrm{Java^E}}, R_{\mathrm{Java^E}})$, $E_{\mathrm{Java^E}} = \Delta_{\mathrm{Java^E}} \uplus B_{\mathrm{Java^E}}$ and its corresponding $\rightarrow_{\mathrm{Java^E}}$ rewriting relation. In the new semantics, program data do not only consist of standard concrete values but each value is decorated with its corresponding confidentiality label. Our approach consists of extending $\mathcal{R}_{\mathrm{Java}}$ (taking advantage of its modularity) by conveniently complementing the concrete domain Value as to consider the extended domain Value $\times$ LValue.

We introduce the sort LValue to represent values Low and High. We write
<Value,LValue> for a pair of a concrete value and its corresponding confiden-
tiality level label. We must also provide appropriate versions of the Java con-
structions and operators for the new extended domain. Recall that the symbols
env and store are the constructor symbols used by the original Java rewriting
semantics for the program environment and the memory store, respectively. The
new constructor symbol lenv is used to store the global confidentiality level
(context label).

Regarding confidentiality, we consider the following Java expressions as a spe-
cial case of the evaluation: literal constants, variable access, binary operators,
assignment expressions, unary pre– and post–fix operators and return expres-
sions. Thanks to the modularity of the rewriting logic approach to formalizing
program semantics, our changes to the semantics of Section 2 are incremental
and minimal. Variables receive an initial confidentiality level, which is stored in
the memory when the variable or parameter is created. Any operation writing a
value in a memory location stores, as the confidentiality label for such variable,
the join of the confidentiality label of the value to be written and the context
level at that moment, as shown in Figure 6. The label of any Integer constant
value, shown in Figure 7, is Low as expected, since constants are public data.
The label of a variable is the confidentiality label of its value in memory and,
therefore, the original equations of Figure 2 need no revision.

For the dynamic labeling of the context, the initial context label of any thread
is Low as usual [12,2,20,19]. Method invocation propagates context label with-
out changes as proposed in [20]. Assignment and expression statements do not
change context label. The context label may change only because of conditional
control flow statements to control indirect information flow, as shown in Fig-
ure 8. The current context label is stored in the continuation using the new
restoreLEnv continuation operator, which restores the previous context label
upon execution; see the last equation of Figure 8. According to [12,2,20,19], the
evaluation of boolean expressions returns a confidentiality level associated to the
resulting true or false value and, possibly, a modified context label. We up-
date the context label in order to reflect the confidentiality level returned by the
evaluation of the boolean expression, and then the two branches of the

```
rl t(k([< Value,LValue > -> L] -> K) id(I) lenv(LEnv) TC)
   store([L, Value', -1] ST)
=> t(k(K) id(I) lenv(LEnv) TC)
   store([L, shared(< Value,LValue join LEnv >), -1] ST) .
```

Fig. 6. Rule for the extended memory write

```
eq k(i(I) -> K) = k(<int(I),Low> -> K) .
eq k(b(B) -> K) = k(<bool(B),Low> -> K) .
```

Fig. 7. Equations for extended constant evaluation

```
--- First evaluates the boolean expression
--- and keeps the current context label
eq k((if E S else S' fi) -> K) lenv(LEnv)
    = k(E -> if(S, S') -> restoreLEnv(LEnv) -> K) lenv(LEnv) .
--- Then updates the context label
eq k(< bool(true), LValue > -> (if(S, S') -> K)) lenv(LEnv)
    = k(S -> K) lenv(LEnv join LValue) .
eq k(< bool(false), LValue> -> (if(S, S') -> K)) lenv(LEnv)
    = k(S' -> K) lenv(LEnv join LValue) .
--- New equation to restore previous context label
eq k(restoreLEnv(LEnv) -> K) lenv(LEnv') = k(K) lenv(LEnv) .
```

Fig. 8. Continuation-based equations for the extended if-then-else statement

```
eq t(k(< V,LValue > -> return -> K) holds(Ll') env(Env')
      lenv(LEnv) fstack(fsi(K',(holds(Ll) env(Env) TC)) Fstack) TC')
= t(k(releaseEnv(Env') -> release(Ll,Ll') -> (<V,LValue join LEnv> -> K'))
      holds(Ll) env(Env) fstack(Fstack) TC) .
```

Fig. 9. Continuation-based equation for return statement

conditional expression will use such a new context confidentiality label for memory updates.

The extended semantics for the `return` statement considers not only the confidentiality label of the value to be returned but also the context confidentiality level, as shown in Figure 9.

4   The Abstract Rewriting Logic Semantics of Java

In this section, we develop an abstract version of the extended rewriting logic semantics of Java developed in Section 3, described by the rewrite theory $\mathcal{R}_{\text{Java}\#} = (\Sigma_{\text{Java}\#}, E_{\text{Java}\#}, R_{\text{Java}\#})$, $E_{\text{Java}\#} = \Delta_{\text{Java}\#} \uplus B_{\text{Java}\#}$ and its corresponding $\rightarrow_{\text{Java}\#}$ rewriting relation. As in Section 3, our approach for the abstract Java semantics consists of extending the original theory $\mathcal{R}_{\text{Java}}$ (taking advantage of its modularity) by abstracting the domain to $\text{LValue} = \{\text{Low}, \text{High}\}$, and introducing approximate versions of the Java constructions and operators tailored to this domain.

An *abstract interpretation* (or abstraction) [11] of the program semantics is given by an *upper closure operator* $\alpha : \wp(\text{State}) \rightarrow \wp(\text{State})$, that is *monotonic* (for all $SSt_1, SSt_2 \in \wp(\text{State})$, $SSt_1 \subseteq SSt_2$ implies $\alpha(SSt_1) \subseteq \alpha(SSt_2)$), *idempotent* (for all $SSt \in \wp(\text{State})$, $\alpha(SSt) \subseteq \alpha(\alpha(SSt))$), and *extensive* (for all $SSt \in \wp(\text{State})$, $SSt \subseteq \alpha(SSt)$). The intuition of this definition is that each Java program state $St \in \text{State}$ is abstracted by its closure $\alpha(\{St\})$. Closure operators have many interesting properties. For instance, when the considered domain is a complete lattice, e.g. $\langle \alpha(\text{State}), \subseteq \rangle$, each closure operator is uniquely determined by the set of its fixed points. In the context of abstract interpretation, closure operators are important because abstract domains can be

equivalently defined by using them or by Galois insertions, as introduced in [11]. Let $\iota : \alpha(\wp(\mathsf{State})) \to A$ be an isomorphism. Then, given an upper closure operator $\alpha : \wp(\mathsf{State}) \to \wp(\mathsf{State})$, the structure $(\wp(\mathsf{State}), \alpha \circ \iota, \iota^{-1}, A)$ is a Galois insertion, where $\alpha \circ \iota$ and ι^{-1} are the abstraction and concretization functions, respectively (see [11] for further details). In our approach [1], we only need an abstract function for each Java variable name x, e.g., $\alpha_{\mathbf{x}} : \wp(\mathsf{Value}) \to \wp(\mathsf{Value})$ and homomorphically extend those abstract functions to an abstract function $\alpha : \wp(\mathsf{State}) \to \wp(\mathsf{State})$. Indeed, for each variable x, α abstracts the values stored in the Java memory for x using $\alpha_{\mathbf{x}}$.

In this section, our abstraction function $\alpha : \wp(\mathsf{State}^E) \to \wp(\mathsf{State}^E)$ is an homomorphism extension to sets of states of the function $2nd : \mathsf{Int} \times \mathsf{LValue} \to \mathsf{LValue}$, meaning that we disregard the actual values of data.

In the abstract Java semantics, several alternative computation steps of $\to_{\mathrm{Java^E}}$ are mimicked by a single abstract computation step of $\to_{\mathrm{Java\#}}$, reflecting the fact that several distinct behaviors are compressed into a single abstract state (i.e. set of states). Consider e.g. the approximate version of the Java > operator. For the case of comparing two abstract states SSt_1 and SSt_2 in $\wp(\mathsf{State}^E)$ for >, an (inaccurate) approximation of the result is the set {<true,Low>, <true,High>, <false,Low>, <false,High>}, since all combinations are possible when we would compare concrete states. As explained in [1], the instrumentalization of the Java semantics to deal with a set of states instead of one single state implicitly means too many modifications.Therefore, we adopt a different approach. When several $\to_{\mathrm{Java^E}}$ rewrite steps are mimicked by a single abstract rewriting state leading to an abstract Java state, and those rewrite steps apply different rules or equations, we use concurrency at the Maude level. That is, we add rules to $R_{\mathrm{Java\#}}$ to reflect the different possible evolutions of the system.

Now, we are ready to formalize the abstract rewriting relation $\to_{\mathrm{Java\#}}$, which intuitively develops the idea of applying only one rule or equation from the concrete Java semantics to an abstract Java state while exploring the different alternatives in a non-deterministic way. By abuse, we denote the abstraction of a rule $\alpha(\{l\}) \to \alpha(\{r\})$ by $\alpha(\{l\} \to \{r\})$.

Definition 1 (Abstract rewriting). *Let $\alpha : \wp(\mathsf{State}^E) \to \wp(\mathsf{State}^E)$ be an abstraction. We define the approximated version of rewriting $\to_{\mathrm{Java\#}} \subseteq \wp(\mathsf{State}^E) \times \wp(\mathsf{State}^E)$ by:*

> $SSt_1 \to_{\mathrm{Java\#}} SSt_2 \qquad using\ \alpha(\{l\} \to \{r\}) \in (R_{\mathrm{Java\#}} \cup \Delta_{\mathrm{Java\#}})$
> $iff\ \forall u \in \alpha(SSt_1), \exists v \in SSt_2\ s.t.\ u \to_{\mathrm{Java^E}} v,\ using\ l \to r \in R_{\mathrm{Java^E}} \cup \Delta_{\mathrm{Java^E}}.$

We denote by $\to_{\mathrm{Java\#}}^*$ the extension of $\to_{\mathrm{Java\#}}$ to multiple rewrite steps. The following result follows straightforwardly by monotonicity, idempotency, and extensivity of the upper closure operator α.

Theorem 1 (Correctness & Completeness). *Let $\alpha : \wp(\mathsf{State}^E) \to \wp(\mathsf{State}^E)$ be an abstraction. Let $SSt_1, SSt_2 \in \wp(\mathsf{State}^E)$. If $SSt_1 \to_{\mathrm{Java\#}}^* SSt_2$, then for all $u \in \alpha(SSt_1)$, there is $v \in SSt_2$ such that $u \to_{\mathrm{Java^E}}^* v$. Let $St_1, St_2 \in \mathsf{State}^E$. If $St_1 \to_{\mathrm{Java^E}}^* St_2$, then there exists $SSt_3 \in \wp(\mathsf{State})$ s.t. $\alpha(St_1) \to_{\mathrm{Java\#}}^* SSt_3$ and $St_2 \in SSt_3$.*

```
rl t(k([LValue -> L] -> K) TC) store([L, Value'] ST)  lenv(LEnv)
=> t(k(K) TC) store([L, LValue join LEnv] ST) lenv(LEnv) .
```

Fig. 10. Abstract rule for the memory write

```
eq k(i(I) -> K) = k(Low -> K) .
```

Fig. 11. Abstract equation for constant evaluation

```
eq t(k(LValue -> return -> K) holds(Ll') env(Env') lenv(LEnv)
       fstack(fsi(K', (holds(Ll) env(Env) TC)) Fstack) TC')
= t(k(releaseEnv(Env') -> release(Ll, Ll') -> LValue join LEnv -> K')
       holds(Ll) env(Env) lenv(LEnv) fstack(Fstack) TC) .
```

Fig. 12. Abstract equation for return statement

Therefore, in the following, we abstract the semantics of Section 3 so that (i) each pair <Value,LValue> in the equations and rules are approximated by the second component LValue; and (ii) those equations that cannot be proved confluent[4] after the transformation are transformed into rules, to reflect the different possible rewrites denoted by an abstract state. We additionally add a rule to deal with confidentiality values alone, shown in Figure 10. For the label of an integer constant value, we return Low as expected, shown in Figure 11. Note that this can be still expressed by means of an equation, since confluence and coherence [10] are preserved. The label of a variable is the confidentiality label of its value in memory and, therefore, we keep the original equations of Figure 2. Analogously, regarding conditionals the equations of Figure 8 still work. Since pairs <Bool,LValue> are handled by the return statement, its abstract semantics is still given by the equation of Figure 9.

However, we must add an extra equation to deal with confidentiality values alone, shown in Figure 12, which is almost identical to equations in Figure 9. The following example illustrates the mechanization of the abstract Java semantics.

Example 5. Consider the Java program together with the call to the main function of Example 1. In the search command below, we ask for all possible values returned by the main Java function of Example 1.

```
search in PGM-SEMANTICS-ABSTRACT :
 java((preprocess(default class t('Safe1NonInterference) imports nil
 extends Object implements none {(public static) int 'mE1((int d('high)),
 (int d( 'low))) throws( noType) {((10@('low = 'high;)) 1 @('low = i(2);))
 12@ return 'low ;} (public static) void 'main (t('String) [] d('args))
 throws(noType) {5 @ ('System . 'out . 'println < 'mE1 < i(1), i(0)
```

---

[4] See the Church-Rosser checker for Maude available at
   http://www.lcc.uma.es/~duran/CRC/

```
 > > ;)}}) noType . 'main < new string [i(0)] > noVal))
=>! X:Output .
Solution 1 (state 1)
states: 2  rewrites: 248 in (7ms real) (0 rewrites/second)
X:Output --> pl(Low)
No more solutions.
```

The search command returns that one unique possible abstract Java execution trace is possible, which leads to the abstract value `Low` as the outcome of the Java instruction "`System.out.println(mE1(1,0));`".

