iFM 2012 ABZ 2012

Joint Conference

in honour of Egon Börger's 65th birthday for his contribution to state-based formal methods

18-21 June 2012

CNR Research Area of Pisa, Italy

Proceedings of the Posters & Tool demos Session

Chairs / Editors

Franco Mazzanti

Gianluca Trentanni

The papers comprised in this book represent the contributions to Posters and Tool Demo Session of the Joint 9th International Conference on Integrated Formal Methods (iFM 2012) and ABZ Conference (ABZ 2012).

Online versions of these proceedings can be retrieved from:

http://puma.isti.cnr.it/linkdoc.php?icode=2012-ED-001&authority=cnr.isti&collection=cnr.isti

iFM 2012 - ABZ 2012

Posters and Tool Demo Session

Table of Contents

The ASMETA framework

Paolo Arcaini[1], Angelo Gargantini[2], Elvinia Riccobene[1], and Patrizia Scandurra[2]

[1] Dip. di Tecnologie dell'Informazione, Università degli Studi di Milano, Italy
{paolo.arcaini,elvinia.riccobene}@unimi.it
[2] Dip. di Ing. dell'Informazione e Metodi Matematici, Università di Bergamo, Italy
{angelo.gargantini,patrizia.scandurra}@unibg.it

1 Introduction

The use of formal methods, based on rigorous mathematical foundations, is essential for system development. However, some skepticism exists against formal methods mainly due to the lack of tools supporting formal development, or to the tools' loosely coupling that does not allow reuse of information. The integration and interoperability of tools is hard to accomplish, so preventing formal methods from being used in an efficient and tool supported manner during the system development life cycle.

The ASMETA (ASM mETAmodeling) framework[3] [4,10] is a set of tools around the Abstract State Machines (ASMs). These tools support different activities of the system development process, from specification to analysis, and are strongly integrated in order to permit reusing information about models during several development phases.

ASMETA has been developed [4,11,13] by exploiting concepts and technologies of the Model-Driven Engineering (MDE), like metamodeling and automatic model transformation. The starting point of the ASMETA development has been the *Abstract State Machine Metamodel* (AsmM) [12], an abstract syntax description of a language for ASMs. From the AsmM, by exploiting MDE techniques of automatic model-to-model and model-to-text transformation, a set of software artifacts (concrete syntax, parser, interchange format, API, etc.) has been developed for model editing, storage and manipulation. These software artifacts have been later used as a means for the development of new more complex tools, and the integration within ASMETA of already existing tools, so providing a powerful and useful tool support for system specification and analysis.

After briefly introducing the ASM formal method and its potentiality as system engineering method, we present the ASMETA toolset which provides basic functionalities for ASM models creation and manipulation (as editing, storage, interchange, access, etc.) as well as advanced model analysis techniques (validation, verification, testing, review, requirements analysis, runtime monitoring, etc.).

A suitable set of ASM benchmark examples will be selected for the demo purposes in order to show all the potentialities of the ASMETA framework over different characteristics of the ASM models (parallelism, non determinism, distributivity, submachine invocations, etc.)

2 Abstract State Machines

The *Abstract State Machine* (ASM) method is a systems engineering method that guides the development of software and embedded hardware-software systems seamlessly from

[3] http://asmeta.sourceforge.net/

requirements capture to their implementation. Within a single precise yet simple conceptual framework, the ASM method supports and uniformly integrates the major software life cycle activities of the development of complex software systems. The process of *requirements capture* results into constructing rigorous *ground models* which are precise but concise high-level system blueprints ("system contracts"), formulated in domain-specific terms, using an application-oriented language which can be understood by all stakeholders. From the ground model, by stepwise refined models, the *architectural and component design* is obtained in a way which bridges the gap between specification and code. The resulting *documentation* maps the structure of the blueprint to compilable code, providing explicit descriptions of the software structure and of the major design decisions, besides a road map for system *(re-)use* and *maintenance*.

Even if the ASM method comes with a rigorous scientific foundation [5], the practitioner needs no special training to use the ASM method since Abstract State Machines are a simple extension of Finite State Machines, obtained by replacing unstructured "internal" control states by states comprising arbitrarily complex data, and can be understood as pseudo-code over abstract data structures. The *states* of an ASM are multi-sorted first-order structures, i.e., domains of objects with functions and predicates (boolean functions) defined on them, while the *transition relation* is specified by "rules" describing the modification of the functions from one state to the next.

The notion of ASMs formalizes simultaneous parallel actions of a single agent, either in an atomic way, *Basic ASMs*, or in a structured and recursive way, *Structured or Turbo ASMs*. It also supports a generalization where multiple agents interact in parallel in a synchronous/asynchronous way, *Synchronous/Asynchronous Multi-agent ASMs.* Appropriate rule constructors also allow non-determinism (**choose** or existential quantification) and unrestricted synchronous parallelism (universal quantification **forall**).

A complete mathematical definition of the ASMs can be found in [5], together with a presentation of the great variety of its successful application in different fields such as: definition of industrial standards for programming and modeling languages, design and re-engineering of industrial control systems, modeling e-commerce and web services, design and analysis of protocols, architectural design, language design, verification of compilation schemas and compiler back-ends, service-oriented applications, etc.

The ASM method allows a *modeling technique* which integrates static (*declarative*) and dynamic (*operational*) descriptions, and an *analysis technique* that combines *validation* and *verification* methods at any desired level of detail. The ASMETA framework makes the application of this modeling technique practically feasible.

3 The ASMETA tool-set

Concrete syntax and other language artifacts To write ASM models in a textual and human-comprehensible form, a platform-independent concrete syntax, *AsmetaL*, is available, together with a text-to-model compiler, *AsmetaLc*, to parse AsmetaL models and check for their consistency w.r.t. the AsmM metamodel OCL constraints. It is also possible to save ASM models into an XMI interchange format, and Java APIs are available to represent ASMs in terms of Java objects[4].

[4] All these software artifacts have been developed in a generative manner from the AsmM metamodel, by exploiting MDE techniques of automatic model-to-model/text transformations.

Simulator Simple model validation can be performed by *simulating* ASM models with the ASM simulator *AsmetaS* [9] to check a system model with respect to the desired behavior to ensure that the specification really reflects the user needs. AsmetaS supports *invariant checking* to check whether invariants expressed over the currently executed ASM model are satisfied or not, *consistent updates checking* for revealing inconsistent updates, *random simulation* where random values for monitored functions are provided by the environment and *interactive simulation* when required inputs are provided interactively during simulation.

Scenario-based validation A more powerful validation approach is *scenario-based validation* by the ASM validator *AsmetaV* [6]. AsmetaV is based on the AsmetaS simulator and on the *Avalla* modeling language; this last provides constructs to express execution scenarios in an algorithmic way as interaction sequences consisting of *actions* committed by the *user actor* to `set` the environment, to `check` the machine state, to ask for the `exec`ution of certain transition rules, and to enforce the machine itself to make one `step` (or a sequence of steps) as reaction of the actor actions.

AsmetaRE Use cases are commonly used to structure and document functional requirements, and should be used for validating system requirements. The AsmetaRE [14] automatically maps use case models – written by the tool aToucan[5] according to the approach *Restricted Use Case Modeling* (RUCM) [15] – into ASM models written in AsmetaL. The result of such model-to-text transformation is an executable ASM specification that serves as basis to perform requirements validation by the ASMETA toolset. In particular, an ad-hoc transformation allows also the generation of Avalla scenarios from use cases for scenarios-based validation with the AsmetaV tool.

Model review Model review is a validation technique aimed at determining if a model is of sufficient quality; it allows to identify defects early in the system development, reducing the cost of fixing them. The *AsmetaMA* tool [2] permits to perform *automatic* review of ASMs; it looks for typical vulnerabilities and defects a developer can introduce during the modeling activity using the ASMs.

Model checking Formal verification of ASM models is supported by the *AsmetaSMV* tool [1]; it takes in input ASM models written in AsmetaL and maps these models into specifications for the model checker NuSMV. AsmetaSMV supports both the declaration of *Computation Tree Logic* (CTL) and *Linear Temporal Logic* (LTL) formulas.

Runtime verification Runtime verification is a technique that allows checking whether a run of a system under scrutiny satisfies or violates a given correctness property. *CoMA* (Conformance Monitoring by Abstract State Machines) [3] is a specification-based approach (and a supporting tool) for runtime monitoring of Java software. Based on the information obtained from code execution and model simulation, the conformance of the concrete implementation is checked with respect to its formal specification given in terms of ASMs. At runtime, undesirable behaviors of the implementation, as well as incorrect specifications of the system behavior are recognized.

ATGT Model-based testing aims to use models for software testing. One of its main applications consists in test case generation where test suites are automatically generated

[5] `http://www.sce.carleton.ca/~tyue/`

from abstract models of the system under test. The ATGT tool [8] is available for testing of ASM models. ATGT implements a set of adequacy criteria defined for the ASMs [7] to measure the coverage achieved by a test set and determine whether sufficient testing has been performed. To build test suites satisfying some coverage criteria, it implements a technique that exploits the capability of a model checker to produce counterexamples, and it uses the model checker SPIN for the automatic test case generation.

References

1. P. Arcaini, A. Gargantini, and E. Riccobene. AsmetaSMV: a way to link high-level ASM models to low-level NuSMV specifications. In *Abstract State Machines, Alloy, B and Z, 2nd Int. Conference (ABZ 2010)*, volume LNCS 5977, pages 61–74. Springer, 2010.
2. P. Arcaini, A. Gargantini, and E. Riccobene. Automatic Review of Abstract State Machines by Meta Property Verification. In C. Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010)*, pages 4–13. NASA, 2010.
3. P. Arcaini, A. Gargantini, and E. Riccobene. CoMA: Conformance monitoring of java programs by abstract state machines. In *2nd International Conference on Runtime Verification, San Francisco, USA, September 27 - 30 2011*, 2011.
4. P. Arcaini, A. Gargantini, E. Riccobene, and P. Scandurra. A model-driven process for engineering a toolset for a formal method. *Software: Practice and Experience*, 41:155–166, 2011.
5. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.
6. A. Carioni, A. Gargantini, E. Riccobene, and P. Scandurra. A Scenario-Based Validation Language for ASMs. In *Proceedings of the 1st international conference on Abstract State Machines, B and Z (ABZ '08)*, volume LNCS 5238, pages 71–84. Springer-Verlag, 2008.
7. A. Gargantini and E. Riccobene. ASM-Based Testing: Coverage Criteria and Automatic Test Sequence Generation. *J. Universal Computer Science*, 7:262–265, 2001.
8. A. Gargantini, E. Riccobene, and S. Rinzivillo. Using Spin to Generate Tests from ASM Specifications. In *Proceedings of ASM 2003*, volume LNCS 2589. Springer Verlag, 2003.
9. A. Gargantini, E. Riccobene, and P. Scandurra. A Metamodel-based Language and a Simulation Engine for Abstract State Machines. *J. Universal Computer Science*, 14(12):1949–1983, 2008.
10. A. Gargantini, E. Riccobene, and P. Scandurra. Model-Driven Language Engineering: The ASMETA Case Study. In *Int. Conf. on Software Engineering Advances, ICSEA*, pages 373–378, 2008.
11. A. Gargantini, E. Riccobene, and P. Scandurra. Integrating Formal Methods with Model-Driven Engineering. In *Software Engineering Advances, 2009. ICSEA '09. Fourth International Conference on*, pages 86 –92, sept. 2009.
12. A. Gargantini, E. Riccobene, and P. Scandurra. Ten Reasons to Metamodel ASMs. In J.-R. Abrial and U. Glässer, editors, *Rigorous Methods for Software Construction and Analysis*, volume LNCS 5115, pages 33–49. Springer Verlag, 2009.
13. A. Gargantini, E. Riccobene, and P. Scandurra. Combining formal methods and mde techniques for model-driven system design and analysis. *International Journal on Advances in Software*, 3(1,2):1 – 18, 2010.
14. P. Scandurra, T. Yue, A. Arnoldi, and M. Dolci. Functional requirements validation by transforming use case models into abstract state machines. In *Proceedings of the 27th Symposium On Applied Computing (SAC 2012)*, 2012.
15. T. Yue, L. C. Briand, and Y. Labiche. A use case modeling approach to facilitate the transition towards analysis models: Concepts and empirical evaluation. In *MoDELS*, pages 484–498, 2009.

The ASMETA framework

http://asmeta.sourceforge.net

FM&SE Lab
Formal methods and software engineering laboratory

Paolo Arcaini[1], Angelo Gargantini[2], Elvinia Riccobene[1], Patrizia Scandurra[2]
[1] DI, Università degli Studi di Milano, Italy – {paolo.arcaini,elvinia.riccobene}@unimi.it
[2] DIIMM, Università di Bergamo, Italy – {angelo.gargantini,patrizia.scandurra}@unibg.it

ASMETA features

- ASMETA (ASM mETAmodeling) [1] is a set of tools for the Abstract State Machines (ASMs)
- ASMETA has been developed [2] by technologies of the Model-Driven Engineering
  - metamodeling (**AsmM** is the ASMs Metamodel)
  - automatic model transformations
- ASMETA supports different activities of the system development process, from specification to analysis
- ASMETA tools are strongly integrated for reusing model information
- ASMETA permits to easily develop new tools or integrating existing ones

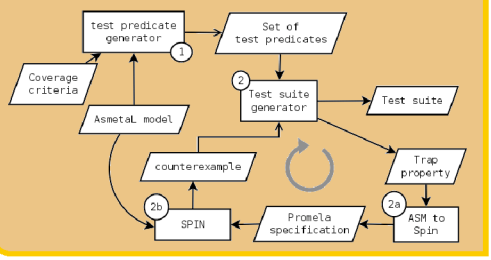
Modeling language: AsmetaL [3]

```
asm ticTacToe
import StandardLibrary
import CTLlibrary

signature:
    domain Coord subsetof Integer
    enum domain Sign = {CROSS | NOUGHT}
    enum domain Status = {TURN_USER | TURN_PC}

    controlled board: Prod(Coord, Coord) -> Sign
    controlled status: Status
    monitored userChoiceX: Coord
    monitored userChoiceY: Coord
    derived winner: Sign -> Boolean
    derived chosenCells: Sign -> Powerset(Prod(Coord, Coord))
    derived endOfGame: Boolean

definitions:
    domain Coord = {1..3}

    function chosenCells($s in Sign) =
        {$x in Coord, $y in Coord | board($x, $y) = $s: ($x, $y)}

    function winner($s in Sign) =
        (exist $r in Coord with (forall $c in Coord with board($r, $c) = $s)) or
        (exist $k in Coord with (forall $t in Coord with board($t, $k) = $s)) or
        (forall $d in Coord with board($d, $d) = $s) or
        (forall $d1 in Coord with board($d1, 4 - $d1) = $s)

    function endOfGame =
        (exist $s in Sign with winner($s)) or
        (forall $x in Coord, $y in Coord with isDef(board($x, $y)))

    rule r_moveUser =
        if(isUndef(board(userChoiceX, userChoiceY))) then
            par
                board(userChoiceX, userChoiceY) := CROSS
                status := TURN_PC
            endpar
        endif

    rule r_movePC =
        par
            choose $x in Coord, $y in Coord with isUndef(board($x, $y)) do
                board($x, $y) := NOUGHT
            status := TURN_USER
        endpar

    invariant over board: abs(size(chosenCells(NOUGHT)) -
                              size(chosenCells(CROSS))) <= 1

    CTLSPEC ef(winner(CROSS))
    CTLSPEC ef(winner(NOUGHT))
    CTLSPEC not(ef(winner(NOUGHT)))
    CTLSPEC ag(not(winner(CROSS) and winner(NOUGHT)))

    main rule r_Main =
        if(not(endOfGame)) then
            if(status = TURN_USER) then
                r_moveUser[]
            else
                r_movePC[]
            endif
        endif

default init s0:
    function status = TURN_USER
```

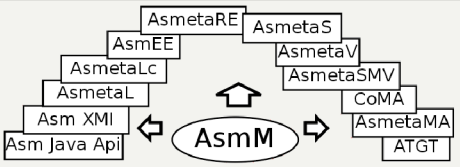
Model checking: AsmetaSMV [5]

```
-- specification EF winner(CROSS)  is true
-- specification EF winner(NOUGHT)  is true
-- specification !(EF winner(NOUGHT))  is false
-- as demonstrated by the following CTL Counterexample
-> State: 1.1 <-
  ...
-> State: 1.6 <-
  board(3,1) = CROSS
  status = TURN_PC
  userChoiceX = 1
  var_$y_0 = 3
-> State: 1.7 <-
  board(2,3) = NOUGHT
  status = TURN_USER
  var_$y_0 = 2
  winner(NOUGHT) = TRUE
  endOfGame = TRUE
```

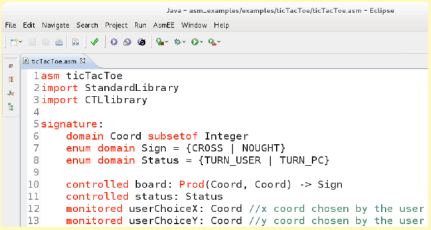
Automatic test case generation: ATGT [6]



ASMETA toolset

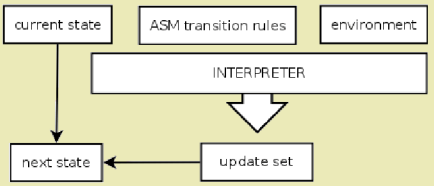


AsmetaL editor: AsmEE



```
1asm ticTacToe
2import StandardLibrary
3import CTLlibrary
4
5signature:
6    domain Coord subsetof Integer
7    enum domain Sign = {CROSS | NOUGHT}
8    enum domain Status = {TURN_USER | TURN_PC}
9
10   controlled board: Prod(Coord, Coord) -> Sign
11   controlled status: Status
12   monitored userChoiceX: Coord //x coord chosen by the user
13   monitored userChoiceY: Coord //y coord chosen by the user
```

Model simulation: AsmetaS [3]



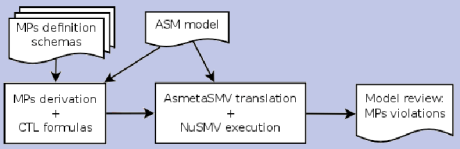
Features

- Interactive simulation
- Random simulation
- Bounded-unbounded simulation
- Invariant checking
- Inconsistent update checking

Simulation trace

```
Insert a constant in Coord of
type Integer for userChoiceX: 1
Insert a constant in Coord of
type Integer for userChoiceY: 1
<State 0 (monitored)>
userChoiceX=1
userChoiceY=1
</State 0 (monitored)>
<State 1 (controlled)>
board(1,1)=CROSS
...
board(3,3)=undef
status=TURN_PC
</State 1 (controlled)>
<State 2 (controlled)>
board(1,1)=CROSS
board(1,2)=undef
board(1,3)=NOUGHT
...
board(3,3)=undef
status=TURN_USER
</State 2 (controlled)>
Insert a constant in Coord of
type Integer for userChoiceX: 3
Insert a constant in Coord of
type Integer for userChoiceY: 1
<State 2 (monitored)>
userChoiceX=3
userChoiceY=1
</State 2 (monitored)>
<State 3 (controlled)>
...
```

Model review: AsmetaMA [7]



Metaproperties

Consistency
- **MP1**: No inconsistent update is ever performed

Completeness
- **MP2**: Every conditional rule must be complete
- **MP7**: Every controlled location is updated and every location is read

Minimality
- **MP3**: Every rule can eventually fire
- **MP4**: No assignment is always trivial
- **MP5**: For every domain element e there exists a location which has value e
- **MP6**: Every controlled function can take any value in its co-domain

Metaproperty violation

Wrong implementation of the main rule

```
main rule r_Main =
    if(not(endOfGame)) then
        par
            r_moveUser[]
            r_movePC[]
        endpar
    endif
```

Violation of the MP1 metaproperty (inconsistent updates are possible)

```
Inconsistent update. Location board(1,1) is
updated to values NOUGHT and CROSS when are
satisfied simultaneously the conditions
((var_$y_0 = 1) & !(endOfGame) & (var_$x_0 = 1)
& (board(1,1) = SIGN_UNDEF)) and (!(endOfGame)
& (1 = userChoiceY) & (board(1,1) = SIGN_UNDEF)
& (1 = userChoiceX)). ...
```

References

[1] A. Gargantini, E. Riccobene, and P. Scandurra. Model-Driven Language Engineering: The ASMETA Case Study. In *Int. Conf. on Software Engineering Advances, ICSEA*, pages 373–378, 2008.

[2] P. Arcaini, A. Gargantini, E. Riccobene, and P. Scandurra. A model-driven process for engineering a toolset for a formal method. *Software: Practice and Experience*, 41:155–166, Feb. 2011.

[3] A. Gargantini, E. Riccobene, and P. Scandurra. A Metamodel-based Language and a Simulation Engine for Abstract State Machines. *JUCS*, 14(12):1949–1983, jun 2008.

[4] A. Carioni, A. Gargantini, E. Riccobene, and P. Scandurra. A Scenario-Based Validation Language for ASMs. In *ABZ 2008*, LNCS 5238, pages 71–84. Springer, 2008.

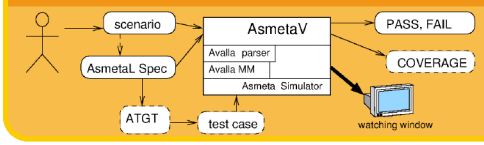
[5] P. Arcaini, A. Gargantini, and E. Riccobene. AsmetaSMV: a way to link high-level ASM models to low-level NuSMV specifications. In *ABZ 2010*, LNCS 5977, pages 61–74. Springer, 2010.

[6] A. Gargantini, E. Riccobene, and S. Rinzivillo. Using Spin to Generate Tests from ASM Specifications. In *ASM 2003*, LNCS 2589, pages 263-277. Springer, 2003.

[7] P. Arcaini, A. Gargantini, and E. Riccobene. Automatic Review of Abstract State Machines by Meta Property Verification. In *Proceedings of NFM 2010*, pages 4–13. NASA, 2010.

[8] P. Arcaini, A. Gargantini, and E. Riccobene. CoMA: Conformance Monitoring of Java programs by Abstract State Machines. In *2nd Int. Conf. on Runtime Verification, 2011*.

