4th International Workshop on

Harnessing Theories for Tool Support in Software
TTSS’10

East China Normal University, Shanghai, China, November 15, 2010

Preliminary Proceedings

Editors: Min Zhang and Volker Stolz

November 2010
UNU-IIST and UNU-IIST Reports

UNU-IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macao, and was founded in 1991. It started operations in July 1992. UNU-IIST is jointly funded by the government of Macao and the governments of the People’s Republic of China and Portugal through a contribution to the UNU Endowment Fund. As well as providing two-thirds of the endowment fund, the Macao authorities also supply UNU-IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU-IIST is to assist developing countries in the application and development of software technology.

UNU-IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Schools and Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU-IIST, and
7. Dissemination, in which UNU-IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU-IIST is on formal methods for software development. UNU-IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU-IIST produces a report series. Reports are either Research [R], Technical [T], Compendia [C] or Administrative [A]. They are records of UNU-IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU-IIST at P.O. Box 3058, Macao or visit UNU-IIST’s home page: http://www.iist.unu.edu, if you would like to know more about UNU-IIST and its report series.

Peter Haddawy, Director
Abstract

This compendium consists of the papers presented at the TTSS’10 workshop for the convenience of the workshop participants.
Contents

1 Keynote
Harnessing theories for Real-Time Java Development
Anders P. Ravn

2 Invited Contributions
The Overture Initiative—Integrating Tools for VDM
Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl,
Marcel Verheoef

Model-Driven Software Engineering In Sustainable Health Care
Zhiming Liu

An Extension to Sequence Diagram for Expressing Runtime Monitoring Concerns
Xian Zhang, Wei Dong and Zhichang Qi

3 Regular Contributions
Models of Rate Restricted Communication for Concurrent Objects
Rudolf Schlatte, Einar Broch Johnsen, Fatemeh Kazemeyni and Silvia Lizeth Tapia Tarifa

Construct Aspectual Models from Requirement Documents for Automotive Software
Xiaoqian Liu

Verification of A Key-Chain Based TTP Transparent CEM Protocol
Zhiyuan Liu, Jun Pang and Chenyi Zhang

Runtime Verification for LTL Schemas
Martin Leucker, Changzhi Zhao, Bin Zheng and Wei Dong

Adaptive-Step-Size Numerical Methods in Rewriting-Logic-Based Formal Analysis of Interacting Hybrid Systems
Muhammad Fadlisyah, Erika Abrahám and Peter Olveczky

On-The-Fly Path Reduction
Sebastian Biallas, Joerg Brauer, Dominique Gueckel and Stefan Kowalewski

A Parallel Approach to Concolic Testing with Low-cost Synchronization
Xiao Yu, Shuai Sun, Zheng Wang, Geguang Pu and Siyuan Jiang
Harnessing Theories for Tool Support in Software

Advisory Committee
Patrick Cousot, ENS, France
Jifeng He, ECNU, China
Mathai Joseph, TATA, India
Zhiming Liu, UNU-IIST, Macau, China
Bertrand Meyer, ETH Zurich, Switzerland
Jim Woodcock, Univ. of York, UK

Program Committee
Luis Barbosa
Universidade do Minho, Portugal
Wei Dong
National University of Defense Technology, China

Clemens Grelck
Univ. of Amsterdam, The Netherlands
Ralf Huuck
NICTA, Australia

Peter Gorm Larsen
Engineering College of Aarhus, DK
Einar B. Johnsen
University of Oslo, Norway

Carlo Montangero
University of Pisa, Italy
Thomas Noll
RWTH Aachen University, Germany

Jun Pang
Univ. Luxembourg, Luxembourg
Anders P. Ravn
Aalborg University, DK

Wuwei Shen
Western Michigan University, US
Axel Simon
TU München, Germany

Meng Sun
Peking University, China
R. Venkatesh
TATA, India

Michael Weber
Univ. Twente, The Netherlands
Naijun Zhan
Institute of Software, CAS, China

Min Zhang (Co-Chair)
East China Normal University
Volker Stolz (Chair)
University of Oslo, Norway
Shanghai, China
& UNU-IIST, Macau SAR, China
Keynote: Harnessing theories for Real-Time Java Development

Anders P. Ravn
Dept. of Computer Science
Aalborg University, Denmark

Abstract

It may still take a while before embedded software is synthesised directly from high level models without interference by fallible human programmers. That raises the question of what researchers can do in the meantime to improve quality and efficiency of the development process? The hypothesis we set up some years ago is that an incremental move away from C and real-time kernels to a more structured language will improve the process. Here Java has been our choice, because it comes with defined profiles for real-time programming and because the profile is supported by platforms that have been demonstrated to work. However, the profile cannot in itself ensure that applications perform predictably. There are many issues, some examples are: ensuring that only allowed features and constructs are used, checking that platform resources (memory and processor time) meet the demands of the executing program, and providing interfaces to special purpose hardware. Therefore we have engaged very much in harnessing theories and adapting tools to assist in verifying profile dictated properties and conformance to platform limitations. The results so far have been encouraging, and we will use them to comment on what kind of theories and tools we expect to see harnessed in a truly supportive R-T Java development environment.

Joint work with Thomas Bøgholm, René Rydhof Hansen, Stephan Korsholm, Kim Larsen, Martin Schoeberl, Arne Skou, Hans Søndergaard, and Bent Thomsen.
The Overture Initiative
Integrating Tools for VDM

www.overturetool.org

Peter Gorm Larsena  Nick Battleb  Miguel Ferreirac
John Fitzgeraldd  Kenneth Lausdahla  Marcel Verhoefe

{pgl@iha.dk, kel@iha.dk} Aarhus School of Engineering, Denmark
b nick.battle@uk.fujitsu.com, Fujitsu, UK
c m.ferreira@sig.nl, Software Improvement Group, Netherlands
d john.fitzgerald@ncl.ac.uk, Newcastle University, UK
e Marcel.Verhoef@chess.nl, Chess, Netherlands

Abstract

Overture is a community-based initiative that aims to develop a common open-source platform integrating a range of tools for constructing and analysing formal models of systems using VDM. The mission is to both provide an industrial-strength tool set for VDM and also to provide an environment that allows researchers and other stakeholders to experiment with modifications and extensions to the tools and language. This paper presents the current status and future vision of the Overture project as an update of [13].

1 Introduction

Formal methods are mathematical techniques for the modelling, analysis and development of software and systems [28]. Their use is motivated by the expectation that, as in other engineering disciplines, performing an appropriate mathematical analysis can promote early discovery of defects and contribute directly to the increased reliability and robustness of a design.

The Vienna Development Method (VDM)\(^1\) is one of the most mature formal methods [12,6]. The method focuses on the development and analysis of a system model expressed in a formal language. The language’s formality

\(^1\) http://www.vdmportal.org/
enables developers to use a wide range of analytic techniques, from testing to mathematical proof, to verify the consistency of a model and its correctness with respect to an existing statement of requirements. The VDM modelling language has been gradually extended over time. Its most basic form (VDM-SL), standardised by ISO [14] supports the modelling of the functionality of sequential systems. Extensions support object-oriented modelling and concurrency [7], real-time computations [19] and distributed systems [24]. There is consequently a need to provide a common basis for supporting the analytic tools covering all these extensions.

Currently, the most feature-rich tool available is VDMTools [5,8], a commercial product which includes syntax- and type-checking facilities, an interpreter to support testing and debugging of models, proof obligation generators that produce formal specifications of integrity checks that can not be performed statically, and code generators. From the perspective of modern Integrated Development Environments (IDEs), VDMTools has some weaknesses, including a relative lack of extensibility.

The Overture project aims to provide at least as much functionality as VDMTools, but built on an open and extensible platform based on the Eclipse framework. An alternative VDM tool, VDMJ [2], provides many of the features of VDMTools. It is small, relatively fast, pure Java and open source but has only a command-line interface. These characteristics make it suitable for integration with the Eclipse IDE and allow it to be distributed freely with the Overture project.

This paper provides a report on the current state of Overture, and plans for its future development. It provides a short introduction to Eclipse (Section 3) and to the Overture plug-in architecture (Section 4). Section 4 then provides a short introduction to the features that currently exist in Overture. Section 5 explains the development strategy for the Overture tool. Finally Section 6 describes future development plans.

2 VDM

A VDM model describes data and functionality. User-defined data types are built from base types and constructors defining compositions such as record and union types, and collections such as sets, sequences and mappings. Types may be constrained by invariants which are arbitrarily complex logical predicates that all elements of the defined type respect. Persistent data is modelled as state variables, again restricted by invariants.

Functionality in VDM is expressed as operations over state variables or as referentially transparent functions over the available data types. Functionality can be defined explicitly by means of simple algorithms or implicitly by means of postconditions characterising the required relationship between the
results and the inputs. In either style, the assumptions under which a function or operation may be called are recorded as logical preconditions. Since functionality can be defined implicitly, and given the expressiveness of the logic for defining functionality, VDM models are not necessarily executable. There is, nevertheless, a large executable subset of the modelling language, allowing validation of models by dynamic testing.

The VDM modelling language has a formal semantics, allowing users to conduct more sophisticated static analyses than regular syntax and type checking. For example, it is possible to analyse models to detect violations caused by potential misuse of partial operators (e.g. indexing out of range) or by potential violation of invariants. The conditions for a consistent model are recorded as proof obligations which are logical predicates that can be proven to hold using the mathematical semantics of the language. Such proof obligations cannot be checked completely automatically in general because of the expressiveness of the modelling language, but they can be proved using modern proof technology or manual inspection.

The VDM extensions that support object-oriented structuring use concepts common to UML and so there is scope for integrating the tools that support formal analysis of VDM models with graphical views of the models via UML class diagrams, for example. Other VDM extensions to support concurrency, distribution and real-time systems open the possibility of providing tools that allow exploration of complex run-time behaviours, again via graphical representations.

3 Eclipse

Eclipse is intended to serve as a common platform that “blends separately developed tools into a well designed suite”\(^2\). The Eclipse platform user interface is based on editors, views, and perspectives. A perspective defines an appropriate collection of views, their layout, and applicable actions for a given user task. A view is a workbench part that can navigate a hierarchy of information or display properties for an object. Figure 2 shows a screenshot of the Overture IDE built on Eclipse.

4 Overture Architecture and Features

Overture uses a plug-in architecture consisting of components that supply core functionality and components that interact directly with the user through the Eclipse GUI. The dependencies between these components are quite complex but it is notable that most of the components depend on the abstract syntax

\(^2\) http://help.eclipse.org/
trees (ASTs) generated from the parsing of a VDM model. The current and envisaged Overture components are shown in Figure 1. The remainder of this section describes each of the main components in more detail.

4.1 **Overture Parser and AST**

VDM models are input in plain text using a standard concrete syntax. A parser is therefore required to transform the textual model into an AST that all plug-ins may use to access the model. The parser developed for Overture is a recursive decent parser written in Java. The parser is for example used in the Overture IDE editor to generate warnings used to highlight syntax problems. This parser also permits VDM source to be embedded within a \LaTeX document.

4.2 **Overture IDE**

The basic Overture IDE [18] is illustrated in Figure 2. The **Explorer view** (on the left of the figure) allows the user to create, select, and delete Overture projects and navigate between the files in these projects. The panel to the right of the Explorer is the **editor area**. Since there are currently several dialects of VDM, the particular editor that opens in the panel depends on the dialect of VDM being used in the current project. The **Outline view**, to the right of the editor, summarises the content of the file currently selected in the editor, displaying declared classes, instance variables, values, types, functions, operations etc. The **Problems view**, shown here at the bottom of the window, displays information generated by Overture, such as warnings and errors relevant to the current project.

Most of the other features of the workbench, such as the menu or the
toolbar are similar to those of other familiar applications. There is also a special menu with Overture-specific functionality. One convenient feature is a toolbar of shortcuts to switch between different perspectives that appears on the right side of the screen; these vary dynamically according to context and history.

4.3 Type Checking

Static type checking is performed automatically as a model is entered via the Overture editor. The simplest errors are often typographical in origin and are readily detected, for example entering the type of a parameter incorrectly, so that the type name used cannot be found. More subtle errors can indicate semantic problems with the model, such as when trying to compose mappings where the domain and range of the operands do not match. Warnings are also generated for unused variables, obviously unreachable code, and so on.

Overture supports type checking with “possible” semantics. An expression is considered type correct if there is an assignment of values that yields a result of the correct type, regardless of whether there are other assignments that could be type incorrect. For example, consider a function with a return type composed of natural numbers constrained by an invariant to be less than 10. If the function has a body that returns a real number, this is considered correct by the checker because a real number is possibly a natural number.
less than 10. There is a risk that, at run time, the real value is actually not a whole number, or is less than zero or greater than 10. The VDM run-time system performs dynamic type checking which catches such errors. By contrast “definite” semantics for type checking raises errors in all cases where there might be a run-time error, regardless of whether this is guarded against.

4.4 Proof Obligation Generation

As indicated above, static type correctness does not guarantee freedom from run-time errors. The formality of VDM’s semantics makes it possible to generate, for a model, a set of logical assertions that must be true if that model is to be dynamically type correct and semantically consistent. We call these assertions proof obligations. Both VDMTools and VDMJ provide tools that automatically generate proof obligations for a given model [3].

Proof obligations are generated for a range of properties relating to the internal consistency of a model. These include obligations checking for potential run-time errors caused by misapplication of partial operators, consistency of results with postconditions and termination of recursion [20,21]. In our type checking example from Section 4.3, proof obligations would be generated that state that, whenever a real number is returned as a result of the function, the real value satisfies the invariant of the return type. The obligations will take account of the context in the model, so for example a result returned from within nested if or else clauses would yield a proof obligation that was qualified with the same tests as the if-expressions which lead to the returned result.

Proof obligations can be verified to different levels of confidence using a range of techniques including manual inspection and formal mathematical proof.

4.5 Proof Support

The highest level of confidence in the validity of proof obligations can be gained by constructing machine-verifiable formal proofs. The production of such proofs manually is a complex and error-prone process, but it is susceptible to automated support. There is currently no VDM-specific theorem prover, but it is possible to exploit off-the-shelf proof technology. A model can be translated into a theory (in the prover’s language) that captures its semantics, and the proof obligations can be set as proof commands. Along with the translation, the connection to the theorem prover is complemented with a set of VDM-specific theorems and proof tactics.

The viability of this approach to automated verification in VDM was established some time ago [1]. More recently, an automated proof support system was developed in the Overture context [26,27]. A VDM prototype model of
a translator tool was developed, converting VDM to HOL. The interaction between the proof obligation generators, the translator tool (automatically generated from the VDM++ prototype) and HOL was later implemented in a Java proof component that integrates all this functionality. Integrating the proof component in Eclipse is now a question of improving interaction and usability of a tool which design is stable. This is a part of the future work in Overture.

4.6 Interpreter/debugger

Automatic verification of proof obligations is an extremely powerful analytic technique, but it is also time consuming, even with tool support. However, a great deal of value can still be gained from less formal analysis of a model. For example, proof obligations may be discharged by informal inspection. The simplest level of validation for a VDM model is to run a simulation using an interpreter, and in the case of problems, to step through the evaluation of expressions using a debugger.

Overture provides an interpreter for all of the VDM language dialects, allowing the evaluation to be stepped through using the normal Eclipse debugging views, i.e. setting breakpoints, inspecting local and state variables, call stack etc. Typically, a model is provided with abstract data to represent the world being modelled, and functions or operations are evaluated over that data to produce abstract results representing the outcome of the action performed by the model. If the interpreter results are as expected, confidence in the model is strengthened. If the results are not as expected, the debugger can be used to find out why.

The interpreters in VDMTools and VDMJ both keep track of how much of the model is covered during an evaluation. This allows a report to be generated highlighting the parts of the model that have not been exercised by a set of tests. The intention is to integrate this with the Overture editor for immediate feedback in the IDE.

4.7 Combinatorial Testing

The combinatorial testing feature supports the automatic execution of a large number of test cases generated from templates provided in the form of trace definitions added to a VDM++ model. Trace definitions are defined as regular expressions describing possible sequences of operation calls, and are conceptually similar to the input provided to model checkers. A plug-in enabling trace expansion, evaluation and inspection is developed in two steps: first as a VDM model prototype and then as an optimised version with direct integration into VDMJ to speed up the evaluation process of large test cases [22,15].
4.8 **Bi-directional UML mapping**

The UML bi-directional mapping [16] feature supports an automatic connection between the object-oriented dialect of VDM (VDM++) and UML 2.1, allowing models developed in both notations to be kept consistent as they evolve. The feature supports two forms of link: between UML class diagrams and VDM models, and between UML sequence diagrams and VDM trace statements.

The link between class diagrams and VDM models relates the static structure of the two models. The link is bidirectional so that modifications to the class structure in one model are reflected automatically in the other.

The link between UML sequence diagrams and VDM trace statements enables a test scenario to be specified in a UML sequence diagram and then transformed into a VDM trace statement which enables the Combinatorial Testing feature to evaluate the test scenario specified in the UML sequence diagram.

4.9 **Realtime Log Viewer**

The VICE dialect of VDM allows the description of real-time concurrent processes which can be distributed over a virtual architecture of CPUs and buses. The interpreter for this dialect produces a time-stamped log file showing every internal event (e.g. operation request, activation and completion), every message handling event on the buses and thread events such as creating, swapping in and out and termination. The *Realtime Log Viewer* plug-in for Overture can read the log file and produce a visualisation of the execution in terms of the CPUs and BUSes as shown in Figure 3. It is possible to see both how busy the components are, and the details of each thread swap in the given scenario. It is also possible to visualize violations of formally specified timing requirements directly [10].

5 **Overture Development**

The Overture project’s main development technologies are Java and VDM itself. The development code, maintained under a Subversion repository at SourceForge\(^3\), is divided into:

- **core** The core components such as VDMJ, proof obligation translation and Realtime log viewer
- **ide** The Eclipse plug-ins which make up the Overture Editor.
- **tools** Additional build tools e.g. Maven Eclipse build plug-ins and Maven

\(^3\) [http://sf.net/projects/overture](http://sf.net/projects/overture)

16
VDMTools integration plug-in. Tools such as treeGen are also located under tools.

The core components are either written directly in Java or in VDM++ and then code generated to Java. The Apache Maven tool is used for project management and build automation. The Maven build tool has been extended with plug-ins to enable building Eclipse plug-ins, enabling an easy integration of VDMTools code generation as a part of the Maven build life cycle.

6 Future Plans

The Overture community plans to further develop the platform by extending existing functionality, integration with other formal methods tools and by integration of more Eclipse technology to improve ease of use. Regarding other tools, work on the link to JML [17] and Alloy [11] is in progress. Work to build a VDM-specific theorem prover is also planned. Regarding better Eclipse integration, style consistency for VDM (with CheckStyle) and task management (with Mylyn) is being considered.

A priority for the VDM community in recent years has been successful industrial deployment [9] and this influenced the priorities for the development of modelling facilities including support for object-orientation, real-time and concurrency as well as the integration of formal modelling into industrial development processes. The European Framework 7 project DESTECS (Design Support and Tooling for Embedded Control Software, www.destecs.org) [4], will develop methods and tools that combine continuous time models of systems (in tools such as 20-Sim) with discrete event controller models in VDM through co-simulation [25,23]. The approach is intended to encourage col-

\[\text{Fig. 3. Overture Realtime Log Viewer}\]
laborative multidisciplinary modelling, including modelling of faults and fault tolerance mechanisms.

Acknowledgments

We are grateful to Thomas Christensen, Jens Kielsgaard Hansen, Hans Kristian Lintrup, Hugo Macedo, David Møller, Paul Mukherjee, Jacob Porsborg Nielsen, Nico Plat, Augusto Ribeiro, Shin Sahara, Adriana Sucena Santos, Pieter van der Spek, Christian Thillermann, Sander Vermolen, Carlos Vilhena and all the other contributors who have helped to create the Overture platform and tools. Finally we would like to thank the EU for funding the FP7 project 248134, called DESTECS (www.destecs.org) where this tool is further enhanced.

References


Model-Driven Software Engineering In Sustainable Health Care

Zhiming Liu

United Nations University—Intl. Institute for Software Technology (UNU-IIST)
Macao S.A.R.

Abstract

Healthcare is of great importance in parts of the world where healthcare is very limited, and equally important in high-income countries where an aging population will need a great deal more and different support and care than in the past. This is why there are recently many international and international initiatives that promotes research on and development of health information systems (HIS) and discusses the major barriers therein. The American Academy of Engineering has also proposed Advancing Health Informatics as of the Grand Engineering Challenge. In this invited talk, we look at the state of the art and into the future of HIS. We will discuss the engineering challenges and identify some research topics. Finally, we will show how a formal model driven design method, such as rCOS, can be extended and applied to address the challenges and problems.
An Extension to Sequence Diagram for Expressing Runtime Monitoring Concerns

Xian Zhang  Wei Dong  Zhichang Qi
School of Computer
National University of Defense Technology
Changsha, China
{zhangxian,wdong,qzc}@nudt.edu.cn

Abstract
As software is becoming more pervasive in everyday life, monitoring software’s execution to prevent damage is of growing importance in many application areas. At present, runtime monitoring capability is usually added to software systems after they have already been realized. Given the complexity of today’s software, we believe that runtime monitoring concerns should be taken into account in the earlier development phase. To this end, we propose an extended design notation to integrate runtime monitoring concerns into software at the design phase. The proposed approach consists of two aspects: (1) an extension to UML sequence diagram which helps users to express runtime monitoring concerns more easily; (2) a automatical translation of the extended sequence diagram into aspect-oriented code.

Keywords: runtime monitoring, sequence diagram, aspect-oriented programming

1 Introduction
Software systems are becoming more pervasive in all aspects of everyday life. In lots of application areas, software systems have to work in a correct, secure, and reliable manner, as life or great loss in property may depend on it. At the same time, the scale and complexity of software systems is increasing dramatically. Despite significant foundational and tool support advances in software development, we are still far from developing error-free software.

Runtime monitoring is a research area that is concerned with monitoring and analyzing software’s execution and preventing from dangerous operations. Runtime monitoring usually consists of two parts: a specification part and a response (reacting behavior) part. The specification part describes the interested execution event patterns that users want to monitor, while the response
Extension | Type | Applies to | Definition
--- | --- | --- | ---
⟨⟨ specification ⟩⟩ | Stereotype | sequence diagram | the sequence diagram is a specification
occurrence | Tagged Value | ⟨⟨ specification ⟩⟩ | the event occurrences which are monitored by the specification
⟨⟨ response ⟩⟩ | Stereotype | sequence diagram | the this sequence diagram is a response
specification | Tagged Value | ⟨⟨ response ⟩⟩ | the specification corresponding to this response
⟨⟨ return ⟩⟩ | Stereotype | message | the message is a return of previous message

Table 1: Extensions to sequence diagram for expressing runtime monitoring concern

describes the behavior that users want to execute when the interested execution event patterns happen.

In this paper, we are trying to provide practical modeling design notations and an algorithm which can translate the design notations to source code automatically. The approach will facilitate programmers to develop software with runtime monitoring capability. The generated code is based on aspect-oriented programming (AOP) language[2].

2 Extend UML Sequence Diagram to Support Runtime Monitoring

As it is natural to use sequence diagrams to represent the execution traces, we based our design notation for runtime monitoring concerns on sequence diagrams. According to OMG’s UML specification [3], ”a sequence diagram describes an Interaction by focusing on the sequence of Messages that are exchanged, along with their corresponding OccurrenceSpecifications on the life-lines” . In order to make it fit for expressing runtime monitoring concerns, we propose the meta-model as described in table 1. We introduce two stereotypes and two tagged values into sequence diagram. The Applies to column specifies where the extension should be used. ⟨⟨ specification ⟩⟩ is used to stereotype a sequence diagram to indicate that it is a runtime monitoring specification sequence diagram. The tagged value occurrence attached to ⟨⟨ specification ⟩⟩ specifies the message occurrences set the user want to monitor. The message occurrence is described by !(or ?) + messagename. The messagename specifies the message the occurrence attached to. ”!” means it is a send oc-
currence of a message, while "?" means a receive occurrence of a message. ⟨⟨response⟩⟩ is used to stereotype a sequence diagram to indicate that it is a response sequence diagram. A tagged value "specification" specifies the specification corresponded to this response. A message in a specification sequence diagram can be stereotyped as ⟨⟨return⟩⟩. This means it is a return message to a previous message. We stereotype the "return" message because the send occurrence of this message usually stands for the end of an execution. The notations used to represent the context information of message is borrowed from JPDD [4]. The argument and return value of a message is stored in the identifier and can be referenced later in response diagram.

3 Convert Sequence Diagrams into Regular Expressions

According to the OMG’s UML specification [3], a sequence diagram is composed of InteractionFragments. InteractionFragments can be categorized into basic InteractionFragment or CombinedFragment. Basic interaction fragments is interaction fragments without high-level operators, while the CombinedFragment is composed of an operator and its operands. The operands can be basic InteractionFragments or CombinedFragments.

Given a basic InteractionFragment, every concerned message occurrence is converted into a Symbol in Tracematches. Tracematches, proposed by Allan et al[1], enables the programmer trigger the execution of extra code by specifying a regular pattern of events in a computation-trace. If we treat every message occurrence as a node, and every partial order relation imposed by basic InteractionFragments as a directed edge, then all the concerned message occurrences in a basic InteractionFragment form a directed acyclic graph (DAG). Every path in the DAG is a sequence of message occurrences. It can be naturally converted into an regular expression (every message occurrence is converted into a Symbol, a sequence of message occurrences is converted into a regular expression by concatenating all the Symbols). The final expression, which is formed by connecting all the regular expressions converted from path through the alternation operator, reflects the runtime monitoring concern depicted by the basic InteractionFragment.

After converting a basic InteractionFragment into a regular expression, the operands of a CombinedFragment can be handled in similar way and then combined according to different operators. At last, a CombinedFragment is also translated into a regular expression in Tracematches.

In a word, the runtime monitoring concerns depicted by a sequence diagram can be finally converted into a regular expression in Tracematches. Every letter in this regular expression stands for a monitored message occurrence. We can weave this regular expression into the system’s implementation, and
thus empower the system the runtime monitoring capability.

4 Future Work

For future work, we are trying to enrich the design notations used to depict runtime monitoring concerns. Wildcard, subClass relationship and control flow are expected to add to increase the expressing power for selecting message occurrences. We are also trying to consider the runtime monitoring concern from the requirement phase and refine them until the corresponding aspect oriented code can be generated.

References


Models of Rate Restricted Communication for Concurrent Objects

Rudolf Schlatte, Einar Broch Johnsen,
Fatemeh Kazemeyni, and S. Lizeth Tapia Tarifa

Department of Informatics, University of Oslo, Norway
{rudi,einarj,fatemehk,sltarifa}@ifi.uio.no

Abstract

Many software systems today are designed for deployment on a range of architectures. However, in formal models it is typically assumed that the architecture is known and fixed; for example, that the software is sequential or concurrent, that the communication environment is synchronous or asynchronous but ordered, etc. In order to specify and analyze models which range over different deployment scenarios, it is interesting to lift aspects of low-level deployment variability to the abstraction level of the modeling language. In this paper, we propose a technique for introducing explicit resource constraints on concurrent objects in a timed extension of Creol, a formally defined high-level object-oriented modeling language. The technique is demonstrated by examples concerning rate restrictions on communication between objects. These restrictions are compositional and non-invasive: no change to the functional parts of the model is required, and restrictions can be selectively applied to parts of the model. In fact, the rate restrictions are captured by parameters in the model, which allows timed simulations to be performed with varying rate restrictions. We demonstrate the usefulness of explicit rate restrictions on communication in the model by a case study of wireless sensor networks. In this domain, rate restrictions may be understood as an abstraction over the collision patterns of broadcast data packets. Simulation results with different rate restrictions show how the timed throughput of data to the sink node in the network varies depending on the available rates.

1 Introduction

Software systems today are often designed for a range of different deployment scenarios, which may even evolve over time. Examples of such systems range from operating systems which may be deployed on sequential, multicore, or distributed architectures, to sensor-based monitoring systems which

* Partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (http://www.hats-project.eu).
may be deployed using various means of inter-sensor communication, such as wired, radio, or even acoustic communication channels. Depending on the deployment scenario, a distributed system may vary in the available processing power or memory provided to its nodes, as well as in the available bandwidth of the communication channels between nodes. In modern architectures such as cloud solutions, virtual infrastructures allow aspects of the deployment scenario to be configured to the needs of the software.

In order to model and reason about such highly configurable software systems, one cannot assume a uniform underlying architecture with given properties. Rather, it is desirable to capture this deployment variability at the abstraction level of the modeling language and subsequently to reason about the model’s behavior in the context of different deployment scenarios. In this paper, we develop a general model of resource restrictions between observable time intervals in the executions of the object-oriented modeling language Creol. The restrictions are expressed inside the language itself, and are used to impose rate restrictions on the communication environment of concurrent objects in Creol. The communication environment of a Creol model is parametric in its rate restriction, which allows the rate to be set without altering the functional part of the model.

Creol is a high-level executable modeling language \cite{14, 21} based on asynchronously communicating concurrent objects. The language abstracts from specific scheduling strategies inside the concurrent objects, and from particular properties of the communication environment. Creol has a formal semantics given in rewriting logic \cite{25}, which can be used directly as a language interpreter in the rewrite system Maude \cite{13}. In order to observe variations in behavior depending on rate restrictions between observable time intervals in the execution, we work with a timed extension to this semantics \cite{6}, which allows the timed behavior of Creol models to be simulated using Maude. This enables us to see the temporal effect of rate restrictions in the model, and compare the timed behavior of the model varying in rates.

We demonstrate by examples how rate restrictions in Creol models can be used to capture properties of radio-based message broadcast as well as of point-to-point communication channels. Channel-based communication concerns itself with communication between two objects, modeling e.g. a low-bandwidth connection between two hardware systems. Arrival-based restrictions describe a single component with its limits of accepting communication from anyone, capturing interference and resending in the context of radio communication. We further show how we can simulate the system behavior, ranging over rate restrictions, using Maude as an interpreter for Creol models.

The paper is structured as follows: Section 2 introduces Timed Creol, a timed extension of the Creol modeling language and illustrates the language by an example of client server communication. Section 3 proposes a model-
Figure 1. The syntax of core Timed Creol. Terms such as \( e \) and \( x \) denote lists over the corresponding syntactic categories, square brackets \([\ ]\) denote optional elements. Expressions \( e \) and guards \( g \) are side-effect free; Boolean expressions \( b \) include comparison by means of equality, greater- and less-than operators. Expressions on other datatypes (strings, numbers) are written in the usual way and not contained in this figure.

2 Concurrent Objects in Creol

Creol is an abstract behavioral modeling language for distributed active objects, based on asynchronous method calls and processor release points. In Creol, objects conceptually have dedicated processors and live in a distributed environment with asynchronous and unordered communication between objects. Communication is between named objects by means of asynchronous method calls; these may be seen as triggers of concurrent activity, resulting in new activities (so-called processes) in the called object. This section briefly introduces Creol (for further details see, e.g., [14, 21]).

Creol objects are dynamically created instances of classes, their declared attributes are initialized to some arbitrary type-correct values. An optional \textit{init} method may be used to redefine the attributes. Active behavior, triggered by an optional \textit{run} method, is interleaved with passive behavior, triggered by method calls. Thus, an object has a set of processes to be executed, which stem from method activations. Among these, at most one process is \textit{active} and the others are \textit{suspended} on a process queue. Process scheduling is by default non-deterministic, but controlled by \textit{processor release points} in a cooperative way. Creol is strongly typed: for well-typed programs, invoked methods are supported by the called object (when not \textit{null}), such that formal and actual parameters match. This paper assumes that programs are well-typed.