5   Certifying Java Source Code

Example 5 above illustrates how our methodology generates a safety certificate which essentially consists of the set of (abstract) rewriting proofs of the form $t_1 \rightarrow^{r_1}_{\text{Java\#}} t_2 \cdots \rightarrow^{r_{k-1}}_{\text{Java\#}} t_k$ that implicitly describe the program states which can (and cannot) be reached from a given (abstract) initial state. Since these proofs correspond to the execution of the abstract Java semantics specification, which is made available to the code consumer, the certificate can be inexpensively checked on the consumer side by any standard rewrite engine by means of a rewriting process that can be very simplified. Actually, it suffices to check that each abstract rewriting step in the certificate is valid and that no rewriting chain has been disregarded, which essentially amounts to use the matching infrastructure available within the rewriting engine. Note that, according to the different treatment of rules and equations in Maude, where only transitions caused by rules create new states in the space state, an extremely reduced certificate can be delivered by just recording the rewrite steps given with the rules, while the rewritings using the equations are omitted.

6   Experiments

The certification methodology presented here has been implemented in Maude[5]. In developing and deploying the system, we fixed the following requirements: 1) define a system architecture as simple as possible, 2) make the certification service available to every Internet requester, and 3) hide the technical details from the user. The prototype system offers a rewriting-based program certification service, which is able to analyze safety properties of Java code which are related to the safe use of types and with program non-interference.

   In Table 1, we study three key points for the practicality of our approach: the size of the reduced certificate versus the Java source code, the size of the reduced certificate versus the size of the full certificate and the relative efficiency of producing certificates w.r.t. their generation. The experiments have been performed on a MacBook with 2 Gb RAM. Programs mE1 and mE2* are Java programs

---

[5] It is publicly available at
   http://www.dsic.upv.es/users/elp/toolsMaude/rewritingLogic.html

Table 1. Certificate sizes, and certification times

Code example	Full Cert. Size (Kb)	Red. Cert. Size (Kb)	Size Relation (Red/Source)	Full C. Gen. Time (ms)	Red. Cert. Gen. Time (ms)
mE1	443	2.62	2.29	16	3.5
mE2*	561	4.65	4.97	213.5	28.5
mE2v1E1	615	4.74	4.42	267	47
mE2mE1	578	4.66	3.98	245.5	31
mE522v1	604	2.91	1.67	14	3.5
mE3	553	4.55	4.33	377	57.5

containing the methods of Examples 1, and 4, respectively. Programs mE2mE1 and mE2v1E1 contain methods which are a sequential composition respectively of Examples 3 and 1 and of a variation of Example 3 with Example 1. Program mE522v1 is a non-interferent variation of Example 2. Program mE3, borrowed from [20], is similar to program mE2mE1 using the equality == operator.

The two columns for "Full Cert." show the size in Kbytes and the generation time, respectively, for the full certificates. Similarly for the two columns of "Red. Cert.". Running times are given in milliseconds and were averaged over a sufficient number of iterations. The experiments are very encouraging, since they show that the reduction in size of the certificate is very significant in all cases, ranging the quotient "Red. Cert. Size/Full Cert. Size" from 8.2% in mE2* to 4.8% for mE522v1. When we compare the time employed to generate the full and reduced certificates we have that the reduced certificate generation time takes only 12% of the full certificate generation time.

7   Related Work

Standard Java verification tools that use standard JML [21] as property specification language do not support non–interference certification. Some sophisticated non–interference policies can be expressed by using the JML extensions of the Krakatoa Java verification tool [13]. These JML extensions were developed for Hoare-style assertions regarding program self-composition [4], which means duplicating the code of the program thus requiring to distinguish the same program variables in its two runs. However, non-interference policies that require labeling data variables with confidentiality levels cannot be expressed by using these JML extensions. The confidentiality aspect of non–interference is expressible using the JML specification pattern suggested in [20,30]. Unfortunately, this proposal abuses notation by identifying confidentiality levels with values of the program variables, and it cannot be applied in all cases [30].

Although non–interference has not been considered in current PCC implementations, there are some (not yet implemented) proposals for a subset of Java [5], Java bytecode [27,7,6] and some simple, toy imperative languages [19,8]. However, none of them uses JML to express non–interference policies. [5] proposes a type system, so that a compiler preserving the information flow type could be developed for Java source code. [6] defines the first information flow type system for a sequential JVM-like language that guarantees non–interference in type

checked programs. The soundness was proven by using the theorem prover Coq, and a certified verifier was extracted from the proof. The certified verifier could be used as a PCC proof checker in the consumer's side. Although we consider only two security levels we can easily extend our methodology to the multilevels of confidentiality of [6]. Our global policies attach security levels to object fields but we do not consider heaps (where objects and their fields are stored). Our local policies are very flexible since the security levels of local variables and parameters of methods may change temporarily as in [19,20].

Some proposals also exist for non–interference verification that are based on information flow analysis by using abstract interpretation [2,3,18,17]. However, these proposals do neither generate a proof as a result of the verification nor use JML to express non–interference policies. The idea of first enriching the original semantics of the language by pairing each data value to its security level, and then approximate it by only considering the security level is also in [2]. A similar idea is developed in [17] where input and ouput channels are associated with security levels. Regarding the values delivered by a `return` statement, our work is similar to [3] and [17]. [18] introduces the notion of abstract non–interference: abstract non–interference can be obtained by weakening the standard notion of non–interference by making it parametric relatively to input/output abstractions. In abstract non–interference, the abstract domains encode the allowed flows that characterize the degree of precision of the knowledge of a potential attacker observing the data.

To verify non–interferent Java source programs, there are other type based proposals that do not use JML either to specify information flow policies, namely the Java extensions JFlow [24] and Jif [25]. These compilers produce secure Java source code for verified programs written in the languages JFlow and Jif by first applying static information flow analysis based on type systems to track the correspondence between the confidential information and the policies that restrict its use.

8   Conclusion

As far as we know, we propose the first sound and complete, fully automatic certification technique that applies to certifying non-interference of Java source code. The proposed methodology features quality attributes (notably reliability and security, but also good performance) through rigorous mechanisms which integrate a wide range of well-established programming language techniques (abstract interpretation, program semantics, meta-programming, etc). Our approach is based on a rewriting logic semantics specification of a sufficiently large subset of the full Java 1.4 language [14]. Certificates are encoded as (abstract) rewriting sequences which can be checked in the abstract Java semantics written in Maude on the consumer side by standard reduction. Our certification methodology extends to other programming languages by simply replacing the concrete semantics by a semantics for the programming language at hand, see [23].

Our work can be easily extended to cope with procedure methods, exceptions, heaps, and multithreading, since they are considered in the Java rewriting logic

semantics. Since we inherit from Maude and the Java rewriting semantics its competitive performance (see [15]), we have a scalable technique that can be further refined to certifying industrial complex Java programs.

References

1. Alba-Castro, M., Alpuente, M., Escobar, S.: Automatic certification of Java source code in rewriting logic. In: Leue, S., Merino, P. (eds.) FMICS 2007. LNCS, vol. 4916, pp. 200–217. Springer, Heidelberg (2008)
2. Barbuti, R., Bernardeschi, C., De Francisco, N.: Abstract interpretation of operational semantics for secure information flow. Information Processing Letters 83(22), 101–108 (2002)
3. Barbuti, R., Bernardeschi, C., De Francisco, N.: Checking security of Java bytecode by abstract interpretation. In: SAC 2002, pp. 229–236. ACM, New York (2002)
4. Barthe, G., D'Argenio, P., Rezk, T.: Secure information flow by self-composition. In: CSFW 2004, pp. 100–114. IEEE, Los Alamitos (2004)
5. Barthe, G., Naumann, D., Rezk, T.: Deriving an information flow checker and certifying compiler for Java. In: SSP 2006, pp. 230–242. IEEE, Los Alamitos (2006)
6. Barthe, G., Pichardie, D., Rezk, T.: A certified lightweight non-interference Java bytecode verifier. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 125–140. Springer, Heidelberg (2007)
7. Barthe, G., Rezk, T.: Non-interference for a JVM-like language. In: TLDI 2005, pp. 103–112 (2005)
8. Beringer, L., Hofmann, M.: Secure information flow and program logics. In: IEEE CSF 2007, pp. 233–248 (2007)
9. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Rustan, K., Leino, M., Poll, E.: An overview of JML tools and applications. IJSTTT 7(3), 212–232 (2005)
10. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
11. Cousot, P., Cousot, R.: Systematic Design of Program Analysis Frameworks. In: POPL 1979, pp. 269–282 (1979)
12. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM 20(7), 504–513 (1977)
13. Dufay, G., Felty, A., Matwin, S.: Privacy-sensitive information flow with JML. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS, vol. 3632, pp. 116–130. Springer, Heidelberg (2005)
14. Farzan, A., Chen, F., Meseguer, J., Rosu, G.: JavaRL: The rewriting logic semantics of Java (2007),
http://fsl.cs.uiuc.edu/index.php/Rewriting_Logic_Semantics_of_Java
15. Farzan, A., Chen, F., Meseguer, J., Rosu, G.: Formal analysis of Java programs in JavaFAN. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 501–505. Springer, Heidelberg (2004)
16. Farzan, A., Meseguer, J., Rosu, G.: Formal JVM code analysis in JavaFAN. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, pp. 132–147. Springer, Heidelberg (2004)
17. De Francesco, N., Martini, L.: Instruction-level security typing by abstract interpretation. Int. J. of Inf. Sec. 6(2-3), 85–106 (2007)

18. Giacobazzi, R., Mastroeni, I.: Abstract non-interference: Parameterizing non-interference by abstract interpretation. In: POPL 2004, pp. 186–197 (2004)
19. Hunt, S., Sands, D.: On flow-sensitive security types. In: POPL 2006, pp. 79–90 (2006)
20. Jacobs, B., Pieters, W., Warnier, M.: Statically checking confidentiality via dynamic labels. In: WITS 2005, pp. 50–56 (2005)
21. Leavens, G., Baker, A., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. ACM SIGSOFT Software Engineering Notes 31(3), 1–38 (2006)
22. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. TCS 96(1), 73–155 (1992)
23. Meseguer, J., Rosu, G.: The rewriting logic semantics project. TCS 373(3), 213–237 (2007)
24. Myers, A.C.: Jflow: Practical mostly-static information flow control. In: POPL 1999, pp. 228–241 (1999)
25. Myers, A.C., Nystrom, N., Zheng, L., Zdancewic, S.: Jif: Java information flow. Software release (2001), http://www.cs.cornell.edu/jif
26. Necula, G.C.: Proof carrying code. In: POPL 1997, pp. 106–119 (1997)
27. Rose, E.: Lightweight bytecode verification. J. Autom. Reason. 31(3-4), 303–334 (2003)
28. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. on Selected Areas in Comm. 21(1), 5–19 (2003)
29. TeReSe (ed.): Term Rewriting Systems. Cambridge U. Press, Cambridge (2003)
30. Warnier, M.E.: Language Based Security for Java and JML. PhD thesis, Radboud University Nijmegen (2005)

Formal Verification of Safety Functions by Reinterpretation of Functional Block Based Specifications[*]

Erzsébet Németh and Tamás Bartha

Systems and Control Laboratory,
Computer and Automation Research Institute, Hungarian Academy of Sciences
Kende u. 13–17, H-1111 Budapest, Hungary
{nemethe,bartha}@sztaki.hu

Abstract. This paper presents the formal verification of a primary-to-secondary leaking (abbreviated as PRISE) safety procedure in a nuclear power plant (NPP). The software for the PRISE is defined by the Function Block Diagram specification method.

Our approach to the formal verification of the PRISE safety procedure is based on the coloured Petri net (CPN) representation. The CPN model of the checked software is derived by reinterpretation from the FBD diagram, using a pre-developed library of CPN subnets. This results in a high-level, hierarchical coloured Petri net, that has an almost identical structure to the FBD specification.

The state space of the CPN model was drastically reduced by "folding" equivalent states and trajectories into equivalence classes. Some of the safety properties could be proven based on the SCC (strongly connected components) graph of the reduced state space. Other properties were proven by CTL temporal logic based model checking.

1 Introduction

Nuclear power plants (NPPs) are highly safety-critical and complex systems, where the correct operation of the safety procedures is of great importance. The plant protection systems must therefore satisfy high safety requirements and minimize spurious forced outages.

Digital Control Systems (DCSs) and Programmable Logic Controllers (PLCs) are widely utilized for control and automation functions in safety-related applications. The experts who create the system specification favour graphical specification formalisms, such as the function block diagram (FBD) defined by the IEC Standard 61131-3 [14]. Graphical tools help to cope with the complexity and the concurrent nature of the plant control and monitoring software.

Most of these graphical specification languages are executable, and the experts use simulation and extensive testing to verify the behaviour during the

---

development process. However, simulation and testing does not guarantee exhaustiveness and completeness. Therefore, formal modelling and analysis of the safety functions is required to prove that the system cannot enter into unsafe states, remains operational (no deadlock or livelock occurs), does not trigger the safety actions unnecessarily (no spurious activation can take place), but it always triggers them when required (no activation masking can happen).

There are several results in the literature to the problem of formal verification and validation of DCS and PLC-based industrial control and monitoring systems. A few examples for formal modelling, verification and validation of PLC applications in various specification languages [6] include:

- The authors of [12] describe an application of formal methods in the development of safety critical software in the nuclear industry over a thirteen year period. The work makes use of *tabular specifications*, and applies formal methods "all the way down" from requirements, through design, implementation and verification.
- In [10] a model based approach using a toolset called PLCTools has been introduced. The FBD programs are modelled and are described as High Level Timed Petri nets (HLTPN). HLTPN are used for validating the design and generating the code. MATLAB/SIMULINK provides suitable means for specifying and simulating the plant.
- In [16] an approach for the automated verification of ladder diagrams [14] and timed function blocks is presented. The algorithms are translated into finite state automata. The SMV tool is used as symbolic model checker to check for the properties. The paper [15] describes a similar approach based on Signal Interpreted Petri nets (SIPNs), also using the SMV model checker tool for analysis.
- In [5] programs described in instruction list [14] are modelled as Petri nets. The model of the program is then composed with Petri net models of the process into one model of the controlled system. The properties to be verified are expressed in CTL and the SMV model checker is used.
- An attempt to combine theorem proving and model-checking to formally verify real-time systems is presented in [13]. The authors use state-event labelled transition systems (SELTS) as a formal model for Time Transition Models (TTMs). State-event observation equivalences formalized in the PVS proof checkers. With appropriate restrictions the PVS models can be translated into input for the SAL model-checker. A simple real-time control system is specified and verified using these theories.