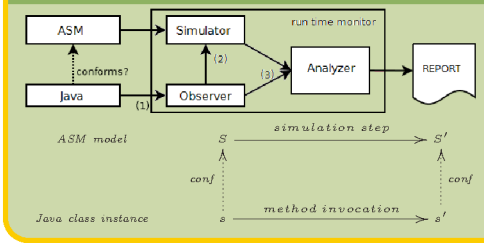
Scenario-based validation: AsmetaV [4]



Scenario

```
scenario ticTacToe1
load ticTacToe.asm

set userChoiceX := 1;
set userChoiceY := 1;
step
check board(1, 1) = CROSS and not(winner(CROSS)) and not(winner(NOUGHT));

exec par
        board(1,3) := NOUGHT
        status := TURN_USER
    endpar ;
check not(winner(CROSS)) and not(winner(NOUGHT));
...
set userChoiceX := 2;
set userChoiceY := 1;
step
check board(2, 1) = CROSS and winner(CROSS) and not(winner(NOUGHT)) and endOfGame;
```

Runtime monitoring: CoMA [8]



Monitored Java class

```
@Asm(asmFile = "models/nonDetModels/ticTacToe.asm")
public class TicTacToe {
    @FieldToFunction(func = "board")
    public Sign[][] board;
    private Random rnd;
    private int numMoves = 0;

    @StartMonitoring
    public TicTacToe() {
        board = new Sign[3][3];
        rnd = new Random();
    }

    @RunStep
    public void execUserMove(@Param(func = "userChoiceX") int x,
                             @Param(func = "userChoiceY") int y) {
        board[x - 1][y - 1] = Sign.CROSS;
        numMoves++;
    }

    @RunStep
    public void execComputerMove() {
        if(numMoves < 9) {
            int moveX, moveY;
            do {
                moveX = rnd.nextInt(3);
                moveY = rnd.nextInt(3);
            }
            while(board[moveX][moveY] != null);
            board[moveX][moveY] = Sign.NOUGHT;
            numMoves++;
        }
    }
    private enum Sign {CROSS, NOUGHT;}
}
```

Tool-based Teaching of Formal Methods

(an extended abstract for a poster presentation)

Shin NAKAJIMA

National Institute of Informatics
also The Graduate University for Advanced Studies (SOKENDAI)
Tokyo, Japan
nkjm@nii.ac.jp

1 Backgrounds

As software-intensive systems constitute the social infrastructure, their safety and reliability are the major concerns in software industry. In Japan, for example, some major IT companies have gotten together to conduct feasibility studies of formal methods [1] to achieve their goals. The number of engineers to learn formal methods is increasing, which implies a strong need for easy-to-access educational materials.

The engineers usually learn a specific method/language to have a particular impression on the formal methods in general, which may hinder them from a proper understanding of the subject matter. Most of the educational materials today focus on a particular method/technique to present an in-depth explanation of each. Some books bundle a collection of several key methods from viewpoints of either theory with indications of mathematical logic (cf. [3]) or practice with a lightweight touch (cf. [4]). They have placed emphasis on the differences between these methods to highlight the novelty of individual approaches.

As formal methods constitute a distinct area in software science and technology, they have some common cores with them. The core concepts are expected to be available in the form of educational materials. Some existing materials (cf. [5]) start with the basics of mathematical logic to explain advanced topics on the current up-to-date formal techniques. Although the approach is quite right in view of tranditional ways to present the subject matter, mathematical logic is a double-edged sword. It is a precise and concise language to define core concepts of formal methods; it plays a role of *meta-language*. All the rigor in specification and analysis in formal methods inherits from that of mathematical logic. These characteristics, at the same time, hinder the readers from the educational materials unless they are patient enough to be accustomed to mathematical logic.

An alternative approach to learning of the common cores is desirable. One may find such materials [2] in computer programs to be used as references. A wide variety of concepts in programming is concretely presented by expressing them in *Scheme* language. The textbook skips mathematical notations usually used in defining the semantics of programming languages. Furthermore, readers can experiment with *Scheme* snippets to learn the concepts without losing precision

or conciseness. A new teaching method in a similar flavor is needed to learn durable ideas in formal methods.

2 Teaching with Alloy

The notion of abstractions plays a key role in software or computing [8]. It has simultaneously been argued that it it not easy to learn how to make abstractions without having concrete representations. Automated tools for formal analysis have been found to be successful in modeling software. We can deepen our understanding of it at an adequate level of abstraction through the iterative process of writing and analysis [6].

We needed to borrow this idea of lightweight formal methods to design the course materials. We particularly adapted *Alloy* [7] for three main reasons.

- The logic behind *Alloy*, i.e., first-order relational logic with a built-in transitive closure operator, is adequate to represent most concepts in the subject matter.
- *Alloy* adapts its syntax similar to what is familiar to software engineers. Mathematical logic is sugared-wrapped.
- Descriptions in *Alloy* can be analyzed automatically, which can provide quick feedback to users.

Note that analysis with *Alloy* is incomplete (Section 5.3 in [7]). The tool employs a bounded analysis method to achieve automation. There are certainly formulas that are not properly handled; such cases result in spurious alarms. We, however, decided to use *Alloy* and not interactive theorem provers, although the latter fully covered first-order logic. This is because automated analysis is preferable for software engineers who do not have much knowledge of mathematical logic. They can use such an automated tool even when they are not familiar with proof techniques. Certainly, much care should be taken in teaching sessions to point out the limitations with bounded analysis.

3 Overview of Contents

We had in mind the statement "light is right, long is wrong" in designing the course. This meant that the overall course was not long, but took a semester, and that the presentation made use of lightweight formal analysis with *Alloy*. Table 1 summarizes the contents of a textbook that provides the educational materials we propose.

The textbook consists of four parts. Chapters 1 and 2 constitute the first part, which is an introduction to the materials. The second part of the textbook consists of Chapters 3 and 4 together with an Appendix. It explains the common core in model-oriented formal methods [9] because of their historical importance. They include the state-based style of specifications and the notion of refinements in these languages.

	Chapter Contents
Chapter 1	Introduction to Formal Methods
Chapter 2	A Quick Tour of Alloy
Chapter 3	State-based Specification
Chapter 4	Refinement
Chapter 5	Class Diagram with OCL
Chapter 6	State Transition Diagram and Logic Model-Checking
Chapter 7	Program Verification and Automated Test Generation
Appendix	History of Model-Oriented Formal Methods
	with 153 references listed

Table 1. Table of Contents

The third part contains Chapters 5 and 6. We should discuss the relationship between (classical) formal methods and object-oriented design methods. Furthermore, we need a basic understanding of logic model checking and state-transition systems. The final part, Chapter 7, focuses on checking of sequential programs written, for example, in C language. It is of practical importance to know how software model-checking and specification-based testing are done since such tools become available for use in industry. The materials are intended to avoid making these technologies as *black-box*.

4 Initial Assessment

Classroom Experience The proposed materials were originally developed for a series of graduate-level lectures at SOKENDAI. The series have been offered every each year since 2009. We obtained some feedback from enrolled students who were part-time and held positions in industry although the number of them is small. They were interested in learning new software technologies such as formal methods, but did not have enough background knowledge in mathematical logic.

Firstly, it was easier for them to use *Alloy* instead of mathematical logic to study the subject matter. The laboratory work to experiment with *Alloy* tool motivated them to gain access to the reading materials. It provided them with good opportunities to look at the originals of classical papers.

Secondly, more discussion was needed on limitations with the scope-bounded analysis of *Alloy* since it was a little difficult to determine whether a given formula had resulted in a spurious alarm. There was always a kind of trade-off between automated analysis and the background knowledge needed for conducting hand proofs with some mechanical support if any.

Furthermore, several seminars were organized for the audiences from industry, each focusing on a particular topic taken from the materials. The view discussed in the fourth part particularly motivated them to understand the core

technology differently than they had before. It was considered that program verification and testing shared nothing in common. The methods of automated verification and test-case generation shared a common underlying mechanism of SAT/SMT or CSP. All the problems were presented in the form of *Alloy* snippets.

Although the teaching experience was limited so far, it can be considered to be an initially positive sign to deem that the materials were effective.

Peer Reviews While assembling the materials into book form, we asked for a few peer reviews. Two main comments are listed below.

- The book is so concise that students who are complete novices will find it difficult to understand without lectures.
- It is suitable for those who are already familiar with at least one single formal method to broaden their knowledge of the subject matter.

These comments are consistent with our plans of using the materials in face-to-face teaching sessions and making them available to who are expected to be *leading* engineers in their projects.

5 Concluding Remarks

The materials have been made public in a textbook in Japanese. We also expect further feedback on the materials gathered from teaching sessions at other institutes in the future.

References

1. DSF (Dependable Software Forum) : http://www.nttdata.co.jp/dsf/en/
2. H. Abelson, and G.J. Sussman : *Structure and Interpretation of Computer Programs*, The MIT Press 1985.
3. D. Bjørner and M. Henson (eds.) : *Logics of Specification Languages*, Springer 2008.
4. H. Habrias and M. Frappier (eds.) : *Software Specification Methods*, ISTE Ltd 2006.
5. M. Huth and M. Ryan : *Logic in Computer Science (2ed.)*, Cambridge University Press 2004.
6. D. Jackson and J. Wing : Lightweight Formal Methods, in H. Saidian (ed.) : An Invitation to Formal Methods, IEEE Computer, pp.16-30, 1996.
7. D. Jackson : Software Abstractions – Logic, Language, and Analysis, The MIT Press, 2006.
8. J. Kramer : Is Abstraction the Key to Computing?, Comm. ACM, vol.50, no.4, pp.37-42, 2007.
9. J. Wing : A Specifier's Introduction to Formal Methods, IEEE Computer, pp.8-24, 1990.

Tool-based Teaching of Formal Methods

Shin NAKAJIMA
National Institute of Informatics
Tokyo, Japan

Backgrounds / Issues

Industry in Japan concerns reliability more than before.

Questions include which FM Should be chosen.

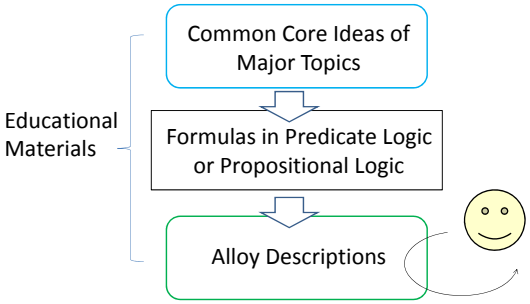
Engineers are lost in in-depth technical details.

Common ideas are important, but the presentation with Logic keeps them away.

Approach

cf. Structure and Interpretation of Computer Programs, MIT

Various Ideas in Programming → Scheme



Educational Materials

Chapter 1 : Introduction

Role of Formal Methods in Development of Software-intensive Systems

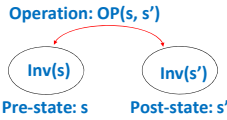
Importance of Abstraction

Chapter 2 : A Quick Tour of Alloy

Composite Pattern : tree structure
Stack : abstract datatype
Birthday Book : functional behavior

Chapter 3 : State-based Specification

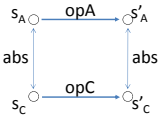
Common Ideas (VDM, Classical B, Event-B, Z Notation, Alloy)



Contractual/Behavioral Interpretation
Notion of Invariants

```
assert VDM-Satisfiability {
  all s1 : State | some s2 : State |
    preOp(s1) and invState(s1) implies postOp(s2) and invState(s2)
}

assert EventB-FIS {
  all s1 : State | some s2 : State |
    guardEvent(s1) and invState(s1) implies actionEvent(s2)
}
```

Chapter 4 : Refinement



Usages : Vertical (Data Refinement)
Horizontal (Superposition Refinement)

Basis : Forward Simulation
Backward Simulation

Technical Differences are seen in Proof Obligations

```
assert EventB-GRD {
  all v, t : StateC,  w, u : StateA |
    I(v) and J(v,w) and H(u) implies G(t)
}

assert EventB-SIM {
  all v, t : StateC,  w, w', u : StateA |
    I(v) and J(v,w) and H(u) and T(u,w,w')
    implies ( some t,v' : StateC |
      G(t) and S(t,v,v') and J(v',w') )
}
```

Chapter 5 : Class Diagram with OCL

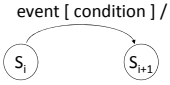
Relationship between OCL and Formal Methods

OCL is more like a functional programming language

Object-oriented Modeling and Formal Methods

Chapter 6 : State-Transition Diagram and Temporal Logic

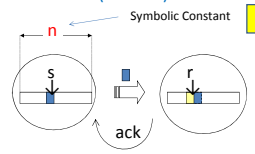
Mealy Diagram



Several ways to connect STD with Logic
(1) Encoding STD in FOL (P. Zave)
(2) Linear Temporal Logic as a fragment of FOL
(3) LTL Bounded Model Checking (BMC)
(4) Abstraction-Aided Verification
Predicate Abstraction, Ranking Abstraction

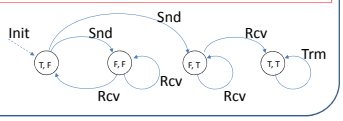
Descriptions to Automated Analysis

File Transfer (Event-B)



Calculating Abstraction Relation

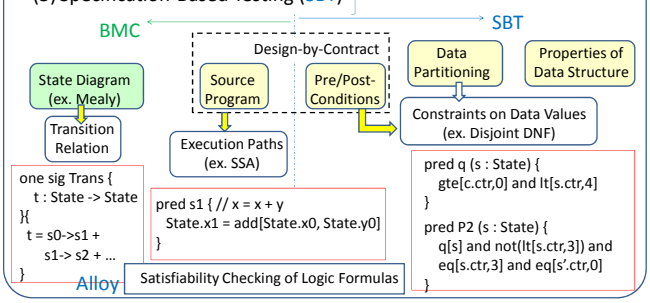
```
pred s1 (a, a' : State) {
  (a.s = a.r) and not(a.s = add[n,1]) and
  (a'.s = add[a.s,1]) and (a'.r = a.r) and (a'.s = a'.r)
}
```



Chapter 7 : Checking of Programs

(1) Floyd-Hoare Style Semantics
(2) Software Model-Checking : BMC
(3) Specification-Based Testing (SBT)

What is Common?



Initial Assessment

Classroom Experiences at SOKENDAI and Seminars before Industrial Audiences

Typical Profile of Students : Interested in new Software Technologies, one of which FM is.
Having no Enough Background Knowledge in Mathematical Logic.

Using Alloy : Pro → "push button style" formal analysis without proof techniques/tactics
Cons → limitation of the analysis due to the bounded search method

Students Feedback : Pro → Systematic Presentation of Software BMC and SBT
Cons → Modeling Exercises, which is another side of the technology

Peer Reviews of Educational Materials

Materials (Textbook) : Pro → Good for who already knows at least one Formal Specification Language
Cons → Too concise

Plan

Further lecture series at graduate schools in Japan are planned (2012, 2013).

Courses will be given based on the textbook.

Acknowledgments

All the people who have done great work in the area of formal methods.

The materials are all dependent on them.

Using the Overture Tool as a More General Platform

Claus Ballegaard Nielsen, Kenneth Lausdahl and Peter Gorm Larsen

Department of Engineering, Aarhus University
Finlandsgade 24, DK-8200 Aarhus N, Denmark
clausbn@iha.dk, kel@iha.dk, pgl@iha.dk

Abstract. The Overture project is aimed at developing an open platform to supply tool support to formal VDM models of systems. This paper presents the recent developments of the Overture project and supplies an overview of recent initiatives building on top of Overture as a platform.

1 Introduction

The Vienna Development Method (VDM) is one of the most mature formal methods [3] and has its focus on the development and analysis of a system model expressed in a formal language. The VDM language allows developers to verify the consistency of a model and its correctness with respect to an existing statement of requirements through the use of a range of analytic techniques, from testing to mathematical proof. Currently, VDM has three language dialects: (1) VDM-SL [5] supporting the modeling of the functionality of sequential systems; (2) VDM++ which supports object-oriented modeling and concurrency [4]; and (3) VDM-RT focused on real-time distributed systems [9]. Overture is built around an extensible open-source platform which from a tool perspective consists of two integrated main parts: an IDE based on the Eclipse framework and the tool VDMJ [1]. VDMJ is a small, relatively fast, open source tool written in Java that provides core support for VDM, such as a parser, a type checker, an interpreter, a debugger and a proof obligation generator. This paper provides a report on the current state of Overture and as such is to be considered as an update of [6]. The Overture tool and its components is introduced in Section 2. Section 3 contains a presentation of new tool support built on the Overture platform.

2 The Overture Tool: Architecture and Features

The Overture tool uses a plug-in architecture comprising the core VDM components and the components that hook into the Eclipse framework. While the relations between these components are fairly complex the principal components all depend on the Abstract Syntax Tree (AST) produced as a result of parsing a

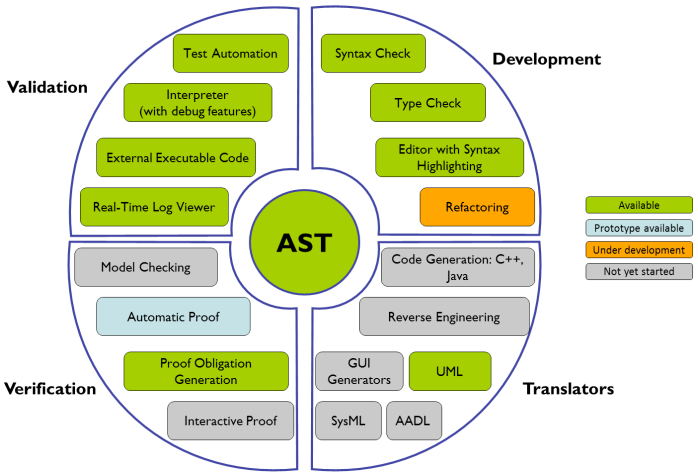


Fig. 1: Overture Tool Components

model. The current state of the Overture tool is illustrated in Figure 1, with the core AST at the center surround by the current and envisioned Overture components. The following presents an overview of the Overture components which all are integrated with the IDE consisting of different perspectives for working with and analyzing VDM models.

Overture Parser and AST: The parser is a recursive decent parser written in Java that accepts plain textual input and outputs an AST which all plug ins can access. The parser integrates with the Overture IDE and generates warnings used to highlight syntax problems.

Type Checking: Static type checking of a models type structure is performed automatically as the model is developed in the editor. Errors will be reported on type errors and warnings are generated for unused variables, unreachable code, etc. Statically an expression is considered type correct if an assignment of a value yields a result of the correct type. Incorrect assignments related to type invariants will be caught by the VDM run-time *dynamic type checking*.

Proof Obligation Generation: The formality of VDMs semantics enables the automated generation of a set of logical assertions that must be true for a specific model in order to ensure that it is dynamically type correct and semantically consistent. These are called *proof obligations* and in Overture they are generated for a range of properties linked to the internal consistency of a model, e.g. consistency of results with post conditions and termination of recursive functions.

Interpreter/debugger: The interpreter allows for simulations by enabling the execution of VDM models. The interpreter keeps track of code coverage during the execution of a model such that the visited and unvisited parts of the model can be color highlighted in the editor. The built-in debugger enable breakpoints to be used for inspecting locals, state variables, call stacks and running threads as well as a step-wise evaluation of a models expressions.

Combinatorial Testing: Combinatorial testing is the automatic generation and execution of test cases, based on regular expressions defined in *trace* definitions added to a VDM model. The expression describes possible sequences of operation calls and inputs from which tests can be generated. This is integrated in the IDE through a perspective [7].

Bi-directional UML mapping: A bi-directional mapping exists which enables a connection between models and UML diagrams defined in the XMI format. Mappings can be made between UML class diagrams and the object-oriented VDM++ models, in addition to UML sequence diagrams and VDM trace statements [8].

Realtime Log Viewer: VDM-RT allows for the definition of a virtual architecture of CPUs, buses and the relation between them. Based on the execution of a VDM-RT model a graphical visualization can be created, showing the internal events (operation request, message, thread swapping etc.).

External Executable Code: Overture defines two interfaces that enable communication with external executable code [10]. The *External Call Interface* enables VDM models to call external code defined in Java libraries, while the *Remote Control Interface* oppositely allows Java code to control the VDMJ interpreter and execute VDM expressions in an executing model.

3 Building on top of the Overture Platform

Originally Overture was built specifically as a platform to support only one VDM dialect, but this was later extended to all dialects. Since, it has developed and the openness of the platform has been used by stakeholders to expand the functionality and scope of use. Figure 2 shows how Overture has been expanding in two directions: (1) the DESTECS path has added an additional tool alongside Overture; and (2) the CML path has built on top of the existing functionality and moved the tool into a new domain.

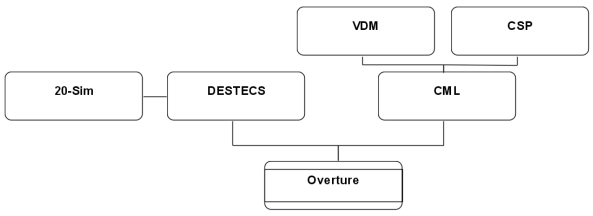


Fig. 2: Current Expansion Overture Tool

DESTECS The DESTECS project[1] is aimed at developing tool support for doing co-simulation between continuous time models and discrete event models [2]. The purpose is to aid the multidisciplinary development of embedded

[1] Design Support and Tooling for Embedded Control Software: http://www.destecs.org

real-time control systems in order to build more dependable systems. In the project, Overture has been extended to establish a link between discrete-event controllers defined in VDM and continuous-time models created in the tool 20-sim[2] using differential equations and the Bond graph notation.

CML CML is a formal notation focused on the complexities found in Systems of Systems (SoS). CML is developed as part of the COMPASS project[3] which is aimed at developing methods and tools to support the construction and analysis of models of SoS. CML is based on a combination between VDM and CSP and Overture is used as the foundation out of which the COMPASS tool will develop.