Figure 1 gives the syntax for a core subset of Timed Creol (omitting, e.g., class inheritance). A \textit{program} consists of interface and class definitions and a \textit{main} method to configure the initial state. \( IF \) defines an interface with
name \( I \) and method signatures \( S_g \). A class implements a list \( \vec{T} \) of interfaces, specifying types for its instances. \( CL \) defines a class with name \( C \), interfaces \( \vec{I} \), class parameters and state variables \( x \) (of type \( I \)), and methods \( M \). (The attributes of the class comprise its parameters and state variables.) A method signature \( S_g \) declares the return type \( I \) of a method with name \( m \) and formal parameters \( \vec{x} \) of types \( \vec{I} \). \( M \) defines a method with signature \( S_g \), a list of local variable declarations \( \vec{x} \) of types \( \vec{I} \), and a statement \( s \). Statements may access class attributes, locally defined variables, and the method’s formal parameters.

**Statements.** Assignment \( x := \text{rhs} \), sequential composition \( s_1; s_2 \), and \textbf{if}, \textbf{skip}, \textbf{while}, and \textbf{return} constructs are standard. The statement \textbf{suspend} unconditionally releases the processor by suspending the active process. In contrast, the guard \( g \) controls processor release in the statement \textbf{await} \( g \), and consists of Boolean conditions \( b \) and return tests \( x? \) (see below). If \( g \) evaluates to false, the current process is \textit{suspended} and the execution thread becomes idle. In that case, any enabled process from the pool of suspended processes may be scheduled. Explicit signaling is therefore redundant.

**Expressions \( \text{rhs} \)** include declared variables \( x \), object identifiers \( o \), Boolean expressions \( b \), and object creation \texttt{new} \( C(\vec{r}) \) and \texttt{null}. The specially reserved read-only variable \texttt{this} refers to the identifier of the object and \texttt{now} refers to the current clock value (explained below). Note that pure expressions are denoted by \( e \) and that remote access to attributes is not allowed. (The full language includes a functional expression language with standard operators for data types such as strings, integers, lists, sets, maps, and tuples. These are omitted in the core syntax, and explained when used in the examples.)

**Communication** in Creol is based on asynchronous method calls, denoted \( o!m(\vec{e}) \), and future variables. (Local calls are written \texttt{this!m(\vec{e}).}) After making an asynchronous call \( x := o!m(\vec{e}) \), the caller may proceed with its execution without blocking on the call. Here \( x \) is a future variable, \( o \) is an object expression, and \( \vec{e} \) are (data value or object) expressions. A future variable \( x \) refers to a return value which has yet to be computed. There are two operations on future variables, controlling external synchronization in Creol. First, the guard \textbf{await} \( x? \) suspends the active process unless a return to the call associated with \( x \) has arrived (allowing other processes in the object to be scheduled). Second, the return value is retrieved by the expression \( x.\texttt{get} \), which blocks all execution in the object until the return value is available. The statement sequence \( x := o!m(\vec{e}); \ v := x.\texttt{get} \) encodes a blocking call, abbreviated \( v := o.m(\vec{e}) \) (often referred to as a synchronous call), whereas the statement sequence \( x := o!m(\vec{e}); \ \textbf{await} \ x?; \ v := x.\texttt{get} \) encodes a non-blocking, preemptable call. Synchronous self-calls \texttt{this.m(\vec{e})} are handled specially in the semantics to avoid the trivial deadlock case.

**Time.** In this paper we work with an extended version of the language Creol which includes an implicit time model [6], comparable to a system clock
which updates every $n$ milliseconds. In this extension, a datatype \texttt{Time} is included in the language. A value of type \texttt{Time} can be obtained by evaluating the expression \texttt{now}, which returns the current time, i.e., the value of the global clock in the current state. Time values form a total order, with the usual less-than operator. Hence, two time values can be compared with each other, resulting in a Boolean value suitable for guards in \texttt{await} statements. While all other time values are constant, the result of comparing the expression \texttt{now} with another time value will change with the passage of time. From an object’s local perspective in this model of timed behavior, the passage of time is indirectly observable via \texttt{await} statements, and time is advanced when no other activity may occur. Note that a global clock is not mandatory in the time model, but is employed in this work to make bandwidth measurements across objects possible. (I.e., in our case the clock models “real” time, not a physical clock that is subject to drift.) The semantics of this model of time, combined with Creol’s blocking and non-blocking synchronization semantics, are powerful enough to express both activity and progress of time.

2.1 Example: Channel-Based Communication

This subsection illustrates how client-server communication may be modeled in Creol. This model will be extended to capture rate restricted communication in Sec. 3, without changing the functionality of the model. The interfaces of the client and server are given in Figure 2 (top). The \texttt{getData} method of the \texttt{Server} interface returns some data, the method \texttt{authenticate} of the \texttt{Client} interface represents an abstraction of an authentication handshake protocol (in the implementation, this would be based e.g. on a shared key, but in this model we are only concerned with the communication patterns).

The implementations of the Client and Server classes are given in Figure 2 (bottom). We see that objects of class \texttt{IClient} are active: upon creation they will request three packets from the server, assembling them in the client. Objects of class \texttt{IServer} are passive. They respond to \texttt{getData} calls by challenging the caller to authenticate itself, and then return a data packet. Running a simulation with one client and one server object will result in the client object containing the string "0,1,2". Figure 5 (left) shows an initial state of the model, which can be executed and tested independently of any communication constraints.

3 Modeling Resource Constrained Behavior

This section presents a technique to capture time-sensitive resource constrained behavior. The technique consists of imposing explicit resource constraints on concurrent objects in Timed Creol. These resource constraints are expressed
interface Client {
  Bool authenticate();
}

class IClient(Server server) implements Client {
  String content = "";
  
  Unit run() {
    Int i = 0;
    while (i < 3) {
      Fut<String> packet;
      packet = server!getData(this);
      await packet?;
      content = content + packet.get;
      i = i + 1;
    }
  }

  Bool authenticate() { return True; }
}

class IServer() implements Server {
  Int packet = 0;

  String getData(Client caller) {
    String result = "";
    Bool auth = caller.authenticate();
    if (auth) {
      result = intToString(packet)+"_";
      packet = packet + 1;
    }
    return result;
  }
}

interface MyInterface {
  Unit resourceConsumingMethod();
}

class MyClass(Int resourceLimit) implements MyInterface {
  Int resourceUsed = 0;

  Unit run() {
    while (True) {
      Time t = now;
      await now > t;
      resourceUsed = 0;
    }

    Unit resourceConsumingMethod() {
      await resourceUsed < resourceLimit;
      resourceUsed = resourceUsed + 1;
      ... // Implement behavior here
    }
}

interface MyInterface {
  Unit resourceConsumingMethod();
}

class MyClass(Int resourceLimit) implements MyInterface {
  Int resourceUsed = 0;

  Unit run() {
    while (True) {
      Time t = now;
      await now > t;
      resourceUsed = 0;
    }

    Unit resourceConsumingMethod() {
      await resourceUsed < resourceLimit;
      resourceUsed = resourceUsed + 1;
      ... // Implement behavior here
    }
}
consumption is modeled by a method which checks the availability of the constrained resource by means of an await-statement (Fig. 3, Line 19), and increments the number of consumed resources within the time interval appropriately when it proceeds (Fig. 3, Line 20). Following this technique, it is straightforward to extend a given model with resource constraints.

We now show how the example of Sec. 2.1 may be extended using the proposed technique in order to add a rate restricted communication channel behavior. The basic idea is to model the channel using the proposed modeling pattern and to leave the original client and server objects unchanged and unaware of the rate restriction imposed on their communication. Figure 4 gives the \texttt{channel} class modeled using the proposed pattern, where rate limits are imposed on all communication between the client and the server. The \texttt{Channel} interface extends both \texttt{Client} and \texttt{Server} interfaces, since channel objects will act in both roles.

Objects of class \texttt{IChannel} are initialized with a parameter \texttt{rate} denoting the “resource limit” per time interval that the channel is prepared to handle; the variable \texttt{nMessages} denotes the “consumed resources” within a time interval. Here, the \texttt{run} method resets the channel capacity when time advances (see Fig. 4, Line 23). The proxy methods implementing the interfaces, starting at Line 24, pass on the method call if the channel has enough capacity left in the current time interval. Otherwise, the call must wait for time to advance. Note that a proxy method is introduced for every method that is restricted by the channel (as explained above). It may sometimes be desirable to fine-tune the bandwidth consumption for the different methods; e.g., a higher bandwidth may be needed in order to transmit data than for completing an authentication handshake. This can be done easily by using the desired values in the proxy methods.

Figure 5 (right) shows how to initialize the constrained model. Note that the only difference with respect to the unconstrained model (left) is in the object initialization phase, where the client is connected to the channel instead of directly to the server.

4 Example: Wireless Sensor Networks

This section presents an extended case study to illustrate the technique introduced in Section 3. We illustrate the effects of arrival-based communication restriction on model behavior by a model of a wireless sensor network (WSN). A typical WSN consists of a number of sensor nodes, equipped with wireless transmitters, and a sink which collects data. The sensors record some sort of data and send it towards the sink. Since wireless sensors can be very small, or deployed in inaccessible terrain, not every sensor will be able to reach the sink directly. Hence, sensors have the additional duty of routing messages from
interface Channel
extends Client, Server {
    Unit setClient(Client client);
    Unit setServer(Server server);
}

class IChannel(Int rate)
implements Channel {
    Client client = null;
    Server server = null;
    Int nMessages = 0;

    Unit setClient(Client client) {
        this.client = client;
    }

    Unit setServer(Server server) {
        this.server = server;
    }

    Unit run() {
        while (True) {
            Time t = now;
            await now > t;
            nMessages = 0; }
    }

    Bool authenticate() {
        await client != null;
        await nMessages < rate;
        Fut<Bool> fauth = client!authenticate();
        nMessages = nMessages + 1;
        await fauth?;
        return fauth.get;
    }

    String getData(Client caller) {
        await server != null;
        await nMessages < rate;
        Fut<String> result = server!getData(this);
        nMessages = nMessages + 1;
        await result?;
        return result.get;
    }
}

Figure 4. Channel interface and implementation

// Unconstrained model
Server server;
Client client;
server = new IServer();
client = new IClient(server);

// Constrained model
Server server;
Client client;
Channel channel;
channel = new IChannel(1);
server = new IServer();
client = new IClient(channel);
channel.setClient(client);
channel.setServer(server);

Figure 5. Initialization of the constrained (left) and unconstrained (right) models

other nodes towards the sink. Wireless sensor networks use scheduling algorithms for channel access and bandwidth management, and routing algorithms for power- and bandwidth-efficient transmitting of messages.

For channel access, there are two different choices when operating on a single shared channel (e.g. multiple senders operating on the same frequency): the Time Division Multiple Access (TDMA) model and the Carrier Sense Multiple Access (CSMA) model with Collision Avoidance (CA). The CSMA/CA scheme is somewhat simpler for ad-hoc networks, but cannot provide bounded channel access delay and guaranteed fairness, which is unacceptable for time-critical applications. The TDMA scheme, on the other hand, has (after an initial setup / negotiation phase) zero overhead and zero collision during data transmission, and can increase the efficiency of the network [12]. Using TDMA, sensor nodes share the same frequency channel and transmit in succession, each using its own time slot.

Routing of messages towards the sink may be done in many different ways.
4.1 Structure of the Model

In our model, the objects representing nodes are not directly connected to each other. Instead, each node object has a reference to a `Network` object which models the behavior of the transmission medium between nodes, following the idea of TDMA. This structure makes it possible to model collisions, message loss, selective retransmission, and the node topology (which nodes can be reached from each node) without local knowledge of the topology inside the node objects, by modifying the behavior of the network object. This object also implements the resource restriction pattern described in Section 3 in order to model limited bandwidth. Figure 6 shows the interfaces of both the nodes and the network. The `broadcast` method of the network gets called by nodes when they wish to broadcast data; the `receive` method of a node is called by the network with data that is broadcast by another node.

When an object of class `SensorNode` (see Figure 7) is generated, its `run` method triggers the two behaviors `senseTask` and `routeTask`. Until the sensor has made the number of sensings it is supposed to do (set via parameter `maxSensings`), the `senseTask` generates a data packet to be sent off. The sensor data, which for simplicity is just a counter, is added to the `sendqueue` list together with the sensor’s id. The `sendqueue` list contains all messages waiting to be sent by the sensor. If there are elements in the `sendqueue` list, the sensor’s `routeTask` will broadcast the first message in the `sendqueue` list by a call to the `broadcast` method of the network object.

When a sensor receives a message from another sensor, a call to the `receive` method is made by the network. If the sensor has not seen this message before, it is added to the `received` set, and queued for re-sending.
This unconditional resending implements a simple flooding routing algorithm: when a sensor senses data, it broadcasts it to all other nodes within range; when a sensor receives a message that it has not seen before, it rebroadcasts this message to all its neighbors. More involved routing algorithms exist where sensors selectively rebroadcast messages depending on whether they are on the path to the sink, but this simple algorithm suffices to illustrate our approach.

The initialization block of the model (not shown) configures the sensor network by first creating all Sensor objects and one Network object. The network topology is defined by a call nw.setTopology(...); that describes which sensors should be able to send to which other sensors. The topology is not necessarily symmetrical; it is possible for a sensor to receive messages from another sensor but not be able to send to it in return. Finally, the nodes start their active behavior once they can see the initialized network (modeled via a call to setNetwork).

Timed behavior of the sensor nodes is modeled in the routeTask method. When executing that method, the current time is stored. The model assumes that sending a message takes time; after broadcasting, routeTask waits until a period of time has passed before continuing execution.

The SinkNode class given in Figure 8 implements the same interface as the sensor nodes, but has a different behavior. The major difference is that the sink has no run method, and hence no activity of its own. The receive method of the sink counts the number of unique messages received and records the time when the last message was received. Figure 9 shows the implementation of the network. Its behavior is implemented by the broadcast method.
class SinkNode implements Node {
    Time lastReceived = Time(0);
    Int noReceived = 0;
    Set<Packet> received = EmptySet;

    Unit setNetwork(Network network) {
        skip;
    }

    Unit receive(Packet packet) {
        if (~contains(received, packet)) {
            noReceived = noReceived + 1;
            received = insertElement(received, packet);
            lastReceived = now;
        }
    }
}

class INetwork(Int bandwidth) implements Network {
    Map<Node, List<Node>> topology = EmptyMap;
    Int usedbandwidth = 0;

    Unit run() {
        while (True) {
            Time t = now;
            await now > t;
            usedbandwidth = 0;
            while (usedbandwidth < bandwidth) {
                await usedbandwidth < bandwidth;
                usedbandwidth = usedbandwidth + 1;
            }
            List<Node> targets = lookup(topology, source);
            while (~isEmpty(targets)) {
                Node target = head(targets);
                target!receive(packet);
                targets = tail(targets);
            }
        }
    }

    Unit setTopology(Map<Node, List<Node>> topology) {
        this.topology = topology;
    }
}

Figure 8. The implementation of the sink node

Figure 9. The implementation of the network

The bandwidth is the number of time slots in a channel. In general, nodes are scalable in sending the messages, but they are limited by the bandwidth of the sinks or gateways. In addition, the number of sent messages may become limited by the transformation strategy regarding to the power efficiency. Since transmission is the biggest source of energy drain in WSNs, having control over bandwidth consumption is important for the efficient power consumption and longevity of the nodes [15].

4.2 Simulation and Analysis

Creol’s rewriting logic semantics allows the Maude rewrite engine to be used as a language interpreter in order to execute Creol models. We run a series of simulations of the WSN, obtained by varying the message arrival rate restriction and the topology of the Network object. The number of nodes was constant, with four sensor nodes and one sink node. The results of these simulations are given in Figure 10. We selected two “outlier” topologies: a star topology with every node directly connected to the sink, minimizing message travel distance, and a linear topology where the nodes form a chain with the sink at one end, maximizing average message travel time.

We further simulated a series of randomly-generated network topologies with constant four sensor nodes and six connections. Each network contained four sensor nodes, where each node was initialized to create and send three
data packets. As expected, the time for all messages to reach the sink goes up as available bandwidth (as modeled by the network arrival rate restriction) goes down. Also, the performance characteristics of the random networks can be observed to lie between those of the two chosen extreme topologies, potentially allowing reasoning about timing behaviors of arbitrary sensor networks of given connectedness based on the behavior of the boundary cases.

Simulation performance has been satisfactory for small to medium-sized models (a few dozen objects). Note that a Creol model contains less instances than the modelled system, since objects are not used as data containers and a Creol object models functionality of a system’s component or subsystem.

5 Related Work

The concurrency model provided by concurrent objects and Actor-based computation, in which software units with encapsulated processors communicate asynchronously, is increasingly attracting attention due to its intuitive and compositional nature (e.g., [1–3,9,14,18,32]). A distinguishing feature of Creol is the cooperative scheduling between asynchronously called methods [21], which allows active and reactive behavior to be combined within objects as well as compositional verification of partial correctness properties [2,14]. Creol’s model of cooperative scheduling has recently been generalized to concurrent object groups in Java [30] by restricting to a single activity within each group. In this paper, we work with Timed Creol [6], a timed extension of Creol in which the passage of time may be observed by means of the await-statements. This allows timing aspects of a model’s behavior to be simulated and related to non-functional properties of the model.

Techniques and methodologies for predictions or analysis of non-functional
properties are based on either measurement or modelling. Measurement-based approaches apply to existing implementations, using dedicated profiling or tracing tools such as JMeter or LoadRunner. Model-based approaches allow abstraction from specific system intricacies, but need parameters provided by domain experts [16]. A survey of model-based performance analysis techniques is given in [5]. Experimentation and simulations is the major source for obtaining an initial understanding of non-functional behavior in distributed networks. In the domain of wireless sensor networks, experiments are performed using simulators such as NS-2 and Omnet++. However, different simulators may give vastly different results, even for simple protocols [8], because the simulators make different assumptions about medium access control and physical layers [4]. In contrast, we have shown in this paper that abstract simulations give initial insights into the behavior of distributed algorithms without having to consider particular assumptions about the lower-level layers. Furthermore, formal models allow a more systematic, in-depth exploration of the execution space not only in terms of model-checking and theorem proving techniques, but also in terms of flexibility with respect to the simulation scenarios [27].

Formal approaches using process algebra, Petri Nets, game theory, and timed automata (e.g., [7, 10, 11, 17, 19]) have been applied in the embedded software and multimedia domains. A family of routing tree discovery algorithms for network diffusion protocols has been specified in TLA [26], where the specification is executable (by generating execution traces) and used to simulate individual runs as well as runs for a given set of parameters. The main performance measure of that paper is the cost of data dissemination during a data interval.

Here, we use Maude [13] to simulate Creol models by executing Creol’s rewriting logic [25] semantics. Maude provides a high-level framework in which flexible and domain-specific communication forms can be specified. Maude and its real-time extension have been used to model and analyze a wide range of protocols (e.g., [27, 29, 31]), but has to our knowledge not been used to capture parametric resources for, e.g., communication rates. In the domain of WSNs, Ölveczky and Thorvaldsen have shown that simulations in Real-Time Maude can provide fairly accurate performance results compared to NS-2 simulations using high-level formal models which provide greater flexibility than traditional simulators in defining appropriate simulation scenarios [27].

Work on modelling object-oriented systems with resource constraints is more scarce. Using the UML SPT profile for schedulability, performance and time, Petriu and Woodside [28] informally define the Core Scenario Model (CSM) to solve questions that arise in performance model building. CSM has a notion of resource context, which reflects the set of resources used by an operation. CSM aims to bridge the gap between UML specifications and techniques to generate performance models [5]. Closer to our work is Hooman
and Verhoef’s extension of VDM++ for simulation of embedded real-time systems [20], in which architectures are explicitly modelled using CPUs and buses. Objects are deployed on CPUs and communication between CPUs takes place over buses which impose a delay for message delivery, but no restriction on the amount of messages that can be delivered concurrently.

In previous work [22, 23], the authors have developed a framework based on concurrent object groups [30] using the technique of resource-constrained deployment components which are parametric in the amount of concurrent activity they allow within a time interval. The work reported here complements that work and will be integrated with it in the future.

6 Conclusions and Future Work

In this paper we present a modeling technique which formalizes patterns for time-sensitive resource constrained concurrent objects, applied to the modeling of communication rate restrictions. With this technique aspects of low-level deployment variability are lifted to the abstraction level of the modeling language, as illustrated here with the Creol language. The examples show how different forms of rate restricted communication can be modeled inside Timed Creol and how simulation techniques in Creol’s simulation environment in the Maude rewrite system can be used to observe the effects of parametric rate restrictions on non-functional properties of models, such as message arrival times, throughput, and abstractions of packet collisions in a network of cooperating nodes. The examples illustrate how both broadcast and channel-based communication models can be modeled with parametric rate restrictions using the proposed technique. The examples capture different network behaviors, e.g., collision patterns for broadcasted data packets, and allow the timed throughput of data in high-level models of radio-based message broadcast as well as of point-to-point communication channels to be observed.

In order to observe the effects of different deployment restrictions, a timed model for Creol is utilized; simulation and testing techniques can be used to gain insights into model behavior and how the effect of the parametric rate restrictions affect non-functional properties of the models.

The technique described in this paper allows communication rate restrictions for single objects to be fully expressed in terms of Timed Creol itself. However, the modeling pattern for resource constraints is not limited to expressing rate restrictions. In future work, we plan to apply the proposed technique to other kinds of resources, e.g., in order to capture resource availability. However, the approach seems best suited for restrictions on single objects. Modeling rate-restricted communications for groups of objects in a natural way requires an extension to the semantics of the Creol language similar to deployment components [22]. A thorough investigation of com-
munication rate restrictions within the framework of deployment components remains future work. Furthermore, it is interesting to investigate stronger analysis techniques than the simulations presented in this paper by combining symbolic analysis with simulations. For example, by using state abstractions a single simulation run can capture a whole class of concrete runs but evades full model checking.

References


Construct Aspectual Models from Requirement Documents for Automotive Software

Xiaojian Liu

Shandong Provincial Key Laboratory of Automotive Electronic Techniques
Institute of Automation, Shandong Academy of Science, Jinan, China, 250014
Emails: liuxijian@keylab.net; liuxiaojian@nwpu.edu.cn

Abstract

One of the main issues in the application of model-driven approach to the development of complex software-intensive systems, is how to obtain models in the very beginning of development. In this paper, we focus on the domain of automotive software, explore how to extract models from complicated requirement documents. We mainly investigate these three problems: (1) what information should be elicited from requirement documents; (2) how to organize these information as aspectual models; and (3) how to integrate the aspectual models together to form a complete specification of requirements. We separate requirements into several orthogonal aspects, and specify them as models by using suitable modeling notations. To integrate these models together, we use 4-variable model as a common framework, and map each elements of the models to the terms of 4-variable model. This integration approach provides a convenient way to understanding how each aspectual model contribute to a complete requirement specification. This work helps designers to organize essential requirement information as rigorous models, which set up the starting point for the model-driven development.

Keywords: Model-driven development; Requirements modeling; Automotive software; Architecture description language; Timed automata

1 Introduction

Model-driven development methodology is a promising approach to developing complex software-intensive systems. It requires that in a development process, each phase is based on the construction of models, models in later phases are constructed from those in earlier phases by model transformations, and code is an executable model generated from models in the design phase [2]. However, the application of model-driven approach to practical development is not an easy task — one of the basic problems is how to obtain models at the very beginning phase of the development, especially at the requirement analysis phase. At this stage, the software to be developed is demonstrated as a collection of documents which are usually written with natural language, the informal descriptions, unstructured organization
and very high-level descriptions of functionalities usually prevent the construction of models.

In this paper, we focus on the domain of automotive software, explore how to construct models from requirement documents. We mainly investigate these four problems: (1) what essential requirement information should be elicited from documents; (2) how to organize and represent these information as models; (3) how to integrate individual models together to form a complete requirement specification; and (4) how to support the construction and analysis of models with tools. Although we focus on the context of automotive domain, we believe that the basic ideas and solutions to these problems are also applicable to general complex software-intensive systems.

In the development of automotive systems, developers usually start development from three kinds of requirement documents: function description document, which describes the functionalities of the ECU (Electronic Control Unit) to be developed; application protocol document, which specifies the signals communicating between the target ECU and its environment (including sensors, actuators and other ECUs); and some constraints, including hardware constraints, nonfunctional requirements etc. Separation of requirements into these individual documents will facilitate the understanding and organization of the requirements of a complex system. However, from software developers’ points of view, this separation does not yet help them to efficiently capture the software and manage the development. For example, the designers usually want to find: what’s the boundary of the software interacting with its environment; what subfunctions are included in a complicated function; how a functionality reacts to the inputs by producing the outputs; and how an ECU exchanges data and control with other ECUs via communication media. These information are usually scattered over one or more places of a document, in some cases even over several different documents. Therefore, when the scale of the software increases, linking the relevant requirement information together and efficiently managing the development process becomes a difficult task.

To tackle this problem, we need to elicit the essential requirement information from documents, and reorganize them as a group of models from which developers can easily understand and design the software. Four kinds of information should be extracted from requirement documents: Structure information, Behavior information, Communication information and Platform and nonfunctional constraints. With these information in hand, designers are able to begin their design activities: structure information will help them to design data structures; behavior information will guide them to design control algorithms; communication information will support them to design messages and communication protocols; and platform and performance constraints will provide information for further software allocations and performance testings. In order to represent these information as models, we should firstly choose suitable modeling notations. To this end, we require the following basic criteria: (1) Each modeling notations only represents a kind of information;
(2) The modeling notations should be domain-specific, which can be easily understood and used by engineers; and (3) The models should be analyzable to support correctness assurance. Following these criteria, we choose notations EAST-ADL2, Timed automata and Communication architecture to model the structure, behavior and communication information respectively.

To capture a comprehensive requirement specification, we need to integrate the constructed models together. To this end, we use 4-variable requirement model \[9\], a relational theory for requirement specifications, as the common base, and map all the elements of the aspectual models to the terms of 4-variable model. This integration approach provides us a way to understanding how each models contributes to a complete requirement specification.

This work is only in its initial stage. Our final aim is to develop a domain-specific language and an environment in order to support model-driven development for automotive software. At present, we have developed a tool to support the ideas proposed above, and also applied it to the actual development to investigate its applicability. So far, the feedbacks from engineers indicate that the proposed ideas help them to clearly shape the requirements for automotive software, and the tool significantly help them to improve the process of development.

The remainder of the paper is organized as follows. Section 2 introduces what requirement information should be elicited through a running example; Section 3 introduces the modeling notations chosen to represent the requirement information; Section 4 describes how to integrate the individual models based on 4-variable requirement model; Section 5 illustrate the tool support for our ideas; Section 6 summarizes related works; and finally Section 7 concludes this paper.

## 2 Requirement Information

In this section, we use BCM (Vehicle Body Control System) as a running example to illustrate what requirement information should be elicited from documents for automotive software.

BCM is an ECU which controls vehicle body components, its main functionalities include locking/unlocking doors, lifting up/down windows, turning on/off a number of lights, and controlling wipers and washers etc. Here, as an example, we select one of the functionalities, locking/unlocking vehicle doors (LOCKCTR for short), to give a concise explanation.

In what follows, we firstly represent some scenarios of LOCKCTR, and then analyze the domain characteristics of LOCKCTR, which will help us to understand what essential requirement information is required to be elicited from documents.
<table>
<thead>
<tr>
<th>Input/Output</th>
<th>I/O</th>
<th>Category</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VehicleSpeed</td>
<td>I</td>
<td>state</td>
<td>Int [0, 254]</td>
<td>The speed of vehicle</td>
</tr>
<tr>
<td>IgnStatus</td>
<td>I</td>
<td>state</td>
<td>{ON, OFF}</td>
<td>The state of ignition</td>
</tr>
<tr>
<td>KeyStatus</td>
<td>I</td>
<td>state</td>
<td>{IN, OUT}</td>
<td>The state of ignition key</td>
</tr>
<tr>
<td>TouchStatus</td>
<td>I</td>
<td>state</td>
<td>{OPENED, CLOSED}</td>
<td>The touch state of doors</td>
</tr>
<tr>
<td>RKEUnlock</td>
<td>I</td>
<td>event</td>
<td></td>
<td>Unlocking the doors by pressing “UN-LOCK” button of remoter only once within 2s, and the pressing keeps at least 0.1s</td>
</tr>
<tr>
<td>DCLUnlock</td>
<td>I</td>
<td>event</td>
<td></td>
<td>Unlocking the doors by central lock</td>
</tr>
<tr>
<td>RKELock</td>
<td>I</td>
<td>event</td>
<td></td>
<td>Locking the doors by pressing “LOCK” button of remoter at least 0.1s</td>
</tr>
<tr>
<td>DCLLock</td>
<td>I</td>
<td>event</td>
<td></td>
<td>Locking the doors by central lock</td>
</tr>
<tr>
<td>VehicleCrash</td>
<td>I</td>
<td>event</td>
<td></td>
<td>Crash signal is raised</td>
</tr>
<tr>
<td>SpeedUp</td>
<td>I</td>
<td>event</td>
<td></td>
<td>Vehicle speed is over 30m/h</td>
</tr>
<tr>
<td>LockEnable</td>
<td>O</td>
<td>state</td>
<td>{0, 1}</td>
<td>Enabling the doors locked</td>
</tr>
<tr>
<td>UnlockEnable</td>
<td>O</td>
<td>state</td>
<td>{0, 1}</td>
<td>Enabling the doors unlocked</td>
</tr>
</tbody>
</table>

Req1: Drivers can lock and unlock vehicle doors by using a remoter;
Req2: Inside the vehicle, drivers can use central lock system to lock and unlock the doors;
Req3: When the vehicle speed is over 30m/h, the doors will be automatically locked;
Req4: When a crash happens, the doors will be automatically unlocked for rescue;
Req5: Driving motors of locking/unlocking doors must keep at least 200ms.

Structure information includes inputs and outputs, and the decomposition of a functionality. Generally, an input or output of a control system is a function from time domain to value domain. For an input, we may want to observe its state at some time instant, or want to observe the change of states, i.e., event, or both of them. For example, for the input of vehicle speed, we want to observe both its values and their changes in different circumstances. Thus, at the requirement level, to obtain abstract and concise descriptions for inputs and outputs, we need to categorize them into two classes: flows and events. A flow input (output) can be considered as a variable only whose values are to be observed; and an event input (output) represents an event which is raised when values of the input (output) are changed. In some cases, identifying and describing an event is difficult, in particular when temporal properties of inputs (outputs) are involved. As an example, Table 1 lists some inputs and outputs of LOCKCTR.
For a complicated functionality, its requirements are usually decomposed into several submodules. For example, the functionality of BCM may include sub-functionalities of locking/unlocking control, lights control and windows control etc. If a functionality is decomposed, we particularly concern about the delegation and assembly relationships between the inputs (outputs) of the top functionality and those of the sub-functionalities.

**Behavior information.** For the BCM application, its behavior demonstrates discrete, reactive and timing characteristics. Thus, the behavior information usually includes the relation between the inputs and outputs, the collaboration of several functionalities, and some timing constraints such as timeout, deadlines etc.

**Communication information.** These information mainly include communication topology and signals which communicate between an ECU and its environments. In automotive systems, bus technologies (CAN, LIN, Flexray and MOST) are widely used to connect an ECU with its environment, and signals transmit over buses to transfer data and control commands. There exists a close relation between the communication information and the behavior information, because the functionalities are usually triggered by the received signals, and their outputs are also transmitted via signals.

**Platform and nonfunctional constraints:** such as mechanical and electronic constraints, predefined hardware architecture, timing constraints, CPU workloads and memory consumptions, etc. These information are essential for hardware design and further software allocations and performance testings.

### 3 Representation of Requirement Information

To represent the requirement information, we choose different modeling notations to represent them: **EAST-ADL2** for structure information, **timed automata** for behavior structure and communication architecture for communication information. Among these formalisms, we will put emphasis on EAST-ADL2 and communication architecture. Timed automata formalism will be introduced briefly, the interesting readers please refer to work [4] for its detail definition.

#### 3.1 EAST-ADL2

As this paper only discusses the requirements modeling, we choose a subset of EAST-ADL2, i.e., the function modeling package of EAST-ADL2, as the modeling language to specify structures of functionalities. Figure 1 illustrates the metamodel of this subset.

