Michael Butler · Alexander Raschke
Thai Son Hoang · Klaus Reichl (Eds.)

LNCS 10817

Abstract State Machines, Alloy, B, TLA, VDM, and Z

6th International Conference, ABZ 2018
Southampton, UK, June 5–8, 2018
Proceedings

Lecture Notes in Computer Science 10817

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

Editorial Board

More information about this series at http://www.springer.com/series/7407

Michael Butler · Alexander Raschke
Thai Son Hoang · Klaus Reichl (Eds.)

Abstract State Machines, Alloy, B, TLA, VDM, and Z

6th International Conference, ABZ 2018
Southampton, UK, June 5–8, 2018
Proceedings

Springer

Editors
Michael Butler
University of Southampton
Southampton
UK

Thai Son Hoang
University of Southampton
Southampton
UK

Alexander Raschke
Universität Ulm
Ulm
Germany

Klaus Reichl
Thales Austria GmbH
Vienna
Austria

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-91270-7 ISBN 978-3-319-91271-4 (eBook)
https://doi.org/10.1007/978-3-319-91271-4

Library of Congress Control Number: 2018942334

LNCS Sublibrary: SL1 – Theoretical Computer Science and General Issues

Printed on acid-free paper

This Springer imprint is published by the registered company Springer International Publishing AG
part of Springer Nature
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

This volume contains the papers presented at ABZ 2018 (6th International ABZ Conference on ASM, Alloy, B, TLA, VDM, and Z) held during June 5–8, 2018, in Southampton, UK. This conference records the latest research developments in state-based formal methods, abstract state machines, Alloy, B, Circus, Event-B, TLS+, VDM, and Z. The 2018 edition followed the success of the previous ABZ conferences in London, UK (2008), Orford, Canada (2010), Pisa, Italy (2012), Toulouse, France (2014), and Linz, Austria (2016).

Four keynotes were presented at ABZ 2018. Janet Barnes and Angela Wallenburg from Altran, UK, jointly gave a talk on making the use of formal methods mainstream within industrial practice and outlined some of the successes and challenges for Altran in using formal methods. Klaus-Dieter Schewe from the Software Competence Centre Hagenberg, Austria, gave a talk on a formal characterization of adaptive distributed systems based on concurrent reflective abstract state machines. Daniel Jackson from MIT gave a talk that argued for the importance of good design in software development. Jean-Raymond Abrial gave a talk that reflected on principles, successes, and challenges around the development and deployment of B and Event-B. We are grateful to the invited speakers for contributing to the success of ABZ 2018.

ABZ 2018 coincided with the 25th anniversary of the first major industrial use of the B Method on METEOR, a railway project for the Paris Metro Line 14, which commenced in 1993. In recognition of this, we organized a panel session at ABZ 2018, with assistance from Laurent Voisin of Systerel, to discuss the evolution of the industrial use of the B Method since 1993.

As successfully practiced at ABZ 2014 and ABZ 2016, the 6th edition of ABZ included special sessions dedicated to a shared real-life case study. The objective of this is to provide points of comparison between ABZ methods and to enrich the set of case studies developed with the methods using a practical and real-life system. This time the case study organizers, Thai Son Hoang and Klaus Reichl, defined a case study from the railway domain with challenging safety requirements. The ABZ 2018 case study is based on the Hybrid ERTMS/ETCS Level 3 standard. These proceedings include an overview of the case study as well as several accepted papers outlining solutions to the case study.

ABZ 2018 received 60 submissions covering a range of formal methods within the scope of the conference. These papers ranged from fundamental contributions, applications in practical contexts, tool developments, and contributions to the case study. Each paper was reviewed by four reviewers and the Program Committee accepted 13 regular research papers, seven papers on the Hybrid ERTMS case study, and 11 short papers presenting work in progress.

We would like to thank the Program Committee members and the external reviewers who carefully reviewed all submissions and selected the best contributions. This event would not exist if authors did not submit their papers. We extend our thanks to all the

people who contributed to the success of ABZ 2018 – reviewers, authors, invited speakers, panelists, Program Committee members, and local organizers. We also thank EasyChair for providing a powerful platform for managing the submissions, reviews, decisions, and proceedings production.