References

1. Battle, N.: VDMJ User Guide. Tech. rep., Fujitsu Services Ltd., UK (2009)
2. Broenink, J.F., Larsen, P.G., Verhoef, M., Kleijn, C., Jovanovic, D., Pierce, K., F., W.: Design support and tooling for dependable embedded control software. In: Proceedings of Serene 2010 International Workshop on Software Engineering for Resilient Systems. pp. 77–82. ACM (April 2010)
3. Fitzgerald, J., Larsen, P.G.: Modelling Systems – Practical Tools and Techniques in Software Development. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, Second edn. (2009), ISBN 0-521-62348-0
4. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object–oriented Systems. Springer, New York (2005)
5. Larsen, P.G., Hansen, B.S., et al.: Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language (December 1996), International Standard ISO/IEC 13817-1
6. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. ACM Software Engineering Notes 35(1) (January 2010)
7. Larsen, P.G., Lausdahl, K., Battle, N.: Combinatorial Testing for VDM. In: 8th IEEE International Conference on Software Engineering and Formal Methods, SEFM 2010 (September 2010)
8. Lausdahl, K., Lintrup, H.K.A., Larsen, P.G.: Connecting UML and VDM++ with Open Tool Support. In: Formal Methods 09. Springer-Verlag (November 2009), lNCS-5850
9. Mukherjee, P., Bousquet, F., Delabre, J., Paynter, S., Larsen, P.G.: Exploring Timing Properties Using VDM++ on an Industrial Application. In: Bicarregui, J., Fitzgerald, J. (eds.) Proceedings of the Second VDM Workshop (September 2000)
10. Nielsen, C.B., Lausdahl, K., Larsen, P.G.: Combining VDM with Executable Code. In: ABZ 2012, LNCS 7316. pp. 266–279. Springer, Heidelberg (2012)

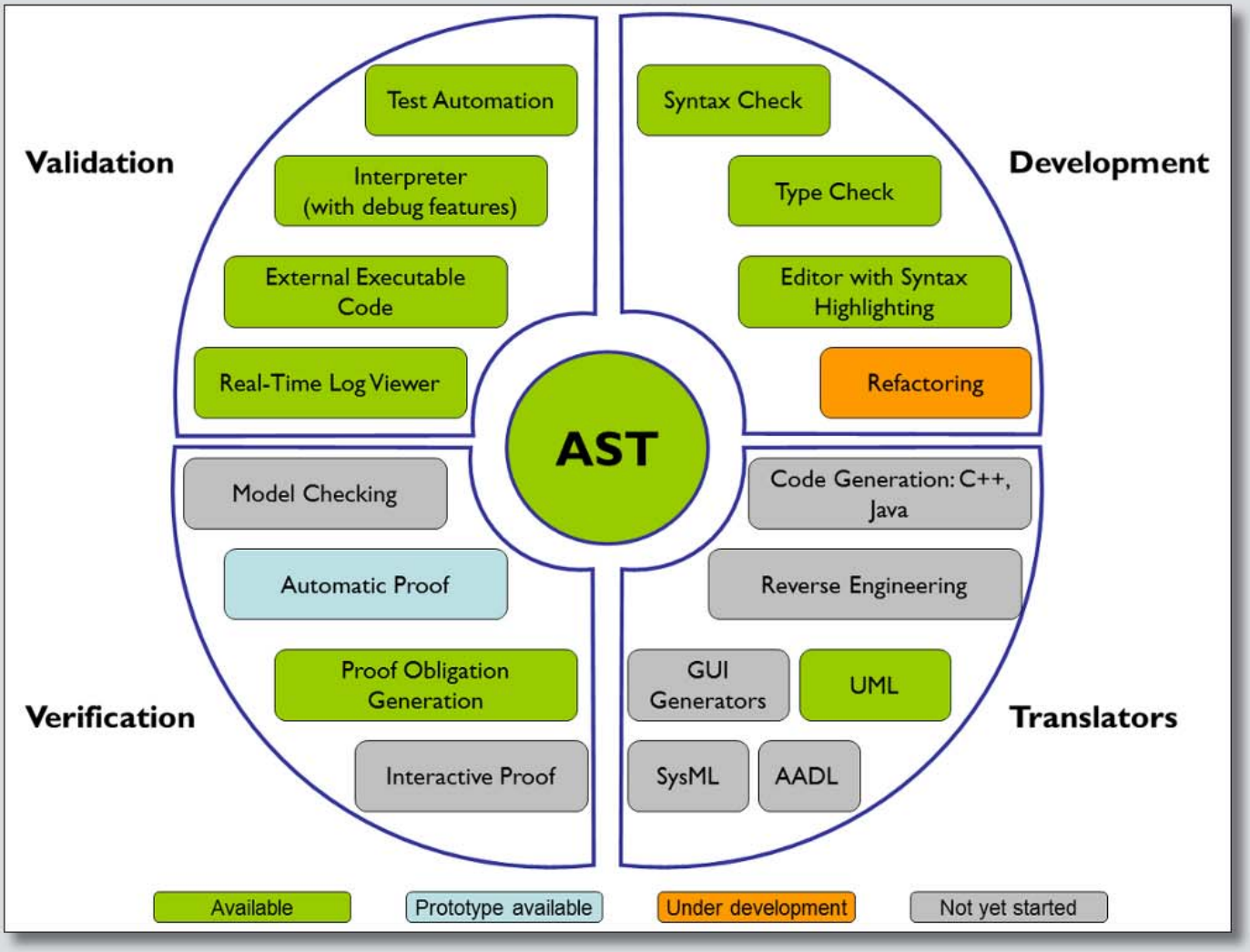
[2] 20-sim: http://www.20sim.com

[3] Comprehensive Modelling for Advanced Systems of Systems: http://www.compass-research.eu/

OVERTURE

A open platform for VDM

Overture is an Eclipse-based platform integrating existing industrial-strength VDM tools, and providing an open platform for new experimental extensions.



Parametric Analysis of Hybrid Systems Using HyMITATOR

Étienne André[1] and Ulrich Kühne[2]

[1]LIPN, CNRS UMR 7030, Université Paris 13, France
[2]Group for Computer Architecture, University of Bremen, Germany

Abstract. Hybrid automata are a powerful formalism for modeling and verifying hybrid systems, that combine continuous and discrete behavior. A common problem for hybrid systems is the good parameters problem, which consists in identifying a set of parameter valuations guaranteeing a certain good behavior of a system. We introduce here HyMITATOR, a tool for efficient parameter synthesis for hybrid automata, providing hybrid systems with a quantitative measure of their robustness.

Keywords: Hybrid automata, Verification, Parameter synthesis, Robustness

1 Motivation and History

Hybrid systems combine discrete and continuous behavior. This corresponds for instance to the discrete control logic and continuous physical variables in an embedded system. Hybrid automata are a popular and powerful model for such systems, where the continuous variables evolve according to ordinary differential equations in each control mode.

In [4], we proposed the *inverse method* for timed automata, a subclass of hybrid systems whose variables (named clocks) all have constant rates equal to 1. Different from CEGAR-based methods, this original semi-algorithm for parameter synthesis is based on a "good" parameter valuation (also named *point*) π_0 instead of a set of "bad" states. This method synthesizes a constraint K_0 on the parameters such that, for each parameter valuation π satisfying K_0, the trace set (i.e., the discrete behavior) of $\mathcal{A}$ under π is the same as for $\mathcal{A}$ under π_0. This preserves in particular linear time properties. This also provides the system with a criterion of robustness, in the sense that the resulting constraint gives a quantitative measure of the allowed "drift" such that the discrete behavior of the system is not impacted. By iterating the inverse method on all integer points within a bounded reference parameter domain, we get a set of constraints (or *tiles*) such that, for each parameter valuation in each such tile, the time-abstract behavior is the same: this gives a *behavioral cartography* of the system [5].

A basic implementation named IMITATOR (for *Inverse Method for Inferring Time AbstracT behaviOR*) has first been proposed, under the form of a Python script calling HyTech. The tool has then been entirely rewritten in IMITATOR 2.0 [3], under the form of a standalone OCaml program making use of the

Parma Polyhedra Library (PPL) [8]. A number of case studies containing up to 60 timing parameters could be efficiently verified in the purely timed framework. The latest version (2.5) includes stopwatches and arbitrary updates, and has been applied to several classes of scheduling problems [6].

The inverse method and the behavioral cartography have been extended to hybrid systems in [11]. Due to the strong syntactic and algorithmic differences between timed automata and hybrid automata, the work of [11] had to be implemented in an experimental "fork" of IMITATOR 2.0, and not in the main version. We present in this paper HyMITATOR, a now mature extension of that prototype, performing parameter synthesis on hybrid systems.

2 Implementation and Features

HyMITATOR takes as input a network of hybrid automata synchronized on shared actions. The input syntax, inspired by HyTech, allows the use of analog variables (such as time, velocity or temperature), rational-valued discrete variables, and parameters (i.e., unknown constants). The dynamics of the analog variables is described by ordinary differential equations. The tool directly supports linear dynamics, while affine dynamics can be approximated with arbitrary precision.

The core of the program is written in the object-oriented language OCaml, and interacts with PPL. Exact arithmetics with unbounded precision is used. A constraint is output in text format; furthermore, the set of traces computed by the analysis can be output under a graphical form (using Graphviz).

HyMITATOR implements the following algorithms for hybrid systems:

Full reachability analysis Given a model, it computes the set of symbolic reachable states.

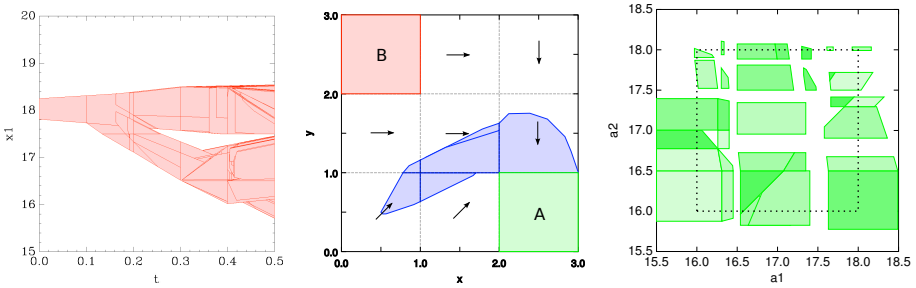
Predicate Abstraction Safety verification can alternatively be performed using a counterexample-guided abstraction refinement loop. The abstract state space is constructed w.r.t. a set of linear predicates [1].

Inverse method Given a model and a reference parameter valuation π_0, it computes a constraint on the parameter guaranteeing the same time-abstract behavior as under π_0 [4,11].

Behavioral cartography Given a model and a bounded parameter domain for each parameter valuation, it computes a set of constraints [5,11].

HyMITATOR uses several algorithmic optimizations, some of which have initially been developed for IMITATOR. In particular, the efficient merging technique presented in [7] has been successfully extended to the hybrid case: we merge any two states sharing the same discrete part (location and value of the discrete variables) and such that the union of their constraint on the analog variables and parameters is convex. This optimization preserves the correctness of all our algorithms; better, the constraint output by the inverse method in that case may be weaker, i.e., covers a larger set of parameter valuations.

For affine hybrid systems, further optimizations are needed. Due to the linear over-approximation by partitioning the state space, a lot of additional branching is introduced, which renders the inverse method ineffective. To solve this



(a) Reachable states (b) Trajectories (c) Behavioral cartography

Fig. 1. Examples of graphics output by HyMITATOR

problem, the algorithm has been extended as described in [11]. Basically, the partitioning is performed locally, and partitions belonging to the same discrete state are merged by taking their convex hull.

The post image computation can be costly for hybrid automata. To overcome this problem, an abstraction technique for the verification of simple safety properties (non-reachability of bad states) has been presented in [1]. Based on a set of linear predicates, reachability is performed on the abstract state space induced by these predicates. Refinement can be performed by discovering separation planes. While the original method is based on flow-pipe construction, we adapted the algorithm to the linear approximation by state space partitioning.

The behavioral cartography has been adapted to the framework of hybrid systems. Different from timed automata, hybrid automata do not restrict coefficients appearing in clock constraints to be integers, and allow variables to be compared with any rational value. For this reason, instead of considering only integer points as starting points for the inverse method, an arbitrary rational step size can be used for each parameter dimension in HyMITATOR. This gives more accurate results, by reducing the size of the possible "holes" not covered by any tile of the cartography.

HyMITATOR (with sources, binaries and case studies) is available on its Web page: http://www.lsv.ens-cachan.fr/Software/hymitator/.

3 Applications

HyMITATOR can be used for the *parametric verification* of hybrid systems. An application to sampled data hybrid systems has been presented in [11]. As a special case, such systems can be parametrized over the initial states. Then, a single run satisfying a desirable reachability property can be generalized to a larger set of initial states. As an example, Figure 1(a) shows the enlarged reachable states of a single run for the *room heating benchmark* from [9]. This also proves the *robustness* of the system w.r.t. the tested property. Figure 1(b) shows an over-approximation of the reachable states for the *navigation benchmark* [9], proving that all trajectories will eventually enter the green target zone.

Another problem that can be addressed using HyMITATOR is *test coverage*. In order to ensure the quality of an implementation of a hybrid system, a set of tests is generated which is then applied to the system. However, since the state space of hybrid systems is infinite in general, it is hard to decide when enough tests have been performed. Using the inverse method, a tile (dense set of points) around each test point is generated which entails the same discrete behavior. This means that any point in this tile can be considered covered. Figure 1(c) shows the coverage of a parameter rectangle for the room heating benchmark.

4 Related Work

One of the first powerful model checkers for analyzing hybrid automata is HyTech [12]. Unfortunately, it can hardly verify even medium sized examples due to exact arithmetics with limited precision and static composition of automata, quickly leading to memory overflows. The tool PHAVer [10] improves on the computation of the reachable states by using efficient over-approximations. Techniques similar to those in PHAVer have also been implemented in HyMITATOR, with additional algorithmic improvements. The work in [2] presents an analysis on Simulink models which shares similar goals with our approach.

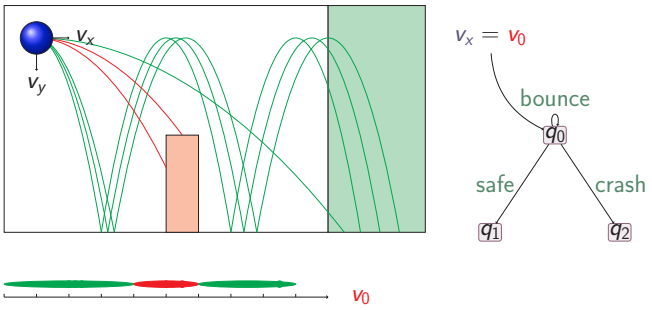
References

1. R. Alur, T. Dang, and F. Ivančić. Predicate abstraction for reachability analysis of hybrid systems. *ACM Trans. Embedded Computing Systems*, 5:152–199, 2006.
2. R. Alur, A. Kanade, S. Ramesh, and K. C. Shashidhar. Symbolic analysis for improving simulation coverage of simulink/stateflow models. In *EMSOFT'08*, pages 89–98. ACM, 2008.
3. É. André. IMITATOR II: A tool for solving the good parameters problem in timed automata. In *INFINITY'10*, volume 39 of *EPTCS*, pages 91–99, 2010.
4. É. André, T. Chatain, E. Encrenaz, and L. Fribourg. An inverse method for parametric timed automata. *International Journal of Foundations of Computer Science*, 20(5):819–836, 2009.
5. É. André and L. Fribourg. Behavioral cartography of timed automata. In *RP'10*, volume 6227 of *LNCS*, pages 76–90. Springer, 2010.
6. É. André, L. Fribourg, U. Kühne, and R. Soulat. IMITATOR 2.5. Available at www.lsv.ens-cachan.fr/Software/imitator/.
7. É. André, L. Fribourg, and R. Soulat. Enhancing the inverse method with state merging. In *NFM'12*, volume 7226 of *LNCS*, pages 100–105. Springer, 2012.
8. R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.
9. A. Fehnker and F. Ivancic. Benchmarks for hybrid systems verification. In *HSCC'04*, volume 2993 of *LNCS*, pages 326–341. Springer, 2004.
10. G. Frehse. PHAVer: Algorithmic verification of hybrid systems past HyTech. *Software Tools for Technology Transfer*, 10(3):263–279, 2008.
11. L. Fribourg and U. Kühne. Parametric verification and test coverage for hybrid automata using the inverse method. In *RP'11*, volume 6945 of *LNCS*, pages 191–204. Springer, 2011.
12. T. A. Henzinger, P. H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:460–463, 1997.

HyMITATOR
Parametric Analysis of Hybrid Systems

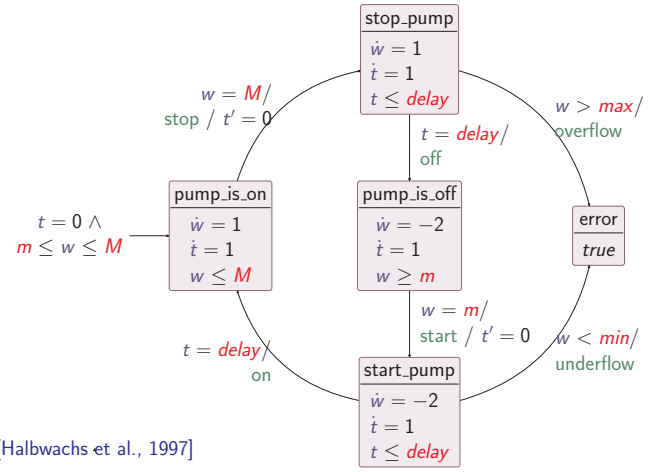
Étienne André[1] and Ulrich Kühne[2]
[1]LIPN, CNRS UMR 7030, Université Paris 13, France
[2]Group for Computer Architecture, University of Bremen, Germany

An Example of Hybrid System: The Bouncing Ball



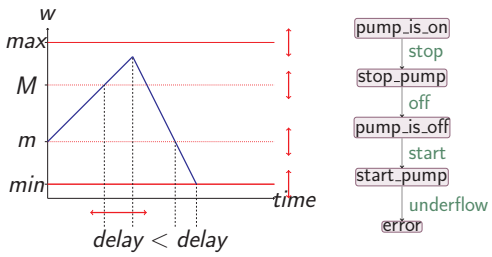
Parameterized Hybrid Automata

- Hybrid Automata (HA): Set of variables, actions, locations (with an activity and an invariant), and discrete transitions (with jumps).
- Parameterized Hybrid Automata: HA augmented with a set of timing parameters (unknown constants)
- Example: Water Tank



[Halbwachs et al., 1997]

The Parameter Synthesis Problem

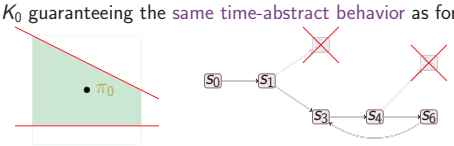


- How to choose min, max, m, M and delay, such that always $min < w < max$?

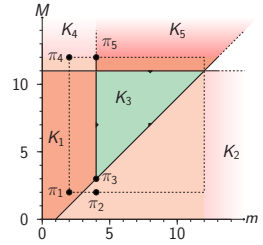
Synthesis problem: "find values for the timing parameters such that the system behaves well".

Parameter Synthesis for Hybrid Automata

- Inverse Method [Fribourg and Kühne, 2011]
  - Given a HA and a reference valuation π_0 for the parameters, synthesize a constraint K_0 guaranteeing the same time-abstract behavior as for π_0



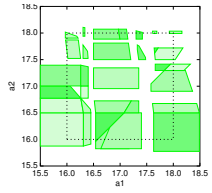
  - K_0 obtained by iterative removal of states incompatible with π_0
- Behavioral Cartography [André and Fribourg, 2010]
  - Performs a tiling of the parametric space, and partition it between good and bad tiles w.r.t. a given property

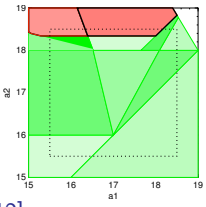


Example of "good" constraint for the water tank:
$M + delay \geq m \wedge m \geq min + 2 \cdot delay \wedge max \geq M + delay$

Features of HyMITATOR

- Algorithms of Parameter Synthesis for Hybrid Systems
  - Implements the inverse method and the behavioral cartography
  - Includes local partitioning with linear over-approximations
  - Makes use of predicate abstraction techniques
  - Features an efficient merging technique [André et al., 2012]
- User-friendly Features
  - Numerous options for analysis
  - Graphical output





- Implementation [André and Kühne, 2012]
  - Implemented in OCaml, using the Parma Polyhedra Library

Try it!

- Distributed under the GNU General Public License
- `www.lsv.ens-cachan.fr/Software/hymitator/`

References

André, É. and Fribourg, L. (2010).
Behavioral cartography of timed automata.
In *RP'10*, volume 6227 of *LNCS*, pages 76–90. Springer.

André, É., Fribourg, L., and Soulat, R. (2012).
Enhancing the inverse method with state merging.
In *NFM'12*, volume 7226 of *LNCS*, pages 100–105. Springer.

André, É. and Kühne, U. (2012).
Parametric analysis of hybrid systems using HyMITATOR.
In *iFM'12*.

Fribourg, L. and Kühne, U. (2011).
Parametric verification and test coverage for hybrid automata using the inverse method.
In *RP'11*, volume 6945 of *LNCS*, pages 191–204. Springer.

Halbwachs, N., Proy, Y.-E., and Roumanoff, P. (1997).
Verification of real-time systems using linear relation analysis.
In *Formal Methods In System Design*, pages 157–185.

Debugging Abstract State Machine Specifications: An Extension of CoreASM

Marcel Dausend, Michael Stegmaier and Alexander Raschke

Institute of Software Engineering and Compiler Construction,
University of Ulm, Germany
{marcel.dausend, michael-1.stegmaier, alexander.raschke}@uni-ulm.de

Abstract. We introduce a debugger component as an extension of CoreASM to simplify validation of (complex) ASM specifications. As a basis, we map well-known debugging concepts of imperative programs to the ASM context. The architecture of our debugger is described and some background information to the implementation is given. We conclude by summarizing the current functionalities of our debugger and outlining further development prospects.

Keywords: Abstract State Machines, CoreASM, Debugging

1 Introduction

Creating and editing different kinds of specifications are key tasks that have to be done throughout the system development process. An accepted executable formalism for the specification of hard- and software systems are Abstract State Machines (ASMs) [1]. ASMs have been used to describe, verify and validate complex formal languages, especially their semantics, e. g. Java and its virtual machine [6] or comprehensive parts of the Unified Modeling Language [5].

A major problem of complex specifications is their maintainability and comprehensibility. In case of ASMs, several methodologies and tools have been developed to support defining, editing, validating, and verifying ASM specifications. These tools differ in their support of ASM concepts and focus on specific application issues [1]. One of these tools is CoreASM. Amongst others, it provides a flexible plugin architecture and an interpreter for ASMs [2].

Debugging is a common method to "identify and remove errors from (computer hardware or software)" [1] and to comprehend specifications. Our approach is to extend CoreASM with a debugging component so that multi-agent ASM specifications can be revised more easily.

In Sect. 2, we clarify our notion and capabilities of debugging ASMs and give a brief overview of existing tools and their concepts for debugging of ASMs. We then explain how debugging concepts for imperative programs can be mapped to concepts for debugging ASM specifications. In Sect. 3, we briefly show existing debugging features of CoreASM before we describe our extension of CoreASM. In Sect. 4, we summarize the current status of our extension and outline our next steps and future work.

[1] definition of *debug* from http://oxforddictionaries.com/definition/debug

2 Debugging Abstract State Machines

According to [4], we consider debugging as an interactive process, where a running instance of a program can be stepwise observed and the program execution can be controlled by the user. This observation provides opportunities to comprehend and deeply understand the program and finally revise it, if necessary.