With EAST-ADL2 language, a functionality is modeled as an element of **ADLFunctio**-**nType**, which contains a set of ports (elements of **ADLFlowPort** or **ADLClien**-**tServerPort**), a set of function prototypes (elements of **ADLFunctionProto**-**type**), and a set of connectors (elements of **ADLConnectorPrototype**). A function
Prototype must appear as a part of an element of ADLFunctionType, and itself is typed by an element of ADLFunctionType. Connectors can be classified as two categories: delegate and assembly, the former connects the ports of a function type and those of its contained function prototypes, and the latter connects the ports of two function prototypes. Every flow port is associated with a data type, which specifies the properties of the data exchanged via this port.

ADLInFlowPort and ADLServerPort can be used to model state inputs and event inputs respectively, and the internal decomposition structure of a functionality can be modeled with function prototypes and the connectors which link function type and all its prototypes together.

Example. Figure 2 illustrates the structure model of BCM. Here, two function prototypes lockCtr and lightCtr are depicted in the function type BCM. The inputs of BCM are delegated to either one of the prototypes or both of them.

The structure model actually specifies the set of common phenomena shared
with the functionality and its environment. To facilitate the following discussions, we divide the ports of a functionality as four sets: \textit{inflow\_ports}, \textit{outflow\_ports}, \textit{server\_ports} and \textit{client\_ports}. For the LOCKCTR functionality, its ports are of:

\textit{server\_ports} = \{RKE\_Lock, RKE\_Unlock, DCL\_Lock, DCL\_Unlock, Vehicle\_Crash, Speed\_Up\}
\textit{inflow\_ports} = \{Vehicle\_Speed, Ign\_Status, Key\_Status, Touch\_Status\}
\textit{outflow\_ports} = \{Lock\_Enable, Unlock\_Enable\}.

### 3.2 Timed Automata

Timed automata formalism is an extension of traditional untimed automata, by introducing time clocks and timed invariants to describe timing behaviors of systems. A timed automaton interacts with its environment through channels and global variables. To simulate and verify a target timed automaton, we must additionally model its environments in terms of timed automata, and compose them parallel to form a closed network of timed automata.

For the simulation and verification purposes, in this paper, we choose timed automata of UPPAAL version \cite{17} as the formalism. UPPAAL version extends original timed automata with a number of features. One of the significant features is that the expressions are allowed to use bounded integer variables (or arrays of these types) as well as clocks, this extension will enhance the expressiveness of timed automata, allowing us to model complicated guard conditions, assignments and invariants. In what follows, if no specific explanation, the term \textit{timed automata} is referred to the UPPAAL extension version.

Let $C$ be a set of clocks, $V$ a set of bounded integer variables. $\Phi(C, V)$ and $R(C, V)$ denote set of conditions and set of reset operations over $C$ and $V$ respectively. A timed automata is a tuple $(L, B, C, V, E, I, l_0)$, where $L$ is the set of locations; $l_0 \in L$ is the initial location; $B$ is the set of channels; $E \subseteq L \times B \times \Phi(C, V) \times R(C, V)^* \times L$ is the set of edges, where $B^\tau = \{a?|a \in B\} \cup \{a!|a \in B\} \cup \{\tau\}$ is the set of co-actions and internal action. An element $(l, \alpha, \varphi, r, l') \in E$ describes an edge from location $l$ to $l'$ with action $\alpha$, guard $\varphi$ and a list $r$ of reset operations; and $I : L \to \Phi(C, \emptyset)$ assigns timing invariants to locations.

### 3.3 Communication Architecture

Communication architecture mainly depicts hardware components and their connections. Hardware components include ECUs, I/O components and power suppliers; Connections usually comprise I/O connections, power connections and bus connections which connect an ECU with other ECUs or I/O components. The only way that an ECU interacts with its environment is through signals, which are transmitted via physical connections. The metamodel of communication architecture is illustrated in Figure 3.

Generally, signals are divided as two categories: \textit{BusSignals} and \textit{IoSignals}. A bus signal is a string of bits, which transmits on bus through frames; An I/O sig-
nal transmits through an I/O physical line, its values are determined by temporal patterns of high/low electric levels.

**Example.** We usually use *signal matrix*, a two-dimension table, to record all the relevant signals of an ECU. The following Table 2 gives some of the signals in LOCKCTR, where above three signals are bus signals which transmit via LIN bus, and the others are I/O signals.

Bus signals always transmit periodically, they need *priorities* to avoid conflicts in transmission. The columns *resolution* and *offset* are used to transform a signal value to its actual value. For instance, the actual value of the signal BCM_InteriorTemp can be calculated from its signal value via the formula:

$$actual\ value = signal\ value \times 0.03125 - 273$$

PIN number of an I/O signal stands for the ID of the hardware interface from which the signal is received or sent. The values of a signal are represented by high/low
electric levels on the physical line. For example, when the signal “DCLUnlock_Signal” is in “Low” level, the central unlock command is activated.

We divide signals of an ECU into input signals and output signals. For the signal matrix of Table 2, the input and output signals of LOCKCTR are of:

\[
\text{input\_signals} = \{ \text{VehicleSpeed\_Signal, InteriorTemp\_Signal, BatteryVoltage\_Signal, DCLUnlock\_Signal, DCLLock\_Signal, RKEUnlock\_Signal, RKELock\_Signal, KeyStatus\_Signal, IgnStatus\_Signal, TouchStatus\_Signal, VehicleCrash\_Signal} \}
\]

\[
\text{output\_signals} = \{ \text{LockEnable\_Signal, UnlockEnable\_Signal} \}
\]

### 4 Integrating Aspectual Models

In this section, we discuss how to combine three aspectual models (structure model, behavior model and communication architecture) to an integral system model. To this end, we must find a common framework on which the aspectual models can be explained consistently. In this paper, we choose the 4-variable requirements model, a relational theory for requirement specifications, as the common base, and map the elements of the aspectual models to the terms of this model.

#### 4.1 4-variable model

An overview of the 4-variable model is shown in Figure 4. According to this model, a sound requirement specification for an embedded software should be specified in terms of four groups of variables and five kinds of relations. The variables in this model are time-dependent:

- **Monitored variables** MON, the environmental quantities that influence a system’s (including both hardware part and software part) behavior;
- **Controlled variables** CON, the environmental quantities that a system controls;
- **Input variables** INPUT, the boundary of the software that the input devices(such as sensors or hardware/software drivers) write to the software; and
- **Output variables** OUTPUT, the boundary of the software that the output devices(such as actuators or hardware/software drivers) read from the software.
The five mathematical relations between these variables are of:

- **NAT** defines nature laws or physical constraints imposed on the variables of MON and CON. For example, a reasonable range on vehicle speed is: 0∼254KHP;
- **REQ**, a relation between MON and CON, which defines the response of the system to the values of the monitored variables by producing the values of the controlled variables;
- **IN**, a relation between MON and INPUT. It is an abstraction of the input device;
- **OUT**, a relation between CON and OUTPUT, which is an abstraction of the output device; and
- **SOF** is a relation between INPUT and OUTPUT, which specifies the software behavior.

### 4.2 Integrating Aspectual Models

The 4-variable model in fact establishes a criterion for determining whether a requirement document is complete and consistent. Therefore, in order to integrate the aspectual models to form a complete and consistent requirement model, we must compare the aspectual models with 4-variable model, that is, establish a corresponding relation between the elements of aspectual models and the terms of 4-variable models. If each elements of 4-variable model can find its counterpart in the aspectual models, then we say that the aspectual models can be integrated.

This corresponding relation is showed in Table 3. The table indicates that variables MON, CON, INPUT and OUTPUT, and relations NAT and REQ have their counterparts in the aspectual models: Ports depicted in EAST-ADL2 models in fact correspond to variables MON and CON; Signals described in hardware architectures constitute variables INPUT and OUTPUT; Nature constraints NAT imposed on MON and CON can be reflected by the data types (See Figure 1) associated with the ports; and Timed automaton model of a functionality in fact discovers the relation REQ between variables MON and CON.

Note that, refinement mappings IN and OUT establish relationships between the high-level observables (i.e., the ports) and the low-level ones (i.e., the signals), they do not find their counterparts in the aspectual models. This means that we have to add both relations to the aspectual models; Relation SOF can be derived from relations IN, OUT and REQ.

In what follows, we use the running example to illustrate IN, OUT and SOF. Before doing this, we firstly represent the formal treatments for ports and signals. Let symbols \texttt{flow\_port}, \texttt{event\_port} and \texttt{signal} be a flow port, a client-server port and a signal respectively, they can be treated as time-dependent state variables:
Table 3  
Mapping the aspectual models to 4-variable model

<table>
<thead>
<tr>
<th>Elements of 4-variable model</th>
<th>Elements of the aspectual models</th>
</tr>
</thead>
<tbody>
<tr>
<td>MON</td>
<td>inflow_ports ∪ server_ports</td>
</tr>
<tr>
<td>CON</td>
<td>outflow_ports ∪ client_ports</td>
</tr>
<tr>
<td>INPUT</td>
<td>input_signals</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>output_signals</td>
</tr>
<tr>
<td>NAT</td>
<td>Data types associated with ports</td>
</tr>
<tr>
<td>IN</td>
<td>Data refinement mapping:</td>
</tr>
<tr>
<td></td>
<td>server_ports ∪ inflow_ports → input_signals.</td>
</tr>
<tr>
<td>OUT</td>
<td>Data refinement mapping:</td>
</tr>
<tr>
<td></td>
<td>client_ports ∪ outflow_ports → output_signals.</td>
</tr>
<tr>
<td>REQ</td>
<td>The relation depicted by timed automaton model.</td>
</tr>
<tr>
<td>SOF</td>
<td>SOF can be derived from REQ, IN and OUT. Let TA be the timed automaton model for a functionality. SOF is a relation which can be depicted by a new timed automaton which is derived by transforming every edge ((t, \alpha, \varphi, r, \tau')) of TA to a new edge ((t, \alpha', \varphi', r', \tau')), such that (\alpha' = \text{IN}(\alpha)), (\varphi') and (r') are expressions about signals, and (\varphi \land \text{IN} \Rightarrow \varphi'), (r \land \text{OUT} \Rightarrow r').</td>
</tr>
</tbody>
</table>

flow_port:  \(\text{Time} \rightarrow \text{Range}\)

event_port:  \(\text{Time} \rightarrow \{0,1\}\)

signal:  \(\text{Time} \rightarrow \text{Range}\)

\text{Time} is a set of real number \(\mathbb{R} \geq 0\), stands for dense and continuous time base; \text{Range} is a finite set of discrete values. For a client-server port, when it is activated, its value equals to 1; otherwise, its value is 0.

\text{IN} and \text{OUT}: are illustrated in Table 4. The descriptions of the ports and the signals please refer to Table 1 and Table 2. It should be noted that, for a client-server port, “RKEUnlock” for example, IN maps it to a formula of duration calculus \([8]\), this means that: the event will be activated when the changes of the signal values satisfy the temporal pattern depicted by the formula. The reason why we select duration calculus as the logic for describing temporal patterns is that it enables the users to describe how time-dependent state variables should behave in certain time intervals. Due to space reason, the introduction of duration calculus is omitted here, interesting readers please refer to \([8]\) for detail.

\text{SOF}: is a relation between input signals and output signals, it can be derived from \text{REQ} (i.e., timed automaton model) through \text{IN} and \text{OUT}. For example, given an edge \((\text{UNLOCKED}, \text{SpeedUp?}, \text{IgnStatus=ON} \& \& \text{TouchStatus=CLOSED}, \text{LockEnable=1}, \text{LOCKING})\) of timed automaton for LOCKCTR (See Figure 6), it can be transformed to a new edge in which the actions, guards and resets are expressed with the signals under
<table>
<thead>
<tr>
<th>ports $\rightarrow$ signals</th>
<th>refinement relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VehicleSpeed $\rightarrow$ VehicleSpeed.Signal</td>
<td>VehicleSpeed=VehicleSpeed.Signal</td>
</tr>
<tr>
<td>IgnStatus $\rightarrow$ IgnStatus.Signal</td>
<td>IgnStatus $= \begin{cases} \text{ON if } \text{IgnStatus.Signal}=\text{Low} \ \text{OFF if } \text{IgnStatus.Signal}=\text{High} \end{cases}$</td>
</tr>
<tr>
<td>KeyStatus $\rightarrow$ KeyStatus.Signal</td>
<td>KeyStatus $= \begin{cases} \text{IN if } \text{KeyStatus.Signal}=\text{Low} \ \text{OUT if } \text{KeyStatus.Signal}=\text{High} \end{cases}$</td>
</tr>
<tr>
<td>TouchStatus $\rightarrow$ TouchStatus.Signal</td>
<td>TouchStatus $= \begin{cases} \text{OPENED if } \text{TouchStatus.Signal}=\text{Low} \ \text{CLOSED if } \text{TouchStatus.Signal}=\text{High} \end{cases}$</td>
</tr>
<tr>
<td>RKEUnlock $\rightarrow$ RKEUnlock.Signal</td>
<td>RKEUnlock=1 if ( F_2((\ell = 2 \land (F_1; F_2)); \text{true} ) holds, where ( F_1 = \text{def } [\text{RKEUnlock.Signal}=\text{High}] \land \ell \in [0.1, 1] ) ( F_2 = \text{def } [\text{RKEUnlock.Signal}=\text{Low}] \lor ) ( (F_1 \land \text{RKEUnlock.Signal}=\text{High}) \land \ell &lt; 0.1 ) ) )</td>
</tr>
<tr>
<td>Description: RKEUnlock is activated if the pattern ( \text{RKEUnlock.Signal} ) is in ( \text{‘High’} ) level at least 0.1s and at most 1s&quot; appears only once within 2s.</td>
<td></td>
</tr>
<tr>
<td>RKELock $\rightarrow$ RKElock.Signal</td>
<td>RKELock=1 if ( \text{[RKELock.Signal}=\text{Low} ; [\text{RKELock.Signal}=\text{High}] \land \ell &gt; 0.1) ; \text{true} ) holds.</td>
</tr>
<tr>
<td>Description: RKELock is activated if RKELock.Signal is in ( \text{‘High’} ) level at least 0.1s.</td>
<td></td>
</tr>
<tr>
<td>DCLUnlock $\rightarrow$ DCLUnlock.Signal</td>
<td>DCLUnlock=1 if ( \text{[DCLUnlock.Signal}=\text{Low} ; [\text{DCLUnlock.Signal}=\text{High}] ; \text{true} ) holds.</td>
</tr>
<tr>
<td>Description: DCLUnlock is activated if DCLUnlock.Signal changes from ( \text{‘Low’} ) level to ( \text{‘High’} ) level.</td>
<td></td>
</tr>
<tr>
<td>DCLLock $\rightarrow$ DCLlock.Signal</td>
<td>DCLLock=1 if ( \text{[DCLLock.Signal}=\text{Low} ; [\text{DCLLock.Signal}=\text{High}] ; \text{true} ) holds.</td>
</tr>
<tr>
<td>Description: DCLLock is activated if DCLLock.Signal changes from ( \text{‘Low’} ) level to ( \text{‘High’} ) level.</td>
<td></td>
</tr>
<tr>
<td>VehicleCrash $\rightarrow$ VehicleCrash.Signal</td>
<td>VehicleCrash=1 if ( \text{[VehicleCrash.Signal}=\text{Low}; \ell \geq 200) \lor \text{[VehicleCrash.Signal}=\text{High}; \ell = 40) ; \text{true} ) holds.</td>
</tr>
<tr>
<td>Description: VehicleCrash is activated if VehicleCrash.Signal is in ( \text{‘Low’} ) level 200ms followed by in ( \text{‘High’} ) level 40ms.</td>
<td></td>
</tr>
<tr>
<td>SpeedUp $\rightarrow$ VehicleSpeed.Signal</td>
<td>SpeedUp=1 if ( \text{[VehicleSpeed.Signal}&lt; 30] ; [\text{VehicleSpeed.Signal} \geq 30] ; \text{true} ) holds.</td>
</tr>
<tr>
<td>Description: SpeedUp is activated if the value of VehicleSpeed.Signal is up to 30KM.</td>
<td></td>
</tr>
<tr>
<td>OUT</td>
<td></td>
</tr>
<tr>
<td>LockEnable $\rightarrow$ LockEnable.Signal</td>
<td>LockEnable=LockEnable.Signal</td>
</tr>
<tr>
<td>UnlockEnable $\rightarrow$ UnlockEnable.Signal</td>
<td>UnlockEnable=UnlockEnable.Signal</td>
</tr>
</tbody>
</table>
the mappings IN and OUT:

\[(\text{UNLOCKED}, \@T(\text{VehicleSpeed Signal} \geq 30), \text{IgnStatus Signal}=\text{Low} \& \& \text{TouchStatus Signal}=\text{High}, \text{LockEnable Signal}=1, \text{LOCKING}), \text{where} \ \@T(\text{VehicleSpeed Signal} \geq 30) \text{ denotes a conditioned event which can be rewritten as} \]

\[\text{VehicleSpeed Signal}^\prime \geq 30 \land \text{VehicleSpeed Signal} < 30\]

Unprimed version of VehicleSpeed Signal denotes the signal in the old state, and primed version denotes the signal in new state.

5 Tool Support

This section will report our experience on the development of the tool which supports the construction of models and model analysis. The tool has these abilities: (1) It allows developers to build aspectual models; (2) It supports developers to establish the linkages between aspectual models; (3) It is able to integrate some external COTS for model simulation and verification; and (4) It has the abilities of code-generation, document-generation and schedulability analysis.

5.1 Structure of the Tool

The development of this tool is based on GME (Generic Modeling Environment) [10] platform, an UML meta-modeling environment for the development of domain specific language for embedded systems. The main strength of GME is that it allows us to develop a modeling environment by the construction of its meta-model. This significantly reduces the development cost and facilitates the management of changes. For example, to build an EAST-ADL2 modeling editor, we just need to build the meta-model of EAST-ADL2 (just like Figure [3]) with GME tool, define the constraints on this meta-model with OCL language, and choose favorable icons and line-styles to demonstrate the modeling elements and relations. After then, GME Meta-Interpreter can interpret the meta-model as a paradigm file (*.xmp file), with which developers can build their EAST-ADL2 models in GME environment.

The tool architecture is illustrated in Figure 5. All the modeling tools and assistant tools are built on the basis of GME environment. The uncolored blocks stand for the build-in mechanisms or interfaces provided by the GME environment, and the grey blocks for the tools developed by ourselves or integrated from external COTS. The main building blocks include:

- **OCL Checker, Add-on Interfaces and Model Access Interfaces** are build-in blocks provided by GME. In GME, constraints are defined on the elements of the meta-model with OCL language. Developers should designate the model element (of the meta-model) where the constraint will be attached, and the time when the constraint checking is triggered. When the constraint is violated in modeling time, an alarm will arise at the triggering time; **Add-on Interfaces** provide developers a means to developing their own programs and integrating them with GME.
For example, in our tool support, UPPAAL tool is integrated through these interfaces; *Model Access Interfaces* provide a set of interfaces allowing us to access the model elements through programming.

- *EAST-ADL2 Model Editor* and *Communication Architecture Model Editors* are developed with GME meta-modeling techniques just mentioned previously; *Signal Matrix Editor* is an individual program which allows us to edit, modify and record the signals of an ECU; UPPAAL tool is also loosely integrated in our toolset, it is only invoked from our tool, thus its integration does not violate the copyright of UPPAAL.
- *Document generator* generates the formatted document from models. It extracts necessary modeling information from models by using model access interfaces, and then exports them to a WORD document; *Code generator* takes timed automata as inputs, and generates segments of C-code to implement them; We have also implemented a schedulability analyzer which is able to compute the WCRT (Worst Case Response Time) [11] of signals for CAN bus. In the future, we also want to combine task schedule and bus schedule together, and integrate holistic schedulability analysis into this tool; *Simulator and Verifier* can simulate a functionality and verify its properties based on its timed automata model. In the next subsection, we will illustrate an example of simulation and verification with the help of UPPAAL tool.

At present, this tool has been used in the BCM project. This project includes total of 8 main functions, 21 subfunctions, more than 80 ports and 200 signals are involved. Obviously, it is difficult to manage such a large model if no tool support was available.

5.2 Simulation and Verification with UPPAAL

In this section, we introduce how to simulate and verify the functionalities with our tool. Since our tool borrows UPPAAL tool to do these tasks, we mainly introduce the procedure of transforming the models to an UPPAAL specification. The transformation procedure will combine the structure model and behavior model together, because the channels, global variables appeared in an UPPAAL specification are actually the ports depicted in the structure model. Therefore, we can
also view the transformed UPPAAL specification as the integration of structure and behavior models.

To simulate and verify the requirements with the UPPAAL tool, we should carry out the following steps to transform the aspectual models to an UPPAAL specification:

- Transform the structure model and the behavior requirements of a functionality to a timed automaton with UPPAAL tool;
- Model the environment of the functionality as timed automata as well;
- Combine the timed automata of both the functionality and its environment together to form an UPPAAL specification, and then perform the simulation and verification activities by using UPPAAL tool.

An UPPAAL specification includes four parts: **global declarations**, **templates** for timed automata, **process assignments** and a **system definition**. Global declarations declare clocks, data variables, channels, and constants, all of which are shared with every timed automaton in a system; templates are timed automata equipped with lists of formal parameters and with local declarations of clocks, data variables, channels, and constants; process assignments instantiate the templates by substituting actual parameters for the formal ones. An instantiated template is called a **process**. A system definition consists of a list of process running parallel.

We propose a number of transformation procedures which transform the structure model and behavior requirements to an UPPAAL specification:

| Step1: Declare *server_ports* and *client_ports* of the structure model as global channels; |
| Step2: Declare *inflow_ports* and *outflow_ports* of the structure model as global data variables or formal parameters of templates; |
| Step3: Model the behavior requirements and local clocks, local data variables as templates of timed automaton; |
| Step4: The environments of a functionality are declared as templates of timed automata as well; |
| Step5: The variables and channels which are expected to be instantiated when creating a process, are declared as formal parameters of the templates; |
| Step6: Compose the functionality and its environment parallel with a system definition. |

**Example** The UPPAAL specification for LOCKCTR is illustrated in Table 5 and Figure 6. The structure model and the behavior requirements for LOCKCTR are illustrated in Figure 2 and “Req1-Req5” in Section 3.

Note that, the environment of LOCKCTR may include drivers, lock actuators, and other ECUs such as EMS (Engine Management System). As an example, here we only select drivers and EMS as the environment of LOCKCTR. They initiate the actions which must be synchronized with the co-actions of LOCKCTR’s automaton. Because variables “VehicleSpeed, IgnStatus, KeyStatus and TouchStatus” come from the other environment outside drivers and EMS, they are specified as the formal parameters of LOCKCTR’s automaton and instantiated when creating
Table 5
UPPAAL specification for LOCKCTR

Global declaration
chan RKSUnlock, RKSLock, DCLUnlock, DCLLock;
chan VehicleCrash, SpeedUp;
const int[0,1] OPENED=1, CLOSED=0, IN=1, OUT=0, ON=1, OFF=0;
int[0,1] UnlockEnable=0, LockEnable=0;

Templates

LOCKCTR(int[0,254] VehicleSpeed, int[0,1] IgnStatus,

    int[0,1] KeyStatus, int[0,1] TouchStatus) // with formal parameters
Drivers();
EMS();

System declarations
const int[0,254] speed=30;
lockcontrol=LOCKCTR(speed,ON,IN,OPENED); // Instantiate formal parameters
drivers = Drivers();
ems = EMS();
system lockcontrol, drivers, ems;

Fig. 6. Timed automata for LOCKCTR and its environments

the process.

With the help of UPPAAL tool, we can simulate and verify the UPPAAL specification. The simulation explores all possible runs of the combined system (LOCKCTR ∥ Drivers ∥ EMS) in a step-by-step or random manner, through which analyzers can detect potential errors in the requirement models. UPPAAL tool also allows us to verify some properties of the specification. Generally, a property may be either a very general purpose one, such as deadlock freedom, which must hold for any valid
requirement model, or a domain-specific one which should be identified according to the knowledge of an application domain. In this paper, we do not discuss how to verify UPPAAL specifications, and how to detect and correct possible requirement errors through verification mechanism. It is a further topic deserved to be explored in the future.

6 Related Works

This work is inspired by the works [1], [2], [5] and [6]. Work [1] proposed the platform-based design methodology for embedded systems, and developed an environment, called Metropolis, to support this methodology. It separates an embedded system as several orthogonal aspects, such as computation and communication, function and architecture, and behavior and performance parameters. Each of the aspects are represented by UML notations. Work [2] proposed rCOS, a theory for component-based model-driven approach. It supports multi-dimension separation of concerns, and incremental development process. The theorem of separation of concerns in [5] allows us to refine the static functionality and reactive behavior separately to preserve the consistency. Work [6] studies how to use different UML models for describing different view and their integration for the overall system requirements modeling and analysis. In that work, use-case diagrams, class diagrams, sequence diagrams and state machines are combined together to describe a system from function, static structure and behavior viewpoints. The author’s paper [7] also discusses the relationship between the structure and behavior aspects in the context of object-oriented programming, where the structure aspect presents the attributes and associations in a class diagram, and the behavior aspect presents the class methods. It investigated how to derive the class methods from the structure aspect by giving a small set of structure refinement rules.

The main ideas of this paper derive from these works, however in this paper, we try to apply them to the domain of automotive software, and try to make them practical for actual development. Domain characteristics and application background make us to choose different modeling notations for automotive software. For example, we choose EAST-ADL2 to describe the structure of software, because on one side EAST-ADL2 is an industrial standard which has been widely recognized by engineers, and on the other side, it is restrictive and expressive enough to present the expected structure information.

SaveCCM [12] and AML (Automotive UML) [14] are works similar to this paper. Like the basic ideas represented in this paper, SaveCCM also separates software into structure and behavior aspects. Its behavior model is specified with task automata formalism [13], an extension of timed automata with tasks. Note that, SaveCCM is a model for software design purpose rather than requirements analysis, because task automata are mainly used to perform schedulability analysis of tasks, however, there is no tasks have been identified yet at the requirement
level. The main weakness of SaveCCM is that it has no notations for describing communication architecture of an automotive system. AML is a modeling language tailored to the development needs of automotive embedded systems. It provides the abstraction levels and necessary modeling elements for automotive applications, such as function, function variant and function networks. The main weakness of AML is that it does not provide the notation for behavior modeling.

For complicated software, a single formalism usually does not work for all system aspects, therefore many works dedicate on the integrated formal methods, which combine several well-built formal methods together to specify the whole system. CSP-OZ [15] is a method which integrates both CSP and Object-Z formalisms. CSP is used to model the behavior aspect and Object-Z for data structures and state-based specifications. To model the time-related behavior, CSP-OZ-DC [16] formalism is proposed, which additionally integrates duration calculus formulas to specify the timing properties of systems. In this paper, the timing behavior is modeled by both timed automata and duration calculus formulas. The former is used to model the top-level operational (reactive) behavior of software, and the latter is used to specify the temporal changes of signal values.

7 Conclusion and Future Works

In this paper, we explored how to extract models from requirement documents to support model-driven approach for automotive software. We also proposed an approach to linking these aspectual models together to form a complete requirement specification. In the future, we want to continue this work in two directions: firstly, we want to extend our framework to include additional modeling concepts and performance characteristics, such as function variants, QoS, safety and reliability, resource and scheduling properties. With these performance parameters, we can deal with perform analysis for the models; The second work is to extend the current tool to support model simulation, holistic schedulability analysis and performance analysis.

References


Abstract

In certified email (CEM) protocols, TTP transparency is an important security requirement which helps to avoid bad publicity as well as protecting individual users' privacy. Recently we have extended the CEM protocol of Cederquist et al. to satisfy TTP transparency. As a continuation, in this paper, we formally verify the security requirement in the extended protocol. The properties of fairness, effectiveness and timeliness are checked in the model checker Mocha, and TTP transparency is analysed in the toolsets µCRL and CADP. The results confirm that our proposed extension achieves our design goals.

Keywords: Verification, fairness, timeliness, TTP transparency, CEM protocols, Mocha, µCRL

1 Introduction

Certified email (CEM) protocols, as an extension of regular email services, require that both senders and receivers be responsible for their roles in the email services. That means, as a protocol successfully runs to the end, neither the sender can deny the dispatch of the email, nor can the receiver deny the receipt. Such requirements are usually implemented by a non-repudiable evidence of origin (EOO) that is to be acquired by the receiver, and a non-repudiable evidence of receipt (EOR) that is to be acquired by the sender. Both the EOO and the EOR may serve as evidences in case of a dispute.

As a special class of fair exchange protocols [20], a CEM protocol is supposed to guarantee fairness with respect to non-repudiable evidences. Informally, at the end of a fair protocol run, either both parties acquire all the evidences, or no party gets an evidence. A trusted third party (TTP) can be
introduced to take charge of the whole procedure and to provide undeniable records of submission (of the sender) and delivery (of the receiver). However in this way, a TTP may easily become a bottleneck, if she has to be involved in a large number of CEM services. A better solution, so called optimistic protocols [5], helps to release this burden from a TTP. In the optimistic protocols, a TTP is only required to be involved in case of unexpected events, such as a network failure or one party’s misbehaviour, to restore fairness. If both the signer and the receiver behave correctly and there is no presence of significant network delays, a CEM protocol terminates successfully without intervention of the TTP. TTP transparency states that if a TTP has been contacted to help in a protocol, the resulting evidences will be the same as those obtained in the case where the TTP has not participated. In other words, by simply looking at the evidences, it is impossible to detect whether the TTP has been involved or not. Transparent TTPs are important and useful in practice, for instance, to avoid bad publicity. Moreover, this property also ensures privacy of the participants for asking for help from TTPs. In the context of CEM protocols, the use of a transparent TTP was first proposed by Micali [17], followed by a number of works, e.g., [16,18,19,21,12], in which different cryptographic schemes are used to achieve TTP transparency.

Recently, we have developed a CEM protocol with a transparent TTP [15], based on the protocol of Cederquist et al. [9] that applies key chains to reduce TTP’s storage requirement. We achieve TTP transparency by adopting the verifiably encrypted signature scheme of [22]. We have shown that our extension is one of the most efficient CEM protocols satisfying TTP transparency, in addition to the other important properties such as strong fairness, effectiveness, and timeliness. The justifications to our claims are carried out on a rather informal level [15]. In this paper, we intend to put our analysis one step further, by incorporating formal verification techniques. The finite-state model checker Mocha [4] is used to verify the properties of fairness, timeliness and effectiveness, that are naturally interpreted in alternating-time temporal logic (ATL) formulas with game semantics [3]. The verification of properties expressed in ATL corresponds to the computation of winning strategies. Another toolset µCRL [7,6] is used for TTP transparency, which requires a comparison of observable traces in various situations. The µCRL toolset has the ability of generating state spaces that can be visualized and manipulated by the toolbox CADP [11] which acts as a back-end of µCRL.

Structure of the paper. We explain our proposed extension of the CEM protocol [9] and discuss its desired properties in Sect. 2. The two verification tools, Mocha and µCRL, are presented briefly in Sect. 3. In Sect. 4 we verify fairness, timeliness and effectiveness in Mocha with a focus on the modelling, and in Sect. 5 we verify TTP transparency in µCRL. Related work is discussed in Sect. 6. We conclude the paper in Sect. 7.
2 A Key Chain Based TTP Transparent CEM Protocol

Our protocol is developed on basis of the protocol [9], to support TTP transparency. Key chains are used to reduce TTP’s storage requirement. Once a key chain is initialized between Alice and Bob, Alice can use any key within it to encrypt messages. Our approach requires the usage of a verifiably encrypted signature scheme to encode a receiver’s commitment to receive the email.

For the sake of readability, we write Alice for the sender and Bob for the receiver. We assume the communication channels are resilient, in the sense that every message is guaranteed to reach its destination eventually. We write \( \{M\}_k \) to denote a message \( m \) encrypted with a symmetric key \( k \), and \( (M)_P \) to denote party \( P \)’s signature on message \( M \).\(^1\) We write \( (M)_{B|T} \) for Bob’s verifiably encrypted (partial) signature on \( M \), by using the public key of TTP to encrypt Bob’s signature on \( M \). Everyone can verify that \( (M)_{B|T} \) is authentic, but only TTP and Bob are able to extract the complete signature \( (M)_B \) out of \( (M)_{B|T} \).