April 2018

Michael Butler
Alexander Raschke
Thai Son Hoang
Klaus Reichl

Organization

Program Committee

Yamine Ait Ameur	IRIT/INPT-ENSEEIHT, Toulouse, France
Paolo Arcaini	National Institute of Informatics, Japan
Richard Banach	The University of Manchester, UK
Egon Boerger	Università di Pisa, Italy
Eerke Boiten	De Montfort University, Leicester, UK
Michael Butler	University of Southampton, UK
Marcel Dausend	e.solutions GmbH, Germany
David Deharbe	Clearsy, France
John Derrick	University of Sheffield, UK
Juergen Dingel	Queen's University, Canada
Roozbeh Farahbod	Huawei Technologies, Germany
Flavio Ferrarotti	Software Competence Centre Hagenberg, Austria
Mamoun Filali-Amine	IRIT, Toulouse, France
Marc Frappier	Université de Sherbrooke, Canada
Leo Freitas	Newcastle University, UK
Angelo Gargantini	University of Bergamo, Italy
Vincenzo Gervasi	University of Pisa, Italy
Uwe Glässer	Simon Fraser University, Canada
Gudmund Grov	Norwegian Defence Research Establishment, Norway
Lindsay Groves	Victoria University of Wellington, New Zealand
Stefan Hallerstede	Aarhus University, Denmark
Klaus Havelund	Jet Propulsion Laboratory, USA
Ian J. Hayes	The University of Queensland, Australia
Rob Hierons	Brunel University, UK
Thai Son Hoang	University of Southampton, UK
Jeremy Jacob	University of York, UK
Regine Laleau	Paris Est Creteil University, France
Peter Gorm Larsen	Aarhus University, Denmark
Thierry Lecomte	ClearSy, France
Michael Leuschel	University of Düsseldorf, Germany
Zhiming Liu	Southwest University, China
Tiziana Margaria	Lero, University of Limerick, Ireland
Atif Mashkoor	Software Competence Centre Hagenberg, Austria
Jackson Mayo	Sandia National Laboratories, USA
Dominique Mery	Université de Lorraine, LORIA, France
Stephan Merz	Inria Nancy, France
Mohamed Mosbah	LaBRI - University of Bordeaux, France
Cesar Munoz	NASA, USA

Uwe Nestmann	TU Berlin, Germany
Jose Oliveira	University of Minho, Portugal
Luigia Petre	Åbo Akademi University, Finland
Andreas Prinz	University of Agder, Norway
Philippe Queinnec	IRIT - Université de Toulouse, France
Alexander Raschke	University of Ulm, Germany
Klaus Reichl	Thales Austria GmbH, Austria
Elvinia Riccobene	University of Milan, Italy
Thomas Rodeheffer	Google, USA
Alexander Romanovsky	Newcastle University, UK
Thomas Santen	TU Berlin, Germany
Patrizia Scandurra	DIIMM - University of Bergamo, Italy
Gerhard Schellhorn	Universitaet Augsburg, Germany
Klaus-Dieter Schewe	Software Competence Center Hagenberg, Austria
Steve Schneider	University of Surrey, UK
Colin Snook	University of Southampton, UK
Michael Stegmaier	University of Ulm, Germany
Jing Sun	The University of Auckland, New Zealand
Loredana Tec	Software Competence Centre Hagenberg, Austria
Laurent Voisin	Systerel, France
Qing Wang	ANU, Australia
Virginie Wiels	ONERA/DTIM, France
Kirsten Winter	The University of Queensland, Australia
Frank Zeyda	University of York, UK