Debugging of ASMs has been addressed formerly by the tools ASMGofer and XASM [1]. Both tools enable you to control an ASM execution by starting, pausing, resuming, and stopping. They offer break conditions to automatically pause an ASM execution. If the execution is paused, both tools allow to investigate the status of ASM functions. CoreASM itself does not support debugging as defined in the previous paragraph, which has been indicated as an open issue [2].

2.1 From debugging of imperative programs towards debugging of ASM specifications

In order to enable fine grained control to debug an ASM execution, an execution **step** has to be defined. Whereas a step in an imperative program means setting the program counter from the current instruction to the following instruction, a step in terms of ASMs means evaluating the machine's program (or Agent programs) and applying the resulting update set to the current state of the ASM. Thus, we use this definition as a debugging step. We do not yet take into account microsteps, which are hidden inside a turbo ASM step (cf. [1], p.174).

A **breakpoint** is a clearly defined point in a program, where the execution stops if this point of the program is reached. We consider different kinds of breakpoints: (line) breakpoints, watchpoints, and method breakpoints.

In an imperative program, a (line) breakpoint is reached if the program counter hits a statement (contained in the line of code) which is marked by a breakpoint. In an ASM, a **(line) breakpoint** is reached, if the marked statement causes an update that is contained within the ASM's update set at the end of the current step. Thereby, it is possible that multiple breakpoints are reached at the same time, which is not possible in an imperative program.

A watchpoint in an imperative program marks a declaration of a variable. The watchpoint is hit if this variable is modified or read in the current step. In an ASM, we define a **watchpoint** as a breakpoint marking either a universe declaration or a function declaration. This includes variable declarations, which are functions of arity zero. The breakpoint is reached if a marked universe or function is changed by any update of the current update set.

Method breakpoints in an imperative program mark the head of a method declaration and are reached if this method is invoked. In ASMs we have macro rules instead of methods, so a **method breakpoint** marks the head of an ASM rule. Instead of stopping the ASM execution when invoking the rule, the breakpoint is reached if at least one statement inside the rule's body causes an update which is contained in the current update set.

Another debugging concept for imperative programs is called "watch expression". A watch expression is a well formed expression of the programming

language. It can be defined as part of the debugging environment so that its current result can be evaluated during the program execution. We define a **watch expression** in ASMs as either a function name or a function name including parameters. The values of all locations of the given function or the value of the given location can be observed at each update step of the ASM.

A **Modification** allows to change a function at a given location when the execution is paused.

3 Architecture and implementation

CoreASM implements different aspects of multi-agent ASM using a flexible plugin based architecture. For example, both, basic ASM and turbo ASM, are implemented as separate plugins.

For the purpose of simple debugging, CoreASM offers the plugin *DebugInfoPlugin*. It allows adding output statements, which are assigned to user defined channels, to a specification. By configuring a set of channels it can be defined which debug info statements are considered for output during the execution.

Additional information, like the current status of an execution, its selected agents, and the current update, can be displayed on the console, but stepwise execution is not possible.

We enhance debugging functionalities of CoreASM using the Eclipse Debug Project (EDP)[2] to implement the concepts introduced in Sect. 2.1. As a basis for debugging we introduce a stepping mode. This mode forces the interpreter to execute exactly one update step and pause the execution afterwards. The user can toggle between the stepping mode and the running mode.

Since our implementation is based on EDP, it is possible to run ASM specifications in *debug mode* and provide a *debug perspective* with views to manage breakpoints, to inspect and modify variables, and to define and inspect expressions. Furthermore, breakpoints can be set or removed directly within the editor. Entries that have been changed in the current step are highlighted to simplify the inspection of updates. The current number of steps and the currently selected agents are displayed at the top of the variables view by default.

In addition to the EDP views, we provide an update view showing all updates of the current update set for a user defined set of agents. Every entry of the update view provides information about the statement which causes the update, its source file and line number, and its executing agent. Inside the update view, all entries of updates matching a breakpoint are highlighted by a special symbol to ease inspection.

The implementation extends the engine driver to enable control of the CoreASM program executions. The Control API of CoreASM provides the information about the current status of the interpreter [3]. This information is used to update the views of our CoreASM debugger after each step.

An ASM specification running in debug mode considers all types of breakpoints (cf. Sect. 2.1) and automatically pauses if any breakpoint is reached.

[2] `http://www.eclipse.org/eclipse/debug/index.php`

4 Conclusion and future work

This work provides the basis for a systematic investigation of complex ASM specifications and opens opportunities to revise them. Our debugger extends CoreASM mainly by using the EDP (cf. Sect. 3) which is an integral part of the Eclipse environment.

Since "Traditional debugging models [...] do not suite ASMs." [2] we propose an adaptation of imperative debugging concepts for the state machine domain of ASMs (cf. Sect. 2.1). In particular, the user interface provided by EDP was adapted and extended to visualize changes of the state of an ASM execution. A new view provides a list containing all updates of a step and allows direct access to its corresponding ASM statements.

Although the debugger is already helpful, there are plenty of possibilities for further extensions. Some ASM constructs are not yet supported: our current definition of a step neglects sequential steps. Derived functions and local rules cannot be debugged, because they do not cause updates which could be observed via the interpreter. We are working on a way to support these constructs.

Additionally, we plan to introduce a history that enables stepping backwards to compare states of an ASM execution, show their differences, and to trace rule calls of specific agents. Furthermore, a record and replay functionality could be used to eliminate non-determinism (e. g. `choose` or the selection of agents for a specific step) in order to enable debugging of an ASM specification repeatedly under the same conditions.

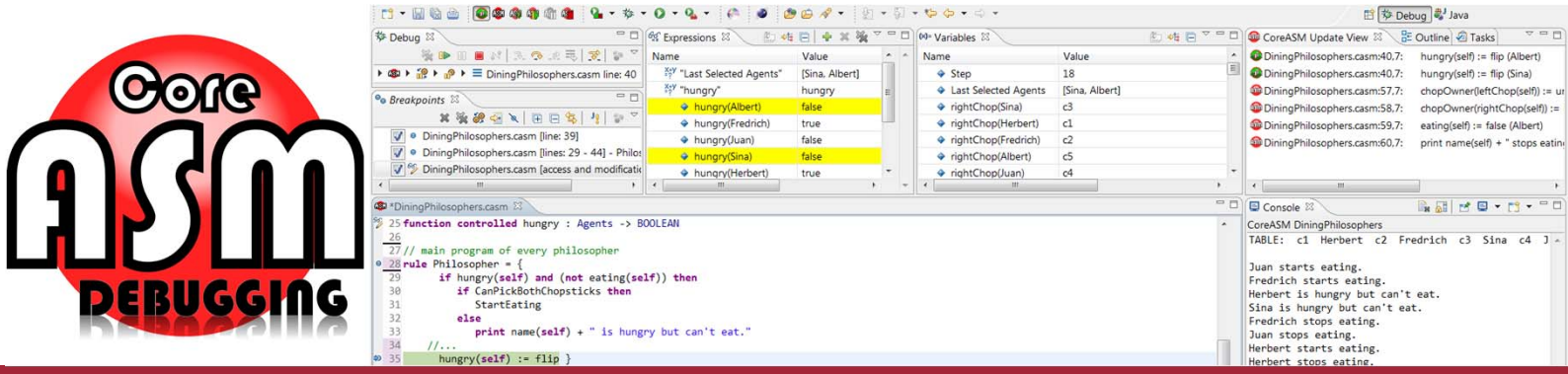
More information about our project and its current status can be found at our website `http://www.uni-ulm.de/en/in/pm/research/projects/coreasm`.

Acknowledgments Thanks to Roozbeh Farahbod for answering numerous questions, trying out the tool, and suggesting some further improvements.

References

1. E. Börger and R. Stärk. *Abstract State Machines – A Method for High-Level System Design and Analysis*. Springer, 2003.
2. R. Farahbod. *CoreASM: An Extensible Modeling Framework & Tool Environment for High-level Design and Analysis of Distributed Systems*. PhD thesis, Simon Fraser University, Burnaby, Canada, 2009.
3. R. Farahbod, V. Gervasi, U. Glässer, and G. Ma. CoreASM Plug-In Architecture. In *Rigorous Methods for Software Construction and Analysis*, pages 147–169, 2009.
4. J. Henkel and A. Diwan. A Tool for Writing and Debugging Algebraic Specifications. In *Proceedings. of the 26th ICSE 2004*, pages 449–458, 2004.
5. J. Kohlmeyer and W. Guttmann. Unifying the Semantics of UML 2 State, Activity and Interaction Diagrams. *Perspectives of Systems Informatics*, 5947:206–217, 2010.
6. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine - Definition, Verification, Validation*. Springer, 2001.

ulm university universität ulm



Debugging Abstract State Machine Specifications

An Extension of CoreASM

Introduction

Although debugging is an integral part of the implementation of software, it is just roughly supported by current Abstract State Machine (ASM) tools.

In order to simplify the validation of (complex) ASM specifications we extend CoreASM [Farahbod2009] by a debugger.

Functionalities

- the control mode "stepping" for the interpreter
- capabilities to debug CoreASM programs based on the Eclipse Debug Project (EDP)
  - line breakpoints
  - watchpoints
  - rule breakpoints (cf. method breakpoints)
  - watch expressions
  - variables view
  - expressions view
  - breakpoint view
- extensions that go beyond EDP
  - updates view
  - agent filter for updates view

Fig. 1 The stepping mode button of the CoreASM control bar

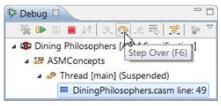


Fig. 2 The standard debug control of Eclipse

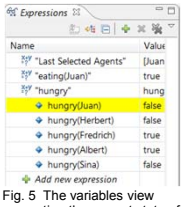


Fig. 5 The variables view presenting the current state of the CoreASM execution

Architecture and Implementation

Architecture and implementation of our debugging component are based on the EDP as a basis for user defined integrated Eclipse debuggers.

The Control API of CoreASM provides information about the current status of the interpreter [Farahbod, Gervasi et al. 2004].

We prepare the conceptual basis for the implementation of the debugger through the transfer of concepts of debugging of imperative programs to concepts of debugging ASM specifications.

Therefore, we characterize the following debugging concepts in terms of ASM: a debugging *step*, a *line breakpoint*, a *watchpoint*, a *method breakpoint*, a *watch expression*, and *modification of data*.

An Example of Using the CoreASM Debugger

As an example, we debug a slightly modified version of the CoreASM sample specification "Dining Philosophers".
A debug execution of CoreASM can be controlled either by using the *extended CoreASM controls* (Fig. 1) or by using the standard eclipse *debug control* (Fig. 2).

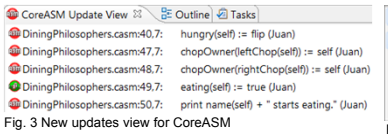


Fig. 3 New updates view for CoreASM

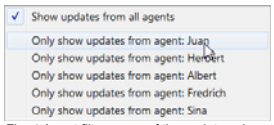


Fig. 4 Agent filter menu of the updates view

The updates of the last execution step are presented within the *update view* (see Fig. 3). Green *highlighted entries indicate* an update which is currently hit by a breakpoint.
A *filter* can be used to focus on a specific agent (see Fig. 4).

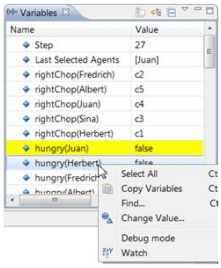


Fig. 6 The expressions view of Eclipse

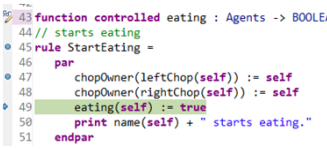


Fig. 7 Editor component with indicators for corresponding breakpoints and the last update of the current update set

The *variables view* (Fig. 5) and the *expressions view* (Fig. 6) can be used to examine the current state of the CoreASM execution.

The main component of CoreASM is the *editor* (Fig. 7).

- A *watchpoint* (see Fig. 7, l. 43) interrupts the interpretation if the marked function at any given location has been changed during the current execution step.
- A *method breakpoint* (rule breakpoint; Fig. 7, l. 45) is hit if any update is caused by any statement within the rules' body.
- A *line breakpoint* (Fig. 7, l. 47) causes the interpreter to pause if a statement of the marked line triggers an update within the current update set.
- The line containing the last update in the current update set is marked by an *indicator* (blue arrow; see Fig. 7, l. 49).

References

[Farahbod 2009] R. Farahbod. *CoreASM: An Extensible Modeling Framework & Tool Environment for High-level Design and Analysis of Distributed Systems*. PhD thesis, Simon Fraser University, Burnaby, Canada, 2009.

[Farahbod, Gervasi et al. 2004] R. Farahbod, V. Gervasi, U. Glässer, and G. Ma. *CoreASM Plug-In Architecture*. In Rigorous Methods for Software Construction and Analysis, pages 147-169, 2009.

Contact

Marcel Dausend marcel.dausend@uni-ulm.de
Alexander Raschke alexander.raschke@uni-ulm.de
Michael Stegmaier michael-1.stegmaier@uni-ulm.de

Project Site
http://www.uni-ulm.de/en/in/pm/research/projects/coreasm/

The Variability Model Checker VMC

Maurice H. ter Beek
ISTI–CNR, Pisa, Italy

1 Introduction

We demonstrate an experimental tool for modelling and analysing (behavioural) variability in product families modelled as Modal Transition Systems (MTSs) [8]. A product family is a compact way for describing different products through their commonalities and variabilities (often defined by *features*). An MTS is a Labelled Transition System (LTS) distinguishing *optional* (may) and *mandatory* (must) transitions [1]. In [6], MTSs were recognized as a formal method for describing the possible operational behaviour of a family's products. The standard derivation methodology for obtaining a product from an MTS modelling a product family is as follows: include all (reachable) must transitions and a subset of the (reachable) may transitions. Each selection is a product (i.e., an LTS). Unfortunately, MTSs cannot model all common variability constraints. The solution adopted in [2, 3] is to enrich an MTS description with a set of constraints defining which of the standardly derivable products should be considered as acceptable valid products. In [2], an appropriate variability and action-based branching-time temporal logic to formalize these constraints is defined, while [3] contains an algorithm to derive all and only LTSs describing valid products. This methodology is implemented in VMC and in this paper we demonstrate how to use VMC by means of the following simple and intuitive case study from [2, 3].

2 Case Study: A Product Family

A family of coffee machines has the following initial list of informal requirements:

1. *Initially, a coin must be inserted: either a euro, exclusively for European products, or a dollar, exclusively for Canadian products*;
2. *After inserting a coin, the user has to choose whether (s)he wants sugar, by pressing one of two buttons, after which (s)he may select a beverage*;
3. *The choice of beverage (coffee, tea, cappuccino) varies, but all products must offer coffee while only European products may offer cappuccino*;
4. *Optionally, a ringtone may be rung after delivering a beverage. However, a ringtone must be rung in all products offering cappuccino*;

We model all valid product behaviour by the MTS of Fig. 1(l) and the additional constraints: (i) euro and dollar are *alternative*; (ii) dollar *excludes* cappuccino; (iii) cappuccino *requires* ring a tone. Note that constraint (iii) cannot invalidate a product ringing a tone before delivering cappuccino; the behavioural description of a product (family) as provided by an LTS (MTS) must impose such orderings.

3 Encoding and Analyzing Product Families with VMC

VMC [8] is part of a family of on-the-fly model checkers developed at ISTI–CNR over the last two decades for verifying logic formulae in an action- and state-based branching-time temporal logic derived from the family of logics based on classic CTL, including FMC [7], UMC [4], and CMC [5].

VMC in particular is derived from FMC. Beyond interactively exploring an MTS, model checking properties over an MTS, and visualizing the interactive explanations of a verification result, VMC furthermore allows the generation of

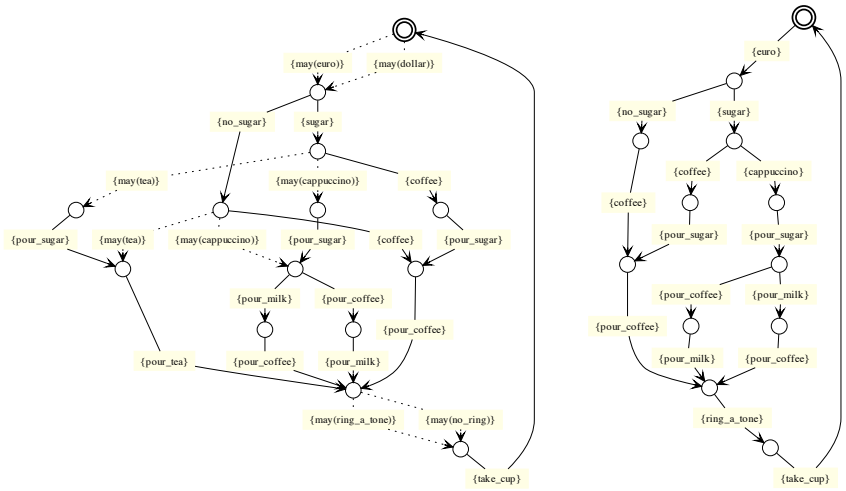


Fig. 1. MTS of coffee machine family (l) and an apparently valid European product (r) as generated by VMC; dashed edges labelled may(·) are may transitions, the others must

all valid products (according to the given constraints) of an MTS describing a product family and the verification of properties over each valid product.

VMC accepts as input a textual process-algebraic encoding of an MTS and a set of constraints of the form ALTernative, EXCludes, REQuires, and IFF (hiding their logic formalization given in [2]). The distinction among may and must transitions is encoded in the resulting LTS by typing action labels corresponding to may transitions as `may(·)`. The MTS in Fig. 1(l) modelling the coffee machine family from the case study was generated by VMC from the following encoding (the associated set of constraints is only taken into account for product generation):

```
T1 = may(euro).T2 + may(dollar).T2
T2 = sugar.T3 + no_sugar.T4
T3 = coffee.T5 + may(cappuccino).T6 + may(tea).T7
T4 = coffee.T8 + may(cappuccino).T9 + may(tea).T10
T5 = pour_sugar.T8
T6 = pour_sugar.T9
T7 = pour_sugar.T10
T8 = pour_coffee.T13
T9 = pour_coffee.T11 + pour_milk.T12
T10 = pour_tea.T13
T11 = pour_milk.T13
T12 = pour_coffee.T13
T13 = may(ring_a_tone).T14 + may(no_ring).T14
T14 = take_cup.T1
```

```
net SYS = T1

Constraints {
  euro ALT dollar
  dollar EXC cappuccino
  cappuccino REQ ring_a_tone
  ring_a_tone ALT no_ring
}
```

The variability logic defined in [2] can be directly encoded in the logic accepted by VMC by considering the typed actions. This latter logic contains the classic box and diamond modal operators $[\,], \langle\,\rangle$, the classic existential and universal state operators E, A (quantifying over paths), and action-based versions of the CTL until operators W, U (resulting also in an action-based version of the 'eventually' operator F). Using VMC it is thus possible to specify and verify properties which are surely preserved in all products by checking them over the family MTS:

(1) *The MTS guarantees that if a euro or dollar action occurs, afterwards for all standardly derivable products it is eventually possible to reach action coffee.*

$$[may(euro)\ or\ may(dollar)]\ E\,[true\ \{not\ may(*)\}\,U\,\{coffee\}\ true]$$

This formula prohibits a path leading to coffee to contain *any* (i.e. $*$) may transition (beyond the initial one). Asked to model check it over the MTS of Fig. 1(l), VMC reports it holds and offers the possibility to explain this result (cf. Fig. 2).

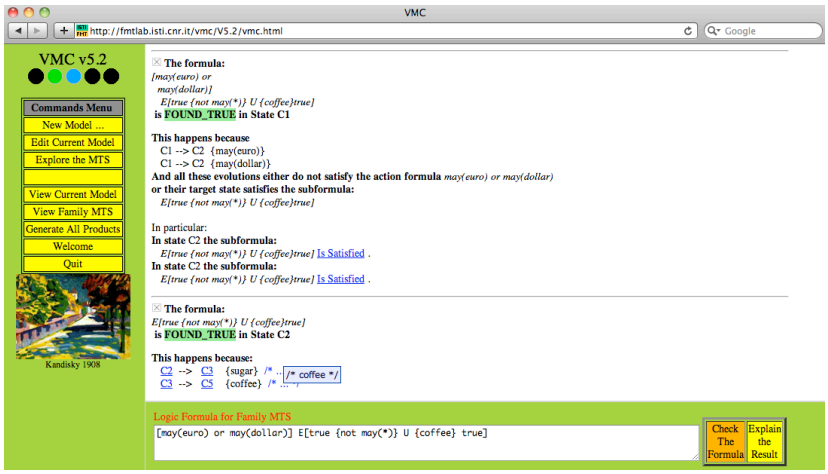


Fig. 2. Explanation of model checking Property 1 over MTS of Fig. 1(l) in VMC

4 Generating and Analyzing Valid Products with VMC

Beyond generating all valid products (LTSs), VMC thus allows browsing them, verifying whether they satisfy a certain property (logic formula), and understanding why a specific valid product does (not) satisfy the verified property. To do so, VMC allows to open for each product a new window with its textual encoding.