2.1 The proposed protocol

The structure of our protocol consists of an exchange sub-protocol, an abort sub-protocol and a recover sub-protocol. The exchange sub-protocol is executed by the communicating parties to deliver an email as well as exchanging undeniable evidences. The other sub-protocols are launched by a party to contact a TTP to deal with awry situations. Each exchange that uses the protocol is called a protocol round, and one initialisation phase followed by a number of protocol rounds is called a protocol session. Each protocol session belongs to a unique pair of communication parties.

Key chain generation. In optimistic CEM protocols, communicating parties will request TTP for help if the exchange process is disrupted. To achieve (strong) fairness, the TTP often needs to store sufficient amount of information, to have the ability to decrypt, retrieve or send out information for the protocol to finally reach a fair state. In most existing CEM protocols, the initiator uses either TTP’s public key [18] or a separate key [21] to encrypt the email for each exchange. This first method normally requires asymmetric key operations, which are more expensive than symmetric key operations. The second method gives TTP burden of storing information of exchanges, such as involved parties, a hash value of email content and so on.

To reduce the TTP’s burden of storing too much information, the protocol [9] uses key chains. A chain of keys is a sequence of keys \( K'_0, \ldots, K'_n \), such that \( K'_i := H(G'(K'_0)) \) for each \( i \geq 0 \), where \( K'_0 \) is the seed, \( H : \kappa \to \kappa \) is a publicly known one-way collision-resistant hash function and \( G : \kappa \to \kappa \) is

\(^1\) In practice a signature is always applied on a hashed value.
a publicly known acyclic function (κ is a key domain). H and G are non-commutative, i.e., given an H(K_i) for which K_i is unknown, it is infeasible to compute H(G(K_i)).

Initialisation. To initialise a session, the initiator Alice (A) sends the key chain seed K_0 and the identity of the potential responder Bob (B), together with a nonce nc to the TTP (T). TTP will check whether there already exists an entry ⟨A, B, K_0, sid⟩ in her database indicating whether the key chain has been established. If yes, TTP just ignores this request. Otherwise, TTP will choose a new session identity sid, and send the message cert := (A, B, sid)_T to Alice, and then store ⟨A, B, K_0, sid⟩ in her database.

Exchange sub-protocol. The i^{th} protocol round in a protocol session sid is described below. The round number i is initially 0 and can arbitrarily grow, Alice incrementing i after each round. For convenience, we use EOR^M to denote (EOO_M)_B[T].

1^{ex}. A → B : A, B, T, i, sid, h(K'_i), \{M\}_{K'_i}, EOO_M, cert
2^{ex}. B → A : EOR^M
3^{ex}. A → B : K'_i
4^{ex}. B → A : EOR^M

At first, Alice sends out message 1^{ex} to Bob. After receiving EOO_M, Bob sends out his partial signature on EOO_M to show his commitment to receive the email. If Alice further sends Bob the key K'_i, Bob will deliver a full signature back to Alice as the final evidence of receipt.

Abort sub-protocol. Only Alice can abort, provided that the protocol has not yet been recovered. Typically, Alice aborts if she does not receive message 2^{ex}. To abort an exchange, Alice sends TTP the following message:

1^{a}. A → T : f_a, A, B, i, sid, h(\{M\}_{K'_i}), abrt

where f_a is a flag used to identify the abort request and abrt is Alice’s signature on the abort request. After receiving this request, TTP checks several things such as the correctness of signatures, identities, entries for the key chain, and status(i) to make decisions. If status(i) has not been initialised, TTP will set it as aborted (status(i) := a) and send back an abort token. If the current round has been recovered, TTP checks whether status(i) = h(\{M\}_{K'_i}). If yes, TTP will send back a recovery token. Otherwise, an error message of the form (error, (error, abrt)_T) is sent back.

Recovery sub-protocol. Alice is allowed to launch the recovery sub-protocol provided she has sent out message 3^{ex}, but has not received message 4^{ex}. 
Similarly, Bob can launch the recovery sub-protocol if he has sent out message $2^c$, but has not received message $3^c$. The first message of the recovery sub-protocol for Alice is

$$1^r_A. \ A \rightarrow T: \ text{fr}, A, B, h(K'_i), h(M), i, sid, EOR_M^i, EOO_M$$

where $f_r$ is a flag used to identify the recovery request. The first message of the recovery sub-protocol for Bob is

$$1^r_B. \ B \rightarrow T: \ text{fr}, A, B, h(K'_i), h(M), i, sid, EOR_M, EOO_M$$

On receipt of a message for recovery, TTP needs to check (1) the correctness of (verifiably encrypted) signatures on $EOO_M$ and $EOR_M$ ($EOR_M^i$), (2) the identity of TTP, and (3) whether there is an entry in its database matching $\langle A, B, *, sid \rangle$. If all the above checks succeed, TTP will retrieve $K_0$ and (4) check whether $h(H(G(K_0)))$ matches $h(K'_i)$. If yes, TTP will check $\text{status}(i)$ for round $i$.

- If $\text{status}(i)$ has not been initialised, TTP will set $\text{status}(i) := h(M)_{K'_i}$.
- Whenever necessary TTP converts $EOR_M^i$ into $EOR_M$. After that, TTP sends out the following messages.

$$2^r. \ T \rightarrow B: \ K'_i, (K'_i)_T$$

$$3^r. \ T \rightarrow A: \ EOR_M$$

- If $\text{status}(i) = h(M)_{K'_i}$, then TTP performs step $2^r$ and step $3^r$ (again).
- If $\text{status}(i) = a$, TTP sends out the abort token to the one that launched the protocol.

$$2^r. \ T \rightarrow A(B): \ \text{abrt}, (\text{abrt})_T$$

If any of the tests (1), (2), (3) and (4) fails, TTP ignores the recovery request and sends back an error message.

$$2^r. \ T \rightarrow A(B): \ \text{error}, (\text{error}, m^r)_T$$

where $m^r$ is the whole message received in step $1^r_A$ or $1^r_B$.

**Evidences and dispute resolution.** When a disputation occurs, both parties are required to provide evidences to an external judge. For each protocol round $i$, $EOO$ (evidence of origin) desired by Bob consists of

$$A, B, T, M, i, sid, K'_i, EOO_M,$$
EOR (evidence of receipt) desired by Alice consists of
\[ A, B, T, M, i, sid, K'_i, \text{cert}, \text{EOR}_M. \]

2.2 Security requirements

The following properties are claimed to be satisfied by the proposed protocol.

**Effectiveness.** If no error occurs then the protocol successfully runs till the end without any intervention from TTP.

**Timeliness.** Both Alice and Bob have the ability to eventually finish the protocol anywhere during the protocol execution. This is to prevent endless waiting of an honest party in case of unexpectancies.

**Fairness.** Honest Alice (Bob) will get her (his) evidences, provided that the other party gets the evidence from her (him).² The evidences can be used to convince an adjudicator that Bob has received the mail, in Alice’s case, or that Alice is the true sender of the message, in Bob’s case. A protocol satisfies fairness if every judgement on Bob’s (Alice’s) non-repudiation can be made solely and independently from Alice’s (Bob’s) evidences, i.e., it does not necessarily involve TTP, nor the participation of Bob (Alice).

**TTP transparency.** The evidence each participant obtains is of the same format regardless of whether TTP is involved in the protocol execution or not.

3 A Brief Description of Mocha and µCRL

To formally analyse whether a security protocol achieves its design goals, first we have to specify the protocol in a formal language, and then express specifications for the desired properties. The model checker Mocha [4] allows specification of models with concurrent game structures, and expression of properties using ATL (Alternating-time Temporal Logic) [3] formulas with game semantics, which is suitable for checking properties such as fairness, effectiveness and timeliness. As to the analysis of TTP transparency, our main idea is to compare traces of getting evidences from different situations.³ Therefore, a process algebraic language µCRL and its toolset [7,6] are used.

3.1 Mocha and ATL

Mocha [4] is an interactive verification environment for the modular and hierarchical verification of heterogeneous systems. Its model framework is in the form of reactive modules. The states of a reactive module are determined by variables and are changed in a sequence of rounds. Mocha can check ATL

² Note that only honest participants need to be protected.
³ This cannot be done with Mocha.
formulas, which express properties naturally as winning strategies with game semantics. This is the main reason we choose Mocha as our model checker. Mocha provides a guarded command language to model the protocols, which uses the concurrent game structures as its formal semantics. The syntax and semantics of this language can be found in [4].

The temporal logic ATL is defined with respect to a finite set \( \Pi \) of propositions and a finite set of players. An ATL formula is one of the following:

- \( p \) for propositions \( p \in \Pi \).
- \( \neg \phi \) or \( \phi_1 \lor \phi_2 \), where \( \phi, \phi_1, \) and \( \phi_2 \) are ATL formulas.
- \( \langle\langle A \rangle\rangle \# \phi, \langle\langle A \rangle\rangle 2 \phi, \) or \( \langle\langle A \rangle\rangle \phi_1 U \phi_2 \), where \( A \subseteq \Sigma \) is a set of players, and \( \phi, \phi_1 \) and \( \phi_2 \) are ATL formulas.

ATL formulas are interpreted over the states of a concurrent game structure that has the same propositions and players [3]. The labeling of the states of a concurrent game structure with propositions is used to evaluate the atomic formulas of ATL. The logical connectives \( \neg \) and \( \lor \) have the standard meaning. Intuitively, the operator \( \langle\langle \cdot \rangle\rangle \) acts as a selective quantification over those paths that the agents in \( A \) can enforce. The path quantifiers \( \circ \) (next), \( \Box \) (globally) and \( U \) (until) carry their usual meanings as in the logic CTL, and \( \Diamond \phi \) is defined as \( \text{true} U \phi \).

3.2 \( \mu \text{CRL and CADP} \)

\( \mu \text{CRL} \) is a language for specifying distributed systems and protocols in an algebraic style. A \( \mu \text{CRL} \) specification consists of two parts: one part specifies the data types, the other part specifies the processes.

The data part contains equational specifications; one can declare sorts and functions working upon these sorts, and describe the meaning of these functions by equations. Processes are represented by process terms. Process terms consist of action names and recursion variables with zero or more data parameters, combined with process-algebraic operators. Actions and recursion variables carry zero or more data parameters. Intuitively, an action can execute itself, after which it terminates successfully. There are two predefined actions: \( \delta \) represents deadlock, \( \tau \) the internal action. \( p.q \) denotes sequential composition, it first executes \( p \) and then \( q \). \( p+q \) denotes non-deterministic choice, meaning that it can behave as \( p \) or \( q \). Summation \( \sum_{d:D} p(d) \) provides the possibly infinite choice over a data type \( D \). The conditional construct \( p \triangleleft b \triangleright q \), with \( b \) a boolean data term, behaves as \( p \) if \( b \) and as \( q \) if not \( b \). Parallel composition \( p \parallel q \) interleaves the actions of \( p \) and \( q \); moreover, actions from \( p \) and \( q \) may synchronise into a communication action, if explicitly allowed by a predefined communication function. Two actions can only synchronise if their data parameters are the same, which means that communication can be used to capture data transfer from one process to another. If two actions are
able to synchronise, then in general we only want these actions to occur in communication with each other, and not on their own. This can be enforced by the encapsulation operator $\partial_H(p)$, which renames all occurrences in $p$ of actions from the set $H$ into $\delta$. Additionally, the hiding operator $\tau_I(p)$ turns all occurrences in $p$ of actions from the set $I$ into $\tau$.

The $\mu$CRL tool set [7] is a collection of tools for analysing and manipulating $\mu$CRL specifications. The $\mu$CRL tool set, together with the CADP tool set [11], which acts as a back-end for the $\mu$CRL tool set, features visualisation, simulation, LTS generation and minimisation, model checking, theorem proving and state-bit hashing capabilities.

4 Verification of the Protocol in Mocha

We give a sketch of our modeling approach and discuss the built models for the extended CEM protocol. Detailed models and analysis can be found in [14].

4.1 Modeling the protocol in Mocha

At first, each participant is modelled as a player (in a game), with the description of its behaviours using the guarded command language of Mocha. Models for honest participants can be easily specified strictly in accordance with the protocol. As to the dishonest models, we mainly consider the dishonest participant’s behaviours, since the security of CEM protocol can be hazarded by dishonest participants instead of outside intruders. Therefore, we build models for both honest and dishonest participants. For each participant, we write $P_i$ and $P_iH$ to represent the dishonest and honest models, respectively. Intuitively, dishonest model $P_i$ allows the player to cheat, while $P_iH$ just follows the protocol honestly. Dishonest behaviours include sending messages derivable from his knowledge at any time, stopping at any time; therefore, a dishonest model may not stop at a point where its role in the protocol is required to stop.

Communication is modelled using shared variables. Evidences (EOO and EOR), key and emails are modelled as boolean variables which are initialised as false and updated by its sender. We model the action of sending out an evidence, or other messages as a guarded command in which the sender resets the corresponding variables as true. In the model for honest participant $P_iH$, the guard consists of all the conditions to be satisfied strictly according to the protocol, and the command consists of all the corresponding actions to be executed. However, for the dishonest $P_i$, the guard just consists of necessary messages to generate the message to be sent.

List. 1 gives the Mocha code describing the behaviours of honest Alice. At first, Alice can do idle actions after she initiates a protocol round by sending
out EOO\textsubscript{M}. For honest Alice, she mainly performs two kinds of actions in the exchange sub-protocol, which includes sending evidence of origin and the key. They are described in step (1) and (2). Step (1) models the action of sending EOO\textsubscript{M}, in which we use boolean variables \( h_k \) and \( p_a\text{eoo} \) to represent the hashed value of \( K'_i \) and the message \((B,T,i,sid,h(K'_i),\{M\}_{K'_i})_A\) signed by Alice, respectively. Setting \( h_k \) and \( p_a\text{eoo} \) to true means Alice has initiated a communication with Bob by sending out her EOO\textsubscript{M}. Step (2) says that if Alice has received the correct verifiably encrypted message, namely \( p_b\text{halfeorm} \) has become true, she can set \( k \) as true, which represents the action of sending out key \( K'_i \). Except for the exchange sub-protocol, Alice is also able to initiate the abort protocol if she does not receive the verifiably encrypted signature \( p_b\text{halfeorm} \) from Bob. This abort request \( A\text{\_abort\_req} \) is described in step (4), in which the guard represents the requirements for asking for abort from TTP, and the commands represent the behaviour of contacting TTP for abort. Besides the abort sub-protocol, Alice can also initiate the recovery sub-protocol which is modelled in step (6). Recovery request is modelled as a boolean variable \( A\text{\_recovery\_req} \), and it will be set to be true if the guard is satisfied, in which the \( k \) and \( p_b\text{halfeorm} \) are true while \( p_b\text{eorm} \) is false. Note that once honest Alice initiates a recovery or abort sub-protocol with TTP, she will not continue the exchange sub-protocol. This mechanism is realized by modeling a boolean variable \( A\text{\_contacted\_T} \). Finally, Alice can stop if she receives final EOR\textsubscript{M} from Bob (step (3)) or recovery token from TTP (step (7)). Abort token (step (5)) can also make Alice stop the protocol round. In a similar way, we can model the honest behaviours of Bob.

Listing 1: Extracted honest model of Alice for the extended CEM protocol

```
-- idle actin while not stopped
[] "pa\_stop & pa\_eoo ->
-- (1) Alice sends EOO to Bob
[] "pa\_stop & "A\_contacted\_T & "pa\_eoo
-> pa\_eoo' := true; h_k' := true
-- (2) Alice sends out key while receiving half EOR\_M
[] "pa\_stop & "A\_contacted\_T & pb\_halferom & "k
-> k' := true
-- (3) Alice can stop when she receives Bob's EOR
[] "pb\_stop & "A\_contacted\_T & pb\_eorm & "pa\_rece_eorm
-> pa\_rece\_eorm' := true
-- (4) Alice can send out abort request to TTP
if she hasn't received half EOR\_M from Bob
[] "pa\_stop & "A\_contacted\_T & pa\_eoo & "pb\_halfeorm
-> A\_contacted\_T' := true; A\_abort\_req' := true
-- (5) Alice stops after receiving abort token from TTP
[] "pa\_stop & A\_contacted\_T & T\_abort\_send\_A
-> T\_abort\_token\_A' := true; pa\_stop' := true
-- (6) Alice can send recovery request
while she possesses pb\_halfeorm
[] "pa\_stop & "A\_contacted\_T & k & pb\_halfeorm & "pb\_eorm
-> A\_contacted\_T' := true; A\_recovery\_req' := true
-- (7) Alice stops after receiving recovery token from TTP
[] "pa\_stop & T\_recovery\_send\_A
-> pa\_rece\_eorm' := true; pa\_stop' := true
```
List. 2 describes the behaviours of dishonest Alice, her malicious behaviours are described as follows. At first Alice is allowed not only to idle, but also to stop and to quit the protocol at any time she wants. The behaviours of sending $EOO_M$ and the key are specified in step (1) and (2). Step (1) models that Alice can send out her evidence of origin by setting variable $pa_{eoo}$ to true at any time she wants, even if she has already contacted TTP and is supposed to stop. Together with $pa_{eoo}$, malicious Alice still has the choice of sending out correct hashed key $hk$ or incorrect hashed key $hke$. Similarly, step (2) specifies that Alice can send out her key at any time she wants. If the variable $k$ is true, it means that the correct key has been sent out. Otherwise, it represents that Alice has not sent out any key or the key that has been sent out is wrong. Moreover, step (3) and (4) models that Alice can contact TTP for abort or recovery as long as she has received enough messages, but she does not set the $A_{contactT}$ as true. The last two steps describe the situations when Alice has received $EOR_M$ or an abort token from TTP.

### Listing 2: Extracted dishonest model of Alice for the extended CEM protocol

```plaintext
-- idle action while not stopped
[] ! pa_stop & pa_eoo ->
-- Alice can stop at any time
[] ! pa_stop & pa_eoo -> pa_stop' := true
-- (1) Alice can send $EOO$ at any time
-- send correct hashed key
[] ! pa_stop & ! pa_eoo & ! hk & ! hke
   -> pa_eoo' := true; hk' := true
-- send incorrect hashed key
[] ! pa_stop & ! pa_eoo & ! hk & ! hke
   -> pa_eoo' := true; hke' := true
-- (2) Alice can send out key at any time
[] ! pa_stop & ! k -> k' := true
-- (3) Alice can send abort request
[] ! pa_stop & pa_eoo -> A_abort_req' := true
-- (4) Alice can send recovery request
[] ! pa_stop & pb_half_eorm -> A_recovery_req' := true
-- (5) Alice receives abort token
[] ! pa_stop & T_abort_send_A -> T_abort_token_A' := true
-- (6) Alice receives recovery token
[] ! pa_stop & T_recovery_send_A -> pa_rece_eorm' := true
```

In a similar way, we can model the dishonest behaviours of Bob.

List. 3 models the corresponding behaviours of TTP. TTP is a special player that has to be modelled in a particular way. It must be objective, and cannot act in collusion with protocol participants. We build the model for TTP that strictly follow the protocol. For each protocol round, we use a variable $T_{stateAB}$ to record the status of protocol. $T_{stateAB}$ has three possible values, which are $abrt$, $recov$ and $empty$ representing aborted, recovered and empty states, respectively. After receiving recovery or abort request, TTP will behave according to the values of $T_{stateAB}$. The first part describes how TTP deals with abort request from initiator Alice. TTP sends out abort token to both Alice and Bob if the status is $empty$ or $abrt$, and the $T_{stateAB}$
is also needed to be set as `abrt` if the original status is `empty`. However, if \( T\_state\_AB \) is `recov`, which means the corresponding round has already been recovered, then the corresponding \( EOR_M \) and key must be sent to Alice and Bob respectively. Part two and three models the behaviours of dealing with recovery requests from Alice and Bob. If the TTP receives a recovery request and its status is `empty` or `recov`, then the required evidences or key must be sent to Alice and Bob respectively. Otherwise, abort token will be sent out.

Listing 3: Extracted model of TTP for the extended CEM protocol

```
-- (1) If TTP receives abort request from Alice
[[]] A\_abort\_req \& (T\_state\_AB = abrt) \& ~T\_response\_A
\rightarrow \ T\_abort\_send\_A := true \& T\_abort\_send\_B := true;
T\_response\_A := true

[[]] A\_abort\_req \& (T\_state\_AB = empty) \& ~T\_response\_A
\rightarrow \ T\_abort\_send\_A := true \& T\_abort\_send\_B := true;
T\_response\_A := true; T\_state\_AB := abrt

[[]] A\_abort\_req \& (T\_state\_AB = recov) \& ~T\_response\_A
\rightarrow \ T\_recovery\_send\_A := true \& T\_recovery\_send\_B := true;
T\_response\_A := true

-- (2) If TTP receives recovery request from Alice
[[]] A\_recovery\_req \& (T\_state = empty) \& ~T\_response\_A
\rightarrow \ T\_state\_AB := recov \& T\_recovery\_send\_A := true;
T\_recovery\_send\_B := true; T\_response\_A := true

[[]] A\_recovery\_req \& (T\_state = recov) \& ~T\_response\_A
\rightarrow \ T\_recovery\_send\_A := true \& T\_recovery\_send\_B := true;
T\_response\_A := true

[[]] A\_recovery\_req \& (T\_state = abrt) \& ~T\_response\_A
\rightarrow \ T\_abort\_send\_A := true \& T\_abort\_send\_B := true;
T\_response\_A := true

-- (3) If TTP receives recovery request from Bob
[[]] B\_recovery\_req \& (T\_state = empty) \& ~T\_response\_B
\rightarrow \ T\_state\_AB := recov \& T\_recovery\_send\_A := true;
T\_recovery\_send\_B := true; T\_response\_B := true

[[]] B\_recovery\_req \& (T\_state = recov) \& ~T\_response\_B
\rightarrow \ T\_recovery\_send\_A := true \& T\_recovery\_send\_B := true;
T\_response\_B := true

[[]] B\_recovery\_req \& (T\_state = abrt) \& ~T\_response\_B
\rightarrow \ T\_abort\_send\_A := true \& T\_abort\_send\_B := true;
T\_response\_B := true
```

4.2 Expressing properties of the protocol in ATL

Given a CEM protocol with just two participants Alice and Bob, the following expressions are suitable for honest participant even if the other is dishonest. Actually, we only care about fairness and timeliness for honest participant. As to effectiveness, it requires that both participants must behave honestly.

**Effectiveness.** If honest participants are willing to exchange emails for receipts, then the protocol will terminate in a state that Alice has obtained \( EOR \) and Bob has received \( EOO \) and \( M \) without the involvement of TTP.

\[
effectiveness \equiv (⟨⟨P_aH, P_bH⟩⟩ ◦(EOO \land M \land EOR))
\]

where \( P_aH \) and \( P_bH \) represent honest participants Alice and Bob, and \( EOR \)
represents the evidence of receipt from receiver Bob. In addition, the EOO and M represents the evidence of origin and the email content from Alice.

**Timeliness.** At any time, an honest participant has a strategy to stop the protocol and thus to prevent endless waiting. Timeliness for Alice and Bob is formulated as:

\[
timeliness_{P_a} \equiv \forall \diamond (\langle\langle P_aH \rangle\rangle \Diamond P_{a-stop}) \quad timeliness_{P_b} \equiv \forall \diamond (\langle\langle P_bH \rangle\rangle \Diamond P_{b-stop})
\]

where \(P_aH\) and \(P_bH\) represent the honest Alice and Bob, and \(P_{a-stop}\) (\(P_{b-stop}\)) represents that Alice (Bob) has already terminated the protocol.

**Fairness.** A protocol is fair for honest Alice \(P_a\) if the following is satisfied: whenever Bob obtains \(P_a\)'s non-repudiation evidence of origin \((EOO)\) and email content \(M\), \(P_aH\) has a strategy to obtain Bob’s non-repudiable evidence of receipt \((EOR)\). In ATL, fairness for honest Alice can be formulated as:

\[
fairness_{P_aH} \equiv \forall \square ((EOO \land M) \Rightarrow \langle\langle P_aH \rangle\rangle \Diamond (EOR))
\]

Similarly, fairness for Bob is formulated as below. If Alice obtains \(P_b\)'s EOR, honest Bob \(P_bH\) has a strategy to get Alice’s EOR and email content \(M\).

\[
fairness_{P_bH} \equiv \forall \square ((EOR) \Rightarrow \langle\langle P_bH \rangle\rangle \Diamond (EOO \land M))
\]

### 4.3 Analysis

We have built three Mocha models, \(P_aH \parallel P_bH \parallel TTP\), \(P_a \parallel P_bH \parallel TTP\), and \(P_aH \parallel P_b \parallel TTP\), combining the aforementioned formulas, to verify fairness, timeliness and effectiveness of our CEM protocol. These properties were successfully checked in Mocha.

### 5 Verification of the Protocol in \(\mu\)CRL

In this section, we only give a sketch on how we model the protocol in \(\mu\)CRL, and focus on how to check TTP transparency of the protocol in \(\mu\)CRL. The detailed models and analysis can be found in [14].

#### 5.1 Modeling the protocol in \(\mu\)CRL

As stated in Sect. 5, each \(\mu\)CRL specification consists of two parts, which are abstract data type definitions and behavioural specifications for participants. Since the execution of protocol mainly depends on the exchange of messages, the contents of the data are not treated in details, instead the data type used and corresponding operations on it are captured. Therefore, we can simplify the complex cryptographic primitives, such as encryption, decryption and verifiably encryption of messages.
TTP transparency states that the final evidences do not reveal whether TTP has intervened in the protocol or not. The main idea of checking TTP transparency is to compare traces obtained from three different models after hiding all unnecessary actions, such as messages between TTP and the users, as well as minimising the generated state space modulo weak trace equivalence. The three models are combinations of honest Alice and honest Bob, honest Alice and malicious Bob and TTP, and malicious Alice and honest Bob and TTP.

Participants are hooked up by communication channels. According to our assumption, the communications channels are resilient, in the sense that every message is guaranteed to reach its destination eventually. Therefore, by using the encapsulation and communication operators in $\mu$CRL, we are able to enforce the actions of participants Alice, Bob and TTP to synchronise. Each participant is defined as a process. The communications between them are composed by actions of sending and receiving messages. The honest and dishonest behaviours of the participants resemble those in the Mocha models.

For instance, the behaviours of the initiator Alice are modelled in a process with a parameter key, which initiates the CEM protocol by sending evidence of origin EOO to receiver Bob. The action $\text{init\_Alice}(x,y,i,A,B)$ shows that Alice initiates a protocol round $i$ for delivering an email $y$ to Bob using a key $x$. Then after receiving the verifiably encrypted message from Bob, honest Alice will send out her key. If Bob’s final reply EOR is correct, Alice will be sure that she has completed one email delivery and successfully obtained the evidence of receipt. Action $\text{evidence\_Alice}(x,y,i,\text{halfeorm},\text{eorm},A,B)$ reports that she has already obtained the evidence for protocol round $i$ which sends email $y$ with key $x$. The sketch of Alice’s behaviour is described as follows.

$$\begin{align*}
\text{Alice}(x : \text{Key}) &= \sum_{y : \text{Item}} \sum_{i : \text{Number}} \text{init\_Send}(A,\text{eoo},B).\text{init\_A}(A,y,x,i,B) \\
&\quad \text{recv}(B,\text{halfeorm},A).\text{send}(A,k,B), \\
&\quad \text{recv}(B,\text{eorm},A).\text{evidence\_A}(A,y,x,i,\text{eorm},B)
\end{align*}$$

where $\text{eoo}$ represents the first message $1^{ex}$ for protocol round $i$. The $\text{halfeorm}$ and $\text{eorm}$ represents Bob’s verifiably encrypted signature and final signature. We need to extend the above process when taking TTP into account to cover when Alice can contact TTP and receive replies from TTP. Similarly, honest Bob, dishonest Alice, dishonest Bob, and TTP can be modelled in $\mu$CRL as well, by specifying their behaviours as discussed before.

5.2 Analysis

Our way to check TTP transparency is by comparing traces of getting evidences between system of only honest participants and systems containing
dishonest participants. After hiding some actions and reducing the model, we obtain a trace from the honest system that is depicted in Fig. 1(a), which shows the situation of getting evidences without TTP. Fig. 1(b) describes traces obtained from the model containing honest Alice, dishonest Bob, and TTP. We can find that Fig. 1(b) has one more trace. Evidences for both traces are of the same form, but the sequence of getting them are different. However, this difference does not affect the correctness of TTP transparency. When checking the evidences possessed Bob and Alice, the only thing that matters is the content of the evidences, and the number of transitions (which might reflect the execution time) is irrelevant due to the asynchrony of the protocol model. Fig. 1(c) depicts the traces obtained from the model containing dishonest Alice, honest Bob and TTP. We can find that this figure has one more trace than Fig. 1(b). This extra trace describes Alice’s malicious behaviours of using the key ($k_2$) that does not match the protocol round ($i_1$). However, the occurrence of this trace manifests that both Alice and Bob get their expected evidences without the intervene of TTP. As if Alice or Bob tries to contact TTP for recovery, they will just obtain error message instead of evidences. Therefore, this trace does not reveal the involvement of TTP. By the above analysis, we can draw a conclusion that our extended CEM protocol satisfies TTP transparency.

6 Related Work

It has been acknowledged that formal verification is important for security protocols, because of the seriousness of security flaws. In this paper, we use the technology of model checking to check automatically whether a given model of CEM protocols satisfy some given specifications. To our knowledge, the lit-
erature of formal verifications of CEM protocols includes the works of Kremer et al. [13], Cederquist et al. [8] and Abadi and Blanchet [1].

Kremer et al. [13] propose an approach for modeling and analysis of CEM protocols using model checker Mocha. The advantage of using Mocha is that it allows to model CEM protocols with concurrent game structures, and specify properties in ATL, a temporal logic with game semantics. Therefore, Mocha is well suited for checking properties such as fairness, timeliness and effectiveness that can be naturally interpreted with game semantics. For similar reasons, Mocha has been used for other fair exchange protocols [10,23]. Besides Mocha, the µCRL toolset, together with CADP which acts as an back-end, has also used to analyse CEM protocols automatically. Cederquist et al. [8] design an optimistic CEM protocol and check both safety and liveness properties using µCRL toolset. The desired properties are specified using µ-calculus. There exists another way to check CEM protocols, which is proposed by Abadi and Blanchet [1]. Their protocol is specified using the applied pi calculus. Taking the protocol specification as input language, the verifier ProVerif automatically checks the property secrecy. As to fairness, it is not checked fully automatically, but with some manual proofs.

7 Conclusion

We have formally verified the protocol [15], an extension of the key chain based CEM protocol [9] by Cederquist et al. to cover an additional requirement TTP transparency. The verification was taken in two steps. First, we checked fairness, effectiveness and timeliness properties, using the model checker Mocha. Then we have modelled the protocol in a process algebraic language µCRL and used its toolsets together with CADP to check TTP transparency. Our analysis showed that the protocol achieves the design goals.

The way to formalize TTP transparency in this paper abstracts a lot from the underlying cryptographic techniques and the ability of the adversary. In the future, we would like to investigate a more appropriate approach, for example, it is interesting to see whether we can interpret TTP transparency using static equivalence in the applied pi calculus [2]. Another direction is to extend the protocol furthermore, to cover other design goals such as stateless TTP and accountability.

References


Abstract
This paper addresses the problem of verifying parameterized properties of programs at runtime. More specifically, we focus on simple, schematic properties, like when file $x$ is opened, file $x$ should be closed before the program exits, in which $x$ is some free parameter, and which should be checked for whatever file is opened. To this end, the paper studies the notion of parameterized traces which can be state sequence, not just event sequence and introduces LTL schemas for expressing properties of parameterized traces. It is explained how to separate a parameterized trace into several non-parametric sub-traces that each correspond to one dedicated parameter instance, and how to transfer the problem of runtime verification for an LTL schema to the easier verification problem of a corresponding LTL formula on a non-parametric state trace. The whole work is carried out using the idea of an impartial and anticipatory three-valued semantics, first introduced for non-parameterized LTL.