Additional Reviewers

Bacis, Enrico
Bodenmüller, Stefan
Bonfanti, Silvia
Boussabbeh, Maha
Cunha, Alcino
González, Senén
Götz, Stefan
Hallerstede, Stefan
Haneberg, Dominik
Hansen, Dominik
Iliasov, Alexei
Kanakis, Georgios
Knapp, Alexander
Krings, Sebastian
Mammar, Amel
Pei, Yu
Stankaitis, Paulius
Tayebi, Mohammad
Thirioux, Xavier
Tounsi, Mohamed
Tueno Fotso, Steve Jeffrey
Wildman, Luke
Yaghoubi Shahir, Amir
Zohrevand, Zahra

How Bugs Led Us Astray (Abstract of Invited Talk)

Daniel Jackson

MIT

Abstract. When the field of formal methods began, it had broad and noble goals. But somehow, over time, these goals were eclipsed by a more reductionist view. Nowadays, quality is measured by defect counts, and eliminating bugs has become the central focus of our field. In this talk, I'll explain how I think this came about, why it's insidious, and what we can do about it.

My key observation will be that bugs are not the causes of problems but are instead symptoms. To improve our software—to make it more secure, safe and usable—we need to move from symptoms to diagnosis, to determine the underlying causes of poor software and fix those. I will argue that design is essential to achieving this, and that we need to reinvigorate design as a central activity in formal methods research and practice. I will give examples of designs, good and bad, drawn from my ongoing work on conceptual design of software.

Contents

Reals and Hybrid Systems

Refinement

Hybrid ERTMS Case Study

Short Papers

Invited Talks

ABZ Languages and Tools in Industrial-Scale Application

Janet Barnes, Jonathan Hammond, Angela Wallenburg(✉), and Thomas Wilson

Altran UK Limited, 22 St Lawrence Street, Bath BA1 1AN, UK
{janet.barnes,jonathan.hammond,angela.wallenburg,
thomas.wilson}@altran.com

Abstract. We give an early view of an ongoing evaluation of ABZ-style languages and their accompanying tools. The target is specifications of safety- and security-critical (software-rich) systems. Our perspective is that of long-term users of formal methods in all parts of the development life cycle. The evaluation's scope is the *production* of specifications. We list requirements for producing specifications, including semantic needs and the resulting requirements on language expressiveness, as well as requirements on tool support for writing, structuring, exploring, and validating specifications. We define criteria for industrial suitability – in our experience – of ABZ languages. We believe that specification structuring is a major discriminating factor for industrial scale-up. So we present an (informal) classification of such mechanisms and illustrate their use by reference to the largest formal specification written by Altran. Our lack of industrial-scale experience in some languages means we are still learning the best mechanisms to use in some cases. We welcome input on this. Finally we discuss remaining work.

1 Background

The SECT-AIR project [6] is a consortium of UK universities and companies with the aim of delivering a step-change improvement in the affordability of aerospace software. The project is looking at how advanced techniques and tooling, including formal methods, can increase productivity across the whole lifecycle.

Altran UK has nearly 30 years of industrial formal methods experience across the lifecycle, including specification in various languages (e.g. VDM, Z, CCS and CSP) and software formal verification and static analysis (e.g. SPARK [5]).

2 Scope: *Production* of Specifications

One SECT-AIR project workpackage is to reduce barriers to the use of formal specification. Although Altran successfully uses formal methods across the lifecycle, we have more challenges with high-level[1] formal specifications. So as part

[1] By high-level we mean expressed in customer/domain, not software, terms and concepts.

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 3–15, 2018.
https://doi.org/10.1007/978-3-319-91271-4_1