Suppose we generate all valid products in VMC and then check for each one:

(2) *If it is possible to obtain a sugared cappuccino, then it is also possible to obtain an unsugared cappuccino.*

$$(EF\,\langle sugar\rangle\,\langle cappuccino\rangle\,true)\ implies\ EF\,\langle no\_sugar\rangle\,\langle cappuccino\rangle\,true$$

Property 2 does not hold for all valid products, revealing ambiguous constraints: the one of Fig. 1(r) satisfies all constraints but offers cappuccino only with sugar. A way to solve such ambiguity is to refine actions by explicitly distinguishing sugared from unsugared ones and to extend the constraints accordingly (cf. Fig. 3).

```
T1 = may(euro).T2 + may(dollar).T2
T2 = sugar.T3 + no_sugar.T4
T3 = sugared_coffee.T5 + may(sugared_cappuccino).T6
   + may(sugared_tea).T7
T4 = unsugared_coffee.T8 + may(unsugared_cappuccino).T9
   + may(unsugared_tea).T10
T5 = pour_sugar.T8
T6 = pour_sugar.T9
T7 = pour_sugar.T10
T8 = pour_coffee.T13
T9 = pour_coffee.T11 + pour_milk.T12
T10 = pour_tea.T13
T11 = pour_milk.T13
T12 = pour_coffee.T13
T13 = may(ring_a_tone).T14 + may(no_ring).T14
T14 = take_cup.T1
net SYS = T1

Constraints {
   euro ALT dollar
   sugared_tea IFF unsugared_tea
   sugared_cappuccino IFF unsugared_cappuccino
   dollar EXC sugared_cappuccino
   sugared_cappuccino REQ ring_a_tone
   ring_a_tone ALT no_ring }
```

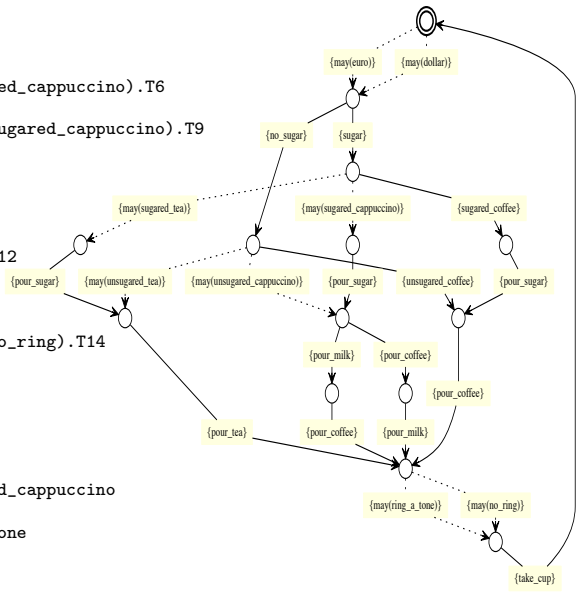


Fig. 3. MTS of a refined coffee machine family: input (l) and output (r) of VMC

From this refined family/MTS, VMC generates the 10 valid products/LTSs depicted in Fig. 4 (listing moreover which of the optional actions they contain), over which VMC can subsequently model check whether all European products offer both sugared and unsugared cappuccino by means of the formula written in Fig. 4 (recall that cappuccino is an optional feature, even for European products).

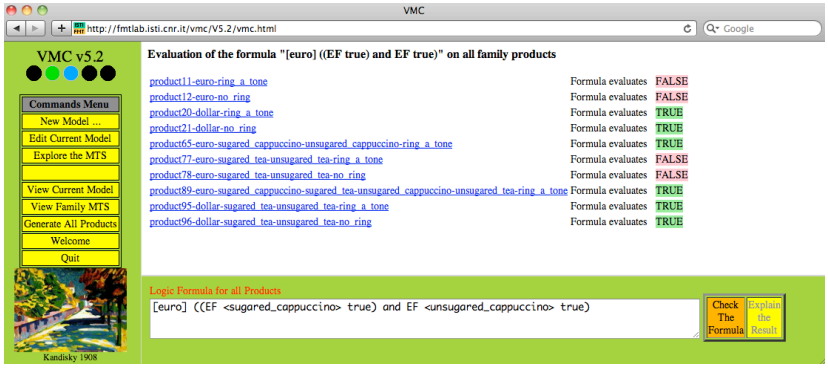


Fig. 4. All valid products/LTSs of the family/MTS of Fig. 3 as generated by VMC and the result of model checking a variant of Property 2 over all valid coffee machines

By clicking on a product, VMC displays it in a new window for further analysis.

Likewise, VMC can verify whether all valid products offer both sugared and unsugared coffee; indeed, it states that the following formula holds for all 10 products:

$$[euro]\,((EF\,\langle sugared\_coffee\rangle\,true)\;and\;EF\,\langle unsugared\_coffee\rangle\,true)$$

The reader is invited to use VMC: the case study is available from the examples.

5 Getting Acquainted with VMC

VMC's core contains a command-line version of the model checker and a product generation procedure, both stand-alone executables written in Ada (easy to compile for Windows/Linux/Solaris/MacOSX) and wrapped with a set of CGI scripts handled by a web server, facilitating a graphical html-oriented GUI and integration with other tools for LTS minimization and graph drawing. Its development is ongoing, but a prototype version is used at ISTI–CNR for academic purposes. VMC is publicly usable online [8] and its executables are available upon request.

Acknowledgements. To P. Asirelli, A. Fantechi, and S. Gnesi for joint research that led to VMC and to F. Mazzanti and A. Sulova for the development of VMC.

References

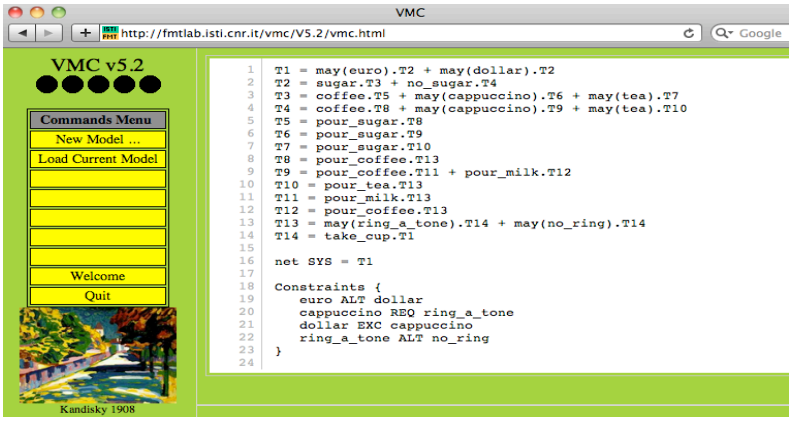
1. A. Antonik, M. Huth, K.G. Larsen, U. Nyman, A. Wąsowski. 20 Years of modal and mixed specifications. B. EATCS 95 (2008), 94–129.
2. P. Asirelli, M.H. ter Beek, A. Fantechi, S. Gnesi. A Logical Framework to Deal with Variability. In *IFM'10*, LNCS 6396, Springer, 2010, 43–58.
3. P. Asirelli, M.H. ter Beek, A. Fantechi, S. Gnesi. Formal Description of Variability in Product Families. In *SPLC'11*, IEEE, 2011, 130–139.
4. M.H. ter Beek, A. Fantechi, S. Gnesi, F. Mazzanti. A state/event-based model-checking approach for the analysis of abstract system properties. *Sci. Comput. Program.* 76, 2 (2011), 119–135.
5. A. Fantechi, A. Lapadula, R. Pugliese, F. Tiezzi, S. Gnesi, F. Mazzanti. A Logical Verification Methodology for Service-Oriented Computing. *ACM TOSEM* 21, 3 (2012).
6. D. Fischbein, S. Uchitel, V.A. Braberman. A foundation for behavioural conformance in software product line architectures. In *ROSATEA'06*, ACM, 2006, 39–48.
7. S. Gnesi, F. Mazzanti. On the Fly Verification of Networks of Automata. In *PDPTA'99*, CSREA Press, 1999, 1040–1046.
8. F. Mazzanti, A. Sulova. VMC v5.2. `http://fmtlab.isti.cnr.it/vmc/V5.2/`

The Variability Model Checker VMC

Maurice H. ter Beek (Formal Methods && Tools Lab, ISTI-CNR, Pisa, Italy)

VMC is an experimental tool for modelling and analysing (behavioural) variability in product families modelled as Modal Transition Systems (MTSs), written by F. Mazzanti and A. Sulova

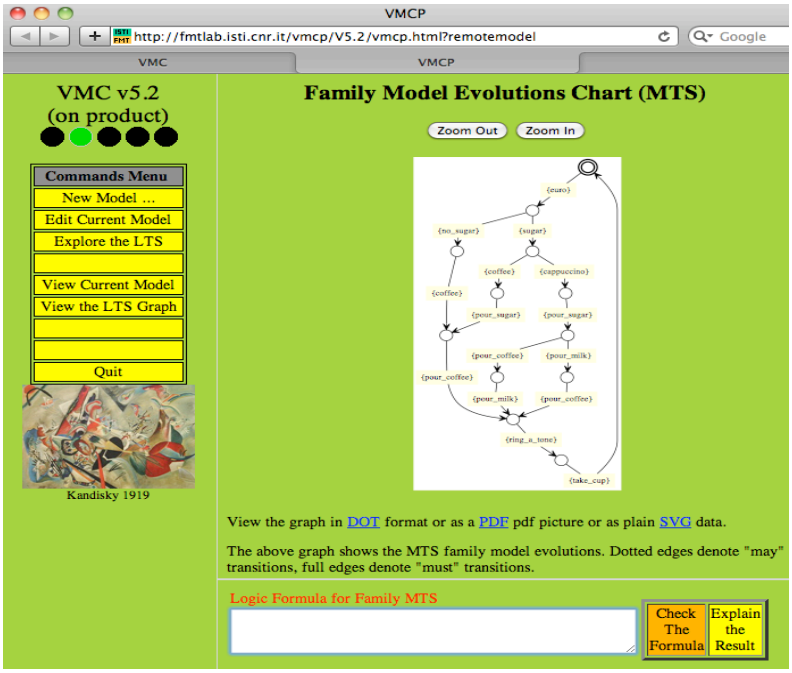
1 VMC accepts as input a textual process-algebraic encoding of an MTS and a set of constraints of the form ALTernative, EXCludes, REQuires, and IFF



VMC can also visualize a product family/MTS (cf. 5)

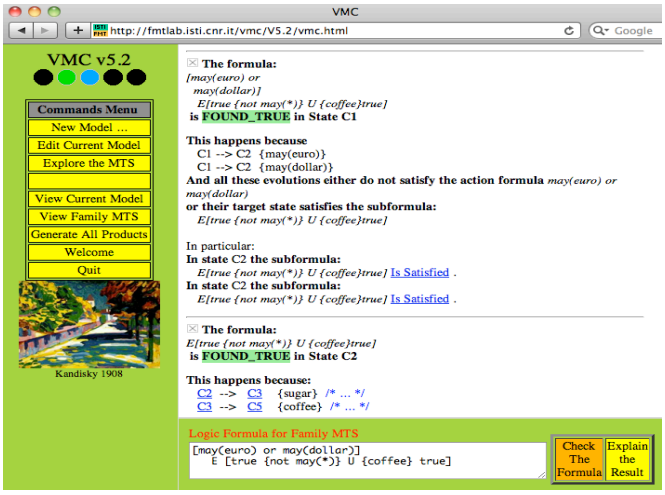
3 VMC can derive all valid products (LTSs) of a family (MTS + constraints) for further analyses (cf. 4)

5 VMC can help understand why a valid product does (not) satisfy a specific verified property by opening a new window for inspecting its encoding (cf. 1) and its visualization

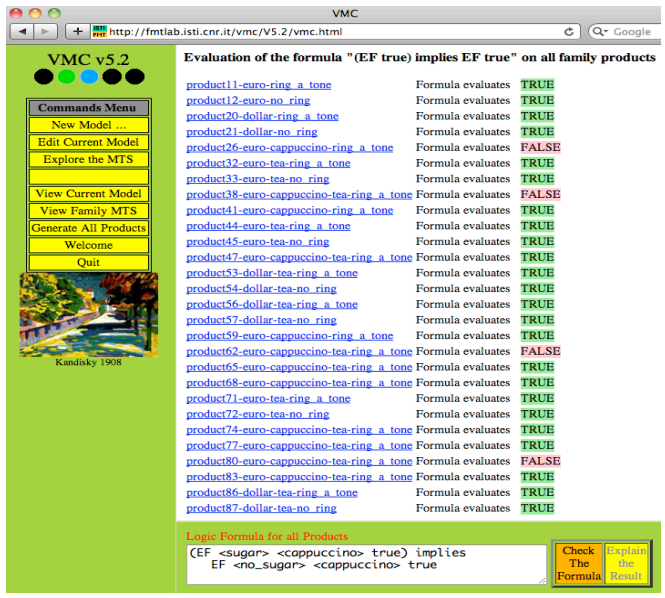


VMC thus allows to interactively explore an MTS of a product family, model check properties (logic formulae) over an MTS, visualize the (interactive) explanations of a verification result, generate all the family's valid products (w.r.t. the given constraints) represented as LTSs, browse and explore these, model check whether they satisfy certain properties, and help understand why a certain valid product does (not) satisfy the verified properties by inspecting it individually

2 VMC can model check whether a product family satisfies a certain property/formula and explain/analyze the result interactively



4 VMC can model check which valid product(s) of a product family satisfy a specific property



VMC is part of a family of on-the-fly model checkers developed at ISTI-CNR for verifying logic formulae in a CTL-like action- and state-based branching-time temporal logic

VMC's core holds a command-line version of the model checker and a product generation procedure, both stand-alone executables in Ada (easy to compile for MacOSX/Windows/Linux/Solaris), and is wrapped with a set of CGI scripts handled by a web server, to help graphical html-oriented GUI and integration with LTS minimization / graph drawing tools

VMC is a prototype, publicly usable online; its executables are available upon request

Using CSP||B and ProB for railway modelling

Faron Moller[1], Hoang Nga Nguyen[1], Markus Roggenbach[1],
Steve Schneider[2], and Helen Treharne[2]

[1] Swansea University, Wales, UK
[2] University of Surrey, England, UK

1 Introduction

One of the goals of the UK research project SafeCap[3] (Overcoming the railway capacity challenges without undermining railway network safety) is to provide railway engineers with a formal modelling framework for analysing safety and capacity of railway systems. To this end, we have proposed a "natural modelling" approach for specifying railway networks in CSP||B [4], and we are developing the capability to model track plans of increasing complexity. We have considered a simple closed track circuit with points, the 'Mini-Alvey' [2]. We have further considered the 'Double Junction' example [3], which includes a track crossing, adjacent points, more complex route locking and open connections. Once we have a model then we are in a position to formulate and verify safety and liveness properties. Introducing more detailed behaviour, such as points in transition, manual release of routes, multi-aspect signalling and more complex driving rules is currently in development. Our approach uses the case studies to drive the development of patterns comprising a generic style for railway modelling.

In our approach, the railway models are as close as possible to the domain model, providing traceability and ease of understanding to the domain expert. This leads to a natural separation between the global modelling of the tracks in B, and the CSP encapsulation of the local views of the individual trains following the driving rules. In this poster we illustrate the modelling approach through the Mini-Alvey case study, and see how the model provides verification through model checking or informative counter example traces if verification fails.

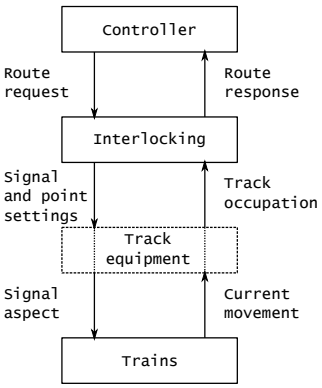
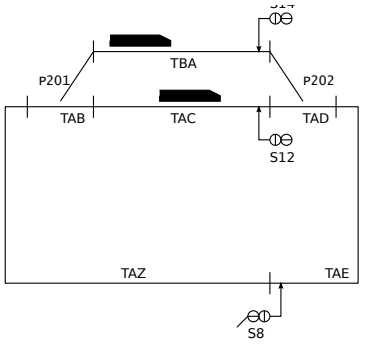


Fig. 1. Information flow.

2 The railway domain

Railways consist of (at least) four, physically different entities: see Figure 1. The *Controller* selects routes for trains and sends requests of routes to the *Interlocking*. The interlocking monitors the *Track equipment* and sends out commands to

[3] SafeCap's web site: `http://safecap.cs.ncl.ac.uk`.



Control table

Route	Normal	Reverse	Clear
8A	P201		TAZ TAB TAC
8B		P201	TAZ TAB TBA
12A	P202		TAD TAE
14A		P202	TAD TAE

Release tables

P201	Occupied
8B	TAC
8A	TBA

s

P202	Occupied
12	TAE
14	TAE

Fig. 2. Mini-Alvey.

control it with respect to the route requests and a pre-defined control table. The track equipments consists of elements such as signals, points and track circuits: signals can show green or red (the yellow aspect of a signal is not modelled at this level of abstraction since we are only interested in whether a train is authorized to enter a section); points can be in normal position (leading trains straight ahead) or in reverse position (leading trains to a different line) and track circuits detect if there is a train on a track. Finally, *Trains* have a driver who determines their behaviour.

Railways are built according to a *Track plan*. Figure 2 depicts a prominent example referred to in the literature as the Mini-Alvey track plan [5, 6]. This plan shows various tracks (TAB, TAC, TAD, ...), signals (S8, S12, S14), and points (P201, P202). This plan is accompanied with a control table describing conditions under which signals at the beginning of every route[4] can show proceed. For example, signal S12 for the route between S12 and S8 can only show proceed if point P202 is in normal (straight) position and tracks TAZ, TAB and TBA are clear. When a signal shows proceed, points on the corresponding route are locked to prevent trains from derailment. They are released according to the *Release tables* associated with each point. For example, locked P201 for route 8B from S8 to S12 will be released if TAC is occupied. In such a railway system, we are interested in verifying *Safety* properties. This means no collision (one train moving into another) and no derailment (points moving under trains, trains moving onto points from the wrong direction, trains travelling too fast).

3 A CSP||B model

The architecture of our model[5] is depicted in Figure 3. The CTRL component is a CSP description which is used to describe the driving rules of trains in order to control their movement (such as never pass a red signal) and enable the Controller to issue route requests. The Interlocking component is a B-machine which

[4] A *route* is a (directed) path which leads from one signal to the next signal.
[5] CSP||B Mini-Alvey model download: `http://www.csp-b.org/mini-alvey.zip`.

describes the general principles of an interlocking such as considering route requests by following the conditions of a control table to determine whether or not the requests are granted and monitoring the state of points and signals and the locations of trains. This component is generic and does not depend on a particular track plan. Conversely, other components are for modelling a specific track plan. ClosedContext declares track equipment such as tracks, signals and points. ClosedTopology describes the connections between tracks as well as the position of signals and points in the track plans. Then, ControlTable and ReleaseTable encode the corresponding components from the track plan. The CSP||B technical descriptions can be found in [2].

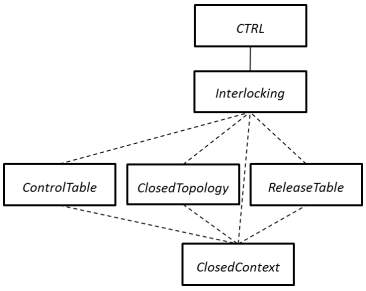


Fig. 3. CSP||B Architecture

4 Verification

Our CSP||B models can be verified using ProB [1] which supports B models that are controlled by CSP controllers. In this section we illustrate the use of ProB to verify safety properties of a railway system, represented as invariants on the *Interlocking* machine. For example, we capture the notions of no-collision and no-derailment in the invariant $pos : TRAIN \rightarrowtail\!\!\!\!\!\!\rightarrow TRACK$ on the *pos* function. This constrains no more than one train on any track circuit, and also that no train is on *nullTrack*, since $nullTrack \notin TRACK$. ProB verifies that this invariant is preserved in our model.

In the following, we consider two faulty scenarios in order to explore how the modelling and analysis exposes errors in the design. In each case ProB discovers violations of the invariant:

CSP||B model with faulty clear tracks: Suppose the control table is adjusted to contain the mistake that TAB is omitted from the tracks which should be clear to grant route $8B$. Then the following trace is produced automatically as a counter-example:

$\langle enter.albert.TAB, enter.bertie.TAE, request.B8.true,$
$nextSignal.bertie.green, move.bertie.TAE.TAZ, nextSignal.bertie.none,$
$move.bertie.TAZ.TAB\rangle$

This leads to a collision of *albert* and *bertie* on TAB.

CSP||B model with faulty points in control table: If the control table contains a mistake on the directions of points, e.g., $P202$ is normal (straight) position for route 14A. Then the check yields the following counter-example trace showing the derailment of *bertie*:

$\langle enter.albert.TAB, enter.bertie.TBA, request.A14.true,$
$nextSignal.bertie.green, move.bertie.TBA.nullTrack\rangle$

This demonstrates a violation of the safety requirement no-derailment.

5 Conclusion

This poster presents our approach to modelling in the railway domain. The "hybrid nature" of railways (namely, that some railway aspects can be directly expressed in an event-based approach while other aspects are more suited for a state-based approach) allows us to construct natural railway models in CSP||B, which are immediately understandable to the railway experts and analysable by current verification technologies.

We are developing our approach by applying it to more complex track designs, with more detailed behaviour and driving rules. We are also extending this approach in order to include the time aspect into railway models which will allow the study of both safety and capacity in an integrated way, to address the goals of the SafeCap research project.

Acknowledgement: The authors would like to thank S. Chadwick and D. Taylor from the company Invensys Rail for their support and encouraging feedback; and also Erwin R. Catesbeiana for keeping us on track.

References

1. M. Leuschel and M. Butler. ProB: an automated analysis toolset for the B method. *Int. J. Softw. Tools Technol. Transf.*, 10(2):185–203, Feb. 2008.
2. F. Moller, H. N. Nguyen, M. Roggenbach, S. Schneider, and H. Treharne. Combining event-based and state-based modelling for railway verification. Technical Report CS-12-02, Department of Computing, University of Surrey, 2012.
3. F. Moller, H. N. Nguyen, M. Roggenbach, S. Schneider, and H. Treharne. CSP‖B modelling for railway verification: the double junction case study. Technical Report CS-12-03, Department of Computing, University of Surrey, 2012.
4. S. Schneider and H. Treharne. CSP theorems for communicating B machines. *Formal Asp. Comput.*, 17(4):390–422, 2005.
5. A. Simpson, J. Woodcock, and J. Davies. The mechanical verification of solid-state interlocking geographic data. In *Formal Methods Pacific '97*. Springer, 1997.
6. K. Winter and N. Robinson. Modelling large railway interlockings and model checking small ones. In *26th ACSC*. Australian Computer Society, Inc., 2003.