Keywords: Runtime Verification, Parameterized Traces, Anticipatory Semantics, LTL Schemas.

1 Introduction

Runtime Verification [11] deals with those verification techniques that allow checking whether an execution of a system under scrutiny satisfies or violates...
a given correctness property while executing the program. In runtime verification, a correctness property $\varphi$ is typically automatically translated into a monitor. Such a monitor is then used to check the current execution of a system or a (finite set of) recorded execution(s) with respect to the property $\varphi$.

Correctness properties in runtime verification specify all admissible individual executions of a system and may be expressed using a variety of different formalisms. Temporal logic-based formalisms, which are well-known from model checking, are also very popular in runtime verification, especially variants of linear temporal logic, such as LTL [13]. LTL is a well-accepted linear-time temporal logic used for specifying properties of infinite traces. Note however that LTL can only express (a subset) of regular properties, but does not allow the specification of, for example, proper nestings of say call and return.

In [5], the scenario to check LTL properties based on finite prefixes of infinite traces was considered, and we follow this view also in this paper. To implement the idea that, for a given LTL formula, its meaning for a prefix of an infinite trace should correspond to its meaning considered as an LTL formula for the full infinite trace, [5] introduced an anticipatory semantics of LTL. Given the monitored property $\varphi \in \text{LTL}$ and a finite trace which has been observed so far, the result of monitoring is one of true, false, or inconclusive (denoted by ?), depending on the possible continuations of the trace. Moreover, based on this three-valued semantics, it is described how to generate a monitor as a (deterministic) finite state machine (FSM, Moore machine) with three output symbols, yielding for a finite traces its three-valued semantics. Finally, the approach is shown to be practically feasible using Dwyer’s specification patterns [9].

LTL is defined using a finite set of atomic propositions, which seems to be a limiting factor in runtime verification. Consider the following typical scenario: A program may allocate some amount of memory and binds the starting address of this memory block to some variable $x$. Using the value of $x$, the corresponding memory block can be deallocated. A typical property to check is that, globally, every allocation is eventually followed by a corresponding deallocation (free), before the program exits, which may be expressed in LTL by $\varphi = \text{G}(\text{malloc} \rightarrow (\neg \text{exit} \cup \text{free}))$. However, the given property should be checked for every allocation, say, each allocation that results in a new memory address. In other words, the intended meaning is to check $\hat{\varphi} = \text{G}(\text{malloc}(x) \rightarrow (\neg \text{exit} \cup \text{free}(x)))$, for any possible value for $x$, or, more practically, for any value for $x$ that was returned by the call to function malloc. In a sense, we consider properties malloc and free that are parameterized by value $x$, yielding malloc$(x)$ and free$(x)$. The latter, however, could be understood to denote the (infinite) set of atomic propositions $\{\text{malloc}_x, \text{free}_x\}$ for each value of $x$. Likewise, we can understand $\hat{\varphi}$ to denote the set of for-
mulas \( \{ G(\text{malloc}_x \rightarrow (\neg \text{exit} \cup \text{free}_x)) \} \). Therefore, we call \( \hat{\varphi} \) an \textit{LTL schema} rather than a plain formula.

In the same manner, program traces are typically not sequences of plain (sets of) propositions but may be seen as parameterized traces. They abound in program executions, because they naturally appear whenever abstract parameters (e.g., variable names) are bound to concrete data (e.g., dynamic objects) at runtime. This is pervasive for each programming language. For example, if one is interested in analyzing the calls to a function in C, then the execution trace of interest may contain events “func1(a),” “func2(b,c)” and so on. Here, “func1(x)” and “func2(y,z)” are different function prototypes, and “x,y,z” are the parameters which belong to particular types and are instantiated for particular arguments from corresponding domains at runtime. In other words, in parametric traces, each event carries a partial instance of property parameters. Likewise, in most situations, we are not concerned with the correctness of a property for a particular object, but we hope the property always to hold for each instance of data structure.

This paper addresses runtime verification for parametric properties, given as LTL schemas, on parametric execution traces which are extracted from a program at runtime. To this end, we formalize the notion of an LTL schema. Its syntax is used to describe the parametric property and the semantics answers the question, when a parametric trace satisfies or violates the parametric property. Then, we show how to leverage the methods and techniques to verify LTL at runtime (conventional, non-parametric traces against conventional, non-parametric properties) to the parametric case. More precisely, we pursue the following road-map. We show that (i) a parametric trace can be projected to a set non-parametric traces, incrementally and thus also at runtime. Moreover, we show that (ii) an LTL schema can be translated to a simple LTL formula, allowing to reuse existing, three-valued monitor generation routines, and that (iii) at runtime, a new monitor instance can be started for each new occurring non-parametric projection, allowing to monitor for each instance of the LTL schema. Overall, we obtain a runtime verification approach for LTL schemas on parametric traces. Following our formal developments, we have implemented the tool \texttt{xt-checker} based on standard Unix trace tools for checking LTL schemas on arbitrary binary code and have, as one example, checked successfully the above LTL schema to find memory leaks in existing Unix tools.

This paper is organized as follow: Section 2 explains some basic concepts and proposes the syntax and anticipatory semantics of LTL Schemas. Section 3 describes the construction of anticipatory monitors for LTL and LTL Schemas. In Section 4, \texttt{xt-checker} is introduced and our experiments are discussed. Section 5 compares our approach with related work, before we conclude the paper.
2 LTL and LTL Schemas

2.1 Preliminaries

LTL is a well-accepted logic to specify properties for systems whose behavior is characterized by a sequence of states which occur at discrete time steps. These states are then abstracted with a set of atomic propositions $AP$ which evaluate to $true$ in such a state. However, in parametric execution traces, we extend atomic propositions to parametric propositions. We introduce the notions of proposition, state, and trace, first non-parametric and then parametric in this section. Moreover, a special operation $Projection$ is proposed which will be used to map a parametric trace to a set of non-parametric traces.

Let $AP$ be a set of non-parametric propositions, called atomic propositions or simply propositions. An $AP$-state $s$ is a set of atomic propositions which are evaluated to $true$ in a corresponding system state. $\Sigma$ is the set of $AP$-states, i.e., $\Sigma = 2^{AP}$. A finite (or an infinite)$AP$-state-trace, or simply a finite (or infinite) non-parametric trace is any finite (or infinite) sequence of $AP$-states over $\Sigma$. Finite traces over $\Sigma$ are elements of $\Sigma^*$, and infinite traces are elements of $\Sigma^\omega$. If an $AP$-state $s \in \Sigma$ appears in a non-parametric trace $w \in \Sigma^* \cup \Sigma^\omega$ then we write $s \in w$.

Next, we extend the above concepts to the parametric case by giving a definition of parametric propositions. Let $[A \rightarrow B]$ and $[A \rightsquigarrow B]$ be the sets of total and partial functions from $A$ to $B$, respectively.

Definition 2.1 [Parametric propositions, states and traces] Let $X$ be a set of parameters and let $V$ be a set of corresponding parameter values. Let $AP$ be the set of atomic proposition names and let $p \in AP^{(n)}$ denote a constructor of arity $n \in \mathbb{N}$. Then the set of parametric propositions $AP(X)$ is defined as:

$$AP(X) = \bigcup_{n \in \mathbb{N}} \bigcup_{p \in AP^{(n)}} \{ p(x_1, \ldots, x_n) \mid x_k \in X, 1 \leq k \leq n \}$$

The set of parametric proposition instances $AP(\Theta)$ is the set of all propositions where each position is instantiated with an element of $V$, i.e., no position contains a variable:

$$AP(\Theta) = \{ p(\theta) \mid \theta \in \Theta \text{ is a partial function in } [X \rightsquigarrow V] \text{ and } \forall i : \theta(x_i) \text{ is defined, } n \in \mathbb{N} \}$$

Parametric states are sets of parametric propositions and parametric traces are then sequences of parametric states:

Definition 2.2 [Parametric states and traces] A parametric state $\Lambda X.s$ is a set of parametric proposition instances which hold in the corresponding system state. $\Sigma(\Theta)$ is the set of parametric states, i.e. $\Sigma(\Theta) = 2^{AP(\Theta)}$. A finite (or
infinite) parametric trace is any finite (or infinite) sequence of parametric states over $\Sigma(\Theta)$. Finite traces over $\Sigma(\Theta)$ are elements of $(\Sigma(\Theta))^*$, and infinite traces are elements of $(\Sigma(\Theta))^\omega$.

Note that atomic propositions can be viewed as special parametric propositions. For example, the begin/end propositions have the form $\text{begin}(i)$ and $\text{end}(i)$, where $i$ is the partial function undefined everywhere.

Definition 2.3 [Least Upper Bound] Partial functions $\theta$ in $[X \rightsquigarrow V]$ are called parameter instances. $\theta_1, \theta_2 \in [X \rightsquigarrow V]$ are called compatible if for any $x \in \text{Dom}(\theta_1) \cap \text{Dom}(\theta_2)$, $\theta_1(x) = \theta_2(x)$. $\Theta = \{\theta_1, \theta_2, \ldots, \theta_n\} \subseteq [X \rightsquigarrow V]$ are compatible if for any $\theta, \theta' \in \Theta$, $\theta, \theta'$ are compatible. We can combine compatible instance $\theta_1, \theta_2, \ldots, \theta_n$, written $\sqcup \Theta$, as follows:

$$\sqcup \Theta(x) = \begin{cases} \theta_i(x) & \text{when } \exists i : \theta_i(x) \text{ is defined} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

$\sqcup \Theta$ is also called the least upper bound (lub) of $\Theta$. $\theta \in [X \rightsquigarrow V]$ is less informative than $\theta' \in [X \rightsquigarrow V]$, or $\theta'$ is more informative than $\theta$, written $\theta \subseteq \theta'$, iff for any $x \in X$, if $\theta(x)$ is defined then $\theta'(x)$ is also defined and $\theta(x) = \theta'(x)$.

When $\{\Theta_i \mid i \in I\}$ is a set of sets in $[X \rightsquigarrow V]$, we define the least upper bound (lub) of $\{\Theta_i \mid i \in I\}$ as:

$$\sqcup \{\Theta_i \mid i \in I\} = \{\theta \in \Theta : \theta_i \in \Theta_i \text{ for each } i \in I \text{ s.t. } \sqcup \Theta_i \text{ exists}\}.$$

Definition 2.4 [Trace Projection[8]] Given the parametric state trace $\tau \in (\Sigma(\Theta))^*$, $\theta \in [X \rightsquigarrow V]$, and $\Lambda.X.s = \{p_1(\theta_1), \ldots, p_m(\theta_m)\}$, let $\Theta = \{\sqcup \{\theta_i, i \in I\} \mid 1 \leq i \leq m\} \cup \{\theta_1, \ldots, \theta_m\}$, the $\theta$-trace projection $\tau \mid_\theta \in \Sigma^*$ is the non-parametric state trace defined as:

$$-\varepsilon \mid_\theta = \varepsilon, \text{ where } \varepsilon \text{ is the empty trace, and}$$

$$-(\tau.\Lambda.X.s(X)) \mid_\theta = \begin{cases} (\tau \mid_\theta s) & \text{when } \exists \theta' \in \Theta, \theta' \subseteq \theta \\ \tau \mid_\theta & \text{when } \forall \theta' \in \Theta, \theta' \not\subseteq \theta \end{cases}$$

with $s = \{p_1, \ldots, p_m\}$, i.e. the non-parametric state which obtained by forgetting the parameters of propositions which hold in $\Lambda.X.s$.

The trace Projection operation $\tau \mid_\theta$ filters out all the states that are irrelevant to the parameter instance $\theta$, i.e., if no parameter instance contained in the current state is compatible with $\theta$, this state will not be included in the corresponding $\theta$-trace projection.
Lemma 2.5 Let $\pi \in (\Sigma(\Theta))^*$ and $\tau \in (\Sigma(\Theta))^\omega$ be finite and infinite non-parametric state traces respectively, $\theta \in \Theta$ be a parameter instance, then

$$(\pi \tau\bigr|_\theta) = (\pi\bigr|_\theta)(\tau\bigr|_\theta)$$

2.2 Syntax and Semantics of LTL and LTL Schema

As a logic to describe the temporal properties on infinite non-parametric traces, we recall the syntax and semantics of LTL [13].

The set of LTL formulae is inductively defined by the grammar:

$$\varphi ::= \text{true} \mid p \mid \neg \varphi \mid \varphi \land \varphi \mid \varphi \lor \varphi \mid \varphi U \varphi \mid \chi \varphi$$

with $p \in AP$.

In addition, we use four abbreviations, namely $\varphi \land \psi$ for $\neg(\neg \varphi \lor \neg \psi)$, $\varphi \rightarrow \psi$ for $\neg \varphi \lor \psi$, $F \varphi$ for $true U \psi$, and $G p$ for $\neg(true U \neg \varphi)$.

Let $w = a_0a_1 \ldots \in \Sigma^\omega$ be an infinite non-parametric trace with $i \in \mathbb{N}$ being a position. Then we define the semantics of LTL formulae inductively as follows:

- $w, i \models \text{true}$;
- $w, i \models p \in AP$ iff $p \in a_i$;
- $w, i \models \neg \varphi$ iff $w, i \not\models \varphi$;
- $w, i \models \varphi_1 \land \varphi_2$ iff $w, i \models \varphi_1$ and $w, i \models \varphi_2$;
- $w, i \models \varphi_1 \lor \varphi_2$ iff $w, i \models \varphi_1$ or $w, i \models \varphi_2$;
- $w, i \models \varphi_1 U \varphi_2$ iff $\exists k \geq i, (w, k \models \varphi_2)$ and $\forall l, i \leq l < k, (w, l \models \varphi_1)$;
- $w, i \models \chi \varphi$ iff $w, i + 1 \models \varphi$.

Further, $w \models \varphi$ holds iff $w, 0 \models \varphi$. Most properties of parametric state traces are also parametric, i.e., refer to each particular parameter instance. In order to describe these properties, we give the definition of LTL Schema below.

Definition 2.6 [LTL Schema] The set of LTL Schema $LTL(X \subseteq X)$ over a set of variable $X$ and a set of parametric positions $AP\langle X \rangle$ is defined by the following grammar:

$$LTL(X \subseteq X) = true$$

$$|\varphi\rangle = p(x_1, \ldots, x_n) \in AP\langle X \rangle, x_1, \ldots, x_n \in X$$

$$|LTL(X) \oplus LTL(X), \oplus \in \{U, \land, \lor\}$$

$$|\neg LTL(X)\rangle$$

$$|\chi LTL(X)\rangle$$

Note that $X \subseteq X$ denotes the set of free variables which are contained in $LTL(X)$ formula. To distinguish an LTL Schema formula from an ordinary,
non-parametric LTL formula, we write it as $\Lambda X.\varphi$. It is obvious that if we replace each parametric proposition in $\Lambda X.\varphi$ with the corresponding atomic proposition, then unique LTL formula $\varphi$ is obtained, and we call $\varphi$ the non-parametric counterpart of $\Lambda X.\varphi$. The relation is defined by a function $P : \Sigma(\Theta) \rightarrow \Sigma$.

**Definition 2.7** [Semantics of LTL Schema] Let $X \subseteq X$ be a set of parameters, $V$ be a set of parameter values, $\tau \in (\Sigma(\Theta))^\omega$ be an infinite parametric state trace with $i \in \mathbb{N}$ being a position, $\Lambda X.\varphi \in LTL(X)$ be LTL Schema formula, $\theta \in [X \sim V]$ be a parameter instance, then we define the semantics of LTL Schema formulae inductively as follows:

$$\tau, \theta, i \models true$$

$$\tau, \theta, i \models p(x_1, \ldots, x_n) \text{ iff } \tau|_\theta, i \theta \models p.$$  

$$\tau, \theta, i \models \Lambda X.\varphi_1 \oplus \Lambda X.\varphi_2 \text{ iff } \tau|_\theta, i \theta \models \varphi_1 \oplus \varphi_2$$

$$\tau, \theta, i \models \neg \Lambda X.\varphi \text{ iff } \tau|_\theta, i \theta \models \neg \varphi$$

$$\tau, \theta, i \models \chi \Lambda X.\varphi \text{ iff } \tau|_\theta, i \theta \models \chi \varphi.$$

with $P(\tau_i) = (\tau|_\theta)_i$. $(\tau|_\theta)_i$ represents the $i_{th}$ element of $\tau|_\theta$. Further $\tau, \theta \models \varphi$ holds iff $\tau, \theta, 0 \models \varphi$ holds.

2.3 Anticipatory Semantics for LTL and LTL Schema

In runtime verification, we aim at checking LTL properties of infinite traces by considering their finite prefixes. To overcome difficulties in defining an adequate boolean semantics for LTL on finite trace, we define our anticipatory semantics to interpret common LTL formulae on finite prefixes to obtain a truth value from the set $\mathbb{B}_3 = \{true, ?, false\}$ below.

Let $u \in \Sigma^*$ denote a finite non-parametric state trace. The truth of LTL formula $\varphi$ with respect to $u$, denoted by $[u \models \varphi]$, is an element of $\mathbb{B}_3$ defined as follows [5]:

$$[u \models \varphi] = \begin{cases} true & \text{if } \forall \sigma \in \Sigma^\omega : \pi \sigma \models \varphi; \\ false & \text{if } \forall \sigma \in \Sigma^\omega : \pi \sigma \not\models \varphi; \\ ? & \text{otherwise.} \end{cases}$$

For the same reason, we will define the anticipatory semantics for LTL Schema next. In the following, we always let $X \subseteq X$ be a set of parameters, $V$ be a set of parameter values, $\pi \in (\Sigma(\Theta))^*$ denote a finite parametric state trace, $\mathbb{B}_3 = \{true, ?, false\}$. $(\mathbb{B}_3, <)$ is a partially ordered set, where $false < ? < true$. It is apparently that $(\mathbb{B}_3, <)$ is a lattice. Then for any $x, y \in \mathbb{B}_3$, a greatest lower bound (glb) of $x$ and $y$, $x \cap y$ exists. Especially,

$$false \cap ? = false; \quad true \cap ? = ?; \quad false \cap true = false.$$
If $\Theta \subseteq \mathbb{B}_3$, the glb of $\Theta$ exists, denoted as $\sqcap \Theta$.

**Definition 2.8** [Anticipatory Semantics of LTL Schema for special parameter instance] For any given parameter instance $\theta \in \Theta$, the truth of LTL Schema $\Lambda X.\varphi$ with respect to $\pi$ and $\theta$, denoted by $[\pi, \theta \models \Lambda X.\varphi]$, is an element of $\mathbb{B}_3$ defined as follows:

$$[\pi, \theta \models \Lambda X.\varphi] = \begin{cases} 
\text{true} & \text{if } \forall \tau \in (\Sigma(\Theta))^\omega : \pi \tau, \theta \models \Lambda X.\varphi; \\
\text{false} & \text{if } \forall \tau \in (\Sigma(\Theta))^\omega : \pi \tau, \theta \not\models \Lambda X.\varphi; \\
? & \text{otherwise.}
\end{cases}$$

Based on the definition of semantics of LTL schema, the anticipatory semantics of LTL Schema can be further defined as:

$$[\pi, \theta \models \Lambda X.\varphi] = \begin{cases} 
\text{true} & \text{if } \forall \tau \in (\Sigma(\Theta))^\omega : (\pi \tau)|^\theta \models \varphi; \\
\text{false} & \text{if } \forall \tau \in (\Sigma(\Theta))^\omega : (\pi \tau)|^\theta \not\models \varphi; \\
? & \text{otherwise.}
\end{cases}$$

Based on the above definition and Lemma 2.5, we can get

**Lemma 2.9** Let $u = \pi|_o \in \Sigma^*$ be a finite non-parametric state trace, then

$$[u \models \varphi] = [\pi, \theta \models \Lambda X.\varphi]$$

**Definition 2.10** [Anticipatory Semantics for LTL Schema] Let $\Theta \subseteq [X \to V]$ be a set of parameter instances, then the truth of LTL Schema $\Lambda X.\varphi$ with respect to $\pi$ and $\Theta$, denoted by $[\pi, \Theta \models \Lambda X.\varphi]$, is an element of $\mathbb{B}_3$ as follows:

$$[\pi, \Theta \models \Lambda X.\varphi] = \sqcap_{\theta \in \Theta} \{[\pi, \theta \models \Lambda X.\varphi]\}$$

Based on the above definitions and Lemma 2.9, we can get

**Lemma 2.11** Let $\Theta = \{\theta_1, \ldots, \theta_n\} \subseteq [X \to V]$ be a set of parameter instances, $\Omega = \{\pi|_o \mid \theta \in \Theta\}$, then

$$[\pi, \Theta \models \Lambda X.\varphi] = \sqcap_{\theta \in \Theta} \{[\pi, \theta \models \Lambda X.\varphi]\} = \sqcap_{u \in \Omega} \{[u \models \varphi]\}$$

Given the LTL schema formula $\Lambda X.\varphi$ and the parameter instance $\theta$, we can combine several monitors $M_\varphi$ which are the anticipatory monitors of the corresponding LTL formula $\varphi$ to get the verdict of $[\pi, \Theta \models \Lambda X.\varphi]$. If there are several parameter instances, then we generate several copies of $M_\varphi$ for each instances. In the end, if and only if all the base monitors give the verdict *true*, the overall monitor will get the same verdict and all the corresponding monitors terminate normally. If and only if it appears that one of the base monitors gives the *false* verdict, the overall monitor give the verdict *false*,

83
the whole monitoring process for the given program should be interrupted, meanwhile, we can tell which parameter instance has resulted in an error. Otherwise, the overall monitor gives the verdict inconclusive. But one more thing that we must do is separating the parametric state trace into several non-parametric state traces according to each parameter instance. It is the duty of Projection operation.

3 Anticipatory Monitor for LTL and LTL Schema

In this section, we develop an automata-based construction procedure of anticipatory monitors for LTL and LTL schema. More specifically, for a given formula $\varphi \in \text{LTL}$, we recall the construction of FSM $M_\varphi$ that reads finite words $\pi \in \Sigma^*$ and outputs $[\pi \models \varphi]$ which is given in [5],[11],[10].

Monitor for LTL.

A nondeterministic Büchi automaton (NBA) is a tuple $A = (\Sigma, Q, Q_0, \delta, F)$, where $\Sigma$ is a finite alphabet, $Q$ is a finite and non-empty set of states, $Q_0 \subseteq Q$ is the set of initial states, $\delta : Q \times \Sigma \to 2^Q$ is the transition function, and $F \subseteq Q$ is the set of accepting states. $L(A)$ is the language accepted by $A$. For $q \in Q$, $A(q)$ is regarded as another Büchi automaton which is same as $A$ except that the initial state is $q$. Let $\varphi \in \text{LTL}$ is the property to be verified, and $A^\varphi = (\Sigma, Q^\varphi, Q_0^\varphi, \delta^\varphi, F^\varphi)$ and $A^{\neg \varphi} = (\Sigma, Q^{\neg \varphi}, Q_0^{\neg \varphi}, \delta^{\neg \varphi}, F^{\neg \varphi})$ for $\varphi$ and $\neg \varphi$ respectively, such that $L(A^\varphi) = L(\varphi)$ and $L(A^{\neg \varphi}) = L(\neg \varphi)$. The corresponding construction is standard [15].

For the automaton $A^\varphi$, we define a function $F^\varphi : Q^\varphi \to \mathbb{B}$ (with $\mathbb{B} = \{\top, \bot\}$) where we set $F^\varphi(q) = \top$ iff $L(A^\varphi) \neq \emptyset$. To determine $F^\varphi(q)$, we identify in linear time the strongly connected components in $A^\varphi$ which can be done using Tarjan’s algorithms [1]. Using $F^\varphi$, we define the NFA $\tilde{F}^\varphi = (\Sigma, Q^\varphi, Q_0^\varphi, \delta^\varphi, F^\varphi)$ with $F^\varphi = \{q \in Q^\varphi \mid F^\varphi \models \top\}$. Analogously, we set $\hat{F}^{\neg \varphi} = (\Sigma, Q^{\neg \varphi}, Q_0^{\neg \varphi}, \delta^{\neg \varphi}, \tilde{F}^{\neg \varphi})$ with $\hat{F}^{\neg \varphi} = \{q \in Q^{\neg \varphi} \mid F^{\neg \varphi} = \top\}$.

Having $\tilde{F}^\varphi$ and $\hat{F}^{\neg \varphi}$ at hand, we evaluate $[u \models \varphi]$ as follows:

**Evaluation of Anticipatory Semantics for LTL**: with the notation as before, we have

$$
[u \models \varphi] = \begin{cases} 
\top & \text{if } u \notin L(\hat{A}^{\neg \varphi}); \\
\bot & \text{if } u \notin L(\tilde{A}^\varphi); \\
? & \text{if } u \in L(\hat{A}^{\neg \varphi}) \cap L(\tilde{A}^\varphi).
\end{cases}
$$

**Monitor $M^\varphi$**: let $\varphi$ be a LTL formula, $\tilde{A}^\varphi$ and $\hat{A}^{\neg \varphi}$ be the NFAs, as defined above. Let $\tilde{A}^\varphi = (\Sigma, Q^\varphi, q_0^\varphi, \delta^\varphi, \tilde{F}^\varphi)$ and $\hat{A}^{\neg \varphi} = (\Sigma, Q^{\neg \varphi}, q_0^{\neg \varphi}, \delta^{\neg \varphi}, \hat{F}^{\neg \varphi})$ be deterministic automata with $L(\tilde{A}^\varphi) = L(\hat{A}^{\neg \varphi})$ and $L(\hat{A}^{\neg \varphi}) = L(\tilde{A}^\varphi)$.  

84
We define the product automaton $\overline{A} \varphi = \overline{A} \varphi \times \overline{A} \neg \varphi$ as the FSM $(\Sigma, \overline{Q}, \overline{q}_0, \overline{\delta}, \overline{\lambda})$, where

\begin{align*}
- \overline{Q} &= Q \varphi \times Q \neg \varphi; \\
- \overline{q}_0 &= (q \varphi_0, q \neg \varphi_0), \\
- \overline{\delta}((q, q'), a) &= (\delta \varphi(q, a), \delta \neg \varphi(q', a)), \text{ and} \\
- \overline{\lambda}: \overline{Q} \to \mathbb{B}_3 \text{ is defined by}
\end{align*}

\[
\overline{\lambda}((q, q')) = \begin{cases}
\top & \text{if } q' \notin \mathcal{F} \neg \varphi; \\
\bot & \text{if } q \notin \mathcal{F} \varphi; \\
? & \text{if } q \in \mathcal{F} \varphi \text{ and } q' \in \mathcal{F} \neg \varphi.
\end{cases}
\]

The monitor $M \varphi$ of $\varphi$ is the unique FSM obtained by minimizing the product automaton $\overline{A} \varphi$.

Correctness of $M \varphi$: let $\varphi \in \text{LTL}$ be a LTL formula and $M \varphi = (\Sigma, Q, q_0, \delta, \lambda)$ be the corresponding monitor. Then, for all $u \in \Sigma^*$, the following holds:

$$[u \models \varphi] = \lambda(\delta(q_0, u))$$

3.1 Monitor for LTL Schema

Now, we extend the construction towards LTL Schema. Starting with the base monitor $M \varphi$ and a set of parameters, the corresponding parametric monitor can be thought as a set of base monitors running in parallel, one for each parameter instance.

Definition 3.1 [Parametric Monitor $\Lambda X.M \varphi$][8] Given parameters $X$ with corresponding values $V$ and monitor $M \varphi = (\Sigma, Q, q_0, \delta, \lambda : [Q \to \mathbb{B}_3])$, the parametric monitor $\Lambda X.M \varphi$ is the monitor $(\Sigma(\Theta), [[X \to V] \to Q], \lambda \theta.q_0, \Lambda X.\delta, \Lambda X.\lambda)$, with $\Lambda X.\delta: [[X \to V] \to Q] \times \Sigma(\Theta) \to [[X \to V] \to Q]$ and $\Lambda X.\lambda: [[X \to V] \to Q] \to [[X \to V] \to \mathbb{B}_3]$ defined as

\[
\Lambda X.\delta(\sigma, \Lambda X.s)(\theta) = \begin{cases}
\delta(\sigma(\theta), s) & \text{if } \exists \theta' \in \Theta, \theta' \subseteq \theta \\
\sigma(\theta) & \text{if } \forall \theta' \in \Theta, \theta' \nsubseteq \theta;
\end{cases}
\]

where $\Theta = (\sqcup\{\{\theta_i, \ell\} \mid 1 \leq i \leq m\} - \{\ell\}) \cup \{\theta_1, \ldots, \theta_m\}$;

\[
\Lambda X.\lambda(\sigma)(\theta) = \lambda(\sigma(\theta))
\]

for any $\sigma \in [[X \to V] \to Q]$ and $\theta, \theta' \in [X \to V]$.

Let $\Lambda X.M \varphi = (\Sigma(\Theta), [[X \to V] \to Q], \lambda \theta.q_0, \Lambda X.\delta, \Lambda X.\lambda)$, $M \varphi = (\Sigma, Q, q_0, \delta, \lambda : [Q \to \mathbb{B}_3])$, $\pi \in (\Sigma(\Theta))^*$ be a finite parametric state trace, $\Theta = \{\theta_0, \theta_1, \ldots, \theta_m\}$ be a set of parameter instances,
Definition 3.2 [Run of $\Lambda X. M^\varphi$] For any $\sigma \in [X \to V] \to Q$, $\sigma$ is called an abstract state of $\Lambda X. M^\varphi$. A runtime state of $\Lambda X. M^\varphi$ on $\Theta$ and $\sigma$ is $m$-tuple $\eta = (\theta_0, q_0), \ldots, (\theta_m, q_m)$, $\forall i : 0 \leq i \leq m, q_i = \sigma(\theta_i)$, and $\sigma$ is called the corresponding abstract state of $\eta$. A run of $\Lambda X. M^\varphi$ on $\pi$ and $\Theta$ is finite runtime state sequence, $\mu = \eta_0 \eta_1 \ldots \eta_n$, $n = |\pi|$ is defined as:

- $\eta_0 = (\langle \theta_0, q_0 \rangle, \langle \theta_1, q_0 \rangle, \ldots, \langle \theta_m, q_0 \rangle)$;
- for any $0 < i < n$, $\eta_i = (\langle \theta_0, \Lambda X. \delta((\sigma_{i-1}, \pi_i)(\theta_0)) \rangle, \ldots, \langle \theta_m, \Lambda X. \delta((\sigma_{i-1}, \pi_m)(\theta_m)) \rangle)$;

where $q_0$ is the initial state of base monitor $M^\varphi$, and $\sigma_{i-1}$ is the corresponding static state of $\eta_{i-1}$.

Definition 3.3 [Output of a Run] Let $\mu = \eta_0 \eta_1 \ldots \eta_n$, $n = |\pi|$ be a run of $\Lambda X. M^\varphi$ on $\pi$ and $\Theta$, then we can define the output of a run as follows:

$$\Lambda X. M^\varphi(\pi, \Theta) = \exists_0 \leq i \leq m \{ \lambda(s_i) \}.$$  

where $\eta_n = (\langle \theta_0, s_0 \rangle, \ldots, \langle \theta_m, s_m \rangle)$, $s_i = \Lambda X. \delta((\sigma_{n-1}, \pi_n)(\theta_i))$, $0 \leq i \leq m$.

4 Implementation and Experiments

We have developed xt-checker\(^5\) as a tool which is tailored to check parameterized properties of system/library call traces of (arbitrary) binary programs and incorporates the formal concepts presented in this paper. Moreover, we have considered several experiments showing the practical benefits of xt-checker.

4.1 Implementation

Figure 1 illustrates the architecture of xt-checker. For a given LTL schema formula $\Lambda X. \varphi$, we first construct a Moore-type FSM monitor model $M^\varphi$ for the corresponding underlying LTL formula $\varphi$, following the construction described in Section 2. To this end, we make use of the LTL3-Tools\(^6\), which produce a textual description of the resulting Moore machine for a given LTL formula. Recall that the resulting monitor is optimal in two respects: First, the size of the generated deterministic monitor is minimal, and, second, the monitor identifies a continuously monitored trace as either satisfying or falsifying a property as early as possible [5].