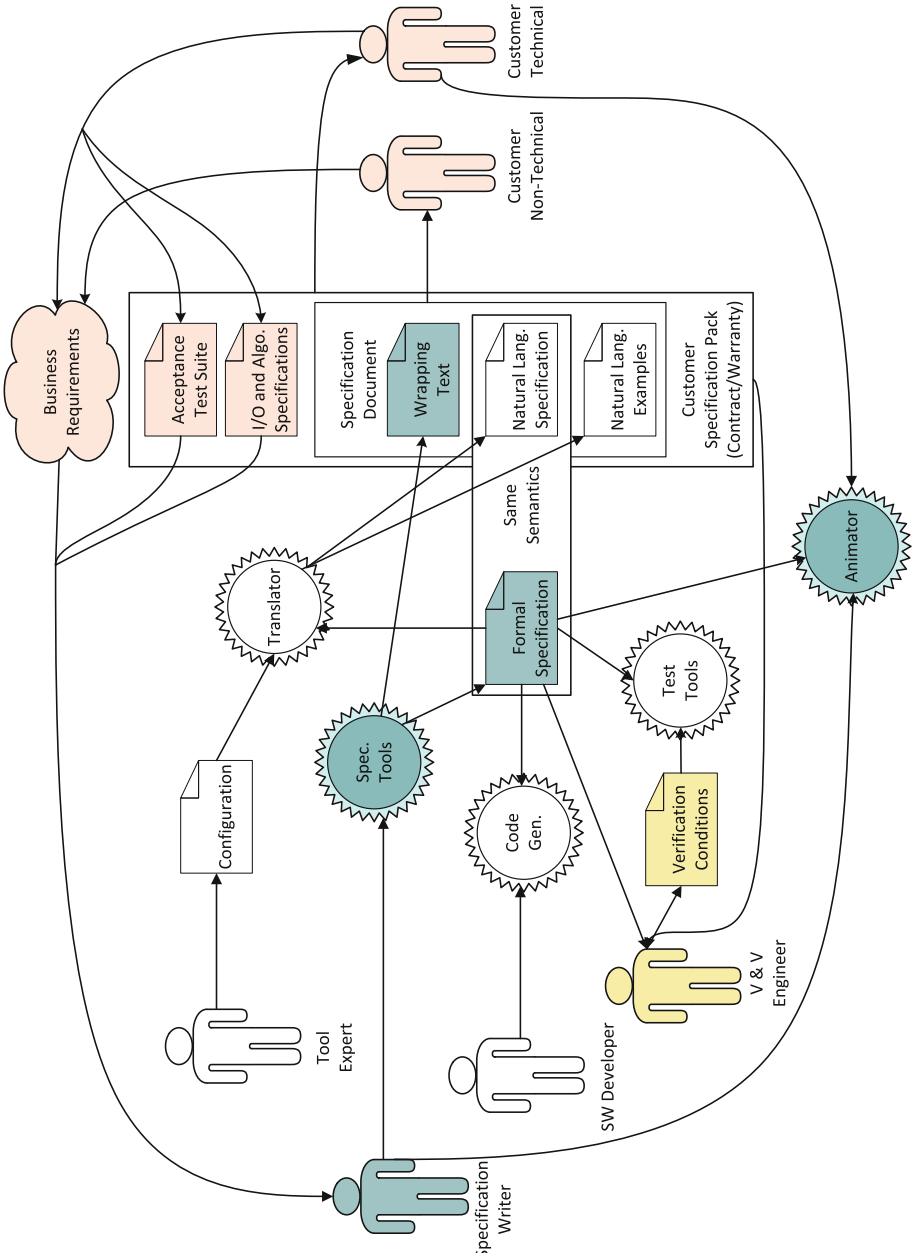


Fig. 1. *Specifications sit at the centre of several key data flows.* The colouring highlights artefacts produced by, and tools driven by, stakeholders, rather than more autonomously. The arrows show the main work flows. Note: the figure shows a wider scope than this paper has space to discuss (e.g. auto-generation of natural language text). The focus of this paper is on the *Spec. Tools* used by the *Specification Writer(s)* to produce the *Formal Specification*, an activity that is beneficial almost regardless of how the formal specification is used later. (Color figure online)

of the SECT-AIR workpackage, Altran is over-hauling its specification solution to improve communication and automation. This solution is to cover the full gamut of specification-related activities, including: writing (editing), exploring (reading, animating, navigating), verifying (e.g. static semantics), validating etc.

Figure 1 summarises the specification solution data flows, showing: (1) stakeholders, (2) artefacts, and (3) tools. The formal specification is at the centre.