USING CSP||B AND ProB FOR RAILWAY MODELLING

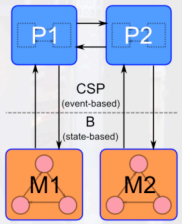
Aims

"Natural modelling" to provide accessible and traceable formal specifications

Tool support for verification with comprehendible feedback

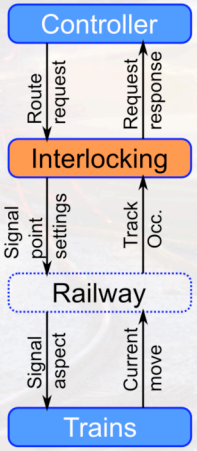
Background

CSP||B Architecture



Railway

hybrid nature



Railway's duality

Event: *"train moves on track T5"*

State condition: *"signal S shows proceed if tracks T1, T2 are free"*

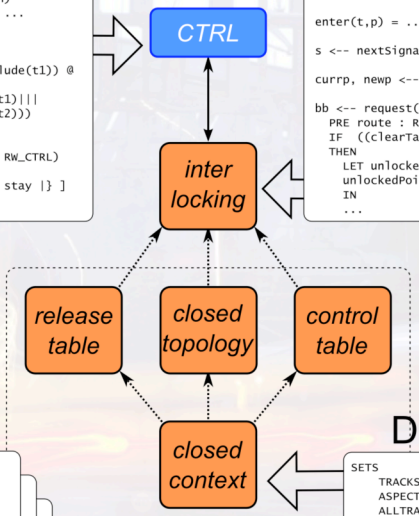
A natural modelling approach

Driving rules

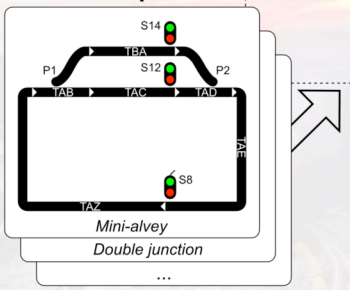
```
RW_CTRL = [] r : ROUTE @
          (request!r -> RW_CTRL)

TRAIN_CTRL(t,currp) =
   nextSignal!t?s ->
   if (s == none or s == green)
   then (move.t.currp?newp -> ...

ALL_TRAINS = [] t1 : TRACK @
   enter.albert.t1 ->
      (([] t2 : diff(TRACK,exclude(t1)) @
         enter.bertie.t2 ->
            TRAIN_CTRL(albert,t1)|||
            TRAIN_CTRL(bertie,t2)))

-TRL  =
   (enter?t?p -> enter?t?p -> RW_CTRL)
   [ {|enter,request |} ||
   {|enter, nextSignal, move, stay |} ]
   ALL_TRAINS
```

Signalling

```
INVARIANT

      pos : TRAIN >+> TRACK &
      ...

OPERATIONS

enter(t,p) = ...

s <-- nextSignal(t) = ...

currp, newp <-- move(t) = ...

bb <-- request(route) =
  PRE route : ROUTE THEN
  IF  ((clearTable(route) <: emptyTracks ))
  THEN
    LET unlockedPoints BE
    unlockedPoints = POINTS-ran(currentLocks)
    IN
    ...
```



Track plans

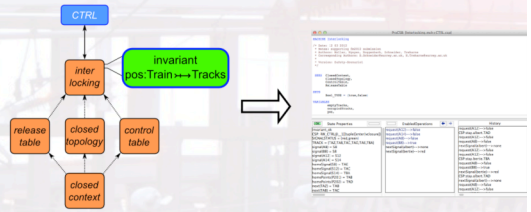


Datatypes

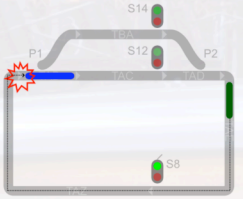
```
SETS
    TRACKSTATUS = {occ,empty};
    ASPECT = {red,green,none};
    ALLTRACK = {TAZ,TAB,TAC,...};
    SIGNAL = {S8,S12,S14};
    TRAIN = {albert,bertie};
    POINTS = {P201,P202};
    POINTPOSITION={normal,reverse};
    POINTSTATUS={locked, unlocked};
    ROUTE = {A8,B8,A12,A14}

CONSTANTS
    SIGNALSTATUS,
    TRACK
...          Mini-alvey
Double junction
...
```

Verification

Safety: no collision and no derailment



Collision caused by a faulty rule for 8A
(TAB not required to be clear)



Derailment caused by a faulty rule for 14A
(P2 in normal position)

Results

CSP||B models which are

- *understandable and (thus) verifiable by our industrial partners*
- *analysable by current verification technology*

References:

Schneider, S. and Treharne, H. *CSP theorems for communicating B machines*, Journal of Formal Aspects of Computing. Volume 17. Pages 390-422. 2005.

F. Moller, H. Nguyen, M. Roggenbach, S. Schneider, H. Treharne, *Combining event-based and state-based modelling for railway verification*, Tech. Rep. CS-12-02, University of Surrey (2012).

Swansea University Prifysgol Abertawe
Faron Moller
Hoang Nga Nguyen
Markus Roggenbach

UNIVERSITY OF SURREY
Helen Treharne
Steve Schneider

SafeCap
invensys Rail

SAM: Stochastic Analyser for Mobility

Michele Loreti

Dipartimento di Sistemi e Informatica
Università di Firenze

Abstract. Network and distributed systems typically consists of a large number of actors that act and interact with each other in a highly dynamic environment. Due to the number of involved actors and their strong dependence on mobility and interaction, performance and dependability issues are of utmost importance for this class of systems. StoKlaim is a stochastic extension of Klaim specifically thought to facilitate the incorporation of random phenomena in models for network-aware computing. Properties of StoKlaim systems can be specified by means of MoSL (*Mobile Stochastic Logic*). This is a stochastic logic that, together with qualitative properties, permits specifying time-bounded probabilistic reachability properties. MoSL is also equipped with operators that permit describing properties resulting from resource production and consumption. SAM (Stochastic Analyser for Mobility) is an automatic tool supporting stochastic analysis of StoKlaim specifications. SAM can be used for: executing interactively specifications; simulating stochastic behaviours; model checking MoSL formulae.

1 Introduction

Network and distributed systems typically consist of a large number of actors that act and interact with each other in a highly dynamic environment. Many programming and specification formalisms have been developed that can deal with issues such as (code and agent) mobility, remote execution, security, privacy and integrity. Important examples of such languages and frameworks are, among others, Obliq [5], Seal [6], ULM [4] and Klaim (*Kernel Language for Agents Interaction and Mobility*) [7, 3].

Performance and dependability issues are of utmost importance for "network-aware" computing, due to the number of involved actors and their strong dependence on mobility and interaction. Spontaneous computer crashes may easily lead to failure of remote execution or process movement, while spurious network failures may cause loss of code fragments or unpredictable delays.

Correctness in network and distributed systems, as well as their safety guarantees, is not a rigid notion "*either it is correct or not*" but has a less absolute nature: "in 99.7% of the cases, safety can be ensured"'.

To facilitate the incorporation of random phenomena in models for network-aware computing a stochastic extension of Klaim [7, 3], named StoKlaim, has been proposed in [8]. Klaim is an experimental language for distributed systems that is aimed at modelling and programming mobile code applications,

i.e., applications for which exploiting code mobility is the prime distinctive feature. In StoKlaim, every action has a random duration governed by a negative exponential distribution.

In [9], MoSL (*Mobile Stochastic Logic*), a logic that allows one to refer to the spatial structure of the network for the specification of properties for StoKlaim models as been proposed. MoSL is a stochastic logic (inspired by CSL [1, 2]) that, together with qualitative properties, permits specifying time-bounded probabilistic reachability properties, such as "the likelihood to reach a goal state within t time units while visiting only legal states is at least 0.92". MoSL is also equipped with operators that permit describing properties resulting from resource production and consumption. In particular, state properties incorporate features for resource management and context verification. Context verification allows the verification of assumptions on resources and processes in a system at the logical level, i.e. without having to change the model to investigate the effect of each assumption on the system behaviour.

2 SAM: Stochastic Analyser for Mobility

SAM, Stochastic Analyser for Mobility[1], is a *command-line* tool, developed in OCaML, that supports the stochastic analysis of StoKlaim specifications. SAM can be used for:

- executing interactively specifications;
- simulating stochastic behaviours;
- model checking MoSL formulae.

Running a specification SAM provides an environment for interactive execution of StoKlaim specification. When a specification is executed, a user can select interactively possible computations.

Simulating a specification To analyse behaviour of distributed systems specified in StoKlaim, SAM provides a simulator. This module randomly generates possible computations. A simulation continues until in the considered computation either a *time limit* or a deadlock configuration is reached.

Fixed a *sampling time*, each computation is described in term of the number of resources (located tuple) available in the system during the computation. At the end of a simulation, the average amount of resources available in the system at specified time intervals is provided.

Model checking SAM permits verifying whether a given StoKlaim specification satisfies or not a MoSL formula. This module, which implements the model checking algorithm proposed in [9], use an existing state-based stochastic model-checker, the Markov Reward Model Checker (MRMC) [11], and wrapping it in the MoSL model-checking algorithm. After loading a StoKlaim specification

[1] SAM website: http://rap.dsi.unifi.it/SAM/

and a MoSL formula, it verifies, by means of one or more calls to MRMC, the satisfaction of the formula by the specification.

Unfortunately, even simple StoKlaim specification can generate a very large number of states. For this reason, the *numerical* model checking cannot always be applied. To overcome the state explosion problem, a *statistical model-checker* has been also implemented in SAM. The statistical approach has been successfully used in existing model checkers [10, 12].

While in a numerical model checker the exact probability to satisfy a path-formula is computed up to a precision ϵ, in a *statistical model-checker* the probability associated to a path-formula is determined after a set of independent observations. This algorithm is parametrised with respect to a given *tolerance* ε and *error probability* p. The algorithm guarantees that the difference between the computed values and the exact ones are greater than ε with a probability that is less than p.

References

1. A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Model checking continuous time Markov chains. *Transations on Computational Logic*, 1(1):162–170, 2000.
2. C. Baier, J.-P. Katoen, and H. Hermanns. Approximate symbolic model checking of continuous-time Markov chains. pages 146–162.
3. L. Bettini, V. Bono, R. De Nicola, G. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The Klaim Project: Theory and Practice. In C. Priami, editor, *Global Computing: Programming Environments, Languages, Security and Analysis of Systems*, volume 2874 of *Lect. Notes in Comput. Sci.*, pages 88–150. Springer, 2003.
4. G. Boudol. ULM: a core programming model for global computing: (extended abstract). In D.A. Schmidt, editor, *Programming Languages and Systems, 13th European Symposium on Programming (ESOP)*, volume 2986 of *Lect. Notes in Comput. Sci.*, pages 234–248. Springer, 2004.
5. L. Cardelli. A Language with Distributed Scope. In *22nd Annual ACM Symposium on Principles of Programming Languages*, pages 286–297. ACM, 1995.
6. G. Castagna and J. Vitek. Seal: A framework for Secure Mobile Computations. In H. Bal, B. Belkhouche, and L. Cardelli, editors, *Internet Programming Languages*, volume 1686 of *Lect. Notes in Comput. Sci.*, pages 47–77. Springer, 1999.
7. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–329, 1998.
8. R. De Nicola, J.-P. Katoen, D. Latella, M. Loreti, and M. Massink. Klaim and its stochastic semantics. Technical report, Dipartimento di Sistemi e Informatica, Università di Firenze, 2006.
9. Rocco De Nicola, Joost-Pieter Katoen, Diego Latella, Michele Loreti, and Mieke Massink. Model checking mobile stochastic logic. *Theoretical Computer Science*, 382(1):42–70, 2007.
10. G. Norman H. Younes, M. Kwiatkowska and D. Parker. Numerical vs. statistical probabilistic model checking. *International Journal on Software Tools for Technology Transfer*, 8(3):216–228, June 2006.

11. J.-P. Katoen, M. Khattri, and I. S. Zapreev. A Markov reward model checker. In *Quantitative Evaluation of Systems (QEST)*, pages 243–244. IEEE CS Press, 2005.
12. Paola Quaglia and Stefano Schivo. Approximate model checking of stochastic COWS. In A. Rauschmayer M. Wirsing, M. Hofmann, editor, *Proc. of TGC 2010*, volume 6084 of *Lecture Notes in Computer Science*, pages 335–347. Springer, 2010.

SAM: Stochastic Analyser for Mobility

Michele Loreti - Dipartimento di Sistemi e Informatica, Università di Firenze
Based on joint work with: R. De Nicola[1], J.P. Katoen[2], D. Latella[3], M. Massink[3]
[1]IMT - Institute for Advanced Studies Lucca, [2]RWTH Aachen University, [3]ISTI - CNR Pisa

AM: Stochastic Analyser for Mobility

AM is a prototype tool, developed in OCaML, to support quantitative analysis f mobile distributed systems specified in StoKlaim
hanks to the its modularity, other languages can be integrated in SAM.

toKlaim

toKlaim [2] is a stochastic extension of Klaim [1] introduced to facilitate the ncorporation of random phenomena in models for network-aware computing.

Klaim is an experimental language for distributed systems that is aimed at nodelling and programming mobile code applications, i.e., applications for which xploiting code mobility is the prime distinctive feature.

n StoKlaim, every action has a random duration governed by a negative ex- onential distribution.

MoSL

MoSL (*Mobile Stochastic Logic*) [3] is a stochastic temporal logic that allows one refer to the spatial structure of the network for the specification of properties or StoKlaim models.

his is a stochastic logic that, together with qualitative properties, permits spec- ying time-bounded probabilistic reachability properties, such as "the likelihood o reach a goal state within t time units while visiting only legal states is at least .92".

MoSL is also equipped with operators that permit describing properties result- ng from resource production and consumption. In particular, state properties ncorporate features for resource management and context verification.

tatistical model checking

Unfortunately, even simple StoKlaim specification can generate a very large umber of states. For this reason, the *numerical* model checking cannot always e applied. To overcome the state explosion problem, a *statistical model-checker* as been also implemented in SAM.

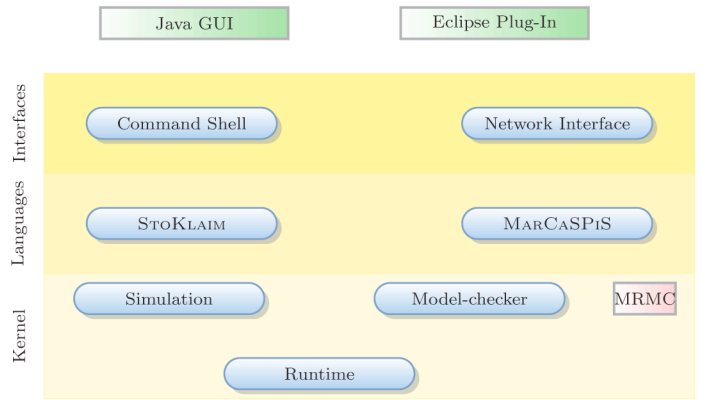
While in a numerical model checker the exact probability to satisfy a path- ormula is computed up to a precision ϵ, in a *statistical model-checker* the prob- bility associated to a path-formula is determined after a set of independent bservations. This algorithm is parametrised with respect to a given *tolerance* and *error probability* p. The algorithm guarantees that the difference between he computed values and the exact ones are greater than ε with a probability hat is less than p.

Main goals

Develop a framework that...

- provides tools supporting quantitative analysis of distributed (adaptive) systems;
- can be easily extended in order to take other specification languages into account;
- simplifies the use of formal tools thanks to a *usable/intuitive* interface;
- can be integrated in a *software development* process;
- supports integration with other (Ascens) tools.

Architecture



Main features

Running a specification SAM provides an environment for interactive execution of StoKlaim specification. When a specification is executed, a user can select interactively possible computations.
Simulating a specification To analyse behaviour of distributed systems specified in StoKlaim, SAM provides a simulator. This module randomly generates possible computations. A simulation continues until in the considered computation either a *time limit* or a deadlock configuration is reached.
Fixed a *sampling time*, each computation is described in term of the number of resources (located tuple) available in the system during the computation. At the end of a simulation, the average amount of resources available in the system at specified time intervals is provided.
Model checking SAM permits verifying whether a given StoKlaim specification satisfies or not a MoSL formula. This module, which implements the model checking algorithm proposed in [3], use an existing state-based stochastic model-checker, the Markov Reward Model Checker (MRMC) [5], and wrapping it in the MoSL model-checking algorithm. After loading a StoKlaim specification and a MoSL formula, it verifies, by means of one or more calls to MRMC, the satisfaction of the formula by the specification.

References

[1] R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–329, 1998.

[2] R. De Nicola, J.-P. Katoen, D. Latella, M. Loreti, and M. Massink. Klaim and its stochastic semantics. Technical report, Dipartimento di Sistemi e Informatica, Università di Firenze, 2006.

[3] Rocco De Nicola, Joost-Pieter Katoen, Diego Latella, Michele Loreti, and Mieke Massink. Model checking mobile stochastic logic. *Theoretical Computer Science*, 382(1):42–70, 2007.

[4] G. Norman H. Younes, M. Kwiatkowska and D. Parker. Numerical vs. statistical probabilistic model checking. *International Journal on Software Tools for Technology Transfer*, 8(3):216–228, June 2006.

[5] J.-P. Katoen, M. Khattri, and I. S. Zapreev. A Markov reward model checker. In *Quantitative Evaluation of Systems (QEST)*, pages 243–244. IEEE CS Press, 2005.

Timed CSP Simulator

Marc Fontaine[1], Andy Gimblett[2], Faron Moller[2],
Hoang Nga Nguyen[2], and Markus Roggenbach[2]

[1] Heinrich-Heine-Universität Düsseldorf, Germany
[2] Swansea University, UK

1 Introduction

Time is an integral aspect of computer systems. It is essential for modelling a system's performance and also affects its safety or security. Timed Csp [5] conservatively extends the process algebra CSP with timed primitives, where real numbers ≥ 0 model how time passes with reference to a single, conceptually global, clock. While there have been approaches for model checking Timed Csp [1, 5], the simulation of Timed Csp was considered only recently [2, 6]. In this poster, we highlight the architecture and a number of selected features of our Timed Csp Simulator, which is a consolidated, mature version of the research prototype presented in [2].

2 Architecture

Timed Csp Simulator is an extension of the CSP animator within the open source tool ProB [3]. In Figure 1, we illustrate the architecture of the Timed Csp Simulator which consists of four main components: a Parser, a Timed Csp Interpreter, a Simulator and a GUI. In principle, the simulator works as follows: A Timed Csp specification is analysed by the Parser (written in Haskell) and translated to a representation in Prolog. This representation is passed into the Timed Csp Interpreter (written in Prolog). The Timed Csp Interpreter implements the "firing rules" of Timed Csp's operational semantics. Process states and the implementation of firing rules are then used by the Simulator for determining the set of actions available, the range of timed transitions as well as their corresponding resultant states. Finally, users interact with the Simulator through a GUI (written in Tcl/Tk) in order to control the simulation progress.

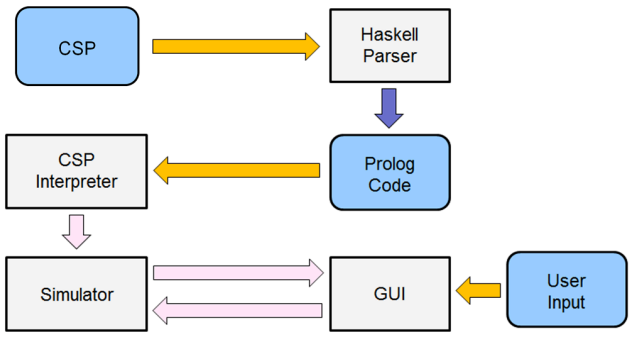


Fig. 1. Timed Csp Simulator's architecture

3 Features

Timed Csp Simulator is characterised by the following features:
Rational time: Timed Csp Simulator restricts processes to rational time only. This is reasonable as [4] proves that Timed Csp is closed under rational time, i.e., rational processes are closed under action transitions and rational delays. Prolog supports proper rationals, i.e., rational time can be expressed. In practice, the limitation to rational time turns out to be negligible. For instance, all examples of Schneider's book [5] can be dealt with in our simulator.
Separate firing rules: Although Timed Csp (as well as CSP) has a number of operators which can be treated as syntactic sugar, e.g., $Wait\ d = Stop \triangleright^d Skip$, we follow the design of ProB where each supported (untimed or timed) operator has its own implementation of the corresponding firing rule. This results in a simulation without change of representation, where ProB can highlight in the specification text which process parts represent the current state.
Extensive set of operators: Compared to the language as given in [5], Timed Csp Simulator supports an extra set of untimed operators such as conditionals, untimed timeout and indexed external choice. To this end, we extend Timed Csp's operational semantics in [4].
Computing upper bounds of timed transitions: Based on the simulation theorem provided in [4], Timed Csp Simulator calculates the largest time step possible for a Timed Csp process in a recursive way. Consider, for instance, the process $T = (P \triangleright^e Q) \triangleright^f R$ with $0 < e < f$ and untimed processes P, Q and R. In T, the process P is enabled within the time interval $[0, e)$. A time step of length e (and a τ-transition) leads to the new state $Q \triangleright^{f-e} R$, where P is not enabled anymore. Thus, the largest time step possible in T is e.
Automatic animation: Timed Csp Simulator supports two animation strategies, where the user selects the number of steps to be performed. **Random:** At each step of the animation, the simulator randomly selects an event or time step available from the interface. **Maximal progress:** At each step of the animation, the simulator selects an event or time step available from the interface in the following priority: (1) randomly select an external event, (2) select the internal event (3) select the maximal time step from a bounded interval, (4) randomly select a time step if arbitrary time steps are possible.
Backward compatibility: Timed Csp Simulator is backwards compatible for untimed CSP specifications. There are two ways to enable the Timed-CSP Simulator in ProB while opening specifications from files. **Explicit:** Files are named with the extension ".tcsp", or **Implicit:** Files (ended with the extension ".csp") contain any timed operator of delay event prefix, wait, timed timeout and timed interrupt.

4 Example

In order to illustrate the use of Timed Csp Simulator, we apply it to the well-known level crossing example [5]. This system consists of three components: a

gate to block the traffic crossing the railway when a train is approaching, a controller to monitor the approach of trains and to instruct the gate to rise or lower appropriately, and a train passing by the crossing. Figure 2 presents the

$$
\begin{aligned}
GATE &= down.command \overset{100}{\rightarrow} down \rightarrow confirm \rightarrow GATE \\
&\quad \Box\ up.command \overset{100}{\rightarrow} up \rightarrow confirm \rightarrow GATE \\
TRAIN &= train.near \rightarrow near.ind \overset{300}{\rightarrow} enter.crossing \overset{20}{\rightarrow} \\
&\quad leave.crossing \rightarrow out.ind \rightarrow TRAIN \\
CONTROLLER &= near.ind \rightarrow down.command \rightarrow confirm \rightarrow CONTROLLER \\
&\quad \Box\ out.ind \rightarrow up.command \rightarrow confirm \rightarrow CONTROLLER \\
CROSSING &= CONTROLLER_C \|_G GATE \\
SYSTEM &= TRAIN_T \|_{C \cup G} CROSSING
\end{aligned}
$$

Fig. 2. Timed Csp's model of the level crossing.

Timed Csp's model of the level crossing as developed in [5]. It is straight forward to write this specification in the concrete syntax of Timed Csp Simulator.