To allow the resulting FSM to be used within our checker, it has to be realized as one of its parts. There are generally two ways to realize an FSM. The first uses a set of nested switch statements. The outer switch has a case for each possible state, while the inner switch has a case for each possible event. Another more concise way of coding is to organize states and affecting state

---

\(^5\) see [http://runtime.in.tum.de](http://runtime.in.tum.de) for further information

\(^6\) see [http://ltl3tools.sourceforge.net/](http://ltl3tools.sourceforge.net/)
transitions as a lookup table. The basic idea is that every state machine can be directly represented as a state transition table. Executing the FSM then becomes a relatively simple loop of finding the current state and event(input) in the transition table and then changing to the next state while generating the output result. The former model is suitable for small-scaled applications which consider only a single instance of FSM. Besides defining the FSM in a cleaner and more portable way, the latter approach is especially beneficial, when several instances of FSM have to be considered, i.e., verifying multi LTL formula instances in parallel. Therefore, we follow the second approach in xt-checker.

The binary program to be checked is executed by xt-checker in the scope of xtrace. xtrace represents a series of Unix and Windows event-based tracers, which observe execution traces and provide runtime information of the program to be verified. More precisely, ltrace and strace in Linux intercepts and records dynamic library calls and system calls which are called by the executed process, respectively. dtrace and truss are similar applications in Solaris and FreeBSD systems. Trace tools allow the observation of programs without any instrumentation at the underlying source code. This has two main advantages. First, the real and no modified code is executed, and, second, especially third-party code can be checked, for which no source code is available. In xt-checker, we currently use strace/ltrace, thus, we trace for system/library calls and focus on the Linux operating system.

The runtime information gathered by xtrace is parametric. The parametric execution trace is then projected to a set of non-parametric traces, that is \( u_1, u_2, \ldots, u_n \), with the instantiation of variables. The online monitors absorb the non-parametric traces and transit to new current states \( \{ s_{ui} \mid 1 \leq i \leq n \} \) if necessary and deliver a truth-value of the respective LTL formula \( \varphi \).

4.2 Experiments

We choose a representative property, safe dynamic memory allocation, for evaluating our framework. Memory leak is a typical problem in C code where
a programmer allocates memory at runtime (in the heap) and fails to deallocate it. A memory leak is a serious problem whenever a program runs for long time and consumes additional memory over time, or whenever system memory is limited. Dynamically allocating and deallocating memory is performed respectively by calls to the functions malloc and free. The property to check is that for each successful malloc there exists a corresponding free before the program exits. The LTL formula of this property is $G(\text{malloc} \rightarrow (\neg \text{exit} \cup \text{free})).$

The constructed FSM monitor is depicted in Figure 2. The sensitive events includes library function calls malloc, calloc, realloc, free and exit. Thus we apply the ltrace tool to gather runtime information.

We applied xt-checker to check for memory leaks of the widely-used GNU wget (v1.11.4) that retrieves content from web servers. When checking the execution wget ftp://ftp.gnu.org/README, the found undeallocated blocks are as follows:

calloc(1,20) malloc(24) malloc(28) malloc(104) malloc(28) malloc(104)

That is, 308 bytes in 6 blocks in total. Here we cautiously choose a relatively stable file on ftp.gnu.org with regard to making the experiment more repeatable. Note that the suspicious memory leak blocks detected by xt-checker match the results of Valgrind’s memcheck tools, which proofs that xt-checker follows a practically relevant concept.

5 Related Work

Parametric properties or dynamic properties have been receiving more and more attention in runtime verification, as shown by the increasing number of RV systems supporting them, e.g. [2], [3], [12]. But most of these techniques have two drawbacks. One is that they tightly couple the handling of parameter instances including binding and using operations with property checking, yielding monolithic monitors [6]. The major challenges these “monolithic monitors” approaches face are when to begin new parameter bindings, when to use them, and how to keep track of the status for each parameter instance during property checking. For example, [14] discussed the formal semantics of an alternative way of specifying formulae with parametric propositions that

---

7 see http://valgrind.org
differentiates between *binging* and *using* variables. In order to identify the
time of binding and using, a special operator is introduced which specifies
and enforces semantic constraints already on a syntactical level: when the left
hand side of the operator is matched, parameter bindings are generated based
on the current state and in the remainder of the formula, these bindings are
only used but do not generate any new bindings.

Such couplings of parameter bindings, using and property checking result
in rather complex algorithms which will be difficult to understand and apply.
Thus the practical expressiveness of the specification formalism may be con-
strained. For example, [14] can only deal with the formula which contains the
special operator correctly that constrains a lot the syntax and semantics.

The other flaw is that the finite trace semantics of a specification formalism
is not consistent. Therefore, for keeping consistency, [14] constrains the syntax
of the logic further, for example, just a formula in Next-free Linear-Time
Logic with free variables and quantification is concerned. Our techniques are
based on decoupling parameter binding and property checking, and on the
anticipatory semantics for *LTL Schema*. It has no further constrains on the
specification formalism.

JavaMop [7] proposes a different solution, based on a complete decoupling
of parameter binding from property checking. JavaMop supports several prop-
erty specification formalisms, including parametric regular expressions, para-
metric temporal logics, and parametric context-free patterns. However, the
technique currently supported by JavaMop can only handle a limited type of
traces, namely ones in which the first event for a particular property instance
binds all the property parameters. This limitation prevents JavaMop from
supporting many useful parametric properties [2]. [6] is another work which
tries to decouple parameter binding from property checking. It tries to sepa-
rate a parametric trace into several trace slices, where each slice corresponds to
a parameter binding. Then the problem of runtime verification for parametric
properties can be transferred to the verification problem of the corresponding
non-parametric properties. But that paper is not concerned with the case that
the parametric trace is a parametric state sequence.

Eagle [3], RuleR [4] are very general trace specification and monitoring sys-
tems, whose specification formalisms allow complex properties with parameter
bindings anywhere in the specification. However, the corresponding verifica-
tion methods differ completely from the approach followed here as no monitor
is synthesized from the given specification. Eagle and RuleR are based on
fixed-point logics and rewriting rules. These systems attempt to define gen-
eral formalisms supporting data binding among many other features.
6 Conclusion

This paper addresses runtime verification for parametric properties expressed as LTL schemas on parametric execution traces, which are extracted from a program at runtime. To this end, we formalized the notion of an LTL schema. We showed how our existing three-valued framework for monitoring LTL properties can be leveraged to parameterized LTL properties. We have implemented the approach as the tool xt-checker using standard Unix trace tools for arbitrary binary code (strace/ltrace) and have, as one example, checked successfully one LTL schema to find memory leaks in existing Unix tools.

References


Adaptive-Step-Size Numerical Methods in Rewriting-Logic-Based Formal Analysis of Interacting Hybrid Systems*

Muhammad Fadlisyah       Peter Csaba Ölveczky

Department of Informatics, University of Oslo, Norway

Erika Ábrahám

Computer Science Department, RWTH Aachen University, Germany

Abstract

This paper focuses on the formal modeling, simulation, and analysis of interacting hybrid systems that influence each other’s continuous behaviors. We define in the rewriting-logic-based Real-Time Maude tool a method for the numerical approximation of the continuous dynamics specified by ordinary differential equations. We adapt the Runge-Kutta-Fehlberg 4/5 method to define an adaptive-step-size technique that allows a more accurate approximation with less computational effort than fixed-step-size techniques. We also present experimental results for two thermal systems using different error tolerances.

Keywords: formal modeling, interacting physical systems, simulation, formal analysis, rewriting logic, Runge-Kutta-Fehlberg method

1 Introduction

Real-Time Maude [15] is a high-performance tool that extends the rewriting-logic-based Maude system [4] to support the formal modeling, simulation, and analysis of object-based real-time systems. Real-Time Maude emphasizes ease and expressiveness of specification, and has proved to be useful for analyzing a wide range of advanced applications that are beyond the scope of timed automata, such as communication protocols [16,13], wireless sensor

* This work was partially supported by the Research Council of Norway through the Rhythm project and the DAAD ppp project HySmart.
network algorithms \[11,17\], and scheduling algorithms that need unbounded data structures \[14\].

This paper is part of an investigation into how Real-Time Maude can be used to formally model, simulate, and analyze hybrid systems with both discrete and continuous behavior. In particular, we consider interacting physical entities, whose continuous behavior can be described by ordinary differential equations (ODEs). The physical entities interact and may influence each other’s continuous behavior. For example, a hot cup of coffee in a room interacts with the room through different kinds of heat transfer.

Our goal is to develop a technique to generate executable models of such systems in Real-Time Maude. For the continuous behavior of physical systems, which is described by ODEs, in our previous work \[7\] the execution was based on fixed-step-size numerical methods giving approximate solutions to ordinary differential equations. That is, to approximate a system’s behavior for a given time duration we approximate the behavior for a series a small time steps, each of them having a fixed duration. We used the Euler and the Runge-Kutta 2nd and 4th order methods for the small-step approximation.

In this paper we describe the integration of adaptive-step-size numerical methods, where the duration of the small time steps is chosen dynamically. Adaptive-step-size approximations have the advantages of: (i) making the analysis more precise by making the time step smaller when needed either to come close to a time instant when a discrete transition must be taken or when it is needed to maintain a desired precision of the approximation, and (ii) making the analysis more efficient by increasing the step size whenever the approximation allows it. In particular, adaptive step-size gives the user the possibility to define his/her own error tolerance to balance between desired precision and computational efficiency.

We develop an adaptation of the Runge-Kutta-Fehlberg 4/5 method (see e.g. \[8\]) that allows us to approximate the continuous behavior of our models with dynamic step-size, based on the error tolerance provided by the user. We describe the implementation of the adapted method in Real-Time Maude. Furthermore, we compare the results and execution times of both simulation and model checking using different error tolerances on two thermal systems with realistic parameters.

There are several simulation tools for hybrid systems based on numerical methods. MATLAB/Simulink \[20\] offers a wide range of numerical methods for simulation. HyVisual \[12\] considers linear multi-step and Runge-Kutta methods. CHARON \[6\] also uses linear multi-step methods with adaptive step size. In contrast to these tools, our approach supports, besides modeling and simulation, also the formal analysis, such as temporal logic model checking, of hybrid systems. Our approach also differs from model checkers for hybrid systems, such as CheckMate \[3\], PHAVer \[9\], d/dt \[5\], and HYPERTECH \[10\]
in that we do not use abstraction or over-approximation, but still support the modeling, reachability analysis, and LTL model checking of the full class of hybrid systems, describing the continuous dynamics by (possibly non-linear) ODEs. Whereas other formal tools use hybrid automata, chart or block models, or formulas for modeling, we use rewriting logic as the underlying modeling formalism. The models of our approach are compositional, where the continuous dynamics of a component may depend on explicitly modeled interactions with other components. Since the logic also supports classes and objects as well as the definition of any computable data type, physical systems controlled by some programs can be intuitively modeled and analyzed.

The paper is structured as follows: Section 2 gives an overview of Real-Time Maude. Section 3 briefly explains our approach for modeling hybrid systems in rewriting logic. Section 4 presents the adaptation of the Runge-Kutta-Fehlberg 4/5 method for our purposes, and Section 5 describes its implementation in Real-Time Maude. The case studies are summarized in Section 6 and concluding remarks are given in Section 7.

2 Real-Time Maude

A Real-Time Maude timed module specifies a real-time rewrite theory \((\Sigma, E, IR, TR)\), where:

- \((\Sigma, E)\) is a membership equational logic \([4]\) theory with \(\Sigma\) a signature\(^1\) and \(E\) a set of confluent and terminating conditional equations. \((\Sigma, E)\) specifies the state space as an algebraic data type, and contains a specification of a sort Time modeling the time domain.

- \(IR\) is a set of (possibly conditional) labeled instantaneous rewrite rules specifying the system’s instantaneous (i.e., zero-time) one-step transitions. The rules are applied modulo the equations \(E\).\(^2\)

- \(TR\) is a set of (possibly conditional) tick rewrite rules that model time elapse, written with syntax

  \[
  r1 [l] : \{t\} \Rightarrow \{t'\} \text{ in time } \tau
  \]

  \[
  crl [l] : \{t\} \Rightarrow \{t'\} \text{ in time } \tau \text{ if } \text{cond}
  \]

  where \(\tau\) is a term of sort Time denoting the duration of the rewrite.

The global states of a system are terms of the form \(\{l\}\); the form of the tick rules then ensures that time advances uniformly in a system. The Real-Time Maude syntax is fairly intuitive (see \([4]\)). For example, a function \(f\) with arguments of sorts \(s_1 \ldots s_n\) and value of sort \(s\) is declared \(\text{op } f : s_1 \ldots s_n\)

\(^1\) i.e., declarations of sorts, subsorts, and function symbols

\(^2\) \(E\) is a union \(E' \cup A\), where \(A\) is a set of equational axioms (associativity, commutativity, identity, etc.). Deduction is performed modulo \(A\). A term is reduced to its \(E'\)-normal form modulo \(A\) before any rewrite rule is applied.
Fig. 1. Physical system components and their interaction in a simple thermal system.

$\rightarrow s$. Equations are written $eq t = t'$, and $ceq t = t'$ if $cond$ for conditional equations. Variables are declared with the keywords $var$ and $vars$.

A class declaration $class C | att_1 : s_1, \ldots, att_n : s_n$. declares a class $C$ with attributes $att_1$ to $att_n$ of sorts $s_1$ to $s_n$. A subclass inherits all attributes and rules of its superclasses. An object of class $C$ is represented as a term $< O : C | att_1 : val_1, \ldots, att_n : val_n >$ of sort $Object$, where $O$, of sort $Oid$, is the object's identifier, and $val_1$ to $val_n$ are the current values of the attributes $att_1$ to $att_n$. In a concurrent object-oriented system, a state is a term of the sort $Configuration$. It has the structure of a multiset made up of objects and possibly messages. Multiset union for configurations is denoted by a juxtaposition operator (empty syntax) that is declared associative and commutative, so that rewriting is multiset rewriting supported directly in Real-Time Maude.

Real-Time Maude specifications are executable under reasonable conditions, and the tool offers a variety of formal analysis methods. The $rewrite$ command ($trew t$ in time $\leq \tau$.) simulates one fair behavior of the system up to duration $\tau$, where $t$ is the initial state and $\tau$ is a term of sort $Time$. The $search$ command uses breadth-first search to analyze all possible behaviors of the system and checks whether a state matching a pattern can be reached from the initial state such that a given condition is satisfied. Real-Time Maude also extends Maude's linear temporal logic model checker to check whether each behavior, possibly up to a certain time bound, satisfies a linear temporal logic formula. Finally, the $find earliest$ command determines the shortest time needed to reach a desired state.

3 Modeling Physical Systems

In [7] we present a framework for the modeling and analysis of physical systems based on the effort and flow approach [21]. One key difference between our work and most other formal approaches to hybrid systems is that, instead of considering the continuous behavior of a component in isolation, we consider a
hybrid system to consist of a set of physical components, where the continuous
dynamics of a component may depend on the continuous dynamics of other
components. The physical interactions between physical entities are therefore
considered as first-class citizens, and a physical system is modeled as a network
of physical entities and physical interactions, as shown in Figure 1.

A physical entity consists of a set of attributes, a real-valued effort variable, and a continuous dynamics. The attribute values can only be changed by discrete events. For example, the phase of a material (solid, liquid, gas, plasma) changes via discrete phase transitions. The effort variable represents a physical quantity, such as temperature, evolving over time. Its continuous dynamics is given as an ordinary differential equation (ODE). A physical entity can have one or more physical interactions with one or more physical entities. A physical interaction represents an interaction between two physical entities. It consists of a set of attributes, a real-valued flow variable, and a continuous dynamics. The flow variable represents a quantity describing the interaction between two entities, e.g., the heat flow rate in a thermal interaction. Its value is determined by the continuous dynamics in the form of an equation.

The continuous dynamics of a physical entity is an ODE with the time
derivative of its effort on the left-hand side and an expression possibly referring
to the entity’s attributes and to the flows of connected interactions on the
right-hand side. Dually, the continuous dynamics of a physical interaction is
an equation with the flow variable on the left-hand side and an expression
possibly referring to the interaction’s local attributes and the efforts of the
connected entities on the right-hand side. This way the direct coupling of the
ODEs of physical entities [2] can be avoided.

Fig. 1 shows a thermal system representing a cup of coffee in a room. In
thermal systems, a physical entity is a thermal entity, whose effort variable \((T)\)
denotes the temperature of the entity and whose continuous dynamics defines
the heat gained or lost by the entity as time evolves and its temperature
changes. Likewise, a physical interaction is a thermal interaction whose flow
variable \((Q)\) denotes the heat flow rate. Examples of thermal interactions are
conduction, convection, and radiation. Their continuous dynamics are given
by equations for the heat transfer rates.

The basic behavior of physical system components is their continuous behavior. We use single-step, initial-value-problem numerical methods [2] to approximate the continuous behaviors of physical system components by advancing time in small discrete time steps, and computing the values of the continuous variables at each “visited” point in time. In previous work, we have integrated the Euler, Runge-Kutta 2nd order, and Runge-Kutta 4th order methods to our modeling technique. However, in these methods, the size of the small time steps in the execution is constant.
Fig. 2. The local and global errors in a numerical approximation.

4 Adaptive-Step-Size Numerical Methods

To approximate some continuous behavior with rapid variations or abrupt changes, for fixed step-size methods we have to choose a small step-size to get satisfactory results. However, small step-sizes come at a very high computational cost. For example, approximating the coffee system in Fig. 1 with our implementation of the Runge-Kutta 4th order method for 1000 time units took 38 minutes using step size 1, but took 285 minutes using step size 0.5. For systems with more stable dynamics, larger step-size can be used to get an adequate approximation more efficiently. The idea of adaptive step-size techniques is to adapt the trajectory of the approximation by estimating and controlling the error at each step. Such error estimates are used as a basis for dynamically increasing or decreasing the step size.

4.1 Approximation Errors

Assume a continuous variable $y$ with time derivative $y'(t) = f(t, y(t))$ and an initial value $y(0) = y_0$. To approximate $y(T)$ for some $T > 0$, small-step numerical methods compute a sequence of values $y_1, y_2, ..., y_N$ that approximate the exact values $y(t_1), y(t_2), ..., y(t_N)$ for some time points $t_1 < t_2 < ... < t_N$, $t_N = T$. Computing $y_{n+1}$ is based on the value of $y_n$, and can be seen as taking a small time step with the step size $h_n = t_{n+1} - t_n$. For the methods with fixed step-size, all $h_n$, $n = 0, \ldots, N - 1$, are equal. This is not the case for methods with adaptive step size.

There are two sources of errors in the above approach. Round-off errors are due to the limitations of computers in representing numbers. Discrete-time approximation errors originate from the fact that the approximations $y_1, y_2, ..., y_N$ deviate from the exact values $y(t_1), y(t_2), ..., y(t_N)$. If we assume that an exact arithmetic is used and thus there are no round-off errors, the deviation $\varepsilon^g_n = y(t_n) - y_n$ is called the global error at time point $t_n$ (see Fig. 2).

The global error $\varepsilon^g_{n+1}$ sums up from the global error at $t_n$ and its propa-
gation, and an additional error due to the last approximation. Let the local solution \( u \) be the solution of 
\[
  u'(t) = f(t, u(t))
\]
for the initial value \( u(t_n) = y_n \). Then the global error at \( t_{n+1} \) is 
\[
  \varepsilon_{g,n+1} = (y(t_{n+1}) - u(t_{n+1})) + (u(t_{n+1}) - y_{n+1}).
\]
The first summand is the global error \( \varepsilon_{g,n} = y(t_n) - u(t_n) \) propagated during the time step from \( t_n \) to \( t_{n+1} \) yielding the error \( y(t_{n+1}) - u(t_{n+1}) \). With other words, it is the difference at time point \( t_{n+1} \) of two solutions of the ODE that differ by \( y(t_n) - y_n \) at time point \( t_n \). The second summand, which we call the local error \( \varepsilon_{l,n+1} = u(t_{n+1}) - y_{n+1} \), is the approximation error of the last step.

4.2 Adjusting the Step Size

We cannot control the global error directly. However, we can control it indirectly by controlling the local error in each time step [19].\(^3\) We need to make the step size adaptive such that the local error in each step is bound by some error tolerance. The error tolerance is determined by a user-given value \( \tau \).

Two commonly used definitions [18] are error per step requiring 
\[
  |\varepsilon_{l,n+1}| \leq \tau,
\]
and error per unit step defining 
\[
  |\varepsilon_{l,n+1}| \leq \tau \cdot h_n
\]
as condition for each \( n \geq 0 \).

To check if these conditions are fulfilled we need to measure the local error, which depends on the order of the method used. The order of a numerical method corresponds to how fast a sequence of approximations generated by a method converges toward the expected solution. The higher the order, the better the approximation. One way to estimate the local error \( \varepsilon_{l,n+1} \) for the approximation \( y_{n+1} \) by a method of order \( p \geq 1 \) is to compute the approximation \( \hat{y}_{n+1} \) also with a higher order \( \hat{p} > p \) method.\(^4\) The local error of the method of order \( p \) can be estimated by comparing the results 
\[
  \varepsilon_{n+1} \approx \epsilon_{n+1} = \hat{y}_{n+1} - y_{n+1}.
\]
It can be shown that it is a correct asymptotic result when \( h \to 0 \) [19].

If the local error passes the test, this error is accepted, and the step size will be increased for the next step. For the error per step condition we define 
\[
  h_{n+1} = \alpha \cdot h_n \text{ with } \alpha = \left(\frac{\tau}{|\varepsilon_{n+1}|}\right)^{1/p+1}. \quad \text{Note that } |\varepsilon_{n+1}| \leq \tau \text{ implies } \alpha \geq 1.
\]
If the local error does not pass the test, this step size is rejected and will be decreased. For the error per step condition we define 
\[
  h_n' = \alpha \cdot h_n. \quad \text{Note that, since the condition was not satisfied, we have } \alpha < 1.
\]

4.3 The Runge-Kutta-Fehlberg Order 4/5 Method

Given a numerical method of a certain order, any other numerical method of a higher order can be used to obtain an estimate of the local error. However, computing a second approximation using a second methods in each time step

\(^3\) Note that the global error cannot be controlled by the numerical methods. If the solutions of the ODEs are unstable, the global error can grow fast even for small local errors.

\(^4\) Note that the computation for \( \hat{p} \) can re-use most of the computations for \( p \), see Section 4.3.
Fig. 3. The execution of a physical system model with adaptive step size.

may be computationally expensive [1]. This problem can be avoided by using methods that share function values which are known as embedded pairs.

The Runge-Kutta-Fehlberg order 4/5 method (RKF45) [8] makes use of such embedded pairs. It uses a 5th order method to estimate the local error of a 4th order method. In each step, for the approximation with the 4th order method 5 values (slopes) must be calculated that are used to compute $y_{n+1}$ as a weighted sum. The 5th order method, used for the error estimation, needs 6 slope values. However, 5 of them are already computed by the 4th order method, thus only one slope must additionally be determined.

The 5th order approximation $\hat{y}_{n+1}$ is computed to estimate the local error of the 4th order approximation $y_{n+1}$. However, since higher order methods yield more exact results than lower order ones, we use $\hat{y}_{n+1}$ instead of $y_{n+1}$ as approximation result. This technique is called local extrapolation.

5 Integrating the Adaptive-Step-Size Method for Modeling Thermal Systems

This section gives an overview on the integration of the adaptive-step-size technique based on the RKF45 method into our modeling framework to support the formal modeling and analysis of hybrid systems with interacting components in Real-Time Maude. Fig. 3 shows the global framework.

5.1 Modeling Thermal Systems in Real-Time Maude

We illustrate our approach thermal systems as the one shown in Fig. 1. Physical entities in thermal systems are thermal entities, and physical interactions are thermal interactions. The temperature of a thermal entity changes according to $\dot{T} = \frac{\Sigma Q}{m \cdot c}$, where $\Sigma Q$ is the sum of the heat flow rates of the connected thermal interactions, $m$ is the entity’s mass, and $c$ is its heat capacity. Like-
wise, the heat flow rate $\dot{Q}$ between two entities through conduction is defined by $\dot{Q} = \frac{k}{L} \cdot A \cdot (T_1 - T_2)$, where $T_1$ and $T_2$ are the current temperatures of the interacting entities, and $k$, $L$, and $A$ are, respectively, the thermal conductivity, the thickness, and the area of the entity from which heat flows by conductivity.

We model thermal entities in Real-Time Maude as objects of the following class ThermalEntity:

```maude
class ThermalEntity | temperature : Rat, mode : CompMode, mass : PosRat, heatCap : PosRat, temp-p : Rat, temp-o1 : Rat, temp-o2 : Rat.
```

The effort variable `temperature` represents the entity’s temperature. The attribute `mode` is used to distinguish between different modes for the continuous dynamics (see below). The attributes `mass` and `heatCap` denote the mass and the heat capacity of the entity, respectively. `temp-p`, `temp-o1`, `temp-o2` are auxiliary attributes used for the computation by the RKF45 method. The entity’s continuous dynamics, described below, specifies the evolution of the temperature, depending on the heat transfer from or to the object.

We can define more specific types of thermal entities as subclasses of the base class ThermalEntity. For example, the following class WaterEntity represents water substance:

```maude
class WaterEntity | phase : MatterState, heatTrans : Rat, heatTrans-p : Rat, heatTrans-o1 : Rat, heatTrans-o2 : Rat.
subclass WaterEntity < ThermalEntity.
```

The attribute `phase` represents the phase of the water substance, which can be one of the main phases solid, liquid, gas, or one of the phase transitions melting, freezing, evaporating, or condensing. The change from a main phase to a phase transition occurs when the temperature reaches a given value, whereas a change from a phase transition to a main phase happens when the value of the heat accumulated during the phase transition divided by the mass of the entity reaches a given value called the latent heat. The attribute `heatTrans` stores the accumulated heat of the water in the phase transitions. The remaining attributes are needed for the computation of the approximations.

The `mode` determines the computation mode for the continuous dynamics. For water, the continuous dynamics of the temperature is the same in all three main phases, whereas the temperature does not change during phase transitions. In addition to the default computation mode for the main phases, we add the mode `phaseTrans` for phase transitions.

Each phase change is modeled by an instantaneous rewrite rule. We show two of the eight such rules for water:\footnote{We do not show the variable declarations; instead we follow the Maude convention that variables are written in capital letters, and that function symbols (including constants) start with a lower-case letter.}

```maude
We do not show the variable declarations; instead we follow the Maude convention that variables are written in capital letters, and that function symbols (including constants) start with a lower-case letter.
```
crl [solid-to-melting] :
  < E : WaterEntity | temp : T, phase : solid >
  => < E : WaterEntity | phase : melting, mode : phaseTrans, heatTrans : 0 > if T >= 0 .
crl [melting-to-liquid] :
  < E : WaterEntity | phase : melting, heatTrans : QT, mass : M >

Thermal Interactions model the heat transfer between thermal entities. Examples are conduction, convection, and radiation. We define a class for general thermal interactions and subclasses for the three heat transfer mechanisms:

class Conduction | thermCond : PosRat, thickness : PosRat .
class Convection | convCoeff : PosRat .
class Radiation | emissive : PosRat .
subclass Conduction Convection Radiation < ThermalInteraction .

The ThermalInteraction class contains common attributes of thermal interactions: entity1 and entity2 are the object identifiers of the two interacting thermal entities; the flow variable hfr specifies the heat flow rate $\dot{Q}$ of the thermal interaction; area is the area of the interaction; hfr-p1 to hfr-p5 are auxiliary attributes used for the computation by the RKF45 method. The subclasses have additional interaction-specific attributes. For conductivity, thermCond is the thermal conductivity of the material and the thickness is the thickness of the material through which the conduction occurs.

5.2 Computing the Step Size

We define the following class to manage the numerical method computation:

ops adj1 adj2 : -> CompStepSize [ctor] .
ops static dynamic : -> StepSizeType [ctor] .
ops eps epus : -> ErrorControlType [ctor] .

class SysMan | numMethod : NumMethod, stepSizeDef : Rat, stepSizeCur : Rat, stepSizeType : StepSizeType, errorTol : Rat, compStepSize : CompStepSize, safetyFactor : Rat, limitStepSize : Bool, stepSizeMin : Rat, stepSizeMax : Rat, limitAdjustRate : Bool, adjustRateMin : Rat, adjustRateMax : Rat, errorControl : ErrorControlType, localExtrapolation : Bool .

The attribute numMethod specifies which numerical method is used. The attribute stepSizeDef stores the initial step-size, and stepSizeCur the current step-size. The stepSizeType determines if fixed or adaptive step-size is used. The errorTol defines the error tolerance in adaptive-step-size computation (assuming that we use a single tolerance value). The compStepSize defines which step-size computation technique is used. The safetyFactor defines a fraction of the locally optimal step size which may be used to reduce the approximation error. The limitStepSize, stepSizeMin, and stepSizeMax limit the value of the step size. The limitAdjustRate, adjustRateMin, and adjustRateMax are used to limit increasing or decreasing rate of the step size. The errorControl chooses either error per step or error per unit step for
controlling the step size. The \texttt{localExtrapolation} specifies whether to use the extrapolation technique in the numerical computation.

The RKF45 method stores the approximations for the temperature values by the 4th and 5th order methods in the attributes \texttt{temp-o1} and \texttt{temp-o2} of the thermal entities. The function \texttt{maxError} computes the maximal local error estimate over all thermal entities in the system:

\begin{verbatim}
op maxError : Configuration -> Rat.
eq maxError(
    < E : ThermalEntity | temp-o1 : TEMP-O1, temp-o2 : TEMP-O2, mode : default >
    < SM : SysMan | errorControl : eps > REST) =
    max(abs(TEMP-O1 - TEMP-O2), maxError(< SM : SysMan | > REST)).
eq maxError(CONFIG) = 0 [otherwise].
\end{verbatim}

The \texttt{adjustRate} function computes the factor of the step size adjustment using error per step (we have a similar function for error per unit step):

\begin{verbatim}
op adjustRate : CompStepSize Rat Rat Rat ErrorControlType -> Rat.
eq adjustRate(adj1, ERR, ERRTOL, SAF, eps) = SAF * root5(ERRTOL / ERR).
\end{verbatim}

The function \texttt{root5(X)} computes $X^{1/5}$ as $exp(1/5 \cdot ln(X))$.

The function \texttt{stepSizeRKF} computes the step size based on the RKF45 method. If the maximal local error is below the tolerance value, it returns a pair of values, consisting of the current step size and the step size for the next time step. If not, the function is recursively called after computing new approximations with a smaller step-size. The following equation defines this function when putting no limitations on the step-size modifications:

\begin{verbatim}
op stepSizeRKF : Configuration -> ErrorStepSize.
ceq stepSizeRKF(
    limitStepSize : false, limitAdjustRate : false, errorControl : ERRCTR > REST) =
if ERR <= ERRTOL then SSCUR ; SSRKF
else stepSizeRKF(compute-EF-RKF45(< SM : SysMan | stepSizeCur : SSRKF > REST)) fi
if ERR := maxError(< SM : SysMan | > REST) /
SSRF := adjustRate(ADJ, ERR, ERRTOL, SAF, ERRCTR) * SSCUR.
\end{verbatim}

We also implemented a similar function that limits the rate of step-size change.