1.1   Verification of the PRISE Safety Procedure: Aim and Approach

This paper proposes a unifying methodology for the formal verification of safety procedures in NPPs. The methodology is applied to the analysis of the Functional Block Diagram (FBD) specification of a nuclear safety procedure called PRISE (Primary-to-Secondary leaking fault). The operation and the activation conditions of the PRISE safety procedure are explained in Section 2.

The FBD based specification of the safety logic was developed by the experts of the nuclear power plant. The development was incremental, i.e. the experts first produced a core logic, tested it in a simulation environment, found some conditions and operating modes not covered by the logic, then extended the logic accordingly. Our task was to prove the correctness and completeness of the final safety logic. The verification goals were identified by the nuclear experts, as described in Section 2.2.

We use *reinterpretation* for deriving our formal model. Thus, we derive the formal model of the checked safety procedure directly from the FBD based specification, instead of the informal written starting specification. In addition, we would like to preserve the advantages of the FBD: the clear and easy to overview graphical representation, and the ability to execute (simulate) the specification. As a further aim, the approach should provide possibility for automatic composition and verification of the formal model.

For this reason, we chose coloured Petri nets (CPNs) [3] as the formal modelling and analysis framework. CPNs possess all of the above mentioned advantages and allow modelling, simulation, and formal analysis based verification [4]. They have been successfully applied in the area of reliability analysis, as well as for verification of safety-critical software and control components in NPPs [17]. A CPN based integrated knowledge base development tool for the verification of the dynamic alarm system has been reported in [8]. A safety-critical software requirements verification using combined CPN and Prototype Verification System (PVS) methods is described in [9].

Our presented approach is non model based, i.e. only the PRISE safety logic is modelled and verified, without including a simplified model of the nuclear process. The model was prepared and analysed using the Design/CPN tool [4].

2   The PRISE Safety Procedure

The subject of our modelling and analysis is a safety procedure, designed for the Paks Nuclear Power Plant (Paks NPP) located in Hungary. This plant operates four VVER-440/213 type pressurized water reactor (PWR) units with a total nominal (electrical) power of 1860 MW.

2.1   The Primary-to-Secondary Leaking Fault Event

The *PRImary-to-SEcondary leaking* (abbreviated as PRISE) is one of the major faults related to non-compensable leaking of parts of the primary circuit. A PRISE event occurs when there is a rupture or other leakage within the steam generator, affecting either a few (3-10) tubes or their collector that contain the high-pressure activated liquid of the primary circuit.

In the unlikely case of a PRISE event, the corresponding safety procedures take care of the reactor trip (emergency shutdown) and the isolation of the faulty steam generator. However, if the event is not handled properly, then there is a possibility to release some of the contaminated water to the environment due to

the unreliable operation of one of the involved sensors. In order to prevent this possibility, a safety valve has been added to each steam generator. These valves drain the contaminated water into the containment when necessary. The nuclear experts developed a new safety procedure (called the *PRISE safety procedure*) to control the operation of the safety valves.

As a preliminary safety analysis step, simulation investigations have been carried out for the PRISE initiating event. The event sequence generated by a PRISE event when the initial plant state is in its *normal operating mode* and *no other fault occurs* has been determined as follows:

1. First the decrease of primary circuit pressure p_{PR} is observed that implies the safety event $p_{PR} < 11.2\ MPa$. This causes an automatic reactor shutdown when the control rods reach their bottom position ($\chi_{RSHUT} = 1$).
2. The shutting down of the reactor initiates the turbine shutdown when the secondary water and steam mass flowrates fall into a nominal low level. This implies an increase in the faulty steam generator level ℓ_{SG}.
3. The increase will eventually cause a level alarm in the faulty steam generator ($\Delta\ell_{SG} > +600\ mm$) that automatically initiates the isolation of the faulty steam generator resulting in even more increase of the ℓ_{SG}.

2.2   The Implemented PRISE Safety Procedure

As described above, the purpose of the PRISE safety procedure us to initiate the draining *if and only if a PRISE event occurs*. This includes:

– Preventing the steam generators from being drained when a fault event (causing similar symptoms but not classified a PRISE event) occurs. Thus, the PRISE safety procedure should be *selective*.
– When the system is not in a normal operating regime, but is either being started or shut down, the PRISE safety procedure is designed not to be active. The operators manually initiate the draining if needed.

Disturbing faults and various operating regimes make the selective detection of a PRISE fault event complicated. Additionally, one of the key sensors, the water level (ℓ_{SG}) sensor in the steam generators is highly unreliable. It tends to show randomly a spuriously high level due to the solid scale content of the secondary water. This measurement error is more frequent in transient operation regime. The water level sensor is not part of the reactor safety system, therefore it is not duplicated.

With the above considerations, the technological and system experts at Paks Nuclear Power plant have designed the timed logical scheme, a *safety procedure*, in a heuristic way. The logical scheme in its FBD representation is shown in Figure 1. The description of the inputs and outputs of the PRISE safety procedure is included in Table 1.

The resulting safety function will be included in the software of the Reactor Protection System (RPS) of the Paks NPP. The RPS is implemented on the basis of the TELEPERM XS (TXS) system platform for digital safety I&C.
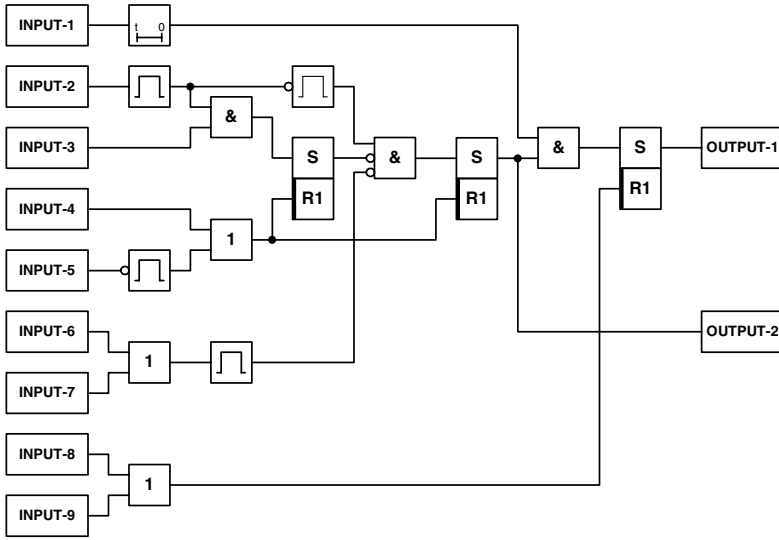
Fig. 1. The Functional Block Diagram of the PRISE safety procedure

Table 1. PRISE safety procedure I/O description

INPUT-1	Short name: SG level high ($\Delta\ell_{SG} > +600\ mm$)
	Description: Steam generator water level is increasing
	(due to closure of the turbine)
INPUT-2	Short name: Primary pressure decreasing ($p_{PR} < 11.2\ MPa$)
	Description: The pressure of the primary water is decreasing
	(due to the PRISE or other leakage)
INPUT-3	Short name: Containment pressure is normal ($p_{CN} < 0.1\ MPa$)
	Description: The pressure of the containment is *not* increasing
	(no primary water inflow due to a non-PRISE fault)
INPUT-4	Short name: Primary temperature below nominal ($T_{CL} < 245^{o}C$)
	Description: Technical condition signifying that the reactor is in
	startup/shutdown operation
INPUT-5	Short name: Control rods fully down ($\chi_{RSHUT} = 1$)
	Description: Technical condition used to reset the operation
	of the PRISE safety procedure
INPUT-6	Short name: SG deltaP
INPUT-7	Short name: SG RAP 1/2
	Description: Technical conditions used to avoid the erroneous draining of
	the secondary water after isolation of the steam generator
INPUT-8	Short name: SG inhibition
	Description: Technical condition used to indicate the SG inhibited state
INPUT-9	Short name: Primary pressure low ($p_{PR} < 5\ MPa$)
	Description: Technical condition signifying that the reactor is in
	startup/shutdown operation
OUTPUT-1	Short name: GFINH1 (SG is inhermetical)
	Description: Primary output, activates the secondary water drain
OUTPUT-2	Short name: ACTIVE
	Description: Auxiliary output used in control operations

The designed safety procedure initiates the draining (OUTPUT-1) when a critical decrease in the primary pressure (INPUT-2) is followed (after a specified time delay) by the increase of the steam generator level (INPUT-1) that lasts for a certain time interval. However, the draining is initiated only if the containment pressure keeps its nominal value (INPUT-3). This assures that the pressure is not increasing due to another, non-PRISE fault causing an inflow of the primary water into the containment. INPUT-1 must hold its value for at least a minimum time interval to prevent the incorrect initiation of draining by an unreliable water level sensor measurement showing temporarily a spuriously high value.

The INPUT-4 and INPUT-9 input conditions inhibit the operation in a startup or shutdown situation. INPUT-5 resets the operation of the PRISE safety procedure in the case when the reactor is shut down. INPUT-6 and INPUT-7 prevent the erroneous draining of the containment after the isolation of a steam generator caused by a non-PRISE fault. INPUT-8 indicates the situation when the steam generator was manually isolated due to a failure indication.

The primary OUTPUT-1 of the procedure is the presence of a PRISE event. Note that the auxiliary OUTPUT-2 signal indicates the presence of all but one of the symptoms of the PRISE situation.

3 Coloured Petri Net Model of the PRISE Safety Procedure

Our choice for the description formalism of the PRISE safety procedure are coloured Petri nets (CPN) [4]. CPN is an extension of Petri nets. Most important differences are:

- places can contain coloured tokens (i.e. multi-sets) that can symbolize the data content in data flow models, and
- CP nets can be hierarchically structured using substitution transitions and subnets.

Figure 2 shows the high-level prime page of our CPN model [17]. The larger rectangles are substitution transitions that denote subnets of the corresponding function blocks. The smaller net elements are simple places and transitions that are only needed for connecting the subnets.

We carried out the development and validation of the CPN model in two phases. In the first phase we studied the related Functional Blocks (FBs) and transformed them into an equivalent CPN diagram. This transformation was not straightforward, as the semantics of the FBs have a heterogeneous, semi-formal description consisting of truth tables, timing diagrams, together with textual information. When the transformation was completed, we generated the entire state space of the resulting CPN model. This was achieved by "plugging in" the CPN model of the FB as a subnet into a simple high-level feedback loop that fed random values to the inputs of the subnet. (We used the same technique to generate the state space of the CPN model of the whole PRISE safety procedure,
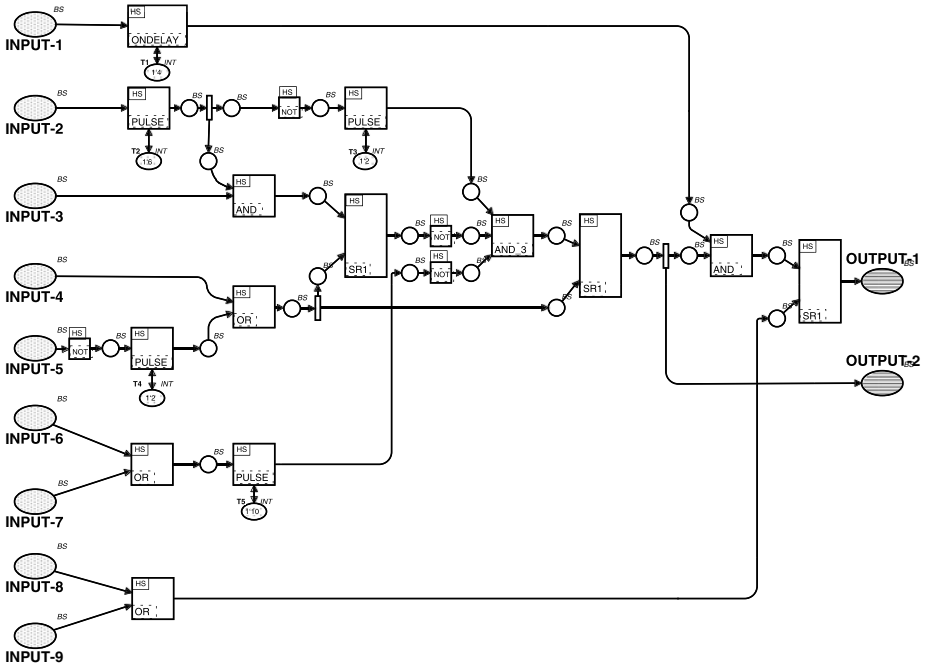
Fig. 2. The Coloured Petri net model of the PRISE safety procedure

as described later.) Then, we compared the state sequences in the alternative trajectories of the state space to the specification.

As an example to the transformation of a functional block to its CPN form, Figure 3 presents the CPN model of the SR1 function block. The SR1 function implements the behaviour of a static RS flip-flop (preferred state on reset, priority on reset) as known from the digital circuit theory. There are 3 copies of the SR1 block used in the timed logical description of the PRISE safety procedure. The subnet shown in Figure 3 is instantiated for each copy of the SR1 substitution transition in the high-level model in Fig. 2.

The operation of the CPN model of the SR1 function can be summarised as follows: if the Set input is activated (BI1 input place is marked with a token coloured with value 1), the output is set to active (BO1 output place receives a token coloured with value 1). Similarly, the activation of the Reset input (BI2 input place) makes the output inactive (a coloured with value 0 is put into the BO1 output place). When both inputs are active, the Reset function dominates. With both inputs inactive, or when any of the input signals is invalid, then the SR1 function block maintains the actual state of the output signal. Initially the output is inactive.