In the following, we show two simulations of the level crossing example which highlight the *rational time only* and *automatic animation* features of Timed Csp Simulator. In Figure 3, we present a timed trace which includes two consecutive

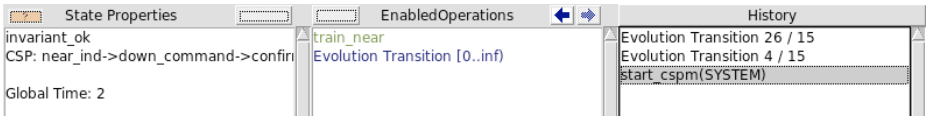


Fig. 3. A trace of two consecutive rational timed evolutions.

timed evolutions. The first timed evolution lasts for $\frac{4}{15}$ time unit while the second for $\frac{26}{15}$ time unit. After the two timed evolutions, the global time reaches $\frac{30}{15}$ time unit which is automatically converted into the simpler representation of 2 time units.

In the second simulation, rather than manually choosing an available action at each step of the simulation, we use the *automatic animation* feature to quickly generate a long timed trace of the level crossing. Figure 4 shows a timed trace generated by an animation of 15 transitions, following the *maximal progress* strategy. This timed trace illustrates the operation of the crossing as a train passing by. When the train approaches the crossing (by *train.near*), the controller requests the gate to move down (by *down.command*). The gate performs the action *down* and replies with a confirmation (by *confirm*) back to the controller. After the train has entered and exited the crossing, the controller is notified (by *out.ind*) so that it will instruct the gate to rise. At the end of this timed trace, the global time is 320 time units.

Timed Csp Simulator comes with an comprehensive test suite, derived from the fundamental algebraic laws of Timed Csp. A typical example is $P \triangleright^5 (Q \triangleright^3 R) = (P \triangleright^5 Q) \triangleright^8 R$, with $P = a \rightarrow Stop, Q = b \rightarrow a \rightarrow Stop$, and $R = c \rightarrow Stop$. Here, we check that simulations of the lhs are possible for the rhs and vice versa. Though these processes are not of much practical use, they highlight

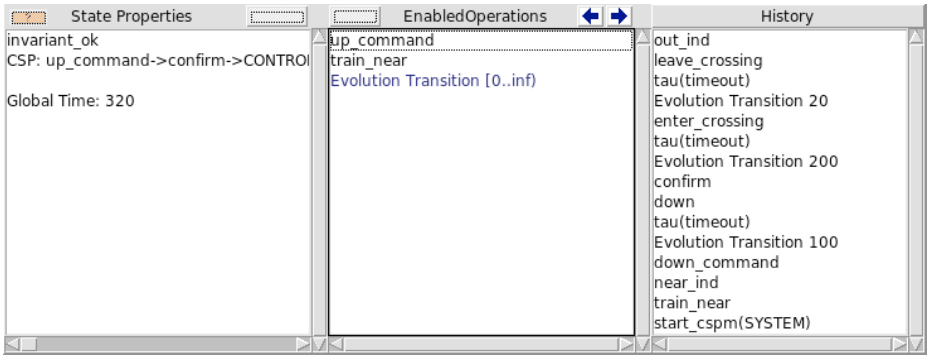


Fig. 4. A trace generated by automatic animation.

tricky features of the Timed Csp semantics and provide an argument that Timed Csp Simulator implements it correctly.

5 Conclusion

We have presented our tool Timed Csp Simulator, which is an extension of the CSP animator within ProB. We discussed architecture and features of the simulator. Besides simulating examples given in [5], we extensively use our tool within the SafeCap project[3] in order to explore how the change of signalling rules affects railway capacity. The ProB team has checked our implementation and made it available at `http://www.stups.uni-duesseldorf.de/ProB/index.php5/Download`. In the future, we plan to complete the simulator and to apply our tool within further application domains.

Acknowledgement We thank Erwin R. Catesbeiana (Jr.) for inspiring us to invest the extra time unit.

References

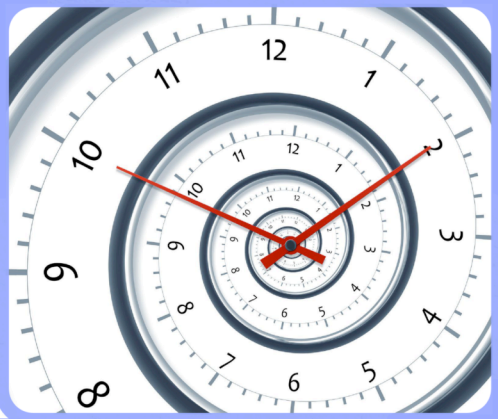
1. J. Dong, P. Hao, J. Sun, and X. Zhang. A reasoning method for Timed CSP based on constraint solving. *Formal Methods and Software Engineering*, 2006.
2. M. Dragon, A. Gimblett, and M. Roggenbach. A Simulator for Timed CSP. In *AVoCS'11*, 2011.
3. M. Leuschel. The ProB model checker. `http://www.stups.uni-duesseldorf.de/ProB/index.php5/Main_Page`.
4. F. Moller, H. N. Nguyen, and M. Roggenbach. Theoretical foundations for simulating Timed CSP. Technical report, Swansea University, In preparation.
5. S. Schneider. *Concurrent and real-time systems: the CSP approach*. Wiley, 2000.
6. T. Yamakawa, T. Ohashi, and C. Fukunaga. Development of an ML-based verification tool for Timed CSP processes. In *CPA'11*. IOS Press, 2011.

[3] SafeCap's website: `http://safecap.cs.ncl.ac.uk/`.

TIMED CSP SIMULATOR

AIM

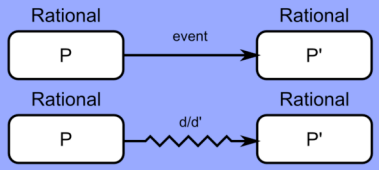
Simulation of timed systems (railways, protocols, controllers,...)



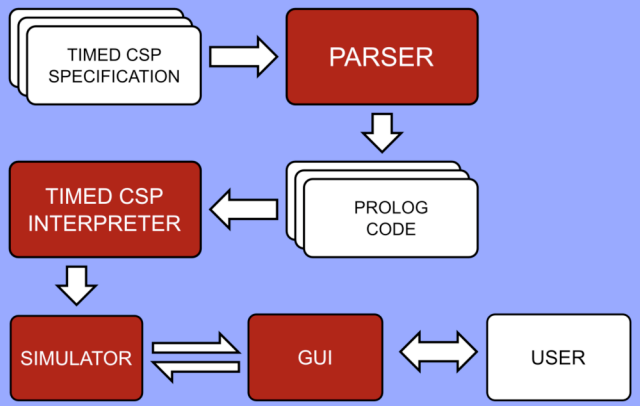
BACKGROUND

Timed CSP [1]:
- Newtonian time
- Real-time behaviour
- Timeouts, delays, interrupts

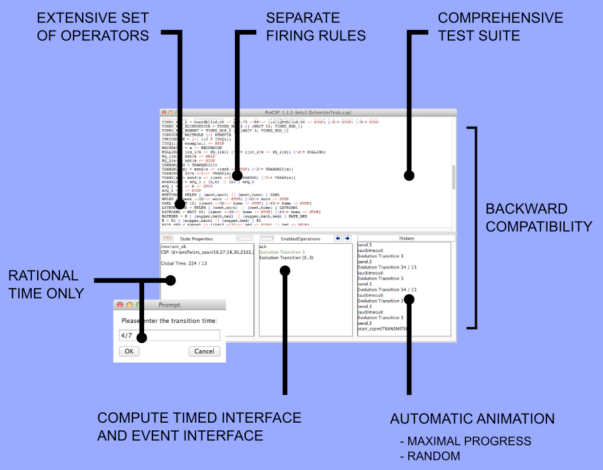
Rational closure's theorem [2]:



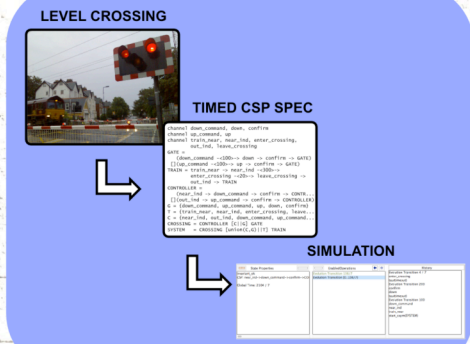
ARCHITECTURE



FEATURES



EXAMPLE



RESULT

A reliable tool for exploring and analysing Timed CSP Processes.

1. S. Schneider. Concurrent and real-time systems: the CSP approach. Wiley, 2000.
2. F. Moller, H. N. Nguyen, and M. Roggenbach. Theoretical foundations for simulating Timed CSP. Technical report, Swansea University, In preparation.
3. Download link: http://www.stups.uni-duesseldorf.de/ProB/index.php5/Download

Swansea University
Prifysgol Abertawe

Andy Gimblett
Faron Moller
Hoang Nga Nguyen
Markus Roggenbach

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Marc Fontaine

SafeCap

Demo: The Margrave Tool for Policy Analysis

Timothy Nelson[1], Daniel J. Dougherty[1],
Kathi Fisler[1], and Shriram Krishnamurthi[2]

[1] Worcester Polytechnic Institute
[2] Brown University

Margrave[3] is a flexible and expressive tool for analyzing policies. Policies abound in today's society: access-control policies, healthcare policies, routing and firewall policies all influence our daily lives. As these policies grow more complex and intertwined, maintaining existing policies and authoring new policies that interact properly with others becomes more challenging. A policy misconfiguration can remain undetected until its consequences emerge, resulting in embarrassment, loss of business, or even physical danger.

Run-time policy testing is one possible defense against these dangers. However, testing cannot cover all possible scenarios. Static analysis techniques allow the off-line exploration of policy behavior, and are often able to make guarantees about all potential inputs. These techniques are known to be useful when designing systems, but they are also valuable for analyzing and comparing existing systems. Thus, tools that use static analysis – like Margrave – are a useful complement to traditional testing. Policies lend themselves to many different types of static analysis. For instance:

Policy Emulation: "What if...?" The most basic kind of analysis is emulation of a policy. Given a request, what will the policy do? This analysis allows test-cases to be run against a policy without creating a separate testing environment. *Example: Tim's doctor wants to access his medical history from the year 2005. Is she permitted to do so?*

Property Verification: "Is this always true?" Policy authors may have a set of goals that they want their policy to meet. Property verification allows such goals to be rigorously checked across all possible requests, without ever invoking the policy. *Example: A patient's employer should never be allowed to access the patient's medical history. Is that goal satisfied by the policy?*

Property-Free Verification: "What changed?" Many policy authors have no formal specification that describes their goal, but can recognize when a policy does not meet their intuitive expectations. In such situations, property-free verification techniques such as the following "change-impact analysis" are especially useful. A user may be performing maintenance on their policy, and want to know in what situations the new policy renders different decisions than the old policy. Model-finding is especially suited to property-free verification; concrete models that illustrate policy behavior can aid in finding bugs as well as in discovering unspoken requirements. *Example: Does the newly modified policy correctly stop employers from reading employee medical records? Are there any unforeseen side effects of the changes?*

[3] www.margrave-tool.org

Rule Responsibility: "Why did this happen?" When investigating policy behavior, it is useful to know which rules are actually responsible for the decision(s) rendered. This type of analysis is especially important when working with an existing policy, which may have been maintained in an ad-hoc manner. *Example: If the prior goal fails, and there are cases where an employer can access employee medical records, which policy rules are responsible for granting that permission?*

Policy Analysis in Margrave

Margrave takes the model-finding approach: given a query over a set of policies, Margrave produces an exhaustive set of scenarios that explain how the policies can meet the behavior described in the query. Users can explore these scenarios to gain further insight into their policies. This outlook is similar to Alloy's. Policies and queries correspond to a specification, and we provide models for that specification. Indeed, we use the Kodkod [TJ07] model finder as our engine, just as Alloy does. However, Alloy is a general-purpose model-finding tool, while Margrave is designed to focus sharply on issues of policy-analysis. Our tool provides several features that are specifically designed to make policy-analysis easier.

- Margrave queries can be expressed either in our SQL-like **query language** or via an **API** for the Racket [FP10] programming language. The Racket API allows users to write scripts that invoke Margrave queries or use Margrave as part of their larger programs.
- Margrave has an **interactive interface** that encourages iterative exploration of policy consequences. Users may write new queries that incorporate the results of prior queries, as well as re-use query results programmatically. The tool itself is embedded in the DrRacket IDE for ease-of-use and to leverage the features of an IDE when writing queries. Figure 1 illustrates this process.
- Margrave queries may refer to specific policy rules. For instance, one can write a query to find the set of rules that never apply. Moreover, Margrave can be instructed to provide **rule-responsibility** information in each model it shows. This information indicates which rules matched or applied for each request. While this sort of information could be useful in general model-finding, when debugging a policy it is especially helpful to know which rules are responsible for a decision, or how they override one another.
- Margrave's query language has explicit support for **change-impact analysis**. The models that Margrave returns for change-impact describe the action of all policies involved in the query. Moreover, Margrave permits change-impact queries over sets of policies that interact with one another. For example, one might ask whether a new perimeter firewall policy permits any traffic that the old policy denied, and is not caught by departmental firewalls.

– For most common queries, Margrave guarantees that its model-finding is **complete**. For more information on this, see our paper [NDFK12], which will be presented at this year's ABZ conference.

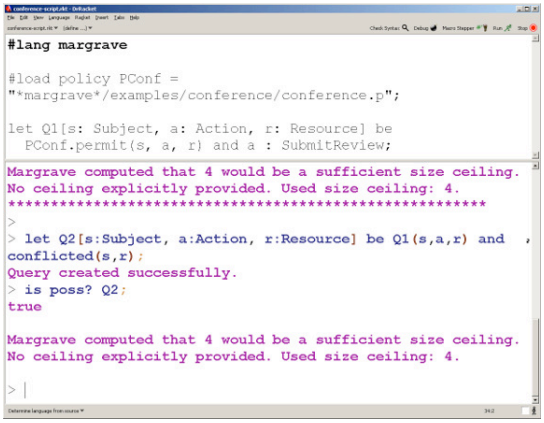


Fig. 1. The Margrave Tool: Refining a Previous Query

Applications

Margrave is designed to apply to a wide range of policy types, rather than only (for instance) access-control. Moreover, a particular type of policy may be written in a plethora of different languages. In spite of the disparity in concrete syntax, policy languages can typically be expressed in relatively simple fragments of first-order logic. Margrave provides an expressive intermediate policy language that encompasses the above languages and more; formally our language is equivalent in expressive power to non-recursive Prolog with negation.

Margrave has built-in support for several real-world policy languages including the majority of the access-control language XACML and a significant fragment of Cisco's IOS configuration language. We do not treat certain XACML features (such as in-line Java code) or stateful inspection in IOS. Margrave analyzes Cisco IOS configurations by partitioning each configuration into its component policies, allowing users to phrase queries about both the firewall and routing behavior of a device. The details of our IOS analyses can be found in our 2010 LISA paper [NBD+10].

Our notion of policy is expressive enough to allow Margrave to reason about other languages that XACML subsumes, e.g. role-based access-control, and other router configuration formats such as IPtables scripts. Prospective users with policies whose language we do not explicitly support may either translate them into our intermediate policy language or use Margrave's Racket API. The API allows policies as well as queries to be created within a program, encouraging the tool's use in automated scripts.

Perspective

Policy analysis is an important application for model-finding technology. Demonstrating concrete examples of policy behavior allows policy authors to see the consequences of a change before it is made. Although Margrave's model-finding approach is similar to Alloy's, it provides several features designed specifically for policy analysis. For instance, Margrave's completeness guarantees add confidence that the concrete examples shown really do cover all possible requests to the policy.

Related Work Readers interested in the relationship to other tools are encouraged to look at our technical papers [NBD+10, Nel10, FKMT05] on Margrave, which all have thorough and extensive related work sections.

Current and Future Work We are working to expand both the languages and policy formalisms that Margrave explicitly supports. Current work on Margrave involves, for instance, the UARBAC formalism of Li and Mao [LM07] and the break-glass policies discussed by Marinovic et al. [MCMD11]. We are also working on ways to better present the models Margrave produces, and better integrate those models with the original policies.

References

[FKMT05] Kathi Fisler, Shriram Krishnamurthi, Leo Meyerovich, and Michael Tschantz. Verification and change impact analysis of access-control policies. In *International Conference on Software Engineering*, pages 196–205, 2005.

[FP10] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR2010-1, PLT Inc., June 7, 2010. racket-lang.org/tr1/.

[LM07] Ninghui Li and Ziqing Mao. Administration in role-based access control. In *ACM Symposium on Information, Computer and Communications Security*, 2007.

[MCMD11] Srdjan Marinovic, Robert Craven, Jiefei Ma, and Naranker Dulay. Rumpole: A flexible break-glass access control model. In *ACM Symposium on Access Control Models and Technologies*, 2011.

[NBD+10] Timothy Nelson, Christopher Barratt, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. The Margrave Tool for Firewall Analysis. In *USENIX Large Installation System Administration Conference*, 2010.

[NDFK12] Timothy Nelson, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Toward a More Complete Alloy. In *ABZ*, 2012. (To Appear).

[Nel10] Timothy Nelson. Margrave: An Improved Analyzer for Access-Control and Configuration Policies. Master's thesis, Worcester Polytechnic Institute, April 2010.

[TJ07] Emina Torlak and Daniel Jackson. Kodkod: A Relational Model Finder. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647, 2007.

Requirements Traceability between Textual Requirements and Formal Models Using ProR

Lukas Ladenberger and Michael Jastram

Institut für Informatik, Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf**
{ladenberger,jastram}@cs.uni-duesseldorf.de

Traceability within a system description is a challenging problem of requirements engineering [1]. In particular, formal models of the system are often based on informal requirements, but creating and maintaining the traceability between the two can be challenging. In [2], we presented an incremental approach for producing a system description from an initial set of requirements. The foundation of the approach is a classification of requirements into *artefacts* W (domain properties), R (requirements) and S (specification) [3]. In addition, the approach uses designated *phenomena* as the vocabulary employed by the artefacts. The central idea is that *adequacy* of the system description must be justified, meaning that $W \wedge S \Rightarrow R$. The approach establishes a traceability, and the resulting system description may consist of formal and informal artefacts.

We created tool support for this approach by integrating Rodin [4] and ProR [5]. Rodin is an Eclipse-based open tool platform for formal modelling in Event-B [6]. ProR is a platform for requirements engineering that is also based on Eclipse and part of the Eclipse Requirements Modeling Framework (RMF)[1].

A seamless integration between ProR and Rodin is possible, as both are based on Eclipse. The integration plug-in is installed into Rodin via an update site[2]. We designed it with the goal to support the approach described in [2] and to ease the integration of natural language requirements and Event-B. Supporting other formalisms is possible in principle, and we are currently working on supporting integration with classical B [7]. Figure 1 shows ProR installed inside Rodin.

The integration allows the identification of phenomena within natural language requirements (Rodin already allows the identification of phenomena in formal model artefacts); it supports the creation of traces between arbitrary artefacts; and it tracks whenever the source or target of a trace changes by marking it as "suspect" (allowing the re-validation of traces).

ProR already supports some features required for an integration. For instance, ProR supports classifying informal and formal artefacts as W, R and S. Other features had to be provided by an integration plug-in, as described below.

There are still some limitations, that we discuss in the conclusion. In the following, we describe the specific features of the tool in more detail.

** Part of this research has been sponsored by the EU funded FP7 project 214158: DEPLOY (Industrial deployment of advanced system engineering methods for high productivity and dependability).

[1] http://eclipse.org/rmf.

[2] The update site URL is http://www.stups.uni-duesseldorf.de/pror_updates.

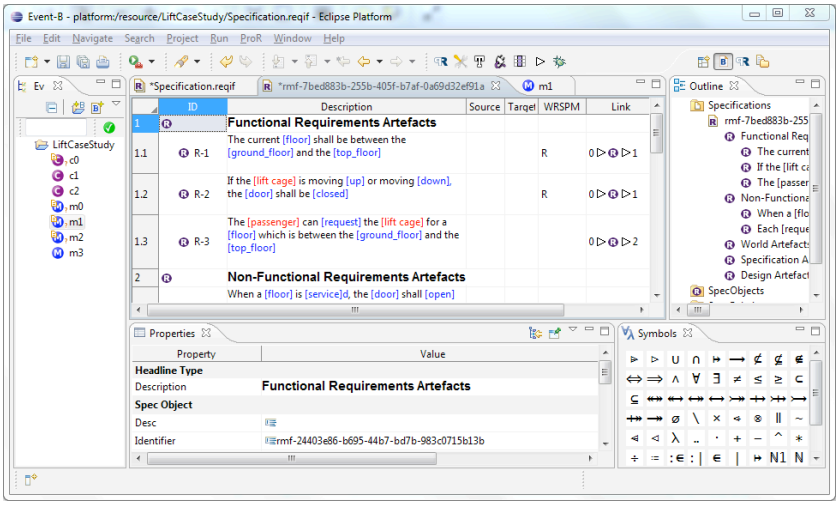


Fig. 1. ProR running inside the Rodin Platform for Event-B modelling.

Tracing of phenomena used in artefacts Textual requirements are rendered by colour-highlighting those text passages that correspond to phenomena (Figure 1). The user has to mark them by square brackets. In doing so, the text passage is rendered in blue, otherwise in red, reminding the user that an undeclared phenomena is used. In addition, unmarked, recognised phenomena are highlighted as well to warn the user about a possible omission (Figure 2). The marked phenomena are automatically synchronized with the model.

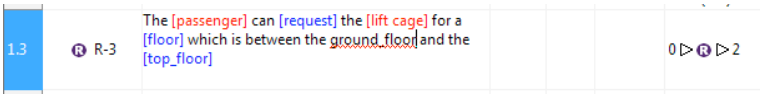


Fig. 2. The tool warns the user about a possible omission.

Annotated traces to modelling elements Manual creation of traces between requirements and formal model elements is supported via drag and drop. Figure 1 shows how the right column "Link" of the specification editor summarizes the number of outgoing (target) and incoming (source) traces. Details of the outgoing trace can be switched on as shown in figure 3. Selecting an outgoing trace shows the targets properties in the Property View. For instance, we see in figure 3 that the trace target which is the event *switch_move* is selected. Selecting the target shows its attributes in the Property View, including the formal event itself. This is a reference to the model, not a copy of the event. As a consequence, whenever the formal model element changes, the reference also changes (and the trace

will be marked as "suspect", as described below). In addition, traces can be annotated if additional information is necessary.