An initial downselection of potential solutions has chosen both reactive control and ABZ languages as key parts of meeting SECT-AIR needs. The rest of this paper: makes observations about our industrial formal specification experience (Sect. 3); defines the requirements for our ideal specification solution (Sect. 4); describes our criteria for evaluating possible solutions (Sect. 5); summarises our evaluation process (Sect. 6); discusses ABZ language structuring mechanisms (Sect. 7); finishes with concluding remarks and future work.

3 Observations on Industrial Formal Specification

This section describes our observations based on informal interviewing and discussion with specification writers about their experience of producing specifications over decades at Altran UK, formerly Praxis. (For a wider, including non-engineering, perspective see Sect. 4.) In this section we focus on the notation that we have most commonly used: Z.

3.1 Cultural Setting: Formal Methods Usage Is Normal

Altran UK enjoys a long tradition of substantial industrial use of formal methods. It is regarded as a normal part of multiple stages in the life cycle. Formal methods are used daily by many engineers, not just by a small group of experts.

3.2 Reading Versus Writing Formal Specifications

Altran trains both specification and implementation (coders and verifiers) engineers, using our customised 4 day-long Z courses. Over 100 have been trained to *read* Z and in our experience the majority learn to read the core Z language fairly well. Where appropriate we have also trained customers to read Z.

A much smaller number are needed, and thus trained, to *write* Z. However, only about a dozen have become productive writers, fewer than we'd ideally like.

3.3 Some Observations About Specification Writers

Our most productive specification writers exhibit the following:

1. Particularly skilled in balancing an overview perspective with attention to detail. Unfortunately, this seems to be a rare skill. Most people seem to be good at one or other, but cannot swiftly change between the two.

2. 85% have a PhD, although this is not an Altran job requirement. The PhDs are varied but all in STEM subjects.
3. *Not* an academic expert in ABZ notations and theories, although one exception has a PhD in CSP.

The above sample of writers is too small to draw significant conclusions from. Furthermore, while we are confident that (1) above is a core required skill, we are not sure what to conclude from (2) and (3). Regardless, our observations help illustrate the challenge to find sufficient productive specification writers. We want a specification solution that more people can use to write productively.

3.4 Specification Style and Structure

The largest formal specification produced by Altran is for an air traffic management (ATM) system, called iFACTS [3,17], developed for NATS (UK National Air Traffic Services).

In style, the iFACTS specification – written in Z – bears much resemblance to the detailed specification (called formal design) of the Tokeneer system (also in Z) [4], which is openly available [2]. Each of these specifications:

- Has a high-integrity aspect (safety for iFACTS and security for Tokeneer)
- Specifies software that has a user interface (UI) and receives inputs from users and the wider system environment (other complex ATM software systems for iFACTS and sensors for Tokeneer)
- Is model-based with: data model, initial state, operations and partial operations.

However, iFACTS' main specification is over 3,000 pages of Z and English and is an order of magnitude larger than the Tokeneer specification. So we use the iFACTS specification to illustrate the structuring discussions in this paper.

Why Is the iFACTS Specification so Large? The level of abstraction is a big factor. However, with the exception of some core algorithm details, it is a high-level specification written in ATM domain terms. The domain is a complex one, due to the variety and richness of its data and to the substantial operational rules governing users' responsibilities and thus what iFACTS does. Furthermore, the UI's role in safe and efficient ATM, led to the modelling of individual UI functions, such as buttons and menu items, as distinct Z operations.

Our view is that the specification's content is not implementation detail to be left for software architects and developers (see also refinement Sect. 7.7).

iFACTS Capability. To help explain the iFACTS specification structure, and illustrate it with examples, a basic understanding of what iFACTS does is useful.

iFACTS provides air traffic controllers with advanced support tools based on predictions of the trajectories flights will take up to 18 min into the future. These predictions are used to:

- Detect potential conflicts (with respect to physical separation) between pairs of flights up to 15 min into the future
- Monitor aircraft and if they deviate from a controller's instructions, alert and identify any potential conflicts that result
- Enable a controller to assess in advance the consequences of different instructions they could issue but have not yet issued

The introduction of iFACTS has in the customer's view [17]: "*... revolutionised our operation, freeing up capacity and improving safety, while at the same time reducing delays and cutting carbon emissions.*".