In the second phase of the development and validation of the CPN model for the PRISE safety procedure, we created the high-level prime page of the CPN model (Figure 2). This prime page connects and instantiates the previously
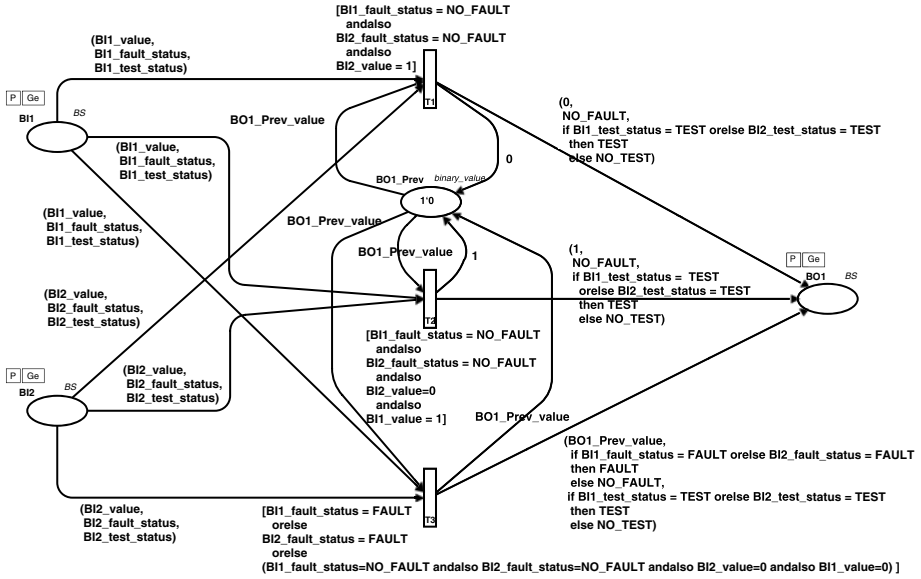
Fig. 3. The CPN model of the SR1 function block

prepared and validated CPN subnets of the different functional blocks. The transformation of the Functional Block Diagram (Figure 1) was rather straightforward, since the structure of the FBD graph and the corresponding CPN graph are isomorphic. The correctness of the translation was also partially validated by the dynamic (behavioural) properties of the CPN, as described in Section 4.

The run-time environment is a safety-critical highly dependable digital process control computer. It uses an explicit 50 millisecond long *scan cycle*. During each scan cycle the controller first samples its inputs, then evaluates all of its functional diagram pages. The evaluation starts from the FBs connected to the inputs and follows the flow of data until they reach the outputs. During the evaluation the controller computes its new internal state, then sets the outputs, and in the remaining time performs self-tests.

This behaviour is reflected by the CPN model the following way: the propagation of the tokens in the net represents the flow of data in the functional diagram. The CPN model has a feedback loop (not included in Figure 2 for simplicity) that puts simultaneously a single coloured token into each input place at the beginning of a scan cycle. The colour of the input tokens carries the input data value. These tokens initiate the execution of the subnets modelling the function blocks. When every subnet has been executed, a single coloured token is generated into each output place. The feedback loop takes away every generated token from the outputs and the scan cycle ends. Then the loop puts a new token into every input place, so that the next cycle can begin.

4   Analysis of the Coloured Petri Net Model

Petri net and CPN models have a broad selection of analysis techniques; some
of which even avoid the state explosion problem [7]:

- *Structural analysis techniques* construct no state space at all, because they
  work directly on the structure of the Petri net. Results are *structural prop-
  erties* and *invariants.*
- *Dynamic (reachability) analysis techniques* are based on the construction and
  exploration of the complete state space (reachability or occurence graph).
- The *lazy state-space construction* method is also available to build reduced
  (interleaving) state spaces. The reduction is based on an appropriate equiv-
  alence function, which maps several states into one.

The previously mentioned problems with the selective detection of a PRISE
fault event, and the heuristic design process of the safety logic made it necessary
to perform a rigorous formal verification of the PRISE safety procedure. We
needed to prove if it initiates the draining

- *always if PRISE occurs* in every normal operation regime coupled with *sen-
  sor fault in ℓ_{SG}* that is highly unreliable,
- *never if PRISE does not occur* even if severe faults causing similar symptoms
  occur.

We could translate these requirements into verification goals the following way:

- *Liveness requirements:* the secondary water draining activity is always acti-
  vated when a real PRISE accident has occurred (no actuation masking).
- *Safety requirements:* the draining activity is not activated if not a real PRISE
  accident has occurred (no erroneous actuation).

4.1   State Space Analysis of Coloured Petri Nets

State space analysis is one of the main formal analysis methods of Petri nets
[1]. It has been used successfully in the verification of concurrent systems like
communication protocols, parallel- and distributed algorithms.

The state space of a CPN is called an *occurrence graph* (O-graph) [4] or a
reachability graph. The O-graph has a node for each reachable marking and an
arc for each step that occurs. The source node of an arc is the start marking of a
step, while the destination node is the end marking. Using the constructed state
space it is possible to algorithmically reason about the behaviour of a system,
such as to verify that the system possesses certain desired properties or to locate
errors in the system. If the state space is finite, it can be used to analyse the
dynamic properties, such as reachability, boundedness, liveness, and fairness.

Although the PRISE safety procedure is relatively simple, its complete state
space is large (it has approx. 10^{14} states). This is due to the cyclic operation (mod-
elled by the feedback loop), and the internal sequential function blocks (the flip-
flop, pulse and delay blocks). Thus, an exhaustive analysis of the state space
cannot be performed with many of the analysis tools (including Design/CPN).

This large state space is partially the result of the non model based approach for the verification: during the model checking phase many *false* counterexamples are found for the analysed requirements that correspond to *invalid state space trajectories*. These trajectories cannot occur in the real system due to the properties and constraints of the monitored plant operation (thus they were not accounted for in the design of the PRISE safety procedure).

We could get round this problem by analysing parts of the state space defining constrained input scenarios. In our case study, we examined the initiation of the OUTPUT-1 secondary water draining activation signal under nominal conditions. It means that all the input signals have constant values or step-function values, except the "SG level high" (INPUT-1) signal, which is unreliable, thus it will be assumed to be arbitrary. The other inputs are set to match the activation conditions of the OUTPUT-1 signal.

Restricting the analysis to the most significant failure scenario cut down the size of the state space by several orders of magnitude: the number of states in the occurrence graph became 46811.

Lazy state space generation using equivalence classes. The CPN models have often some markings or state space trajectories which, for certain purposes, are alike or similar, so that we may want to ignore the difference between them. This notion of similarity provides a potential to reduce the state space for faster and more thorough analysis, and motivated the introduction of the so-called *occurrence graphs with equivalence classes* (OE-graphs).

The similarity of states can be formalized by defining *equivalence relations* on the set of states and the set of actions. All states in an equivalence class are then represented by a single node in the resulting OE-graph, therefore the nodes correspond to equivalence classes of states and the arcs correspond to equivalence classes of actions. The constructed *condensed* state space is called the *OE-graph*. It is typically orders of magnitude smaller than the ordinary full state space, but from which the same kind of dynamic properties can be directly verified and analysed without unfolding it to the full state space.

The Design/CPN tool (from version 3.1) contains an OE/OS Graph Tool. The OE/OS Graph Tool has a large number of built-in standard queries. These can be used to investigate the dynamic properties of CPNs, such as reachability, boundedness, home properties, liveness and fairness, and gives possibility to the user to formulate his own queries. The theoretical background of the OE-graphs including the proofs of reachability, boundedness, home, liveness and fairness properties can be found in detail in [4].

Reduced state space construction for the PRISE CPN. Unfortunately, due to the nature of the modelled safety procedure, the model and its state space do not contain any symmetries or permutation invariances, thus these cannot be used for reduction.

On the other hand, by a closer examination of the occurrence graph one can observe a particular property: from the initial nodes there is a large "fan out" into many alternate trajectories, but these soon converge into 3–4 destination nodes. The reason for this behaviour is the internal nondeterminism of

the PRISE CPN model. This means that the execution order of the CPN sub-nets located on parallel data paths is not deterministic (in other words they are *concurrent, casually independent*): all substitution transitions that meet the enabling conditions (have enough coloured tokens on their input places) can fire. (This nondeterminism models our lack of knowledge about the real execution order of concurrent paths, since the TXS source code is confidential.)

We could formally verify that these alternate execution trajectories starting from a particular input produce the same set of outputs, that is they form *observation equivalences*. This can be used for state space reduction by removing ("folding" together) the redundant trajectories, resulting in a single equivalent trajectory.

For the purpose of the Design/CPN OE/OS Graph Tool, the equivalence specification of markings and/or binding elements must be defined by the user. It is achieved by implementing two equivalence functions and a hash function. One defines when two markings are equivalent, the other defines when two binding elements are equivalent. The hash function maps each equivalence class into a unique hash key.

Instead of trying to formulate the above outlined observation equivalences in terms of equivalence and hash functions, we rather created a minimum representation for the PRISE CPN model in the chosen constrained input scenario. For this purpose we substituted the subnet determined by the constrained inputs (INPUT-2 to INPUT-9) with an equivalent minimum sequential CPN subnet. (The minimum sequential CPN subnet does not have concurrent data paths, but produces the same results in the selected internal data points as the original nonminimal model.) The observation equivalence of the original model and the minimum representation was proven.

With this modification the size of the state space of the PRISE CPN model was further reduced down to 387 states. Due to this small size we could examine the structure of the state space directly by the SCC graph method.

The SCC-graph of the reduced state space. The SCC (Strongly Connected Components) graph represents the strongly connected components of an occurrence graph by a single node. The strongly connected components of an O-graph are important, because every state in an SCC is a *home state* [1], that is each state of an SCC is reachable from any other state of the same SCC. Thus, strongly connected components represent repeatable, cyclic activities. The structure of the SCC-graph derived from the reduced state space of the PRISE CPN is shown in Figure 4.

As it can be seen, the state space has a large initial sequence (172 nodes) of singular (non-SCC) states. This sequence represents the enabling of the PRISE detection logic. At the end of the sequence the OUTPUT-2 signal becomes active, meaning that the safety logic is ready to activate on a valid INPUT-1 signal.

Two SCC nodes (~39 and ~181) can be found in the SCC-graph, these are emphasized by bold text and grey background. The following interpretation was found for the states contained in these SCC nodes by careful analysis using both state space search methods and model checking:
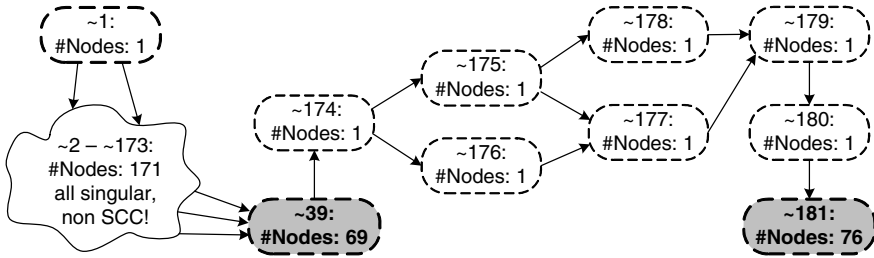
Fig. 4. The structure of the SCC graph for the PRISE CPN

- The ~39 node contains the states in which the PRISE safety logic is enabled, but no valid INPUT-1 signal (the unreliable steam generator level measurement signal) is received. The various short input "spikes" on the INPUT-1 are correctly filtered out by the ONDELAY functional block.
- On the other hand, the ~181 node includes all of those states, where a valid INPUT-1 signal was received (the 'SG level high' signal was active for a sufficient number of scan cycles).

From the structure of the SCC-graph in Figure 4 it is easy to see that the PRISE safety procedure can be initiate iff the enabling conditions are valid (OUTPUT-1 cannot be activated before OUTPUT-2), and only iff a reliable steam generator level measurement signal is received. However, further analysis is required to prove that the initial activation sequence is also correct. This will be proven in the next sections.

Dynamic properties of the PRISE CPN. The analysis results from the reduced state space of the PRISE CPN provide a lot of important information for both the validation of the model and the verification of its correctness. The investigated *dynamic* (or *behavioural*) properties are summarized in Table 2.

Table 2. Dynamic properties of the PRISE CPN model

Property	Result
Boundedness	The PRISE CPN is *multi-set bounded*.
	The PRISE CPN is *safe in the integer sense*.
Liveness	The PRISE CPN with feedback is *deadlock-free*.
	All transitions related to the primary output signal are *live*.
Fairness	Each live transition is at least *impartial* or *fair*.

The derived dynamic properties of the PRISE CPN give additional proof for the validation of the transformation from the FBD based specification of the safety procedure to its formal CPN model (see Section 3), in the following way:

- All places in the net are *multi-set bounded*. The *upper multi-set bounds* of places describe the operational range of the corresponding signals. The *lower multi-set bounds* of places prove that the resources (e.g. the inner state of time-dependent blocks) are preserved, as the corresponding places contain a token in all states of the operation.
- The net is *safe in the integer sense*, meaning that each place contains at most one coloured token in any state. This confirms that the both the intended data-flow behaviour and the functional structure is correctly expressed in the CPN model.
- The PRISE CPN with the feedback loop is *deadlock-free*, there are no dead markings: the safety logic will not "freeze" in any state of operation. All transitions involved in the activation of the primary output signal (OUTPUT-1) are *live*. Thus, the PRISE safety procedure is able to activate the emergency activity, and retains this capability during the whole operation.
- The fairness property of each live transition is at least *impartial* or *fair*. This implies both that they can fire infinite times (the functionality is repeatable); and neither "domination", nor "starvation" of the activities can occur.

4.2   Analysis of the PRISE CPN by Model Checking

After the dynamic analysis of the CPN model the main characteristics of the PRISE safety logic can already be seen. With these properties and with the interpretation of the SCC-graph the model is partially verified.

In order to prove the safety and liveness requirements (defined in Section 4), we have proved several subconditions:

1. The OUTPUT-2 and OUTPUT-1 signals are activated in *all trajectories of the state space* (this is a liveness condition, since the initial activation conditions are always present in the scenario under analysis).
2. In all trajectories of the state space the OUTPUT-1 signal *can only be activated after the OUTPUT-2 signal*, and not in the reverse order.
3. *Neither* the OUTPUT-2 nor the OUTPUT-1 signal *can be activated incorrectly* by the 'SG level high' signal when the enabling conditions are not present (that is while the INPUT-2 signal is still delayed).
4. The "ONDELAY" functional block connected to INPUT-1 *correctly filters the transient behaviour* of the 'SG level high' signal: the filtered signal will only be activated if the 'SG level high' signal remains continuously active during the filtering interval. Shorter "spikes" of this signal cannot make the filtered signal to become active.
5. The activation of the filtered 'SG level high' signal *will always activate* the OUTPUT-2 and subsequently the OUTPUT-1 signals when the other enabling conditions are present.