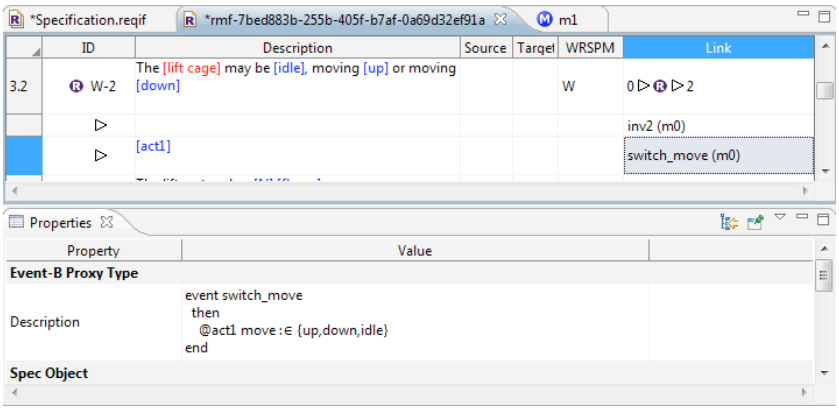


Fig. 3. The unveiled traces of an element. As the link target is selected, the link target's properties are shown in the Property View (the lower pane).

Change management to both the requirements and the formal model Last, when traced formal model elements change, the trace is marked as "suspect" by showing a small icon as demonstrated in figure 4. Two columns exist for the source and the target of the trace, respectively. The user sees at a glance which requirements or formal model elements need to be revalidated. This is particularly useful if the requirements document becomes large. By double-clicking on the "suspect" icon, the user can mark the trace as "revalidated" and the icon will be removed.

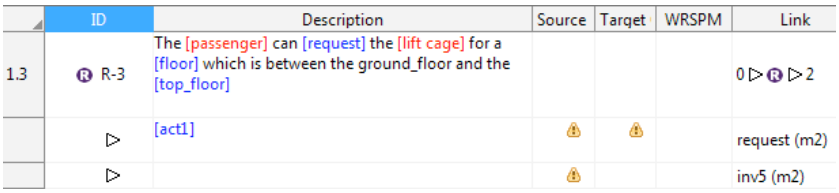


Fig. 4. A small icon indicates whenever the source (the requirement) or the target (formal model element) needs to be revalidated.

Conclusion We believe that the integration between Rodin and ProR supports the user in managing requirements in natural language and the corresponding traces to formal model elements, as outlined by our approach described in [2].

There are still some limitations, however. While all required data structures exist, the tool would benefit from more sophisticated reporting. In particular, [2] lists a number of properties of a correct system description. While the presence of these properties does not guarantee correctness, their absence indicates a problem. Reporting on the state of these properties would be valuable.

Furthermore, we believe that the integration is useful even beyond supporting our approach. For instance, the capability of marking traces as "suspect" if the source or the target change could be useful in many situations, even without the use of formal methods.

Last, we believe that this integration brings two complimentary fields of research, requirements engineering and formal modelling, closer together.

References

[1] O. Gotel and A. Finkelstein: An Analysis of the Requirements Traceability Problem IEEE Computer Society (1994)
[2] M. Jastram and S. Hallerstede and L. Ladenberger: Mixing Formal and Informal Model Elements for Tracing Requirements AVOCS 2011 (2011)
[3] C. A. Gunter and M. Jackson and E. L. Gunter and P. Zave: A Reference Model for Requirements and Specifications IEEE Software Vol. 17, 37–43 (2000)
[4] J.-R. Abrial and M. J. Butler and S. Hallerstede and T. S. Hoang and F. Mehta and L. Voisin: Rodin: An Open Toolset for Modelling and Reasoning in Event-B STTT Vol. 12, 447–466 (2010)
[5] M. Jastram: ProR, an Open Source Platform for Requirements Engineering based on RIF SEISCONF (2010)
[6] J.-R. Abrial: Modeling in Event-B: System and Software Engineering Cambridge University Press (2010)
[7] J.-R. Abrial: The B-Book: Assigning programs to meanings Cambridge University Press (1996)

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

More Information on web:
http://wiki.event-b.org/index.php/ProR
http://www.eclipse.org/rmf/

Requirements Traceability between Textual Requirements and Formal Models Using ProR

Lukas Ladenberger and Michael Jastram
Institut für Informatik, Universität Düsseldorf

Introduction

Traceability within a system description is a challenging problem of requirements engineering [1]. In particular, formal models of the system are often based on informal requirements, but creating and maintaining the traceability between the two can be challenging. In [2], we presented an incremental approach for producing a system description from an initial set of requirements. The foundation of the approach is a classification of requirements into *artefacts* W (domain properties), R (requirements) and S (specication) [3]. In addition, the approach uses designated *phenomena* as the vocabulary employed by the artefacts. The central idea is that *adequacy* of the system description must be justified, meaning that W ∧ S => R. The approach establishes a traceability, and the resulting system description may consist of formal and informal artefacts. We created tool support for this approach by integrating Rodin [4] and ProR [5]. We designed it with the goal to support the approach described in [2] and to ease the integration of natural language requirements and Event-B [6]. We demonstrate the usefulness of the tool by describing its features (marked with 1 2 3 4) in a typical requirements engineering process. We distinguish the different activities of requirements elicitation, requirements specication, system modelling, requirements validation and requiremen

Requirements Specification

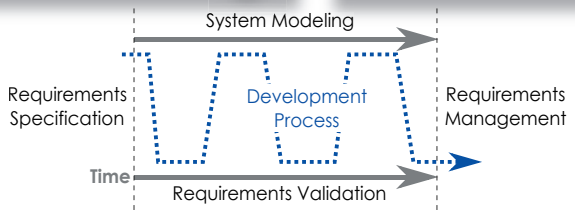
During the requirements specification phase requirements and domain properties are first identified. The resulting classification into corresponding artefacts W (domain properties), R (requirements) and S (specification) and phenomena is the starting point for modelling and validation.

1

Requirements Management

We see that requirements management as a continuation of modelling and validation in a later phase of a project lifecycle. The underlying assumption is that the development of a system description fully finished. The ongoing work includes change management and requirement evolution.

4



Requirements Validation

The objective of requirements validation is to validate the relationship between informal artefacts and formal constructs and to validate the adequacy of the specification elements. The validation relies on the model for tracing artefacts and phenomena and its use for tracing artefacts into formal model and refinements.

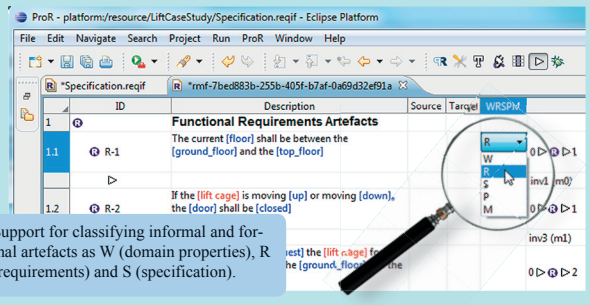
2 3 4

System Modelling

The objective of system modelling is the formal modelling of a subset of the system description as well as the elaboration of the specification elements. Artefacts can be incorporated gradually into the formal model using refinement.

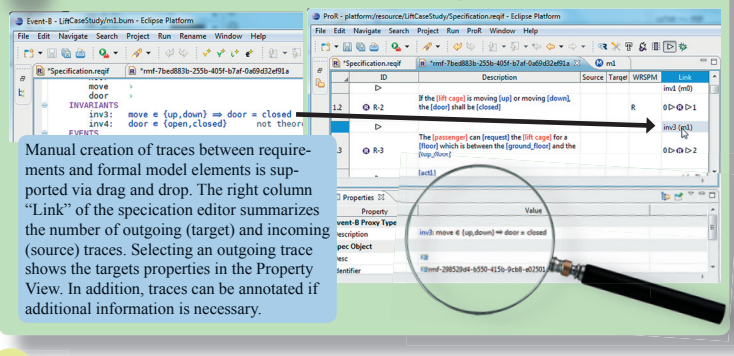
2 3 4

1 Classifying informal and formal artefacts



Support for classifying informal and formal artefacts as W (domain properties), R (requirements) and S (specification).

2 Annotated traces to modelling elements

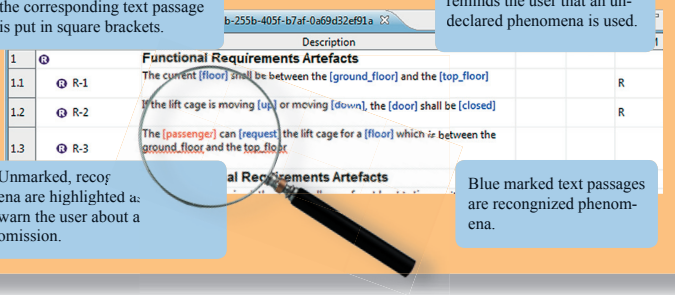


Manual creation of traces between requirements and formal model elements is supported via drag and drop. The right column "Link" of the specication editor summarizes the number of outgoing (target) and incoming (source) traces. Selecting an outgoing trace shows the targets properties in the Property View. In addition, traces can be annotated if additional information is necessary.

3 Tracing of phenomena used in artefacts

In order to add a uses-trace for an phenomenon to an artefact the corresponding text passage is put in square brackets.

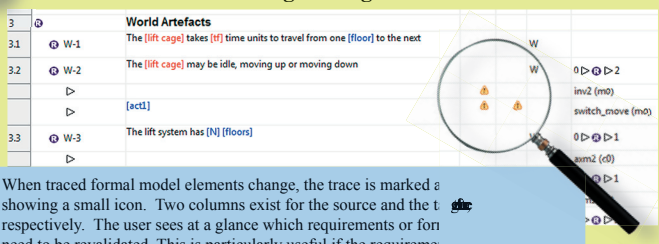
Red marked text passages reminds the user that an undeclared phenomena is used.



Unmarked, recog ena are highlighted a warn the user about a omission.

Blue marked text passages are recongnized phenomena.

4 Changemanagement



When traced formal model elements change, the trace is marked a showing a small icon. Two columns exist for the source and the t respectively. The user sees at a glance which requirements or for need to be revalidated. This is particularly useful if the requireme comes large. By double-clicking on the "suspect" icon, the user can mark the trace as "revalidated" and the icon will be removed.

References:
[1] O. Gotel and A. Finkelstein: *An Analysis of the Requirements Traceability Problem*, IEEE Computer Society (1994)
[2] M. Jastram and S. Hallerstede and L. Ladenberger: *Mixing Formal and Informal Model Elements for Tracing Requirements*, AVOCS 2011 (2011)
[3] C. A. Gunter and M. Jackson and E. L. Gunter and P. Zave: *A Reference Model for Requirements and Specications*, IEEE Software Vol. 17, 37-43 (2000)
[4] J.-R. Abrial and M. J. Butler and S. Hallerstede and T. S. Hoang and F. Mehta and L. Voisin: *Rodin: An Open Toolset for Modelling and Reasoning in Event-B*, STTT Vol. 12, 447-466 (2010)
[5] M. Jastram: *ProR, an Open Source Platform for Requirements Engineering based on RIF*, SEISCONF (2010)
[6] J.-R. Abrial: *Modeling in Event-B: System and Software Engineering*, Cambridge University Press (2010)
[7] J.-R. Abrial: *The B-Book: Assigning programs to meanings*, Cambridge University Press (1996)

UML-B Modelling and Animation Tool Demonstration

Colin Snook, Vitaly Savicks and Michael Butler

University of Southampton

UML-B [1] is a 'UML-like' diagrammatic modelling environment for the Event-B language [2]. UML-B has become established and is being used in a number of industrial research and teaching scenarios. While an understanding of Event-B and its verification is still required, UML-B isolates the modeller, to some extent, from the Event-B models. For some users this is a plus point but for others, who are more adept with formal modelling, a closer integration between diagrams and text formats would be preferred. Therefore, we are developing a suite of alternative tools which, for now, we will collectively call iUML-B (integrated UML-B). Whereas UML-B is a complete and separate model from the generated target Event-B, iUML-B diagrams are embedded into an existing Event-B model and generate contributions to it. In iUML-B much of the modelling is still performed using Event-B with the diagram contributing extra information.

In this demonstration I show and contrast both approaches and highlight the relative benefits of both so that the audience can decide for themselves which approach they prefer. It is hoped to get feedback from the audience in this respect.

Both UML-B and iUML-B are available as plug-ins for the Rodin platform [3] that is established as the modelling and verification platform for formal systems modelling based on the Event-B notation. UML-B provides several different diagrammatic notations: a project diagram showing the relationships between machines and contexts, a Context diagram for describing the static aspects of a system in an entitiy-relationship style, a Class diagram for describing the data aspects of a model in a class-oriented style and a state machine diagram for describing the behaviour of the model. iUML-B [4] currently includes a project diagram plug-in and a state machine diagram plug-in. A new class diagram plugin is being developed which will allow static and variable data to be modelled in a common notation and tool.

The main advantages of using UML-B that are demonstrated are that,

1. it is quick to produce and alter models because a simple placement of a diagram symbol generates many lines of Event-B text,
2. it is more concise because contextual information is often inferred from the placement of items within diagrammatic model elements,
3. the diagrammatic representation provides a ready visualisation of the model,
4. the model can be communicated more easily (especially to Event-B illiterates),
5. it is easier to experiment with different abstractions (and finding useful abstractions can be hard).

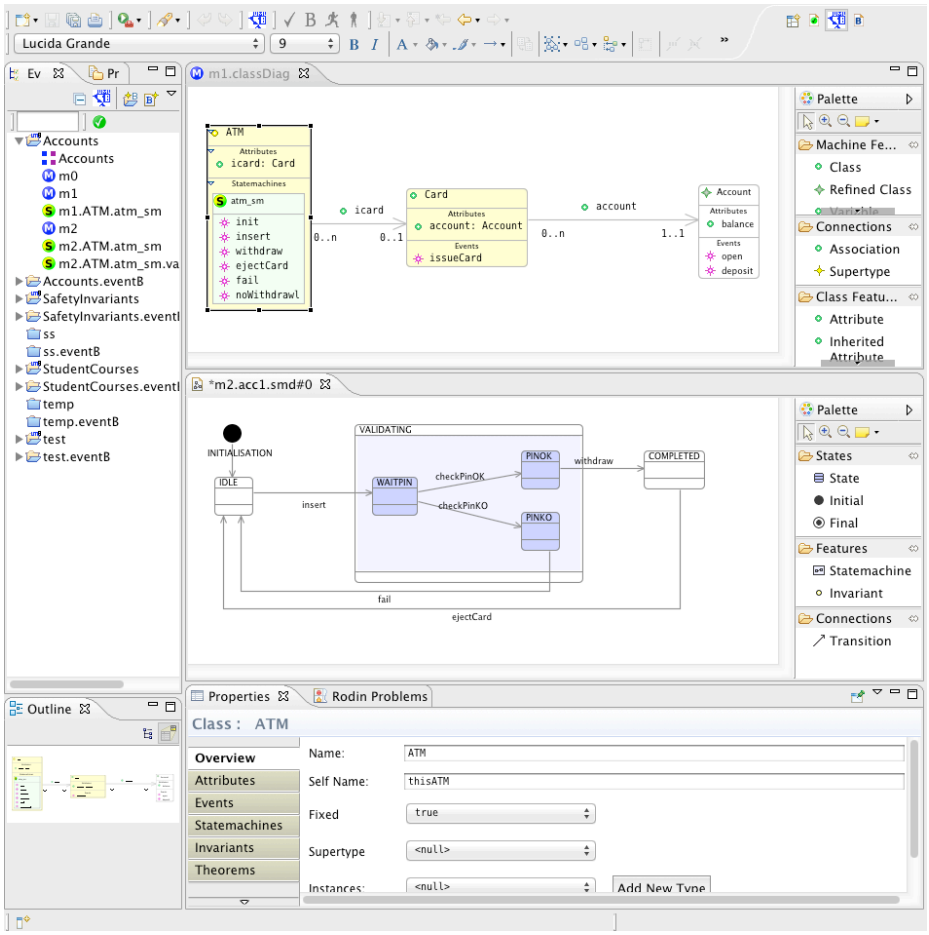


Fig. 1. Modelling with UML-B/iUML-B

It is essential to validate formal models to ensure that a useful model has been achieved. This, by definition, requires a stakeholder to examine the behaviour of the model to provide a verdict on whether the model behaves as required. To do this, we use the animation capabilities of the ProB [5] model-checker plug-in. Since, with UML-B, we have a diagrammatic visualisation of the model, the task can be made more approachable by animating the diagrams. The demonstration shows this feature using our state-machine animation plug-ins [6]

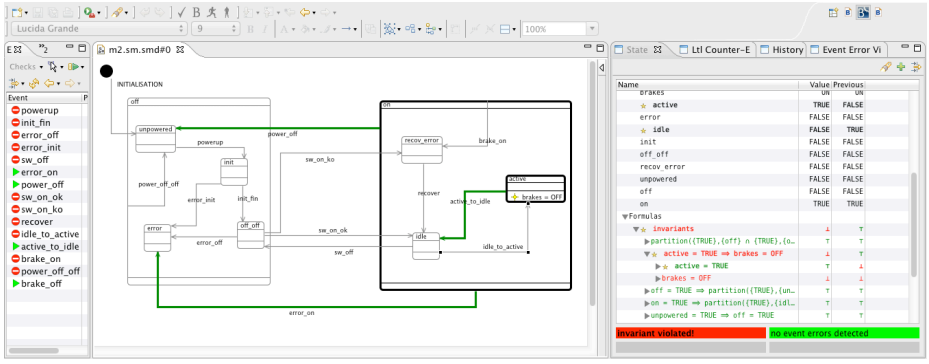


Fig. 2. Animating with iUML-B/ProB

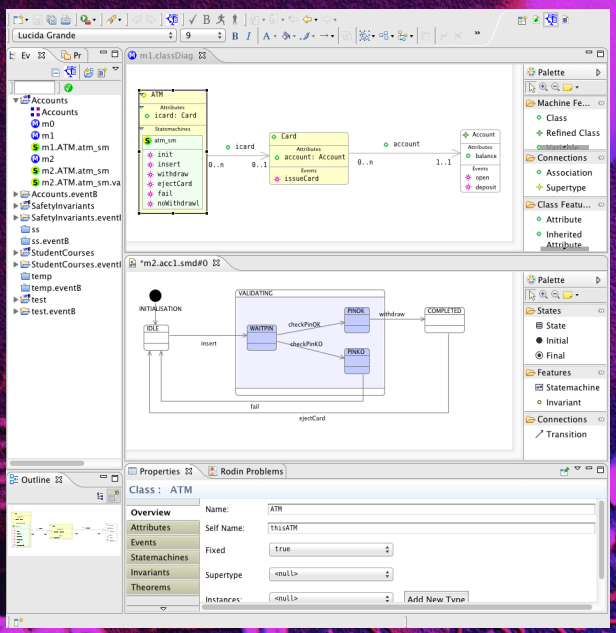
References

1. Snook, C., Butler, M.: UML-B and Event-B: an integration of languages and tools. In: The IASTED International Conference on Software Engineering - SE2008. (February 2008)
2. Abrial, J.R.: Modelling in Event-B: System and Software Engineering. Cambridge University Press (2009)
3. Butler, M., Hallerstede, S.: The Rodin Formal Modelling Tool. BCS-FACS Christmas 2007 Meeting - Formal Methods In Industry, London. (December 2007)
4. Savicks, V., Snook, C., Butler, M.: Event-B Statemachines. `http://wiki.event-b.org/index.php/Event-B_Statemachines` (2011) Accessed April 2012.
5. Leuschel, M., Butler, M.: ProB: A model checker for B. In Araki, K., Gnesi, S., Mandrioli, D., eds.: FME 2003: Formal Methods. LNCS 2805, Springer-Verlag (2003) 855–874
6. Savicks, V., Snook, C., Butler, M.: UML-B Statemachine Animation. `http://wiki.event-b.org/index.php/UML-B_-_Statemachine_Animation` (2010) Accessed April 2012.

UNIVERSITY OF Southampton

UML-B Modelling and Animation

A plug-in for the Rodin platform



UML style modelling with refinement.

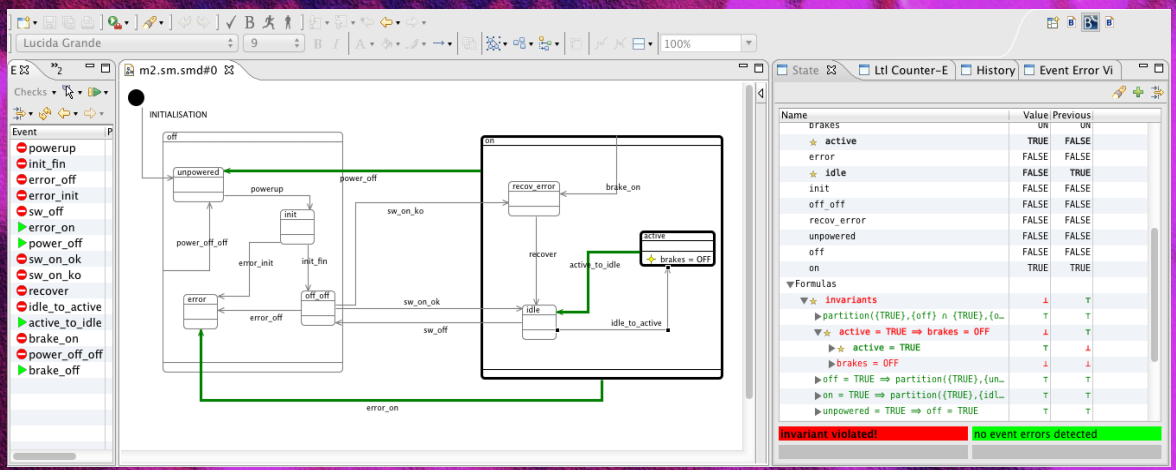
- Visualise models
- Explore useful abstractions
- Create models quickly

Automatic translation to Event-B for verification.

- Static type checker
- Prove consistency (invariants and refinements)

Animation of diagrams using the Pro-B model checker/animator.

- Visualise behaviour
- Validate model
- Demonstrate to stakeholders



Info: http://www.event-b.org/
Contact: cfs@ecs.soton.ac.uk

ADVANCE

Advanced Design and Verification Environment for Cyber-physical System Engineering

www.advance-ict.eu

ADVANCE is an FP7 Information and Communication Technologies Project funded by the European Commission. The objective of ADVANCE is to develop a unified tool framework for automated formal verification and simulation-based validation of cyber-physical systems.

Dr Colin Snook,
Prof. Michael Butler,
Dr Mar yah Said
Vitaly Savicks

Published by:

Consiglio Nazionale delle Ricerche
Piazzale Aldo Moro 7 - 00185 Roma, Italy

Istituto di Scienza e Tecnologie
dell'Informazione “A. Faedo”
Via Moruzzi, 56124 Pisa, Italy

ISBN 978-88-7958-008-3
ISTI Editorial 2012-ED-001