Data Model Packages. The data model is divided into packages (as per the UML term). Each package describes a group of related data items, with the aim to have high cohesion of items within each package and low coupling between packages.

There are a couple of dozen iFACTS packages in the following groupings:

- *UI*: each major UI element (such as each tool) is a separate package.
- *Core algorithms*: each different type of algorithmic processing iFACTS performs (trajectory prediction, deviation monitoring and conflict detection) has its own package.
- *Live domain data*: iFACTS receives and uses a wide range of data that changes in real-time. Each type of data has its own package, for example:
  - Radar data
  - Flight plans
  - Clearances (the ATM term for instructions issued by controllers)
  - Weather forecasts
- *Configured domain data*: such as airspace definitions and aircraft performance models that are fixed at run-time.

The packages form a hierarchy (strictly a partial ordering) where data items in a package may depend on data items in packages beneath it in the hierarchy. The groupings are listed above in top-down order, as elements of the UI present data generated by the core algorithms which in turn use live domain data etc.

The data items in a package are primarily described as classes and relevant associations. UML class diagrams are used to give an overview, with Z schema types, relations and invariants specifying the detail. There is a wide range of package sizes. For example, the radar data package class diagram has two classes, whereas the trajectory prediction package class diagrams total over thirty classes.

Operations and Partial Operations. An *operation* specifies the state change for the entire data model that results from a single external stimulus (such as a user input). A *partial operation* specifies the state change of a single package. An operation is specified as a particular combination (conjunction) of partial operations (plus Ξ-schema predicates for unchanged packages).

The iFACTS specification has approximately 165 operations. Each represents a distinct, discrete piece of functionality that iFACTS provides. About 120 operations can be user initiated and about 50 by either the receipt of data from an external system or due to the expiry of a timer. Note therefore that a handful of the 165 operations have multiple stimuli. For example, a cancel operation may occur either due to user action or because a timer has expired. For iFACTS the number of partial operations varies significantly between packages. For example, the radar data package has just four partial operations, whereas the clearances and some of the UI packages have over thirty each.

4 Specification Solution Requirements

Drawing on our above experience and wider business context (including input from managers as well as engineers), Altran's ideal specification solution would:

R1. *Have a sufficiently low adoption hurdle.* Particularly for specification users, e.g. non-technical readers, for whom minimal/no training should be needed.
R2. *Be amenable to translation to a format suitable for sign-off.* Specifications must be accessible to customers.
R3. *Be amenable to tool assisted validation.* Both readers and writers need ways to check their understanding is correct.
R4. *Have a mathematical underpinning.* Vital to enable meaningful analysis.
R5. *Be expressive enough to capture high-level concepts easily.* Drives ease of understanding and productivity.
R6. *Be scalable to very large systems.* Applies to both tool support and readability.
R7. *Execute as a test oracle or facilitate code generation.* A key business driver to avoid separately duplicating semantics in all of specification, code and test.
R8. *Have a minimum number of languages and tools for the required domains.* We work across domains and cannot afford to maintain competency in a wide range of languages/tools.

There is deliberately no requirement for *formal* proof support. Although we have had notable success with (informal) specification proof [14], we do not believe the cost vs benefit is worthwhile in most cases. That would change if significant automation is possible, for the scale of systems we build.

5 Evaluation Criteria

To evaluate whether languages and tools achieve the above requirements, we have derived the evaluation criteria below. Tracing is given back to the requirements. The weighting reflects the relative importance we place on the criteria.