In the Design/CPN tool we could use two different techniques to prove these subconditions: reachability graph search functions and ASKCTL temporal operators with model checking.

```
fun search_GF ( p : BS) : Node list
 = PredAllNodes (fn n => cf(p, Mark.PRISE_MAIN'GFINH1 1 n) >0);

val lista=search_GF (1,NO_FAULT,NO_TEST);
length lista;
```

```
val search_GF = fn : BS -> Node list
val lista = [45862,43623,43622,42128,32485,32484,23896] : Node list
val it = 7 : int
```

```
fun search_e (p : BS) : Node list
 = PredAllNodes ( fn n => (
        not((Mark.PULSE'Time 3 n) = 1'0)
          andalso
        (Mark.PRISE_MAIN'GFINH1 1 n) = 1'(1,NO_FAULT,NO_TEST)
));

val lista = search_e (1,NO_FAULT,NO_TEST);
length lista;
```

```
val search_e = fn : BS -> Node list
val lista = [] : Node list
val it = 0 : int
```

Fig. 5. Example verification results using state space search and reachability analysis

Figure 5 demonstrates the use of reachability graph search functions. The two example search formulas are on the left (written in CPN ML language), the result of their execution is shown on the right. The first formula says that the OUTPUT-1 (GFINH in the figure) signal is active (contains a token with the value 1) in 7 states of the state space. The second formula proves that the OUTPUT-1 signal cannot be active in any of the states before the counter of the PULSE block connected to the INPUT-2 place reaches zero — a prerequisite to the activation of the OUTPUT-2 (ELES in the figure) signal.

The Design/CPN program has an explicit-state branching time model checker called ASKCTL for evaluating CTL temporal expressions. This model checker operates directly on the state space generated by the OE/OS Graph Tool.

```
fun is_ELES_Set a
 = (Bind.SR1'T2 (2,
    {BI1_fault_status = NO_FAULT, BI1_test_status = NO_TEST,BI1_value=1,
     BI2_fault_status = NO_FAULT, BI2_test_status = NO_TEST, BI2_value=0,
     BO1_Prev_value = 0 })
    = ArcToBE a );

val myELESSet = EV ( MODAL ( AF ("Is ELES Set", is_ELES_Set) ) );

eval_node myELESSet  InitNode;
```

```
val is_ELES_Set = fn : Arc -> bool
val myELESSet = FORALL_UNTIL (TT,MODAL (AF (#,#))) : A
val it = true : bool
```

```
fun is_ELES n
 = (Mark.PRISE_MAIN'ELES 1 n) = 1'(1,NO_FAULT,NO_TEST);

fun is_GF n
 = (Mark.PRISE_MAIN'GFINH1 1 n) = 1'(1,NO_FAULT,NO_TEST);

val myNotGFuELES = FORALL_UNTIL ( NOT ( NF ("Is GF", is_GF) ),
NF ("Is ELES", is_ELES));

eval_node myNotGFuELES  InitNode;
```

```
val is_ELES = fn : Node -> bool
val is_GF = fn : Node -> bool
val myNotGFuELES = FORALL_UNTIL (NOT (NF (#,#)),NF ("Is ELES",fn)) : A
val it = true : bool
```

Fig. 6. Example verification results using ASKCTL temporal expressions

Figure 6 gives two examples for the use of model checking with temporal operators (again, temporal formulas are on the left, and the evaluation results are on the right). The first formula confirms that all possible execution paths eventually reach a state from where an action activating the OUTPUT-2 signal will be executed. The second formula proves that the OUTPUT-1 signal cannot be activated before the OUTPUT-2 signal would be set.

5   Conclusions

We presented an approach for the formal modelling and verification of a Function Block based specification for the PRISE safety procedure. The approach uses coloured Petri nets as the description formalism and state space analysis together with model checking as the verification method.

The example demonstrated that the CPN formalism is well suited to FDB based specifications, highlighting several advantages over other popular formal analysis methodologies:

- Since both the FDB and the derived coloured Petri net express the flow of data within the modelled system, their structure are very similar. Many advantages of the Functional Block Diagrams are preserved, including the clear and easy to overview structure and the simulation capabilities.
- Thanks to the hierarchical modelling capabilities of the Design/CPN tool, the elementary Functional Blocks can be modelled separately by CPN subnets, and a model library can be created.
- Due to the similarity of the FBD and CPN models, the construction of the coloured Petri net model from a Function Block Diagram is straightforward.
- In addition to the dynamic properties, the Design/CPN tool provides other strong analysis tools, like the equivalence classes, the SCC-graph, the state space query and search methods, and branching time temporal logic.

Based on the successful model checking of the subconditions defined in Section 4.2 we were able to prove that the PRISE safety procedure fulfils its requirements in the constrained input scenarios. We could obtain further results for the correctness of the PRISE safety procedure by using a model based verification method, i.e. by developing a simplified coloured Petri net plant model and connecting it in a closed-loop structure to the CPN model of the PRISE safety procedure. (These results are not covered by this paper, the interested reader can find more details in [18].)

As a future work, we would like to create a model of the PRISE in a symbolic model checking tool, such as SAL (Symbolic Analysis Laboratory) [13]. Symbolic model checkers use a very compact representation of the state space, thus can handle much larger state spaces than e.g. Design/CPN. It would be interesting to evaluate the advantages and disadvantages of the CPN based modelling and verification approach compared to a purely model checking based solution.

References

1. Murata, T.: Petri Nets: Properties, Analysis and Application. Proceedings of the IEEE 77(4), 541–580 (1989)
2. Design/CPN – Computer Tool for Coloured Petri Nets, CPN group at the University of Aarhus, Denmark (2002), http://www.daimi.au.dk/designCPN/
3. Jensen, K.: Coloured Petri Nets – Basic Concepts, Analysis Methods and Practical Use. In: Basic Concepts. Monographs in Theoretical Computer Science, vol. 1. Springer, Heidelberg (1992)

4. Jensen, K.: Coloured Petri Nets – Basic Concepts, Analysis Methods and Practical Use. In: Analysis Methods. Monographs in Theoretical Computer Science, vol. 2. Springer, Heidelberg (1997)
5. Mertke, T., Menzel, T.: Methods and tools to the verification of safety-related control software. In: Proc. of the IEEE Int. Conf. on Sys., Man and Cybernetics (SMC 2000), Nashville, USA, pp. 2455–2457 (2000)
6. Younis, M.B., Frey, G.: Formalization of existing PLC programs: A survey. In: Proc. of the IEEE/IMACS Multiconf. on Comp. Eng. in Sys. App. (CESA 2003), Lille, France, Paper No. S2–R–00–0239 (2003)
7. Heiner, M.: Verification and optimization of control programs by Petri nets without state explosion. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 69–84. Springer, Heidelberg (1997)
8. Park, J.H., Seong, P.H.: An integrated knowledge base development tool for knowledge acquisition and verification for NPP dynamic alarm processing systems. Annals of Nuclear Energy 29, 447–463 (2002)
9. Son, H.S., Seong, P.H.: Development of a safety critical software requirements verification method with combined CPN and PVS: a nuclear power plant protection system application. Reliability Engineering and System Safety 80, 19–32 (2003)
10. Baresi, L., Mauri, M., et al.: PLCTools: Design, Formal Validation, and Code Generation for Programmable Controllers. In: Proc. of the IEEE Conf. on Sys., Man, and Cybernetics (SMC 2000), Nashville, USA, pp. 2437–2442 (2000)
11. Hanisch, H.M., Lobov, A., et al.: Formal Validation of Intelligent Automated Production Systems towards Industrial Applications. Int. J. of Manufacturing Tech. and Management 8(1), 75–106 (2006)
12. Wassyng, A., Lawford, M.: Lessons Learned from a Successful Implementation of Formal Methods in an Industrial Project. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 133–153. Springer, Heidelberg (2003)
13. Lawford, M., Pantelic, V., Zhang, H.: Towards Integrated Verification of Timed Transition Models. Fundamenta Informaticae 70(1–2), 155–164 (2006)
14. International Standard IEC 61131-3: Programmable Controllers - Part 3: Programming Languages. International Electrotechnical Commission, Geneva, Switzerland (1993)
15. Minas, M., Frey, G.: Visual PLC-Programming using Signal Interpreted Petri Nets. In: Proc. of the American Control Conference 2002 (ACC 2002), Anchorage, Alaska, pp. 5019–5024 (2002)
16. Rossi, O., Schnoebelen, P.: Formal Modelling of Timed Function Blocks for the Automatic Verification of Ladder Diagram Programs. In: Proc. 4th Int. Conf. Automation of Mixed Processes: Hybrid Dynamic Systems (ADPM), Dortmund, Germany, pp. 177–182. Shaker Verlag, Germany (2000)
17. Németh, E., Bartha, T.: Formal verification of function block based specifications of safety-critical software. In: Modern Information Technology in the Innovation Processes of the Industrial Enterprises (MITIP 2006), Budapest, Hungary, pp. 211–218 (2006)
18. Németh, E., Fazekas, C., Szederkényi, G., Hangos, K.M.: Modeling and simulation of the primary circuit of the Paks nuclear power plant for control and diagnosis. In: Proceedings of the EUROSIM 2007, Ljubljana, Slovenia (2007) (on CD)

Using Datalog and Boolean Equation Systems for Program Analysis⋆

María Alpuente, Marco A. Feliú, Christophe Joubert, and Alicia Villanueva

Universidad Politécnica de Valencia, DSIC / ELP
Camino de Vera s/n, 46022, Valencia, Spain
{alpuente,mfeliu,joubert,villanue}@dsic.upv.es

Abstract. This paper describes a powerful, fully automated method to evaluate Datalog queries by using Boolean Equation Systems (Bess), and its application to object-oriented program analysis. Datalog is used as a specification language for expressing complex interprocedural program analyses involving dynamically created objects. In our methodology, Datalog rules encoding a particular analysis together with a set of constraints (Datalog facts that are automatically extracted from program source code) are dynamically transformed into a Bes, whose local resolution corresponds to the demand-driven evaluation of the program analysis. This approach allows us to reuse existing general purpose verification toolboxes, such as Cadp, providing local Bes resolutions with linear-time complexity. Our evaluation technique has been implemented and successfully tested on several Java programs and Datalog analyses that demonstrate the feasibility of our approach.

Keywords: program analysis, Datalog, boolean equation system, demand-driven evaluation.

1   Introduction

Program analysis is a technique for statically determining dynamic properties of programs. Static analysis generally executes an abstract version of the program's semantics on abstract data, rather than on concrete data. While originally established as a technique used in optimizing compilers, program analysis is also commonly used in software-development tools that help to find program errors and also derive safety properties of programs.

Recently, a large number of program analyses have been developed in Datalog [15,18], a simple relational query language rich enough to describe complex interprocedural program analyses involving dynamically created objects.

The advantages of formulating dataflow analyses as a Datalog query are twofold. On the one hand, analyses that take hundreds of lines of code in a traditional language can be expressed in a few lines of Datalog [18]. On the

---

other hand, an important number of optimization techniques for Datalog have been studied extensively in logic programming and deductive databases [1,4]. The two general approaches for evaluating Datalog queries are the top-down and the bottom-up methods. Given a set of rules, the bottom-up approach computes all facts that can be inferred from the program and then selects those that unify with the given query. The top-down, goal-directed approach computes on-demand. While bottom-up computation may be very inefficient, the top-down approach is prone to infinite loops and redundant computations. Optimization methods for both approaches that resolve the major drawbacks have been developed, such as bottom-up transformations based on magic sets [3] and top-down evaluation with tabling [4]. In the *Query-Sub-Query* (QSQ) optimization technique [16], goals are generated top-down, but whenever possible, goals are propagated in sets at a time, rather than one at a time, and all generated goals and facts are memoized.

This paper describes the use of Boolean Equation Systems (BES) [2] to evaluate Datalog queries and its application to object-oriented program analysis. Our technique is based on top-down evaluation guided by the given query, and makes use of tables and finite data domains to ensure termination. Our method is not a direct evaluation method because it transforms the rules prior to evaluate them. Similarly to the QSQ technique [16], computation is done by proceeding with a set tuples at a time. This can be a great advantage for large datasets since it makes disk accesses more efficient. In our program analysis methodology, Datalog rules encoding a particular analysis, together with a set of constraints (Datalog facts that are automatically extracted from program source code), are dynamically transformed into a BES, whose local resolution corresponds to the demand-driven evaluation of the program analysis. This approach allows us to reuse existing general purpose verification toolboxes, such as CADP, providing local BES resolutions with linear-time complexity.

Related Work. The description of data-flow analyses as a database query was pioneered by Ullman [15] and Reps [13] who applied Datalog's bottom-up magic-set implementation to automatically derive a *local* implementation.

Recently, BESs with typed parameters [11], called PBES, have been successfully used to encode several hard verification problems such as the first-order value-based modal μ-calculus model-checking problem [12], and the equivalence checking of various bisimulations [5] on (possibly infinite) labeled transition systems. However, PBESs have not yet been used to compute complex interprocedural program analyses involving dynamically created objects.

The closest related work proposes the use of Dependency Graphs (DGs) for representing satisfaction problems, including propositional Horn Clauses satisfaction and BES resolution [10]. A linear time algorithm for propositional Horn Clauses satisfiability is described in terms of the least solution of a DG equation system. This corresponds to an alternation-free BES, which can only deal with propositional logic problems. The extension of Liu and Smolka's work [10] to Datalog query evaluation is not straightforward. This is testified by the encoding of data-based temporal logics in equation systems with parameters in [12], where each

boolean variable may depend on multiple data terms. DGs are not sufficiently expressive to represent such data dependencies on each vertex. Hence, it is necessary to work at a higher level, on the PBES representation.

Recently, a very efficient Datalog program analysis technique based on binary decision diagrams (BDDs) has been developed in the BDDBDDB system [18], which scales to large programs and is competitive w.r.t. the traditional (imperative) approach. The computation is achieved by a fixed point computation starting from the everywhere false predicate (or some initial approximation based on Datalog facts). Datalog rules are then applied in a bottom-up manner until saturation is reached, so that all solutions satisfying each relation of a Datalog program are exhaustively computed. These sets of solutions are then used to answer complex formulas.