The following \texttt{tick rule} advances time in the system by the step size computed by \texttt{stepSizeRKF}, and computes the new values of the effort variables for all thermal entities. These values are the 5th order approximations if the extrapolation is used, and the 4th order approximations otherwise.

\begin{verbatim}
crl [tick-adaptive-stepsize] :
    < SM : SysMan | stepSizeDef : SS, stepSizeType : dynamic, errorTol : ERRTOL > REST
=>
delta(< SM : SysMan | stepSizeCur : firstES(SSPAIR), stepSizeDef : secondES(SSPAIR)> REST) in time firstES(SSPAIR)
if TimeCanAdvance(< SM : SysMan > REST) /
SSPAIR := stepSizeRKF(compute-EF-RKF45(< SM : SysMan | stepSizeCur : SS > REST)).
\end{verbatim}

\begin{verbatim}
eq delta(< SM : SysMan | numMethod : rkf45, localExtrapolation : true > REST) =
    compute-EF-RKF45-Order5(< SM : SysMan > REST).
eq delta(< SM : SysMan | numMethod : rkf45, localExtrapolation : false > REST) =
    compute-EF-RKF45-Order4(< SM : SysMan > REST).
\end{verbatim}
5.3 Integrating the RKF45 Method

The general model for adapting numerical methods in our effort/flow framework is depicted in Fig. 4. We use time discretization, and compute approximations for each small time-step. To compute the approximations by a numerical method, some slopes $k_1$ to $k_n$ must be computed.\(^6\) For the RKF45 method we need six slopes $k_1$ to $k_6$, as explained in Section 4. For each $k_i$ we need to compute a linear approximation of the behavior for a small time-step, starting from some initial point. This is done by (1) first calculating the heat flow rates of all thermal interactions at the initial point, (2) summing up the heat flow rates for all connected interactions for each entity, and (3) linearly approximate the effort, i.e. the temperature, after a small time-step, assuming that the computed heat flow rates are constant. Due to lack of space, in the following we restrict ourselves to explain these computation steps for $k_1$ (up to $k_n$, the other cases are similar but use different auxiliary attributes).

The function `computeFlow-IP` computes the heat flow rate of each thermal interaction for the initial point according to the laws of physics as described in [7]. We only show the case for thermal conductions:

```plaintext
op computeFlow-IP : Configuration -> Configuration .

ceq computeFlow-IP(< E1 : ThermalEntity | temperature : T1 >
                   < E2 : ThermalEntity | temperature : T2 >
                   < TI : Conduction | entity1 : E1, entity2 : E2, area : A, thermCond : K, thickness : L | REST>) =

  computeFlow-IP(< E1 : ThermalEntity | >
                  < E2 : ThermalEntity | > REST)
  if QDOT-T := Qdot-Conduction(K, L, A, T1, T2).

eq computeFlow-IP(CONFIG) = CONFIG [wise] .
```

The equation above computes the initial heat flow rate for a thermal interaction TI of type Conduction between two thermal entities E1 and E2, and then recursively applies the function to the remaining configuration. The function

---

\(^6\) In our previous works, we have applied this technique for the Euler, Runge-Kutta 2nd order, and Runge-Kutta 4th order methods.
Qdot-Conduction defines the dynamics as $\dot{Q} = \frac{K}{L} \cdot (T_1 - T_2)$.

The function sumFlows-IP computes the sum of the initial heat flow rates of all thermal interactions connected to a thermal entity:

$$\text{eq sumFlows-IP}(\text{CONFIG}, E) = 0 \quad \text{[otherwise]}.$$ 

The function computeEffort-P1 linearly approximates the temperature of each thermal entity in the system after a time step, assuming constant flow rates over the time step. It invokes the function Tdot representing the continuous dynamics given by $\dot{T} = \frac{\sum \dot{Q}}{m \cdot c}$, where $\sum \dot{Q}$ is the sum of the heat flow rate values of the thermal interactions of the entity as computed by sumFlows-IP, $m$ the mass, and $c$ the heat capacity. The attributes numMethod and stepSize of SM determine the numerical method and time step size, respectively:

$$\text{eq computeEffort-P1}(\text{CONFIG}) = \text{CONFIG} \quad \text{[otherwise]}.$$ 

6 Case Studies

This section investigates how the adaptation of the adaptive step size technique based on the RKF45 method affects the accuracy and performance of the simulation and time analysis of thermal systems. We start with a cup of hot coffee in a room. Then we add a heater giving a constant heat flow to the coffee. In the analysis we use the error per step, and the extrapolation for the computation.

6.1 Case Study 1: A Cup of Coffee in a Room

We first model a cup of hot coffee in a room, as shown in Fig. 1, with conduction and convection as thermal interactions, and with realistic physical parameters. The initial state consists of a SysMan object managing the numerical computation, the thermal entity objects coffee and room, and two
thermal interaction objects that model the heat flow:

\[ \text{eq cs1} = \{ \langle \text{coffe} : \text{WaterEntity} | \text{temperature} : 70, \text{heatCap} : \text{coffeeHC}, \text{mass} : \text{coffeeMass}, \text{mode} : \text{default}, \text{temp-p} : 0, \text{temp-o1} : 0, \text{temp-o2} : 0, \text{phase} : \text{liquid}, \text{heatTrans} : 0, \rangle \text{heatTrans-p} : 0, \text{heatTrans-o1} : 0, \text{heatTrans-o2} : 0 \} \]

\[ \langle \text{room} : \text{ThermalEntity} | \text{temperature} : 20, \text{heatCap} : \text{roomHC}, \text{mass} : \text{roomMass}, \text{mode} : \text{default}, \text{temp-p} : 0, \text{temp-o1} : 0, \text{temp-o2} : 0 \} \]

\[ \langle \text{crCond} : \text{Conduction} | \text{entity1} : \text{coffe}, \text{entity2} : \text{room}, \text{hfr} : 0, \text{therCond} : k, \text{area} : \text{condArea}, \text{thickness} : \text{cupThick}, \text{hfr-p1} : 0, \text{hfr-p2} : 0, \text{hfr-p3} : 0, \text{hfr-p4} : 0, \text{hfr-p5} : 0 \} \]

\[ \langle \text{crConv} : \text{Convection} | \text{entity1} : \text{coffe}, \text{entity2} : \text{room}, \text{hfr} : 0, \text{convCoeff} : h, \text{area} : \text{convArea}, \text{hfr-p1} : 0, \text{hfr-p2} : 0, \text{hfr-p3} : 0, \text{hfr-p4} : 0, \text{hfr-p5} : 0 \} \]

\[ \langle \text{sm} : \text{SysMan} | \text{numMethod} : \text{rkf45}, \text{stepSizeCur} : \text{INIT-TIME-STEP}, \text{stepSizeDef} : \text{INIT-TIME-STEP}, \text{stepSizeType} : \text{dynamic}, \text{errorTol} : 1/1000, \text{compStepSize} : \text{adj1}, \text{safetyFactor} : 9/10, \text{limitStepSize} : \text{false}, \text{stepSizeMin} : 1/100, \text{adjustRateMin} : 1/4, \text{errorControl} : \text{eps}, \text{localExtrapolation} : \text{true} \} \].

The behavior of the system until time 500 can be simulated using the following timed rewriting command:

\[ \text{Maude} \langle \text{trew cs1 in time <= 500 .} \rangle \]

Fig. 5 shows the simulation results using the error tolerance $10^{-3}$. The diagram on the left shows how the temperature of the coffee decreases and the one of the room increases as the heat flows. The small diagram inside shows the change of the relative values of the global error of the approximation of both temperatures at each time step.\(^8\). The diagram on the right shows the change of the step size which is getting larger as time advances.

The following table compares the simulation results using different error tolerances for the simulation time of 500:

<table>
<thead>
<tr>
<th>Error Tolerance</th>
<th>Effort</th>
<th>Error Abs (+ Rel %)</th>
<th>CPU Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^{-3}$</td>
<td>$T_c$</td>
<td>3.471e-06 (4.9815e-06)</td>
<td>0.0101 (0.0354)</td>
</tr>
<tr>
<td></td>
<td>$T_r$</td>
<td>8.1800e-08 (4.0930e-07)</td>
<td>2.3796e-04 (0.0011)</td>
</tr>
<tr>
<td>$10^{-4}$</td>
<td>$T_c$</td>
<td>3.471e-06 (4.9815e-06)</td>
<td>0.0006 (0.0123)</td>
</tr>
<tr>
<td></td>
<td>$T_r$</td>
<td>8.1800e-08 (4.0930e-07)</td>
<td>8.5957e-05 (4.1164e-04)</td>
</tr>
<tr>
<td>$10^{-5}$</td>
<td>$T_c$</td>
<td>3.471e-06 (4.9815e-06)</td>
<td>0.0012 (0.0041)</td>
</tr>
<tr>
<td></td>
<td>$T_r$</td>
<td>8.1800e-08 (4.0930e-07)</td>
<td>2.8990e-05 (1.3895e-04)</td>
</tr>
</tbody>
</table>

\(^{8}\) The relative approximation error percentage is computed using \[\frac{|\text{val}_{\text{exact}} - \text{val}_{\text{approx}}|}{|\text{val}_{\text{exact}}|}\]. Its percentage error is 100 times the relative error.
It shows the values of minimum, maximum, and average of absolute and relative values of global errors of both temperatures of the coffee $T_c$ and the room $T_r$, including the execution time. The results show that by decreasing the error tolerance we increase indirectly the accuracy of the computation, with the consequent increase in the computation time. The values of maximum and average errors which are greater than the error tolerances remind us again that the error tolerance does not control directly the global error. Note that the error values of the coffee are greater those of the room because the change rate of the temperature of the coffee is greater than the room.

6.2 Case Study 2: Keeping the Coffee Warm

To illustrate hybrid behavior in thermal systems, we add a heater providing a constant heat flow of 1.5 KW to the cup of coffee. We start with an initial coffee temperature of $-10^\circ C$ to go over the phase transitions from solid to liquid through the melting phase transition. Due to lack of space, we refer to [7] for a detailed model description.

Fig. 6 shows the simulation results using the error tolerance of $10^{-3}$. The diagram on the left shows how both temperatures of the coffee and the room increase as the coffee receives constant heat flow from the heater. It shows how the discrete behavior of the coffee, namely the changes from one physical state to the other (here solid to melting, melting to liquid, and liquid to evaporating), affect its continuous dynamics. The diagram on the right shows the changes of the step size, but unlike the previous case study, here the step size increases and decreases following the changes of the dynamics of the coffee.

We model the changes in the phase transition phenomena as discrete events that change the dynamics of the physical entities. We can use the find earliest Real-Time Maude command to find out discrete changes in our model. For example, we use this command to check when our coffee starts melting:

```maude
Maude> (find earliest cs2 =>*
   C:Configuration < coffee : WaterEntity | phase : melting > .)
```

The following table compares the results and execution time of the above
command using different error tolerances:

<table>
<thead>
<tr>
<th>Error Tol</th>
<th>Discrete Jump</th>
<th>CPU Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^{-3}$</td>
<td>11.3135531852</td>
<td>0.385197647</td>
</tr>
<tr>
<td>$10^{-4}$</td>
<td>15.9889535171</td>
<td>4.501970120</td>
</tr>
<tr>
<td>$10^{-5}$</td>
<td>11.6013479838</td>
<td>0.6425970213</td>
</tr>
<tr>
<td>$10^{-6}$</td>
<td>11.038434197</td>
<td>0.1414667933</td>
</tr>
</tbody>
</table>

The results show an expected correlation between the error tolerance and the execution time. However, the results do not show the 'gradual' changes of the discrete jump, as we may expect. The coffee is supposed starting the melting process at temperature 0°C. Thus we expect that as the error tolerance decreases, the approximate value of the coffee temperature will be closer to zero. But the results above cannot show our expectation. The following table shows the corresponding traces to the jump points from the simulation using different tolerances:

<table>
<thead>
<tr>
<th>Time point</th>
<th>$t_{j}$</th>
<th>$t_{j-1}$</th>
<th>$t_{j-2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^{-3}$</td>
<td>-0.0569967424</td>
<td>-0.051039563</td>
<td>-0.051039563</td>
</tr>
<tr>
<td>$10^{-4}$</td>
<td>-0.0510841430</td>
<td>-0.0510841430</td>
<td>-0.0510841430</td>
</tr>
<tr>
<td>$10^{-5}$</td>
<td>0.385197647</td>
<td>4.501970120</td>
<td>0.6425970213</td>
</tr>
<tr>
<td>$10^{-6}$</td>
<td>0.385197647</td>
<td>4.501970120</td>
<td>0.6425970213</td>
</tr>
</tbody>
</table>

The results shows that the changes of the coffee temperature value from one time point to another correlate to the error tolerances. However, the execution of a discrete event depends on the execution strategy for hybrid behaviors. For the implementation presented in this paper, the check of the occurrence of a discrete event is performed before the time step is taken, but a discrete event that should occur between time $t_n$ and $t_{n+1}$ is executed at $t_{n+1}$.

7 Concluding Remarks

In this paper we describe how the adaptive-step-size technique based on the Runge-Kutta-Fehlberg 4/5 method can be adapted to an effort-flow-based modeling of interacting physical systems, and how the methods can be implemented in Real-Time Maude. We have compared the precision and execution times for some thermal systems, and showed that decreasing the error tolerances increases both the accuracy of the approximation of the continuous behavior and the computational effort. We also found a weakness in our execution strategy of the hybrid behavior when using adaptive step size techniques.

Making these methods, and a modeling framework, available within the Real-Time Maude rewriting logic tool should make it a suitable candidate for the object-based formal modeling, simulation, and model checking of advanced hybrid systems due to the tool’s expressiveness, support for concurrent objects, user-definable data types, different communication models, etc. In future work, this should be validated this on more advanced systems.

Acknowledgment. We thank Lawrence F. Shampine for helpful discussions.
References


On-The-Fly Path Reduction

Sebastian Biallas\textsuperscript{1} Jörg Brauer\textsuperscript{1} Dominique Gückel\textsuperscript{2}
Stefan Kowalewski\textsuperscript{1}

\textit{Embedded Software Laboratory}
\textit{RWTH Aachen University}
\textit{Aachen, Germany}

Abstract
Path reduction is a well-known technique to alleviate the state-explosion problem incurred by explicit-state model checking, its key idea being to store states only at predetermined breaking points. This paper presents an adaptation of this technique which detects breaking points on-the-fly during state-space generation. This method is especially suitable for the detection of breaking points in systems where static analyses yield coarse over-approximations. We evaluate the effectiveness of this technique by applying it to binary code verification.

Keywords: verification, model checking, state explosion, path reduction

1 Introduction
Despite a significant amount of research on abstractions, state explosion is still a major obstacle for the applicability of (explicit-state) software model checking to real-world applications [5]. One such abstraction for CTL model checking is the so-called path reduction [18]. The key idea of path reduction is to collapse single-successor chains in the state space if intermediate states cannot influence the validity of a specification. This means that states are only stored when visiting program locations that cause a branching in the state space or influence the validity of the CTL specification. These program locations are called \textit{breaking points}. For instance, a program statement that alters the value of a variable used in an atomic proposition or reads a nondeterministic value, which causes a branching in the state space, is called a breaking point.

\textsuperscript{1} Email: \{lastname\}@embedded.rwth-aachen.de
\textsuperscript{2} Email: gueckel@embedded.rwth-aachen.de
Storing states only at these specific locations reduces the memory footprint of the state space, possibly at the cost of increased runtime.

In their seminal paper, Yorav and Grumberg [18] have described the detection of breaking points using static analysis for the model checker MUR\(\phi\). Due to the rather limited nature of the input language used in MUR\(\phi\), breaking points can be determined accurately for this specific tool. However, this is not always so. A domain where this approach may lead to coarse over-approximations is binary code verification for embedded systems [13,14]. This has different reasons, for example:

- Programs for low-level platforms are frequently interrupt-driven. In this case, states have to be stored at any program location where interrupts may fire because the execution of an interrupt handler is optional, and thus causes a split in the state space.
- Nondeterminism is often introduced through the hardware itself. Reading the value of the same register, say, an input port, may lead to either deterministic or nondeterministic values, depending on the exact hardware configuration. Unfortunately, no static analysis techniques are known that can infer the state of the hardware as precisely as required.
- To guarantee termination of the model-checking process, states need to be stored in possibly nonterminating loops (for fixed-point detection). Thus, at least one location in each loop has to be breaking. Despite the advances in termination proofs for high-level programs [7], these techniques are not yet applicable to low-level code.

Consequently, path reduction for binary code model checking based on static analysis does not reach the effectiveness known from other domains [14, Sect. 6.2]. Binary code model checkers, however, typically generate state spaces using dedicated simulators of the target microcontroller. The exact configuration of the microcontroller is thus known during state-space building. For example, when simulating an indirect store instruction, the concrete target address of this instruction is known, and breaking-point detection does not have to rely on conservative over-approximations.

Our main contribution in this paper is a new technique called on-the-fly path reduction, which performs state-space reductions dynamically while state spaces are built (see Sect. 2). This technique is novel in that it performs tasks such as detecting fixed points while states are generated. We also detail how to expand counterexamples obtained with path reduction to concrete counterexamples [3,6] (see Sect. 3). To evaluate the effectiveness of on-the-fly path reduction, we compare its performance to results obtained using static detection of breaking points in Sect. 4. Finally, Sect. 5 presents related work and Sect. 6 concludes the paper.
2 Reducing Paths On-the-fly

We implemented the path reduction by static analysis (SPR) and the new on-the-fly path reduction (OPR) for the model checker [mc]SQUARE [13] which we used for our case studies. This section details the motivation and implementation of the algorithms.

2.1 Preliminaries

[mc]SQUARE uses an on-the-fly model checking algorithm according to [8]. This means the state space is generated on-the-fly: A state corresponds to a configuration of the microcontroller and if the model checker needs to follow successor states that are not created yet, the simulator is used to create them on demand. A path reduction algorithm needs to cope with the on-the-fly generation of states. That is, instead of returning the direct successors by executing a single instruction, the simulator should follow a path of subsequent instructions returning the next state that should show up in the reduced state space. What qualifies a state as the next state (and hence determines the reduced state space) needs to be selected in a way that the reduction is indistinguishable by $\text{CTL}_X$ logic. Here, $\text{CTL}_X$ denotes the logic $\text{CTL}$ without the next time operator $X$, which cannot be used for reduced state spaces. We call states which match such a criterion and thus are used as the new successor states breaking. A path $(a, b_1, b_2, \ldots, b_n, c)$ where at most $a$ and $c$ are breaking is called elementary path. So each transition in the reduced state space corresponds to an elementary path in the original state space.

As an example, Fig. 1 shows a state space with 14 states labelled $a$ to $n$. The corresponding reduced state space is shown in Fig. 2. Although only the four breaking states $a$, $c$, $f$ and $n$ remain, the state space is stuttering bisimilar [2,16].
to the original state space. This is achieved by merging elementary paths like \((c, e, i, n)\), where the intermediate states have only one successor, into a single transition \((c, n)\). Note that such a pruning of states is also possible for loops like \((c, d, h, m, l, c)\) which is merged into the transition \((c, c)\). Note also that state \(b\) does not show up in the reduced state space although it has two predecessors. The loop \((f, j, g, k, f)\), however, requires special treatment, since none of its states have multiple successors.

We will now evaluate the criteria for a state to be breaking, i.e., to show up in the reduced state space. For SPR, this criterion solely depend on the program counter. Here, breaking program locations (breaking points) are found using static analysis. Afterwards, for each state with a program counter matching these values the path pruning algorithm stops during state space generation. The static analysis \([18,14]\) selects each location as a breaking point,

- where a nondeterministic interrupt can be triggered, or
- where the corresponding instruction performs some nondeterministic operation as reading from an input register, or
- where the corresponding instruction alters the value of a variable (memory location) used as an atomic proposition in the \(\text{CTL}\) formula, or
- which is target of a jump (or other control transfer) instruction.

The first two conditions make locations breaking where the corresponding state has more than one successor in the state space. The third condition guarantees that all changes of variables used in the formula are visible to the model checker. Finally, the fourth condition ensures the existence of a breaking point in every loop in the program, such that every loop in the state space has at least one breaking state for fix-point detection.

As mentioned in the introduction, using static analysis to determine the breaking points usually yields a coarse over-approximation and thus assumes more locations breaking than necessary. For example, a read-operation on an I/O port which is configured for input returns a nondeterministic value, which in turn leads to a branching in the state space. The corresponding program location is thus marked as breaking. However, an I/O port may also be configured for output, which means that it stores a deterministic value. In this case, the corresponding program location does not need to be breaking. Since static analysis fails to accurately capture such bit-level dependencies of the hardware, it computes overly pessimistic results. Additionally, the fourth condition makes it impossible to prune long running loops of the program into a single transition. In the next subsection, we will focus on the new OPR that uses criteria which can be evaluated on-the-fly to determine whether a state is breaking and thus can handle these issues.
2.2 On-the-fly Path Reduction

The general idea of OPR is that we decide for each state (instead of each program location) whether it is breaking by evaluating its local neighborhood. This decision is made during state space generation, and we therefore need conditions that can be checked on-the-fly: While generating states along an elementary path, the OPR assumes a state as breaking if

- it has more than one successor, or
- the truth value of a atomic proposition changes after the transition to its (sole) successor, or
- it was already visited in this elementary path.

The first condition assures that states where a nondeterministic decision has to be taken (such as the execution of an interrupt or a read from a hardware register) show up in the state space. To maintain visibility of all changes to the model checker, the second condition assures that at most one transition might influence the formula in each elementary path. The last criterion is needed to guarantee termination and will be studied in detail later on. First, we formally describe the algorithm for the OPR successor state generation:

**Algorithm 1 Generate successors of state in reduced state space**

**Input:** sourceState

**Output:** successors of sourceState in the reduced state space

1: successors ← createDirectSuccessors(sourceState)
2: resultSuccessors ← {} 
3: for all state in successors do
4:    current ← state
5:    visited ← {} 
6:    repeat 
7:       visited ← visited ∪ {current} 
8:       nextStates ← createDirectSuccessors(currentState) 
9:       next ∈ nextStates 
10:      breaking ← |nextStates| ≠ 1 or atomics(current) ≠ atomics(next) 
11:      if not breaking then 
12:         current ← next 
13:      end if 
14:    until breaking or current ∈ visited 
15:   resultSuccessors ← resultSuccessors ∪ {current} 
16: end for 
17: return resultSuccessors

For each direct successor, the inner loop (lines 6–14) of the algorithm follows its elementary path (line 12) until a breaking state is found. The first
two breaking conditions are checked in line 10. In the next section it will be described how to implement the loop detection in line 7 and line 14, which is used to meet the third condition for breaking states.

2.3 Loop Detection

We describe three different criteria for the detection of loops in elementary paths, which can be checked one-the-fly:

- Stop if the same program counter is encountered twice.
- Stop if the same state is encountered twice.
- Stop if the same hash code of a state is encountered twice.

The first criterion is inspired by SPR and guarantees that every loop (in the program and thus in the state space) contains at least one breaking instruction. The drawback is that each loop without nondeterministic control flow (for example a memory copy or initialization), will create at least one state in the state space for each iteration.

To prune such loops, the second criterion takes the whole microcontroller configuration into account, that is, all states along elementary paths are temporarily stored and a state is assumed breaking once is already in this list. Since states have finite size, this is guaranteed to terminate for all loops. This detangles loops in the control flow from loops in the state space and hence allows for representing loops as single transitions.

The third criterion is an improvement of the second criterion. It takes just the 64 bit hash code of the raw state data as a criterion for detecting already encountered states. This is faster and less memory intense, since only the hashes of all intermediate states have to be stored while simulating along an elementary path. As our case study will show, the third criterion offers the high accuracy while being a very fast possibility to detect a loop in the state space.

To summarize, the advantages of the OPR introduced so far are:

- There is no need for static analysis, which yields more accurate results.
- Our algorithm is independent of the microcontroller simulator used for the state generation. For SPR, on the other hand, detailed knowledge of the microcontroller is necessary to detect breaking states.
- It is possible to prune program loops with many iterations into single transitions.

While this section dealt with reducing paths to alleviate the state explosion, we will discuss how to re-expand paths to create meaningful counterexamples in the next section.
Counterexamples/Witnesses are paths (possibly with loops) in the state space, showing how some undesired property is reached or some desired property is never reached. Counterexamples provide crucial information to help understanding why formulae are valid or violated \cite{b1}. A counterexample for the formula $\text{AG } x \neq 5$, for instance, would show a path (and thus all nondeterministic inputs) that leads to a state where $x$ equals 5. A counterexample for the formula $\text{AF } x = 6$, on the other hand, would show a path into a loop, such that $x = 6$ is never valid.

In a reduced state space, counterexamples are less useful. To illustrate, consider Fig. 3 where the counterexample trace $(a, c, d)$ is shown; the formula is false in state $d$. States omitted by the path reduction ($b$ and $b'$) are shown as dotted circles. To understand such a counterexample trace, it is crucial to know which nondeterministic decision has been taken for the $(a, b)$ transition. Unfortunately, this information is not readily available from the $(a, c)$ transition visible in the counterexample trace: State $c$ might be too far away to distinguish the $(a, b)$ transition from the $(a, b')$ transition without manual investigation.

To remedy this problem, we implemented means to reverse the effect of path reduction on given counterexample traces in [MC]SQUARE by re-expanding all reduced paths. Such re-expanded counterexamples are then identical to their corresponding traces in the original state space.

This is achieved in two steps. In the first step, all states omitted by the path reduction are recreated using the simulator. For states with only one successor, it is sufficient to create the direct successors until the next state on the reduced path is reached. For states with more than one successor, however, we have to find out which transition actually belongs to the counterexample. This corresponds to the decision between the $(a, b)$ and $(a, b')$ transitions in Fig. 3. To decide which of these nondeterministic transitions belongs to the counterexample, a breadth-first search for the target state $c$ is started at node $a$. The search terminates when state $c$ is found. States with more than one successor do not need to be followed because they are breaking and thus part of the reduced state space. The path leading to state $c$ is then added to the counterexample making the nondeterministic decision shallow.
In the second step, states are dropped at the end of elementary paths where the violation of the formula manifests itself in the first transition but is noticed at the end of this path. Such a situation is depicted in Fig. 4 (upper part). Let us assume that after the \((a, b)\) transition the formula is violated, i.e. the formula is true in state \(a\) but false in states \(b\) to \(c\) (recall that on each elementary path only the first transition might influence the formula). Since only \(a\) and \(c\) are stored in the reduced state space while the states in between are omitted, the violation of the formula is detected in state \(c\), yielding a counterexample ending in \(c\). It is desirable to have shortest counterexamples (which highlight the first instruction invalidating a formula), and we thus drop the states \(b\) to \(c\). Formally, a path that ends in \((\ldots, a, c)\) in the reduced counterexample is transformed into \((\ldots, a, b)\), where \(b\) is the direct successor of \(a\) in the original state space. This is shown in the lower part of Fig. 4.

The remaining disadvantage of path reduction is the loss of the \(X\) operator for \(\text{CTL}\). As we perform model checking on the level of machine instructions, the \(X\) operator is not of practical relevance anyway.

4 Case Studies

This section describes different experiments to examine the impact of OPR with respect to different evaluation criteria: the effects of different approaches for loop detection (see Sect. 4.1), a comparison to path reduction based on static analysis (see Sect. 4.2), and the effects of \(\text{CTL}\) specifications on the generated state spaces (see Sect. 4.3). All experiments were run on a SUN Fire X4600 M2 server equipped with eight AMD Opteron dual-core processors and 256 GiB of RAM. However, only a single processor was used in order to obtain unbiased results.

4.1 Variants of On-the-fly Path Reduction

The first case study focuses on the effects of different loop-detection criteria (cp. Sect. 2.3), which determine the termination of a path compression step. In order to obtain realistic results, the program to be verified has to execute several loop iterations, ideally with as little overlapping of the iterations as possible. A program called vector from our benchmark set satisfies this requirement. This program continuously reads inputs from the environment in a nonterminating loop. The values are then interpreted as integer vectors and used for different typical vector operations.

For each criterion, the model checker had to generate the entire state space of the program. The results of these runs are shown in Tab. 1. The reference values without any abstraction are shown in Tab. 2, in the entry for vector.

As expected, the “same pc” criterion results in the largest state space
of the three criteria. Compared to the state space size without applying path reduction, this still amounts to a reduction by 96.83%. Comparing states for identity in a byte-wise fashion results in a much smaller state space, which is unsurprising. The interesting aspect of this experiment is thus to determine whether the effort for identity checking has significant advantages over the simpler checking for hash collisions. Regarding the number of stored states, there is no difference between hash collision detection and state identity detection for this particular program. This means that no two different states are mapped to the same hash values. The time required when checking for state identity, however, was slightly higher than for detecting hash collisions.

Judging from these results, we conclude that the checking for hash collisions is an adequate compromise between runtime and memory consumption concerns. Hence, in the following case studies, we use hash collision checking as the default criterion in OPR.

4.2 Comparison to Other Abstraction Techniques

For the second set of experiments, we have used [MC]SQUARE to generate state spaces on different levels of abstraction: (i) no abstraction, (ii) SPR, and (iii) OPR. For a thorough evaluation, we present experimental results for five different microcontroller programs. The results of the different runs are shown in Tab. 2.

The first program is called lightSwitch and models a reactive electrical switch based on a state machine. The program uses two hardware timers, but no interrupts. Model checking this extremely simple program with SPR results in a reduction of the state space size by 73.88%, but it increases the number of states created by 156.69%. In comparison, OPR reduces the state space even further by 88.43%, relative to the original results. The lower number of states stored in the state space also influences the number of states that have to be recreated during model checking. Hence, the number of states created increased by approx. 300%. For this small program, neither technique had a noticeable effect on the memory consumption or the runtime. The memory footprint in this case is largely influenced by the initial sizes of the hash tables used for storing state spaces.
The second program, called `plant`, controls a fictive chemical plant. Consisting of 225 lines of code, it is slightly longer than `light_switch` (162 lines). Two interrupts and one timer are used in `plant`. SPR has a significant effect, lowering the number of states stored by 93.44%. Again, OPR achieves better results by reducing the number of states by as much as 97.54%. The increase in the number of created states amounts to 13.96% for SPR, and 22.92% for OPR. Hence, compared to the according numbers for `light_switch`, the increase is rather modest. This means that either the model checker has to revisit fewer states, or that the length of the compressed paths is shorter. Memory consumption using any of the path reduction techniques dropped to the vicinity of the initial size of the hash tables.

In the next program, `reentrance`, a 16-bit integer variable is accessed concurrently in the main process and in an interrupt handler. As the ATmega has an 8-bit architecture, such accesses are non-atomic, thus leading to race conditions. The reduction in states stored achieved by SPR is 93.84%. OPR reduces the state space further by halving the number of remaining states, yielding a reduction of 96.92%. The increase in runtime due to revisits was 10.85% for SPR and 11.94% for OPR.

An automotive application was used as the fourth program. The program called `window_lift` implements the functionality of an electric window lift for cars. It is based on a state machine, which generates outputs depending on its current state. To fulfill its task, it uses three interrupts and one 16-bit timer. The state space for this program without any abstraction is rather large compared to the previous programs, consisting of more than 2.3 million states. SPR decreases this value by 92.2%, while OPR decreases it by 94.72%. The runtime required for model checking is approximately doubled, with a slight advantage for SPR. Memory consumption was reduced by 89.89% in case of SPR and by 92.05% in case of OPR.

Our fifth case study used the aforedescribed program `vector` (cp. Sect. 4.1). SPR resulted in a decrease of the number of states stored by 89.76%. OPR outperformed this by three orders of magnitude, reducing the number of states stored by 99.99%. This effectively reduced the amount of memory required for the state space from more than 11.5 GB (no abstraction) to 23.7 MB (OPR). SPR results in a reduction of 87.96% to approximately 1.4 GB, which can still render the program manageable for model checking on desktop computers. The time required when using SPR increased only by 1.55%, whereas OPR results in an increase of 533%.

Considering that far less states are stored using OPR, this increase is actually surprisingly low. An explanation for the large difference between the two approaches to path reduction is the different handling of loops. In order to guarantee termination of the state space generation in the presence of program loops, SPR has to assume at least one position in the loop to be
breaking (cp. [13]). In our implementation, this position is indicated by the head of the loop. Thus, on each revisit of the program counter position of the head, SPR terminates the current chain and stores a state. OPR, on the other hand, does not have to store at the head of a loop (in fact, it is unaware of the existence of the program loop), unless it uses the same program counter approach for termination detection. Hence, OPR compresses loop iterations far more efficiently.