Evaluation criteria	Weight	Rationale
Format suitable for sign-off (R2)	10	Top requirement to reduce risk with client and enable more use of spec. as basis for contract/warranty
Be amenable to tool assisted validation (R3)	5	Examples: type checking, animation, example generation, consistency checking
Be scalable to very large systems (R6)	8	Both for tools and readability. Often overlooked, key for exploitation
Execute as a test oracle or facilitate code generation (R7)	10	Main cost driver
Writers' satisfaction w.r.t. semantics, expressiveness, abstraction mechanisms (R5)	8	Writing formal specifications pays off by the activity itself, providing writing is sufficiently productive
Have a sufficiently low adoption hurdle (R1)	5	More important for organisations with less experience in applying formal methods
Have a mathematical underpinning (R4)	8	A must for disambiguation, risk reduction, and reliable tools
Be applicable to reactive control systems (R8)	8	Control software projects, often embedded, for example: safety control, engine control, brake control, power control, fly-by-wire, alarm handling, etc.
Be applicable to state based systems (R8)	8	Data and history-rich projects, for example: systems for database and configuration validity, tracking, and book-keeping
Evidence of successful adoption (R1, R6)	8	Is there evidence of repeat and/or widespread (industrial) use (beyond case studies)? Technologies with significant tool building efforts but without repeated use score low. This helps balance any "hyped" technologies. New technologies also score low. Note: this does not exclude new technologies as alternatives, but the reduced score is intended to reflect risk
Tools cost, development and maintenance (R6, R8)	6	Cost matters but has lower weight than most criteria due to expected benefits being worth the investment

These criteria come from both our own experience (e.g. [10]) and business drivers, as well as others' published experiences, such as [16].

6 Evaluation Process

6.1 DAR Process

We are using a Decision Analysis and Resolution (DAR) process to choose our specification solution. DAR is part of Capability Maturity Model Integration (CMMI) [8]. Its purpose is to analyse and document possible decisions using a structured process with evaluation against established criteria.

6.2 First Pass: Creating and Choosing from a Short-List

Based on the requirements (Sect. 4) and an assessment of available language classes, a short-list of 5 possible solutions was created. Grid analysis combined with balanced pairwise comparison led to a preferred solution: a combination of natural language (English) with significant auto-generation from a formal language – either a reactive control or ABZ language, depending on domain. The possible language choices, for more detailed evaluation, are:

1. Reactive control language: Lustre, SCADE, ArgoSim.
2. ABZ language: Event-B, Z, TLA+, VDM

The rest of this paper focuses on the ABZ languages.

6.3 Second Pass: More Detailed Evaluations

Each of the above possible languages is now subject to a more detailed evaluation. This includes trying the languages and associated toolsets on both tutorial examples and real specifications (such as Tokeneer [2]). The final language choices will then be made from the results of these more detailed evaluations.

7 Discussion of Structuring Mechanisms

The ABZ languages have a lot in common; most use standard mathematical notation from axiomatic set theory, lambda calculus, and first-order predicate logic. With regards to typing, they differ. Event-B, Z and VDM are all strongly-typed, unlike TLA+. We find automated type-checking very cost-effective.

Most of all, ABZ languages differ in their structuring mechanisms. In our experience the structuring features are the most discriminating for being able to scale-up to large industrial projects (such as iFACTS). Fundamentally this is because the readability, and thus usability, of a large specification is highly dependent on how it is structured. As Jackson says in his essay [13] on abstraction in software development:

> An abstraction can be represented in more than one way. Whether it is comprehensible depends not only on its formal content but also – vitally – on its representation.

We discuss below some ABZ language structuring concepts and relate them to our experience (e.g. iFACTS). We welcome input from the ABZ community.

7.1 Vertical and Horizontal Abstractions

Jackson [13] distinguishes the concepts of *vertical* and *horizontal abstraction*. A *vertical* abstraction introduces a new (higher-level) concept that corresponds to a particular collection of (lower-level/more detailed) phenomena. For example, a circle, with centre and radius, abstracts a specific class of closed curves. A *horizontal* abstraction simply selects those concepts or phenomena that are significant for the purpose in hand. For example, the London Underground map preserves connectivity and ordering but eschews geographical accuracy.