In contrast, our approach focus on demand-driven techniques to solve a set of queries with no *a priori* computation of the derivable atoms. In the context of program analysis, note that all program updates, like pointer updates, might potentially be inter-related, leading to an exhaustive computation of all results. Therefore, improvements to top-down evaluation remain attractive for program analysis applications. Recently, Zheng and Rugina [19] showed that demand-driven CFL-reachability with worklist algorithm can compare favorably with an exhaustive solution, especially in terms of memory consumption. Our technique to solve Datalog programs based on local BES resolution goes towards the same direction and provides a novel approach to demand-driven program analyses.

Plan of the Paper. The rest of the paper is organized as follows: Section 2 recalls Datalog definitions and the BES formalism with its parameterised extension. Our methodology to transform Datalog query to an implicit BES with parameters is described in Section 3. Section 4 illustrates the application of Datalog and BES to program analysis, together with experimental results on JAVA programs and context-insensitive pointer analysis. Finally, Section 5 concludes and highlights future research directions.

2   Preliminaries

2.1   Datalog

Datalog [15] is a relational language using declarative *rules* to both describe and query a deductive database. A Datalog rule is a function-free Horn clause over an alphabet of *predicate* symbols (e.g. relation names or arithmetic predicates, such as $<$) whose *arguments* are either variables or constant symbols. A *Datalog program* $\mathcal{R}$ is a finite set of Datalog rules.

Definition 1 (Syntax of Rules). *Let $\mathcal{P}$ be a set of* predicate *symbols, $\mathcal{V}$ be a finite set of* variable *symbols, and $\mathcal{C}$ a set of* constant *symbols. A Datalog* rule *r, also called* clause*, defined over a finite alphabet $P \subseteq \mathcal{P}$ and arguments from $V \cup C$, $V \subseteq \mathcal{V}$, $C \subseteq \mathcal{C}$, has the following syntax:*

$$p_0(a_{0,1}, \ldots, a_{0,n_0}) \; :- \; p_1(a_{1,1}, \ldots, a_{1,n_1}), \; \ldots, \; p_m(a_{m,1}, \ldots, a_{m,n_m}).$$

where each p_i is a predicate symbol of arity n_i with arguments $a_{i,j} \in V \cup C$ ($j \in [1..n_i]$).

The atom $p_0(a_{0,1}, \ldots, a_{0,n_0})$ in the left-hand side of the clause is the rule's *head*, where p_0 is neither arithmetic nor negated. The finite conjunction of *subgoals* in the right-hand side of the formula is the rule's *body*, *i.e.*, atoms that may optionally be negated or arithmetic, and contain all variables appearing in the head. Following logic programming terminology, a rule with empty body ($m = 0$) is called a *fact* whereas a rule with empty head and $m > 0$ is called a *goal*. To keep the presentation simple, we restrict our syntax to predicate symbols of arity 1. A syntactic object (argument, atom, or rule) that contains no variables is called *ground*. The *Herbrand Universe* of a Datalog program R defined over P, V and C, denoted U_R, is the finite set of all ground arguments, *i.e.*, constants of C. The *Herbrand Base* of R, denoted B_R, is the finite set of all ground atoms that can be built by assigning elements of U_R to the predicate symbols in P. A *Herbrand Interpretation* of R, denoted I (from a set $\mathcal{I}$ of Herbrand interpretations, $\mathcal{I} \subseteq B_R$), is a set of ground atoms.

Definition 2 (Fixed point semantics). *Let R be a Datalog program. The least Herbrand model of R is a Herbrand interpretation I of R defined as the least fixed point of a monotonic, continuous operator $T_R : \mathcal{I} \to \mathcal{I}$ known as the immediate consequences operator and defined by:*

$$T_R(I) = \{h \in B_R \mid h : -b_1, \ldots, b_m \text{ is a ground instance of a rule in } R,$$
$$\text{with } b_i \in I, i = 1..m, m \geq 0\}$$

Note that T_R computes both, ground atoms derived from applicable rules–called *intentional database* (or *idb*)–, and ground instances of rules with an empty body ($m = 0$), also called *extensional database* (*edb*). The choice of minimal model as the semantics of a Datalog program is justified by the assumption that all facts that are not in the database are false.

The number of Herbrand models being finite for a Datalog program R, there always exists a least fixed point for T_R, denoted μT_R, which is the least Herbrand model of R. In practice, one is generally interested in the computation of some specific atoms, called *queries*, and not in the whole database of atoms. Hence, queries may be used to prevent the computation of facts that are irrelevant for the atoms of interest, *i.e.*, facts that are not derived from the query.

Definition 3 (Query Evaluation). *A Datalog query q is a pair $\langle G, R \rangle$ where:*

- *R is a Datalog program defined over P, V and C,*
- *G is a set of goals.*

Given a query q, its evaluation *consists in computing $\mu T_{\{q\}}$, $\{q\}$ being the extension of the Datalog program R with the Datalog rules in G.*

The evaluation of a Datalog program augmented with a set of goals deduces all the different constant combinations that, when assigned to the variables in the goals, can make one of the goal clauses true, *i.e.*, all atoms b_i in its body are satisfied.

2.2   Parameterised Boolean Equation System

Given $\mathcal{X}$ a set of boolean variables and $\mathcal{D}$ a set of data terms, a *Parameterised Boolean Equation System* [11] (PBES) $B = (x_0, M_1, ..., M_n)$ is a set of n blocks M_i, each one containing $p_i \in \mathbb{N}$ fixed-point equations of the form

$$x_{i,j}(\boldsymbol{d}_{i,j} : \boldsymbol{D}_{i,j}) \stackrel{\sigma_i}{=} \phi_{i,j}$$

with $j \in [1..p_i]$ and $\sigma_i \in \{\mu, \nu\}$, also called *sign* of equation i, the least (μ) or greatest (ν) fixed point operator. Each $x_{i,j}$ is a boolean variable from $\mathcal{X}$ that binds zero or more data terms $d_{i,j}$ of type $D_{i,j}{}^1$ which may occur in the *boolean formula* $\phi_{i,j}$ (from a set Φ of boolean formulae). $x_0 \in \mathcal{X}$, defined in block M_1, is a boolean variable whose value is of interest in the context of the local resolution methodology. Boolean formulae $\phi_{i,j}$ are formally defined as follows.

Definition 4 (Boolean Formula). *A boolean formula ϕ, defined over an alphabet of (parameterised) boolean variables $X \subseteq \mathcal{X}$ and data terms $D \subseteq \mathcal{D}$, has the following syntax given in positive form:*

$$\phi, \phi_1, \phi_2 ::= \mathsf{true} \mid \mathsf{false} \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid X(e) \mid \forall d \in D.\ \phi \mid \exists d \in D.\ \phi$$

where boolean constants and operators have their usual definition, e is a data term (constant or variable of type D), $X(e)$ denotes the call *of a boolean variable X with parameter e, and d is a term of type D.*

A *boolean environment* $\delta \in \Delta$ is a partial function mapping each (parameterised) boolean variable $x(d : D)$ to a predicate $\delta(x) : \mathcal{X} \rightarrow (D \rightarrow \mathbb{B})$, with $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$. *Boolean constants* true and false abbreviate the empty conjunction $\wedge \emptyset$ and the empty disjunction $\vee \emptyset$ respectively. A *data environment* $\varepsilon \in \mathcal{E}$ is a partial function mapping each data term e of type D to a value $\varepsilon(e) : D \rightarrow D$, which forms the so-called support of ε, noted $supp(\varepsilon)$. Note that $\varepsilon(e) = e$ when e is a constant data term. The *overriding* of ε_1 by ε_2 is defined as $(\varepsilon_1 \oslash \varepsilon_2)(x) =$ *if* $x \in supp(\varepsilon_2)$ *then* $\varepsilon_2(x)$ *else* $\varepsilon_1(x)$. The *interpretation function* $[\![\phi]\!]\delta\varepsilon$, where $[\![.]\!] : \Phi \rightarrow \Delta \rightarrow \mathcal{E} \rightarrow \mathbb{B}$, gives the truth value of boolean formula ϕ in the context of δ and ε, where all free boolean variables x are evaluated by $\delta(x)$, and all free data terms d are evaluated by $\mathcal{E}(d)$.

Definition 5 (Semantics of Boolean Formula). *Let $\delta : \mathcal{X} \rightarrow (D \rightarrow \mathbb{B})$ be a boolean environment and $\varepsilon : D \rightarrow D$ be a data environment. The semantics of a boolean formula ϕ is inductively defined by the following interpretation function:*

$$
\begin{aligned}
[\![\mathsf{true}]\!]\delta\varepsilon &= \mathsf{true} \\
[\![\mathsf{false}]\!]\delta\varepsilon &= \mathsf{false} \\
[\![\phi_1 \wedge \phi_2]\!]\delta\varepsilon &= [\![\phi_1]\!]\delta\varepsilon \wedge [\![\phi_2]\!]\delta\varepsilon \\
[\![\phi_1 \vee \phi_2]\!]\delta\varepsilon &= [\![\phi_1]\!]\delta\varepsilon \vee [\![\phi_2]\!]\delta\varepsilon \\
[\![x(e)]\!]\delta\varepsilon &= (\delta(x))(\varepsilon(e)) \\
[\![\forall d \in D.\ \phi]\!]\delta\varepsilon &= \forall\, v \in D,\ [\![\phi]\!]\delta(\varepsilon \oslash [v/d]) \\
[\![\exists d \in D.\ \phi]\!]\delta\varepsilon &= \exists\, v \in D,\ [\![\phi]\!]\delta(\varepsilon \oslash [v/d])
\end{aligned}
$$

---

[1] To simplify our description in the rest of the paper, we intentionally restrict to one the maximum number of data term parameter $d : D$.

Definition 6 (Semantics of Equation Block). *Given a* PBES *$B = (x_0, M_1, ..., M_n)$ and a boolean environment δ, the solution $[\![M_i]\!]\delta$ to a block $M_i = \{x_{i,j}(d_{i,j} : D_{i,j}) \stackrel{\sigma_i}{=} \phi_{i,j}\}_{j \in [1,p_i]}$ $(i \in [1..n])$ is defined as follows:*

$$[\![\{x_{i,j}(d_{i,j} : D_{i,j}) \stackrel{\sigma_i}{=} \phi_{i,j}\}_{j \in [1,p_i]}]\!]\delta = \sigma_i \Psi_{i\delta}$$

where $\Psi_{i\delta} : (D_{i,1} \to \mathbb{B}) \times \ldots \times (D_{i,p_i} \to \mathbb{B}) \to (D_{i,1} \to \mathbb{B}) \times \ldots \times (D_{i,p_i} \to \mathbb{B})$ is a vectorial functional defined as

$$\Psi_{i\delta}(g_1, ..., g_{p_i}) = (\lambda v_{i,j} : D_{i,j}.[\![\phi_{i,j}]\!](\delta \oslash [g_1/x_{i,1}, ..., g_{p_i}/x_{i,p_i}])[v_{i,j}/d_{i,j}])_{j \in [1,p_i]}$$

where $g_i : D_i \to \mathbb{B}$, $i \in [1..p_i]$.

A PBES is *alternation-free* if there are no mutual recursion between boolean variables defined by least ($\sigma_i = \mu$) and greatest ($\sigma_i = \nu$) fixed point boolean equations. In this case, equation blocks can be sorted topologically such that the resolution of a block M_i only depends upon variables defined in a block M_k with $i < k$. A block M_i is *closed* when the resolution of all its boolean formulae $\phi_{i,j}$ only depends upon boolean variables $x_{i,k}$ from M_i.

Definition 7 (Semantics of alternation-free PBES). *Given an alternation-free* PBES *$B = (x_0, M_1, ..., M_n)$ and a boolean environment δ, the semantics $[\![B]\!]\delta$ to B is the value of its main variable x_0 given by the semantics of M_1, i.e., $\delta_1(x_0)$, where the contexts δ_i are calculated as follows:*

$$\delta_n = [\![M_n]\!][] \text{ (the context is empty because } M_n \text{ is closed)}$$
$$\delta_i = ([\![M_i]\!]\delta_{i+1}) \oslash \delta_{i+1} \text{ for } i \in [1, n-1]$$

where each block M_i is interpreted in the context of all blocks M_k with $i < k$.

3 Datalog Queries and Boolean Equation Systems

An elegant and direct intermediate representation of a Datalog query can be given as an implicit BES parameterised with typed boolean variables. In this section, we present reductions between Datalog query evaluation and PBES resolution for both directions of reducibility. The reductions are linear-time with a suitable representation of the problem instances. As in [18], we assume that Datalog programs have stratified negation (no recursion through negation), and totally-ordered finite domains, without considering comparison operators.

3.1 Datalog Query Representation

We propose a transformation of the Datalog query into a related query, expressed as a parameterised boolean variable of interest and a PBES, which is subsequently evaluated using traditional PBES evaluation techniques.

Proposition 1. *Let* $q = \langle G, R \rangle$ *be a Datalog query, defined over* P, V *and* C, *and* $B_q = (x_0, M_1)$, *with* $\sigma_1 = \mu$, *a* PBES *defined over a set* $\mathcal{X}$ *of boolean variables* x_p *in one-to-one correspondence with predicate symbols* p *of* P *plus a special variable* x_0, *a set* $\mathcal{D}$ *of data terms in one-to-one correspondence with variable and constant symbols of* $V \cup C$, *and* M_1 *the block containing exactly the following equations, where fresh variables are existentially quantified after the transformation:*

$$x_0 \stackrel{\mu}{=} \bigvee_{:-\; q_1(d_1),\; ...,\; q_m(d_m).\; \in G} \bigwedge_{i:=1}^{m} x_{q_i}(d_i) \tag{1}$$

$$\{x_p(d : D) \stackrel{\mu}{=} \bigvee_{p(d)\; :-\; p_1(d_1),...\; p_m(d_m).\; \in R} \bigwedge_{i:=1}^{m} x_{p_i}(d_i) \mid p \in P\} \tag{2}$$

Then q *is satisfiable if and only if* $[\![B]\!]\delta(x_0) = \mathsf{true}$.

Boolean variable x_0 encodes the set of Datalog goals G, whereas (paremeterised) boolean variables $x_p(d : D)$ represent the set of Datalog rules R modulo renaming.

In our framework, the reverse direction of reducibility consists in the transformation of a parameterised boolean variable of interest, defined in a PBES, into a related relation of interest expressed as a Datalog query, which could be evaluated using traditional Datalog evaluation techniques.