<table>
<thead>
<tr>
<th>Program</th>
<th>Options used</th>
<th>States stored</th>
<th>States created</th>
<th>Size [MB]</th>
<th>Time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>light_switch</td>
<td>none</td>
<td>4,268</td>
<td>6,296</td>
<td>21.6</td>
<td>0.42</td>
</tr>
<tr>
<td>162 lines</td>
<td>SPR</td>
<td>1,115</td>
<td>16,175</td>
<td>20.9</td>
<td>0.79</td>
</tr>
<tr>
<td></td>
<td>OPR</td>
<td>494</td>
<td>25,223</td>
<td>20.7</td>
<td>0.88</td>
</tr>
<tr>
<td>plant</td>
<td>none</td>
<td>130,524</td>
<td>135,949</td>
<td>52.28</td>
<td>2.33</td>
</tr>
<tr>
<td>225 lines</td>
<td>SPR</td>
<td>8,552</td>
<td>154,921</td>
<td>22.6</td>
<td>2.81</td>
</tr>
<tr>
<td></td>
<td>OPR</td>
<td>3,205</td>
<td>167,114</td>
<td>21.4</td>
<td>3.03</td>
</tr>
<tr>
<td>reentrance</td>
<td>none</td>
<td>107,649</td>
<td>110,961</td>
<td>44.4</td>
<td>2.60</td>
</tr>
<tr>
<td>147 lines</td>
<td>SPR</td>
<td>6,628</td>
<td>123,003</td>
<td>22.0</td>
<td>2.08</td>
</tr>
<tr>
<td></td>
<td>OPR</td>
<td>3,312</td>
<td>124,207</td>
<td>21.3</td>
<td>1.40</td>
</tr>
<tr>
<td>window_lift</td>
<td>none</td>
<td>2,342,564</td>
<td>2,589,665</td>
<td>633.9</td>
<td>47.59</td>
</tr>
<tr>
<td>289 lines</td>
<td>SPR</td>
<td>182,709</td>
<td>3,818,060</td>
<td>64.1</td>
<td>57.78</td>
</tr>
<tr>
<td></td>
<td>OPR</td>
<td>123,585</td>
<td>4,123,385</td>
<td>50.4</td>
<td>59.49</td>
</tr>
<tr>
<td>vector</td>
<td>none</td>
<td>47,477,797</td>
<td>48,419,003</td>
<td>11,508,422</td>
<td>772</td>
</tr>
<tr>
<td>930 lines</td>
<td>SPR</td>
<td>4,860,321</td>
<td>55,584,435</td>
<td>1,385,914</td>
<td>784</td>
</tr>
<tr>
<td></td>
<td>OPR</td>
<td>4,656</td>
<td>496,768,475</td>
<td>23.7</td>
<td>4,891</td>
</tr>
</tbody>
</table>

Table 2
Effects of different path reduction techniques on five microcontroller programs

4.3 Influence of Formulae

So far, we examined the effect of OPR when checking the formula $AG \ TT$, which is true in every state. In this section, we will now evaluate the effects of OPR when checking actual formulae whose validity depend on variables. Since
path reduction needs to store states when a transition influences the formula, we expect an increase in the size of the state spaces.

For the first examinations, we decided to use the window\_lift program. The results are shown in Tab. 3. As described in Sect. 4.2, the program models an automotive electrical window lift, and is based on a state machine. The state machine is implemented using a global integer variable called mode, which is expected to assume only the values 0 to 6 at any time during execution. Hence, our first test was to check this using the formula

\[(1) \ AG (\text{mode} \geq 0 \land \text{mode} \leq 6),\]

which could be verified after 54.83s.

The second test was to verify whether the sequence of states assumed by mode satisfies a certain property. Whenever a sensor reports that there is an object stuck in the window (mode = 5), the window lift is expected to open completely (mode = 6) before allowing normal operation again (mode = 0), which can be specified by the formula

\[(2) \ AG (\text{mode} = 5 \Rightarrow \neg E (\text{mode} = 5 \land \neg \text{mode} = 6) \ U (\neg \text{mode} = 5 \land \neg \text{mode} = 6)).\]

The program window\_lift contains a subtle error which prevents this property from being satisfied. The error is based on the simultaneous occurrence of two interrupts, which allows mode to skip the value 6. [MC]SQUARE correctly discovered this error and created a counterexample consisting of 912 states.

Our second test program for this case study was plant, also described in detail in Sect. 4.2. The first property to verify was, similar to window\_lift, to verify that a global variable satisfies certain constraints, which can be specified by the formula

\[(3) \ AG (\text{tank} \geq 0 \land \text{tank} \leq 4),\]

which could also be verified by [MC]SQUARE. The second property was then to ensure the correct behavior of the plant in case of an emergency. For this purpose, [MC]SQUARE had to check the formula

\[(4) \ AG (\text{PORTA} = 0x20 \Rightarrow AG \text{PORTA} = 0x20) \land EF (\text{PORTA} = 0x20),\]

which resulted in a counterexample with 1,643 states after expansion.

The truth value of all four formulae was the same compared to model checking without OPR. Since formulae (2) and (4) were violated, the model checker could prematurely stop the state space generation. Thus, the time for model checking and the size of the state space is not comparable to the
Table 3
Influence of formulae on state space sizes

<table>
<thead>
<tr>
<th>Program</th>
<th>Formula</th>
<th>States stored</th>
<th>States created</th>
<th>Size [MB]</th>
<th>Time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>window_lif</td>
<td>(1)</td>
<td>226,452</td>
<td>3,792,507</td>
<td>78.33</td>
<td>54.83</td>
</tr>
<tr>
<td></td>
<td>(2)</td>
<td>11,665</td>
<td>207,964</td>
<td>24.52</td>
<td>3.64</td>
</tr>
<tr>
<td>plant</td>
<td>(3)</td>
<td>3,678</td>
<td>167,114</td>
<td>21.54</td>
<td>2.32</td>
</tr>
<tr>
<td></td>
<td>(4)</td>
<td>77</td>
<td>1,839</td>
<td>20.67</td>
<td>0.6</td>
</tr>
</tbody>
</table>

other case studies. For the verification of formula (1), we have an increase of 83.24% of the state space size, while the time decreased slightly, due to the smaller number of revisits. For formula (3), the increase of the state space size is negligible.

5 Related Work

Path reduction based on static analysis for the model checker MuRϕ was first described by Yorav and Grumberg [18]. This technique is used in a similar fashion in the SPIN model checker [9] using a static analysis for its input language PROMELA. SPIN uses an intraprocedural static analysis (using inlining), and compared to binary code, PROMELA is much simpler since (1) communication between concurrent processes can only be performed using distinguished statements and (2) it does not contain indirect control statements.

Later, Quiros [12] has adapted the approach of Yorav and Grumberg to a bytecode language used in a virtual machine. This bytecode language is similar to a parallel while language. This means that function calls are handled using inlining, communication is performed at certain program locations, and indirect control is not supported. Hence, SPR turns out to be effective for this domain. Our own prior work [14] adapts these earlier approaches to the domain of binary code verification by introducing tailored static analyses and revising breaking conditions for binary code.

Behrmann et al. [1] implemented a similar technique for the model checker UPPAAL, which focuses on timed automata. Their approach is similar to our implementation of SPR: they decided for a static analysis of the control structure of the automata in order to obtain a so-called covering set of edges. This set is used in order to guarantee termination in case of loops in the state space. States that are targets of edges in the covering set have to be stored, which exactly corresponds to the breaking property used in SPR. As we have illustrated, this property can prove a significant disadvantage of SPR.
in the presence of long-running but terminating loops. Our contribution, OPR, can handle such loops without storing states in each iteration. Pelanek [11] conducted a survey of on-the-fly state space reduction techniques. He subsumes techniques preserving stutter equivalence under the term of transition merging. His survey, however, focuses on high-level representations.

Recently, Yang et al. [17] introduced dynamic path reduction for bounded model checking of sequential programs. However, even though their technique is named similarly, its purpose is to prune out infeasible executions paths introduced by nondeterministic conditionals, and thus, must not be confused with path reduction in the sense used in this paper. Their algorithm computes weakest preconditions and unsatisfiable cores using SMT solving. Thus, both their approach and their goals are fundamentally different from our work.

6 Concluding Discussion

6.1 Conclusion

This paper describes a new technique for dynamic path reduction and shows the predominance of this method over approaches based on static analysis for the specific application of binary code model checking. Further, it shows how counterexamples generated using this abstraction technique can be expanded in order to ease their comprehensibility. In terms of effectiveness, the OPR approach allows for formidable state space reductions, comparing it to static path reduction techniques. The smaller memory footprint, however, may lead to higher runtimes. Thus, OPR provides a technique that allows to trade runtime for memory.

6.2 Future Work

Another abstraction technique discussed by Yorav and Grumberg [18] is dead variable reduction (DVR), the key idea being to reset variables whose value is not going to be read in any subsequent program execution. However, DVR for binary code suffers particularly from the presence of indirect reads in binary code, where the source memory locations can often not be determined accurately using static analysis [14, Sect. 6.1]. Consequently, it will be of interest to evaluate if state space reductions as significant as those obtained through OPR can be achieved using an on-the-fly adaptation of DVR [10,15].

Acknowledgement

This work was supported by the DFG Cluster of Excellence on Ultra-high Speed Information and Communication (UMIC), German Research Foundation grant DFG EXC 89. Further, the work of Sebastian Biallas was supported
by the DFG. The work of Jörg Brauer and Dominique Gückel was, in part, supported by the DFG Research Training Group 1298 Algorithmic Synthesis of Reactive and Discrete-Continuous Systems (AlgoSyn). We thank Bastian Schlich for sharing his thoughts on the ideas described in this paper.

References

A Parallel Approach to Concolic Testing with Low-cost Synchronization

Xiao Yu, Shuai Sun, Geguang Pu and Siyuan Jiang, Zheng Wang

Software Engineering Institute, East China Normal University, Shanghai, China

Abstract
This paper presents a practical approach to parallelize the test data generation algorithm by which computing resources can be fully used. The test data generation approach that we are using is based on the dynamic symbolic execution (concolic testing). The basic idea of parallelizing the algorithm is to distribute analysis processes of different paths to different computing units. Although a centralized scheduler with several sub processes can directly achieve the goal of parallelism, it may cause global idle time when parallel processes frequently end at the same time. In our approach, a runtime deterministic scheduler is introduced to reduce the potential global idle time. Our experiments show some notable results when using a proper scheduling function. Compared with the sequential concolic testing, our approach can save nearly 70% computing time in some cases on a system with eight CPU cores from our experiments.

Keywords: Parallel Algorithm, Automatic Test Generation, Symbolic Execution

1 Introduction and Motivation
Software testing is the most popular methodology to find bugs. Some tools pay attention to automate the generation of test cases. Pex [17] is an automated unit testing tool that can automatically generate test inputs for program units on .NET platform. CUTE [16] is another tool that can analyze and generate test inputs for C program units. The techniques of automated test case generation tools have experienced a long time of development. Symbolic execution [12], initially introduced in 1970s, uses symbolic values to replace initial concrete inputs while the target program is being executed. By using a constraint solver, it is possible to obtain a set of precise concrete inputs from a set of path constraints to strictly satisfy that path. The basic idea of symbolic execution is novel and useful, which inspires following works on test-
Yu

ing [9, 14, 19, 18, 21, 22]. Recently, the concept of concolic testing [11, 16, 8] has been proposed. The concolic testing is a variant of symbolic execution with the advantage that concrete program states are also stored to guide the process of symbolic execution. Constraints of paths are incrementally collected while some of those are replaced by concrete states. It is an enhanced dynamic symbolic execution [13] technique. The constraints collected along one path can be simplified with this technique. It is a novel way to improve the usability and the performance of pure symbolic execution. Based on this improvement, many other techniques [15, 7, 6, 10, 5] have been proposed to further improve the usability of concolic testing.

Test case generation techniques face with the path-state explosion problem since program paths increase radically with the growth of program scale. For example, the following code fragment has only 35 lines, it tries to find the word ‘web’ and the word ‘ebay’ in a given string with 5 characters. It uses a typical method of state machine. Every loop iteration contains 6 first-level branches, each of which contains 3 or 4 second-level branches. Because the first-level branches are not overlapped with each other, in all, there are 23 branch conditions in one single loop iteration. Considering the looping condition, the feasible paths can be deep and the number of feasible paths can be enormous. We tested this function on an Intel Core i7 platform. The result showed that the sequential concolic test cost about 84.6 seconds to complete the searching of about 3400 feasible paths.

```c
void foo(char c[]) {
    int state = 0, idx = 0;
    while (c[idx] > 0) {
        if (state == 0) {
            if (c[idx] == 'w') state = 1;
            else if (c[idx] == 'e') state = 2;
            else state = 0;
        }
        else if (state == 1) {
            if (c[idx] == 'w') state = 1;
            else if (c[idx] == 'e') state = 3;
            else state = 0;
        }
        else if (state == 2) {
            if (c[idx] == 'w') state = 1;
            else if (c[idx] == 'e') state = 2;
            else if (c[idx] == 'b') state = 4;
            else state = 0;
        }
        else if (state == 3) {
```
Yu

```c
if (c[idx] == 'w')    state = 1;
else if (c[idx] == 'e')  state = 2;
else if (c[idx] == 'b')  state = 6;
else    state = 0;
}
else if (state == 4) {
    if (c[idx] == 'w')    state = 1;
    else if (c[idx] == 'e')  state = 2;
    else if (c[idx] == 'a')  state = 5;
    else    state = 0;
}
else if (state == 5) {
    if (c[idx] == 'w')    state = 1;
    else if (c[idx] == 'e')  state = 2;
    else if (c[idx] == 'y')  state = 7;
    else    state = 0;
}
else if (state > 5) {
    printf("\nHit\n");
    break;
}
idx++;
}
```
pattern that consists of a centralized scheduler and several sub processes can
directly achieve the goal of parallelism, it may cause global idle time when
parallel processes frequently end simultaneously. Rather than building the
centralized scheduler, we introduce a deterministic scheduler on each working
unit, which considerably reduces the synchronization time cost. The detail of
the method will be expanded in the following sections. The main contributions
of our work are:

- Enhancing parallel capability to traditional concolic testing
- Introducing the concept of runtime deterministic scheduler in order to re-
duce synchronization time
- Implementing a parallel concolic testing framework with positive exper-
imental results.

In this paper, Section 2 describes the basis of concolic testing and details
about how to achieve parallelism. Section 3 shows the results of experiments
and the discussion. The last section gives the conclusion.

2 Parallel Approach

The parallel approach of test case generation is introduced in this section.

2.1 Background: Concolic Testing

Concolic testing [16], which is also referred as the dynamic symbolic execution
testing [13], is a variant symbolic execution. The idea is simple but powerful.
It combines the symbolic execution and concrete one together to generate
test inputs dynamically. By monitoring the path execution of the test unit,
the branch conditions along the execution path are collected. In order to
explore new paths, conditions of the branches are collected as constraints,
using symbols related to the input variables. The symbolic constraint solvers
then processed the constraints, returning the solution, which forms the input
of the next execution iteration. The above process is iterated until no new
path can be generated, which indicates the path tree of this test unit is fully
explored. The whole process is the process of test case generation, while each
iteration contains one path execution, constraints processing and constraints
solving.

During the concolic testing of one test unit, two structures are always
maintained between two successive iteration. One structure is the global path
decision tree \( T \), which contains the path recorded from every iteration and
shares path information between each iteration. The other is a sequence of
values \( M \) which provides concrete values for the sequence of input variables
\( I \). The whole process of sequential concolic testing is within a main iteration.
In each iteration, it substitutes the input $I$ with the value $M$ into the target program $P$, and starts to execute $P$ concretely and symbolically. The result from the execution of target program can be treated as a triple which consists of the execution trace $t$, the decision path $p$ and the path feasibility, which is either feasible or infeasible indicating whether the execution of the target program goes through the expected path or the execution is aborted abnormally. The algorithm terminates when no more expected path prefix can be explored. Some technique details can be found in [20].

2.2 Parallel Model

This subsection presents the parallel algorithm of the concolic test case generation. We design a parallel model that makes the process of test case generation run concurrently. An interesting point in this parallel algorithm is that we divide the whole path space into different disjoint areas dynamically that can be managed and updated by different computing units. Thus, each computing unit can freely access and analyze the paths belonging to its own allocated area. This means the global synchronization is fundamentally removed among parallel computing units. This technique can further improve our performance of parallelized concolic testing. The removal of global synchronization is implemented by a runtime task scheduler which allows each computing unit safely updates its own data on a shared global decision tree.

2.2.1 Architecture

Figure 1 shows the basic architecture of our parallel model. There are three roles in the parallel model. There are Worker, WorkerStub and Coordinator, respectively. Instances of Worker perform actual concolic testing simultaneously on different computing units. Each of instances of Worker is managed by a corresponding instance of WorkerStub. The WorkerStub is the essential part of our parallel algorithm for the reduction of global synchronization. Instead of a centralized task scheduler, WorkerStub holds a runtime deterministic task scheduler, which will be explained later. Specifically, the WorkerStub takes charge of (1) starting one instance of Worker each iteration, (2) assigning path computing tasks to the Worker, (3) collecting feedback from the Worker, (4) exchanging computing tasks with other instances of WorkerStub, (5) maintaining a partial path decision tree and reporting the path decision tree it manages to the Coordinator.

The algorithm of Worker is shown in Algorithm 1. A Worker takes $(P, I, \text{choice, trace})$ assigned by WorkerStub as inputs where $P$ is the target program to be tested, $I$ is the sequence of input variables, $\text{choice}$ is the truth-value assignments for the expected path prefix and $\text{trace}$ is the program trace which relates to the expected path prefix. After solving constraints of $\text{choice}$ and
Worker \((P, I, \text{choice}, \text{path}, \text{trace})\)

Inputs:
\(P\) is the target program to be tested.
\(I\) is the sequence of input variables.
\text{choice} is the truth-value assignments for the expected path prefix.
\text{path} is one path which relates to the prefix \(c\) from the entire decision tree.
\text{trace} is the program trace which relates to the \text{path}.

Returns:
\((\text{feasible}, p, t, S_p)\) where \(p\) is the result path of \(P\),
\(t\) is the result trace and \(S_t\) is the set of all prefix paths related to \(p\).
\text{infeasible} when no solution satisfies the \text{choice} on \text{path}.

Let \(M\) be the constraint solution values corresponding to the input \(I\)
\(M := \langle \rangle\)
if \(\text{path} \neq \text{nil}\) then
\(c_1, c_2, ..., c_n\) be all non-leaf nodes of \text{path}
and \(C\) be the final whole sequence of constraints
for \(i = 1\) to \(n\) do
\(C := C \cdot \text{GetRealConstraint}(c_i, \text{trace}, I)\)
end for
\(M := \text{SolveConstraints}(C, \text{choice})\)
else
\(M := \text{GenerateRandomInput}(I)\)
end if
if \(M = \langle \rangle\) then
return \text{infeasible}
end if
\((t, p, s) := \text{ConcreteAndSymbolicExecution}(P, I, M)\)
if \(s = \text{infeasible}\) then
return \text{infeasible}
end if
\(S_p := \phi\)
\(c := \text{GetBranchChoice}(p)\)
while \(c \neq 0\) do
\(i_1, i_2, ..., i_n\) be the branch choices in \(c\)
\(\text{expected} := \text{FlipLastChoice}(c)\)
\(S_p := S_p \cup \{\text{expected}\}\)
\(c := c - i_n\)
end while
return \((\text{feasible}, p, t, S_p)\)

Algorithm 1. The algorithm on the worker
executing the target program with the solving result, the Worker returns (feasible, \(p, t, S_p\)) for a feasible path or infeasible for an infeasible path according to the current performance. If it is a feasible path, the complete executed path \(p\) (obviously the prefix of \(p\) is choice) and the corresponding program trace \(t\) will be sent back to WorkerStub in order to build a partial path decision tree. If it is an infeasible path, the original expected path prefix will be marked as infeasible. Besides, the \(p\)-related path prefix set \(S_p\) is also returned to WorkerStub to create new tasks. The set \(S_p\) is computed by negating every constraint assignment (around the call to FlipLastChoice in Algorithm 1) on the path \(p\).

When one WorkerStub has been started up, it begins to maintain an individual task list and a partial decision tree which will be built from the complete tasks in the individual task list. If the started WorkerStub is the first instance and the task list is empty, it will start an instance of Worker with empty inputs in order to get the first path data from the target program. Otherwise, the WorkerStub will wait for some tasks sent from other WorkerStubs to the individual task list and then run a series of Worker iteratively to compute tasks. When an instance of WorkerStub has received a set \(S_p\) from its worker, it firstly uses the local deterministic task scheduler to decide for a specific path prefix in \(S_p\) which WorkerStub should receive it as an individual task. After the sorting, every item in \(S_p\) with related program trace will be sent as individual task to corresponding WorkerStub told by the scheduler.

The Coordinator maintains a global view of the path decision tree by
Yu periodically collecting and merging partial trees from all instances of WorkerStub. It initially starts several instances of WorkerStub (the exact number of instances is decided by the number of processors installed on the target computer). It terminates the whole testing process when the global path decision tree is full.

To the generalized view of the parallel model (see Figure 2), the Workers with WorkerStubs are the parallelized units. The Worker only communicates with its own WorkerStub by which it receives computing tasks and sends results. The Coordinator mainly controls the termination. The key of low-cost synchronization is the Deterministic Task Scheduler that makes the task scheduling free from a global serialized task list. The shared tasks and tree are divided into a set of disjoint areas by the deterministic task scheduler, which will be explained in the following section.

<table>
<thead>
<tr>
<th>Parallel Concolic Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coordinator</td>
</tr>
<tr>
<td>WorkerStub</td>
</tr>
<tr>
<td>Worker</td>
</tr>
</tbody>
</table>

**Fig. 2. Processes**

### 2.2.2 Deterministic Task Scheduler

A naive task scheduler (Figure 3a) usually maintains a centralized task list to schedule tasks. When there are more than one free workers waiting tasks, the scheduler has to serialize the assignment of tasks to avoid data races among workers (e.g., avoiding two workers get the same task). The serialization means the workers have to wait in a queue, which leads to a waste of computing time. The underlying reason of the imperative serialization is the nondeterminism in the scheduling plan. It simply distributes tasks to any free worker. A task in the list can be accomplished by any workers.

In our parallel model introduced previously, dynamically generated paths can be scheduled among different computing units by a uniform deterministic
task scheduler. The deterministic task scheduler (Figure 3b) is designed to overcome disadvantages of the nondeterministic scheduler. As we can see in Figure 3b, the centralized structure along with its connections with the workers is eliminated. The workers connect directly with each other to exchange path record only when necessary (determined by the deterministic scheduler on each worker). The effect of the deterministic scheduler is that for each task generated by the testing process, the scheduler tells which worker should compute the task by a universal independent algorithm instead of randomly allowing some free worker to compute the task. Thus, the deterministic scheduler can be placed in each computing unit instead of a global one, which makes the serialized task list be separated to each computing unit. Finally, the global synchronization is eliminated.

Several observations should be kept to implement the deterministic task scheduler. The global path decision tree of a program is dynamically combined through concolic testing iterations. Each Worker iteration consumes only one task that consists of a path prefix which is corresponded to program history traces, and generates only one path from the prefix despite the path is feasible or infeasible. Each path along with its prefix on the tree is computed independently from others. All paths can be projected to a discrete space, which means they can be divided into partitions. By dividing and projecting paths and prefixes on the tree into the discrete space, the whole computation of the target program is naturally classified to several disjoint regions on the space. Every working unit takes charge of one region so that all working units have the knowledge of a specific task which working unit should take charge of.

Formally, the deterministic task scheduler is implemented by a function \((\mathcal{F} \circ \mathcal{H})(p)\) where the sub function \(\mathcal{H}\) takes its domain as all possible paths on the binary decision tree, and its image is a bounded range of positive integer to present the abstract path space. Function \(\mathcal{F}\) maps the image of \(\mathcal{H}\) to integers in the range \([0, \text{MAX\_UNIT\_CNT})\) to present regions on the space.
The function must ensure that on different computing units it should get the same result for the same path \( p \) (namely deterministic). The sub function \( \mathcal{H} \) can have many types of implementations. Different implementations have different effects to the parallel concolic testing.

A good design of the schedule function is a hard problem. One reason for its difficulty is that the condition of the path tree can be varied. Different programs, even different units in the same program, have different kinds of path distribution. This means there can hardly be a universal scheduler which performs the same well on every testing unit. Another reason for the difficulty is the inability in the prediction of the running time of every worker iteration. Even if we find a scheduler which can balance the number of paths on workers, the total time cost on different workers may be still unbalanced.

Although we have not got an excellent scheduler function, we come up with some standards which a good scheduler function should observed. A good scheduler function should divide the space as uniformly as possible, each computing unit having almost the same number of tasks. Moreover, it is even better if the scheduler could balance the overall computing time on each computing unit. On the other hand, a poor scheduler function fails to equitably distribute the number of tasks and computing time on each computing unit. This could lead to that a certain number of computing units are extremely busy, while the others are just wasting time on waiting.

For instance, let \( p \) be the length of a path to be scheduled, and \( \mathcal{H} \) be the hash policy used in the scheduler. Then, we have the following definitions for the scheduler function \((\mathcal{F} \circ \mathcal{H})(p)\):

\[
\mathcal{H}(p) = p \mod \text{MAX\_UNIT\_CNT}, \quad \mathcal{F}(p) = \mathcal{H}(p).
\]

This scheduler assigns a task to the working unit identified by the length of path prefix modulo the number of computing units. If the number of computing units is larger than the length of the longest path in the target program, some computing units will stay in starving state for a long time. Thus, the design of a fair function \( \mathcal{H} \) is important to improve the whole testing performance.

2.2.3 Motivating Example Revisited

The motivating example in Section 1 can be efficiently processed. We tested the example on an Intel Core i7 platform which is equipped with eight logical processors. Compared to the result of the sequential testing (84.6 seconds), the parallel testing cost only 21.4 seconds to complete the searching of about 3400 feasible paths. During the parallel testing in this case, eight processes are started at the same time to explore the path space. The CPU workload is fully utilized for solving constraints of long paths and exploring more paths from existing paths. The percentage of the performance improved is determined
not only by the increasing of computing units but also by the complexity of the target program under test. In the extreme case by the example we present here, the performance improvement is huge. It shows that the parallel approach gains better advantage for large program involving longer paths and more complex path conditions.

3 Evaluation

We have implemented the parallel algorithm and integrated it into the unit testing toolkit CAUT [1]. In this section, some details on the experiments will be given. Also, some typical results will be shown, and the explanations to them will be given. Our experiments are conducted on 2.66GHz Intel Core i7 CPU running Windows 7 with 6GB RAM, which provides eight logical processors.

3.1 Experiment preparation

The experimental examples mainly come from SIR [2], including bash, flex, grep, make, printtoken2 and schedule. Other examples are algebra linear [3] and micro OpenGL core (conGL) [4]. We selected parts of those programs but not whole programs to ensure that the testing time of each experiment was less than 15 minutes in the single core mode. For each example, we tested every separated function one by one in the target program and then summed the data of every tested unit (such as feasible paths) up as the result data. The calling dependencies in the unit under test were replaced by mock functions. Besides, environment inputs shall be transformed to arguments of the testing unit, as they may disturb program paths.

The scheduler function $H$ we adopted is a general hash function with the range $[0, 2^{32})$, while the function $F$ divides the range of the $H$ equitable to every computing unit. The functions are defined as follows:

```c
unsigned int H(p) {
    unsigned int hash = 0;
    for each node value n in p
        hash = (hash << 5) + hash + n;
    return hash;
}

unsigned int F(p) {
    unsigned int worker = H(p) mod MAX_UNIT_CNT;
    return worker;
}
```

The above scheduler we adopted is based on our experimental tries. As
Table 1
Experimental Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Units</th>
<th>Lines</th>
<th>Feasible Paths</th>
<th>Single-Core(ms)</th>
<th>Dual-Core(ms)</th>
<th>Single:Dual</th>
</tr>
</thead>
<tbody>
<tr>
<td>algebra linear</td>
<td>27</td>
<td>3240</td>
<td>1657</td>
<td>723598</td>
<td>553444</td>
<td>130.74%</td>
</tr>
<tr>
<td>bash</td>
<td>35</td>
<td>1170</td>
<td>2002</td>
<td>336139</td>
<td>257466</td>
<td>130.56%</td>
</tr>
<tr>
<td>c00nGL</td>
<td>26</td>
<td>1282</td>
<td>226</td>
<td>76242</td>
<td>60864</td>
<td>125.27%</td>
</tr>
<tr>
<td>flex</td>
<td>25</td>
<td>538</td>
<td>3150</td>
<td>758809</td>
<td>587220</td>
<td>129.22%</td>
</tr>
<tr>
<td>grep</td>
<td>19</td>
<td>1215</td>
<td>505</td>
<td>101050</td>
<td>95835</td>
<td>105.44%</td>
</tr>
<tr>
<td>make</td>
<td>26</td>
<td>786</td>
<td>1769</td>
<td>136716</td>
<td>128294</td>
<td>106.56%</td>
</tr>
<tr>
<td>printtoken2</td>
<td>13</td>
<td>359</td>
<td>47</td>
<td>176574</td>
<td>132333</td>
<td>133.43%</td>
</tr>
<tr>
<td>schedule</td>
<td>16</td>
<td>147</td>
<td>100</td>
<td>6140</td>
<td>5659</td>
<td>108.50%</td>
</tr>
</tbody>
</table>

it is discussed in the last section, it is selected according to the observations, although this instance of scheduler is not guaranteed to be the best solution.

3.2 Results

The experimental result is shown in Table 1, where the fourth column shows all the feasible paths of each example we tested. The last three columns show the time cost ratio between sequential (Single Core) and parallel (Dual Core) mode. It is clearly demonstrated that time cost of the parallel is less than the one of sequential mode with the given hardware resources given. Because of the scheduler function behaves differently on different examples the accelerated percentage (the last column) floats in a wide range (the lowest for grep costs 105.44% while the highest for printtoken2 costs 133.43%). In a good one, e.g. printtoken2, the scheduler function may assign the generated paths uniformly on the two processors. The statistic data also supports our reasoning. In printtoken2, 315 cross-cpu tasks were sent to the one processor while the other processor received 304 cross-cpu tasks. Taking grep as another example, the performance of our parallel algorithm behaves not well enough. The experimental data shows that the paths allocation is not average for two processors: one is assigned to 782 cross-cpu tasks, while the other even only has 208 cross-cpu tasks, and computing tasks from 14 of 19 functions completely cannot be parallelized. To explain this reason, we analyzed the source code of grep. We found that many paths in grep program are short, which leads to the bad performance of our adopted scheduler function, because it may map the short paths to the same processor. The other examples which behaves not well in the parallel algorithm have the same reason.

The other experiment shows the trend of performance boosting by increasing processors for five examples we listed previously. Figure 4 shows the result where the x axis presents the number of total processors and the y axis is the ratio between the time cost of parallel testing and the cost of sequential one. In Figure 4, we can easily see that the performance increases with
the adding of processor numbers. The acceleration ratio can reach almost 30% with eight processors for those examples, which means that our parallel algorithm is very effective and can save nearly 70% time cost compared to the sequential concolic testing. Furthermore, the five curves also tell that the threshold of performance boosting may be arrived. It is obvious that some of curves drop more rapidly than others, but the trend of all of them tends to be flat. The reason is that in our model all the path schedulers are local and run in parallel, and they will send the paths to other processors to be handled. With the increasing number of processors, the communication cost (even it is asynchronous) will increase as well.

4 Conclusion

This paper gives a different perspective on the performance improvement of concolic testing technique by introducing a parallel algorithm. This kind of method is able to fully utilize resources of hardware, so the performance can be gained by increasing hardware processors or computation nodes in the distributed system. The contribution of our work is to apply parallel capability to traditional concolic testing with a low-synchronization framework. The parallel algorithm has been implemented and integrated into CAUT [1]. The usability of the parallel approach is further confirmed by the application of CAUT. Comparing with other scalable test case generation techniques, the parallel model provides a practical approach for them.

5 Acknowledgement

Xiao Yu is partially supported by 973 Project No.2005CB321904. Sun Shuai is partially supported by NFSC No. 90818024. Geguang Pu is partially supported by Fundamental Research Funds for the Central Universities and
References