Proposition 2. *Let* $B = (x_0, M_1)$, *with* $\sigma_1 = \mu$, *be a* PBES *defined over a set* $\mathcal{X}$ *of boolean variables and a set* $\mathcal{D}$ *of data terms, and* $q_B = \langle G, R \rangle$ *be a Datalog query defined over a set* P *of predicate symbols* p *in one-to-one correspondence with boolean variables* x_p *of* $\mathcal{X} \setminus \{x_0\}$, *a set* $V \cup C$ *of variable and constant symbols in one-to-one correspondence with data terms of* D, *and* $\langle G, R \rangle$ *containing exactly the following Datalog rules:*

$$G = \left\{ \begin{array}{l} :-\quad q_{1,1}(d_{1,1}),\ldots,\quad q_{1,n_j}(d_{1,n_j})., \\ \qquad\qquad\vdots \\ :-\, q_{n_i,1}(d_{n_i,1}),\ldots,q_{n_i,n_j}(d_{n_i,n_j}). \end{array} \right. \left| \; x_0 \stackrel{\mu}{=} \bigvee_{i=1}^{n_i} \bigwedge_{j=1}^{n_j} x_{q_{i,j}}(d_{i,j}) \in M_1 \right\}$$

$$R = \left\{ \begin{array}{l} p(d) :-\quad p_{1,1}(d_{1,1}),\ldots,\quad p_{1,n_j}(d_{1,n_j})., \\ \qquad\qquad\vdots \\ p(d) :- p_{n_i,1}(d_{n_i,1}),\ldots,\, p_{n_i,n_j}(d_{n_i,n_j}). \end{array} \right. \left| \; x_p(d) \stackrel{\mu}{=} \bigvee_{i=1}^{n_i} \bigwedge_{j=1}^{n_j} x_{p_{i,j}}(d_{i,j}) \in M_1 \right\}$$

Then $[\![B]\!]\delta(x_0) = \mathsf{true}$ *if and only if* $q_B = \langle G, R \rangle$ *is satisfiable.*

Example 1. We illustrate the reduction method from Datalog to PBES by means of a simple Datalog example. Let $q = \langle G, R \rangle$ be the following Datalog query with $D = \{mary, alice, mark, X, Y, Z\}$:

```
:- superior (mary, Y).
supervise(mary, alice).
```

```
supervise(alice, mark).
superior(X, Y) :- supervise(X, Y).
superior(X, Y) :- supervise(X, Z), superior(Z, Y).
```

By using Proposition 1, we obtain the following PBES:

$$x_0 \stackrel{\mu}{=} \exists Y \in D \,.\, x_{superior}(mary, Y)$$
$$x_{supervise}(mary, alice) \stackrel{\mu}{=} \mathsf{true}$$
$$x_{supervise}(alice, mark) \stackrel{\mu}{=} \mathsf{true}$$
$$x_{superior}(X:D, Y:D) \stackrel{\mu}{=} x_{supervise}(X, Y) \vee$$
$$\exists Z \in D.(x_{supervise}(X, Z) \wedge x_{superior}(Z, Y))$$

In the rest of this paper, we will develop the use of PBESs to solve Datalog queries.

3.2   Instantiation to Parameterless BES

Among the different known techniques for solving a PBES, such as Gauss elimination with symbolic approximation, and use of patterns, under/over approximations, or invariants, we consider the resolution method based on transforming the PBES into a parameterless boolean equation system (BES) that can be solved by linear time and memory algorithms [11,7] when data domains are finite.

Definition 8 (Boolean Equation System). *A Boolean Equation System (*BES*)* $B = (x_0, M_1, ..., M_n)$ *is a* PBES *where data domains are removed and boolean variables, being independent from data parameters, are considered* propositional.

To obtain a direct transformation into a parameterless BES, we first described the PBES in a simpler format. This simplification step consists in introducing new variables, such that each formula at the right-hand side of a boolean equation only contains at most one operator. Hence, boolean formulae are restricted to pure disjunctive or conjunctive formulae.

Given a Datalog query $q = \langle G, R \rangle$, by applying this simplification to the PBES of Proposition 1, we obtain the following PBES:

$$x_0 \stackrel{\mu}{=} \bigvee_{:- \; q_1(d_1),...,q_m(d_m). \; \in G} g_{q_1(d_1),...,q_m(d_m)}$$

$$g_{q_1(d_1),...,q_m(d_m)} \stackrel{\mu}{=} \bigwedge_{i:=1}^{m} x_{q_i}(d_i)$$

$$x_p(d:D) \stackrel{\mu}{=} \bigvee_{p(d) \; :- \; p_1(d_1),...,p_m(d_m). \; \in R} r_{p_1(d_1),...,p_m(d_m)}$$

$$r_{p_1(d_1),...,p_m(d_m)} \stackrel{\mu}{=} \bigwedge_{i:=1}^{m} x_{p_i}(d_i)$$

By applying the instantiation algorithm of Mateescu [11], we eventually obtain a parameterless BES, where all possible values of each typed data terms have been enumerated over their corresponding finite data domains.

The resulting implicit parameterless BES is defined as follows, where $\preceq$ is the standard preorder of relative generality (instantiation ordering).

$$x_0 \stackrel{\mu}{=} \bigvee_{:-\ q_1(d_1),...,q_m(d_m). \ \in G} g_{q_1(d_1),...,q_m(d_m)} \tag{3}$$

$$g_{q_1(d_1),...,q_m(d_m)} \stackrel{\mu}{=} \bigvee_{1\leq i\leq m,\ e_i\in D_i\wedge d_i\preceq e_i} g^c_{q_1(e_1),...,q_m(e_m)} \tag{4}$$

$$g^c_{q_1(e_1),...,q_m(e_m)} \stackrel{\mu}{=} \bigwedge_{i:=1}^{m} x_{q_{i_{e_i}}} \tag{5}$$

$$x_{p_d} \stackrel{\mu}{=} \bigvee_{p(d)\ :-\ p_1(d_1),...,p_m(d_m). \ \in R} r_{p_1(d_1),...,p_m(d_m)} \tag{6}$$

$$r_{p_1(d_1),...,p_m(d_m)} \stackrel{\mu}{=} \bigvee_{1\leq i\leq m,\ e_i\in D_i\wedge d_i\preceq e_i} r^c_{p_1(e_1),...,p_m(e_m)} \tag{7}$$

$$r^c_{p_1(e_1),...,p_m(e_m)} \stackrel{\mu}{=} \bigwedge_{i:=1}^{m} x_{p_{i_{e_i}}} \tag{8}$$

Observe that Equation 1 is transformed into a set of parameterless equations (3, 4, 5). First, Equation 3 describes the set of parameterised goals $g_{q_1(d_1),...,q_m(d_m)}$ of the query. Then, Equation 4 represents the instantiation of each variable parameter d_i to the possible values e_j from the domain. Finally, Equation 5 states that each instantiated goal $g^c_{q_1(e_1),...,q_m(e_m)}$ is satisfied whenever the values e_j make all predicates q_i of the goal true. Similarly, Equation 2 (describing Datalog rules) is encoded into a set of parameterless equations (6, 7, 8).

3.3 Optimizations

The parameterless BES described above is inefficient since it adopts a brute-force approach that, in the very first steps of the computation (Equation 4), enumerates all possible tuples of the query. It is well-known that a Datalog program runs in $O(n^k)$ time, where k is the largest number of variables in any single rule, and n is the number of constants in the facts and rules. Similarly, for a simple query like `:- superior(X,Y).`, with X and Y elements of a domain D of size 10 000, Equation 4 will generate D^2, *i.e.*, 10^8, boolean variables representing all possible combinations of values X and Y in relation `superior`. Usually, for each atom in a Datalog program, the number of facts that are given or inferred by the Datalog rules is much lower than the *domain's size* to the power of *atom's arity*. Ideally, the Datalog query evaluation should enumerate (given or inferred) facts only *on-demand*.

Among the existing optimizations for top-down evaluation of Datalog queries, the so-called *Query-Sub-Query* [16] technique consists in minimizing the number of tuples derived by a rewriting of the program based on the propagation of bindings. Basically, the method aims at keeping the bindings of variables between atoms $p(a)$ in a rule. In our Datalog evaluation technique based on BES, we adopt a similar approach: two boolean equations (Equations 4 and 7 slightly modified) only enumerate the values of variable arguments that appear more than once in the body of the corresponding Datalog rule, otherwise arguments are kept unchanged. Moreover, if the atom $p(a)$ is part of the Extensional Database, the only possible values of its variable arguments are values that reproduce a given fact of the Datalog program. We note D_i^p the subdomain of D that contains all possible values of the ith variable argument of p if p is in Extensional Database, otherwise $D_i^p = D$. Hence, the resulting BES resolution is likely to process fewer facts and be more efficient than the brute-force approach.

Following this optimization technique, a parameterless BES can directly be derived from the previous BES representation which we define as follows:

$$x_0 \overset{\mu}{=} \bigvee_{:-\ q_1(d_1),\dots,q_m(d_m).\ \in G} g_{q_1(d_1),\dots,q_m(d_m)} \tag{9}$$

$$g_{q_1(d_1),\dots,q_m(d_m)} \overset{\mu}{=} \bigvee_{\substack{\{a_1,\ \dots,\ a_m\}\in(\{V\cup D_1^{q_1}\}\times\dots\times\{V\cup D_1^{q_m}\})\ | \\ if\ (\exists\ j\in[1..m],\ j\neq i\ |\ d_i=d_j\wedge d_i\in V) \\ then\ a_i\in D_1^{q_i}\ \wedge\ (\forall\ j\in[1..m],\ d_i=d_j\ |\ a_j:=a_i)\ else\ a_i:=d_i}} g_{q_1(a_1),\dots,q_m(a_m)}^{pc} \tag{10}$$

$$g_{q_1(a_1),\dots,q_m(a_m)}^{pc} \overset{\mu}{=} \bigwedge_{i:=1}^{m} x_{q_{i_{a_i}}} \tag{11}$$

$$x_{q_a} \overset{\mu}{=} x_{q_a}^f \vee x_{q_a}^r \tag{12}$$

$$x_{q_a}^f \overset{\mu}{=} \bigvee_{(e:=a\ \wedge\ a\in C)\ \vee\ (e\in D_1^q\ \wedge\ a\in V)\ |\ q(e).\in R} x_{q_e}^c \tag{13}$$

$$x_{q_e}^c \overset{\mu}{=} \text{true} \tag{14}$$

$$x_{p_a}^r \overset{\mu}{=} \bigvee_{p(a)\ :-\ p_1(d_1),\dots,p_m(d_m).\ \in R} r_{p_1(d_1),\dots,p_m(d_m)} \tag{15}$$

$$r_{p_1(d_1),\dots,p_m(d_m)} \overset{\mu}{=} \bigvee_{\substack{\{a_1,\ \dots,\ a_m\}\in(\{V\cup D_1^{p_1}\}\times\dots\times\{V\cup D_1^{p_m}\})\ | \\ if\ (\exists\ j\in[1..m],\ j\neq i\ |\ d_i=d_j\wedge d_i\in V) \\ then\ a_i\in D_1^{p_i}\ \wedge\ (\forall\ j\in[1..m],\ d_i=d_j\ |\ a_j:=a_i)\ else\ a_i:=d_i}} r_{p_1(a_1),\dots,p_m(a_m)}^{pc} \tag{16}$$

$$r_{p_1(a_1),\dots,p_m(a_m)}^{pc} \overset{\mu}{=} \bigwedge_{i:=1}^{m} x_{p_{i_{a_i}}} \tag{17}$$

Observe that Equations 9, 11, 15 and 16 correspond respectively to Equations 3, 5, 6 and 8 of previous BES definition with only a slight renaming of generated boolean variables. The important novelty is that, instead of enumerating

all possible values of the domain, as it is done in Equation 4, the corresponding new Equation 10 only enumerates the values of variable arguments that are repeated in the body of a rule, otherwise variable arguments are kept unchanged *i.e.*, $a_i := d_i$. Indeed, the generated boolean variables $g^{pc}_{q_1(a_1),...,q_m(a_m)}$ may still refer to atoms containing variable arguments. Thus, the combinatorial explosion of possible tuples is avoided at this point and delayed to future steps. Equation 12 generates two boolean successors for variable x_{q_a}: $x^f_{q_a}$ when q is a relation that is part of the Extensional Database, and $x^r_{q_a}$ when q is defined by Datalog rules. In Equation 13, each value of a (variable or constant) that leads to a given fact $q(e)$. of the program, generates a new boolean variable $x^c_{q_e}$, that is true by definition of a fact. Equation 15 simply infers Datalog rules whose head is p_a. Note that Equations 10, 13, and 16 enumerate possible values of subdomains $D^{p_i}_1$ instead of full domain D. With the Datalog program described in Example 1, this restriction would consist in using two new subdomains $D^{supervise}_1 = \{mary, alice\}$ and $D^{supervise}_2 = \{alice, mark\}$ instead of domain $D = \{mary, alice, mark\}$ for the values of each variable argument in relation *supervise*.

3.4   Solutions Extraction

Considering the optimized parameterless BES defined above, the query satisfiability problem is reduced to the local resolution of boolean variable x_0. The value (true or false) computed for x_0 indicates whether there exists at least one satisfiable goal in G. We can remark that the BES representing the evaluation of a Datalog query is only composed of one equation block containing alternating dependencies between disjunctive and conjunctive variables. Hence, it can be solved by optimized depth-first search (DFS) for such a type of equation block. However, since the DFS strategy can only conclude the existence of a solution to the query by computing a minimal number of boolean variables, it is necessary to use a breadth-first search (BFS) strategy to compute all the different solutions of a Datalog query. Such a strategy will "force" the resolution of all boolean variables that have been put in the BFS queue, even if the satisfiability of the query has been computed in the meantime. Consequently, the solver will compute all possible boolean variables $x^c_{q_e}$, which are potential solutions for the query. Upon termination of the BES resolution (ensured by finite data domains and table-based exploration), query solutions, *i.e.*, combinations of variable values $\{e_1, \ldots, e_m\}$, one for each atom of the query that lead to a satisfied query, are extracted from all boolean variables $x^c_{q_e}$ that are reachable from boolean variable x_0 through a path of true boolean variables.

4   Application to JAVA Program Analysis

There is a strong interest in developing efficient demand-driven evaluation techniques that are applicable for program analysis since they naturally fit into *Integrated Development Environments* (IDEs) that dynamically provide analysis results to a programmer during the development of its code. Actually, demand-driven techniques are often considered better than global approaches during the

program development since they usually encounter errors more rapidly by exploring only a portion of the code.

This also applies to Datalog queries: the more specific the query (*i.e.*, the higher the number of constant arguments), the better demand-driven resolution of the query, as compared to a global-based method, since only facts from the Datalog program that are necessary to answer the query will be inferred.

4.1 Datalog-Based Program Analysis

The Datalog approach to static program analysis [18] can be summarized as follows. Each program element, namely variables, types, code locations, function names, are grouped in their respective *domains*. Thus, each argument $a_{i,j}$ of a predicate symbol p_i is typed by a domain $A_{i,j}$ of values. Hence, atoms $p_i :$ $\wp(A_{i,j})^{n_i} \to \mathbb{B}$ are considered as *relations* among program's elements defined in their respective domains. By considering only finite program domains, Datalog programs are ensured to be *safe* (query evaluation generates a finite set of facts). Each program statement is decomposed into *basic program operations*, namely load, store, assignment, and variable declarations. Each kind of basic operation is described by a relation in a Datalog program. A program operation is then described as a tuple satisfying the corresponding relation.

Example 2. Consider the simple JAVA program [18] on the left-hand side of the following example:

```
public A foo { ...   p = new Object(); /* o1 */        vP_0(p, o1).
                     q = new Object(); /* o2 */        vP_0(q, o2).
                     p.f = q;                    ⇒     store(p, f, q).
                     r = p.f; ...  }                   load(p, f, r).
```

where `o1` and `o2` are heap allocations (extracted from corresponding bytecode). The Datalog pointer analysis approach consists first in extracting Datalog constraints (relations on the right-hand side of the above example) from the program. Then, it deduces further pointer-related information as output, like points-to relations vP from local variables and method parameters to heap objects as well as points-to relations hP between heap objects through field identifiers. The relation `vP_0` consists of initial points-to relations (v, h) of a program, *i.e.*, `vP_0(v,h)` holds if there exists a direct assignment within the program between a reference to a heap object `h` and a variable `v`. Other Datalog constraints such as `store` and `load` relations are calculated similarly.

In this framework, a *program analysis* consists in either querying extracted relations or computing new relations from existing ones. Datalog is both used to specify a static code analysis as well as to evaluate queries on given and inferred facts from the analysis.

Example 3. Consider the Datalog program that defines context-insensitive points-to analysis given in Fig. 1 (`pa.datalog` [18]).

The program consists of three parts:

```
### Domains
V 262144 variable.map
H 65536 heap.map
F 16384 field.map

### Relations

vP_0      (variable : V, heap : H)            inputtuples
store     (base : V, field : F, source : V)   inputtuples
load      (base : V, field : F, dest : V)     inputtuples
assign    (dest : V, source : V)              inputtuples
vP        (variable : V, heap : H)            outputtuples
hP        (base : H, field : F, target : H)   outputtuples

### Rules

vP (V1, H1)     :- vP_0(V1, H1).
vP (V1, H1)     :- assign(V1, V2), vP(V2, H2).
hP (H1, F1, H2) :- store(V1, F1, V2), vP(V1, H1), vP(V2, H2).
vP (V2, H2)     :- load (V1, F1, V2), vP(V1, H1), hP(H1, F1, H2).
```

Fig. 1. Datalog specification of a context-insensitive points-to analysis

1. A declaration of *domains*, with domain names and size (number of elements).
2. A list of *relations* specified by a predicate symbol, its arguments over specific domains and whether it is derived from an applicable Datalog rule (value `outputtuples`), or extracted from the program source code (value `inputtuples`).
3. A finite set of Datalog *rules*, defining the `outputtuples` relations.

A Datalog query consists of a set of goals over the relations defined in the Datalog program, *e.g.*, `:- vP(X,Y).` where X and Y are variable arguments of vP, meaning computing the complete set of variables in the domain of X that may point to any heap object Y at any point during program execution. From the initial relations of Example 2, we deduce by inferring the Datalog rules given by Fig. 1 the following output relations: `vP(p, o1)`, `vP(q, o2)`, `hP(o1, f, 02)`, and `vP(r, o2)`. For instance, the output relation $vP(r, o2)$ indicates that variable r points to same heap allocation $o2$ as variable q.

4.2   Datalog-Based Program Analyzer

We implemented the Datalog query transformation to BES in a powerful, fully automated Datalog solver tool, called DATALOG_SOLVE, developed within the CADP verification toolbox [8]. Without loss of generality, in this section, we describe the DATALOG_SOLVE tool focusing on JAVA program analysis. Other source languages and classes of problems can be specified in Datalog and solved by our tool.

DATALOG_SOLVE takes three different inputs (see Fig. 2): the domain definitions (`.map`), the Datalog constraints or *facts* (`.tuples`), and a Datalog query

$q = \langle G, R \rangle$ (`.datalog`, like `pa.datalog` in Fig. 1). The domain definitions state the possible values for each predicate's argument of the query. These are meaningful names for the numerical values that are used to efficiently described the Datalog constraints. For example, in the context of pointer analyses, variable names (`var.map`) and heap locations (`heap.map`) are two domains of interest. The Datalog constraints represent information relevant for the analysis. For instance, `vP0.tuples` gives all direct references from variables to heap objects in a given program. For efficiency reasons, these combinations are described by numerical values in the range $0..(domain\ size - 1)$. Both, domain definitions and facts are specified in the `.datalog` input file (see Fig. 1) and they are automatically extracted from program source code by using the JOEQ compiler framework [17] that we slightly modified to generate *tuple-based* instead of BDD-*based* input relations.

DATALOG_SOLVE (120 lines of LEX, 380 lines of BISON and 3 500 lines of C code) proceeds in two steps:

1. The front-end of DATALOG_SOLVE constructs the optimized BES representation given by Equations 9-17 by extracting from the inputs (a particular analysis) the set of Datalog goals, rules and facts defining each boolean variable.
2. The back-end of our tool carries out the demand-driven generation, resolution and interpretation of the BES by means of the generic CÆSAR_SOLVE library of CADP, devised for local BES resolution and diagnostic generation.

This architecture clearly separates the implementation of Datalog-based static analyses from the resolution engine, which can be extended and optimized independently.

Upon termination (ensured by safe input Datalog programs), DATALOG_SOLVE returns both the query's satisfiability and the computed answers represented numerically in various output files (`.tuples` files). The tool
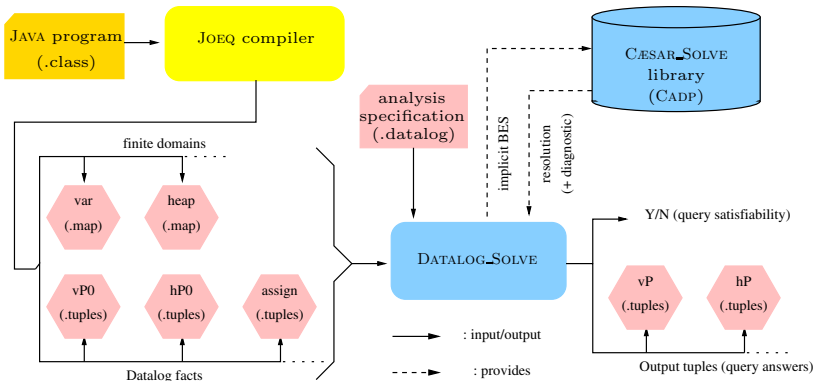

Fig. 2. JAVA program analysis using DATALOG_SOLVE tool

Table 1. Description of the Java projects used as benchmarks

Name	Description	Classes	Methods	Vars	Allocs
freetts (1.2.1)	speech synthesis system	215	723	8K	3K
nfcchat (1.1.0)	scalable, distributed chat client	283	993	11K	3K
jetty (6.1.10)	server and servlet container	309	1160	12K	3K
joone (2.0.0)	Java neural net framework	375	1531	17K	4K

Table 2. Times (in seconds) and peak memory usages (in megabytes) for each benchmark and context-insensitive pointer analysis

Name	time (sec.)	memory (Mb.)
freetts (1.2.1)	10	61
nfcchat (1.1.0)	8	59
jetty (6.1.10)	73	70
joone (2.0.0)	4	58

takes as a default query the computation of the least set of facts that contains all the facts that can be inferred using the given rules. This represents the worst case of a demand-driven evaluation and computes all the information derivable from a Datalog program.

4.3 Experimental Results

The Datalog_Solve tool was applied to a number of Java programs by computing the context-insensitive pointer analysis described above. To test the scalability and applicability of the transformation, we applied our technique to four of the most popular 100% Java projects on Sourceforge that could compile directly as standalone applications and were used as benchmarks for the Bddbddb tool [18]. They are all real applications with tens of thousands of users each. Projects vary in the number of classes, methods, variables, and heap allocations. The information details, shown on Table 1, are calculated on the basis of a context-insensitive callgraph precomputed by the Joeq compiler. All experiments were conducted using Java JRE 1.5, Joeq version 20030812, on a Intel Core 2 T5500 1.66GHz with 3 Gigabytes of RAM, running Linux Kubuntu 8.04.

The analysis time and memory usage of our context insensitive pointer analysis, shown on Table 2, illustrate the scalability of our Bes resolution and validate our theoretical results on real examples. Datalog_Solve solves the (default) query for all benchmarks in a few seconds. The computed results were verified by comparing them with the solutions computed by the Bddbddb tool on the same benchmark of Java programs and analysis.

5 Conclusions and Future Work

This paper presents a novel approach to solve Datalog queries based on Boolean Equation System (Bes) resolution and its application to program analysis. By

using a local fixed point computation of BESs, our technique not only keeps the robustness of bottom-up over top-down evaluation semantics (problem of repeated computations and infinite loops), but also preserves the effectiveness of demand-driven techniques by taking advantage of constants and constraints that are part of the query's goals in order to reduce the search space. A new deductive database solver, called DATALOG_SOLVE, was designed and implemented, and several JAVA programs were analyzed modulo a context-insensitive pointer analysis encoded in Datalog. The tool architecture is based on the well-established verification framework CADP, which provides a generic library for local BES resolution.

As a future work, we plan to endow DATALOG_SOLVE with new, optimized strategies for local BES resolution, *e.g.*, rewriting of Datalog rules to allow goal-directed bottom-up evaluation, as in the *Magic sets* approach with complexity guarantees [14]. Another interesting improvement we plan to explore is to use the rewriting logic framework implemented in the reflective, functional programming language Maude [6] as a solver for Java program analyses using reflection. Finally, we could benefit from the regular structure of our BES encoding by distributing the BES resolution over a network of workstations with balanced partitioning preserving locality, similarly to the work of [9] that successfully applied a distributed BES resolution algorithm to numerous verification problems.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Reading (1995)
2. Andersen, H.R.: Model checking and boolean graphs. Theoretical Computer Science 126(1), 3–30 (1994)
3. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.D.: Magic Sets and Other Strange Ways to Implement Logic Programs. In: Proc. 5th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems PODS 1986, pp. 1–15. ACM Press, New York (1986)
4. Ceri, S., Gottlob, G., Tanca, L.: Logic Programming and Databases. Springer, Heidelberg (1990)
5. Chen, T., Ploeger, B., van de Pol, J., Willemse, T.A.C.: Equivalence Checking for Infinite Systems Using Parameterized Boolean Equation Systems. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 120–135. Springer, Heidelberg (2007)
6. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
7. van Dam, A., Ploeger, B., Willemse, T.A.C.: Instantiation for Parameterised Boolean Equation Systems. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigun, H. (eds.) ICTAC 2008. LNCS, vol. 5160, pp. 440–454. Springer, Heidelberg (2008)
8. Garavel, H., Mateescu, R., Lang, F., Serwe, W.: CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 158–163. Springer, Heidelberg (2007)

9. Joubert, C., Mateescu, R.: Distributed On-the-Fly Model Checking and Test Case Generation. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 126–145. Springer, Heidelberg (2006)

10. Liu, X., Smolka, S.A.: Simple Linear-Time Algorithms for Minimal Fixed Points. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 53–66. Springer, Heidelberg (1998)

11. Mateescu, R.: Local Model-Checking of an Alternation-Free Value-Based Modal Mu-Calculus. In: Proc. 2nd Int'l Workshop on Verication, Model Checking and Abstract Interpretation VMCAI 1998 (1998)

12. Mateescu, R., Thivolle, D.: A Model Checking Language for Concurrent Value-Passing Systems. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 148–164. Springer, Heidelberg (2008)

13. Reps, T.W.: Solving Demand Versions of Interprocedural Analysis Problems. In: Fritzson, P.A. (ed.) CC 1994. LNCS, vol. 786, pp. 389–403. Springer, Heidelberg (1994)

14. Tekle, K.T., Hristova, K., Liu, Y.A.: Generating specialized rules and programs for demand-driven analysis. In: Meseguer, J., Roşu, G. (eds.) AMAST 2008. LNCS, vol. 5140, pp. 346–361. Springer, Heidelberg (2008)

15. Ullman, J.D.: Principles of Database and Knowledge-Base Systems. The New Technologies, vol. I and II. Computer Science Press (1989)

16. Vieille, L.: Recursive Axioms in Deductive Databases: The Query/Subquery Approach. In: Proc. 1st Int'l Conf. on Expert Database Systems EDS 1986, pp. 253–267 (1986)

17. Whaley, J.: Joeq: a Virtual Machine and Compiler Infrastructure. In: Proc. Workshop on Interpreters, Virtual Machines and Emulators IVME 2003, pp. 58–66. ACM Press, New York (2003)

18. Whaley, J., Avots, D., Carbin, M., Lam, M.S.: Using Datalog with Binary Decision Diagrams for Program Analysis. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 97–118. Springer, Heidelberg (2005)

19. Zheng, X., Rugina, R.: Demand-driven alias analysis for C. In: Proc. 35th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages POPL 2008, pp. 197–208. ACM Press, New York (2008)

Author Index