7.2 Aggregations

A very common idiom in object-oriented systems development is *aggregation*, which encapsulates tightly-coupled state. This is used to group sub-components into a whole component. A classical example would be to model a `Car`, as an aggregation of its components e.g. `Wheels`, `Engine`, `Seats`, and so on. Typically encapsulation is achieved by not allowing the outside of the main component to tamper with its subcomponents, for example disallowing the outside of `Car` to manipulate the `Engine` that is owned by that car. Aggregation is therefore a kind of vertical abstraction. The concept of a `Car` is very useful when discussing relationships with other objects, such as owners, passengers, other traffic etc, without needing to refer to its components.

Aggregation (strictly, composition in UML) is commonly used in our formal specifications. For example, UI elements are sometimes defined as a hierarchy of constituent components. More complex data types may also be aggregations. Two iFACTS examples are trajectories and clearances. Encapsulation is partially, and purely stylistically, enforced by operations only specifying state changes by using the defined partial operations on the packages containing the aggregations.

7.3 Generalization and Specialization

Supertypes (generalization) and subtypes (specialization) are frequently used in our data models. Subtypes tend to arise from specific (specialised) needs, but supertypes can have a useful role as a horizontal abstraction, particularly when specifying the relationships (associations) with other concepts. Each subtype's schema type in Z 'inherits' the supertype's schema type via schema inclusion.

For example, in iFACTS there are several specific types of flight predictions that are natural subtypes of a unifying prediction supertype. The supertype can be used to relate to concepts like a conflict (which arises between a pair of predictions) or to define the different predictions that may be needed for the same flight (such as current expected behaviour and potential future behaviour if a controller were to issue a different clearance).

7.4 Partial Operations

It is very natural to specify separately the effects of an external stimulus on different data model packages and then combine those separate pieces to specify the complete effects. In our Z style the pieces are the partial operations. Event-B has a similar concept for sharing, or splitting, an event between machines [1,12].

If the state changes in two packages are truly independent then decomposition into separate pieces presents no problem. Of course, in general, such independence is not the case. The data model package hierarchy often means that one package's state change depends on another package's state change lower in the hierarchy. This is true in the iFACTS specification. For example, the state changes triggered by receiving new radar data for a flight include:

1. Updating the stored radar data associated with the flight
2. Generating an updated prediction using the new radar data
3. Checking for any unacceptable deviation from the flight's clearance
4. Potentially generating new predictions if deviations are found
5. Displaying the appropriate results of all the above items in the tools

Each item from 2 onwards depends on the results of at least one earlier item. Specifying such dependencies is straightforward if the lower-level package after state variables are visible in higher-level package partial operations (as in the iFACTS specification). Event decomposition may be another approach, see Sect. 7.7.

7.5 Condition Hierarchies

Another important vertical abstraction in our experience is the ability to specify hierarchies of conditions (predicates). Such conditions may be guards, at either whole operation or partial operation level, or the definition of cases or alternative courses of action within an operation or partial operation. Being able to abstract cases/groups of conditions, by naming combinations of predicates, makes the overall logic much clearer than a long list of potentially complex predicates.

The iFACTS specification makes extensive use of Z schemas as predicates to build up complex logic hierarchically. For example, deviation monitoring schemas check a flight conforms to a clearance's altitude. There are different cases for flying level, climbs and descents. Each case has subcases, such as if a descent needs to be now, or only in time to reach the required altitude at a defined point.

7.6 Abstract Data Types

An effective use of abstract data types (ADTs), in the specification of a train control system, is described in [9]. A key motivation for this ADT use is ease of refinement proof. Event-B contexts are used to give abstract types that embody domain concepts of rail network connectivity and train occupancy. Subsequent successive instantiations of the ADT, by more concrete representations, enables formal refinement to a practical implementation.

In our usage of Z, abstract data types (ADTs) are not a central concept.[2] Some schemas could be viewed as ADTs, but only by convention because encapsulation is not enforced. Typically our specifications have not required ADTs. We wrote a Z specification [15] without ADTs in the same domain as the train control system, but we had no intention to refine as far as implementation.

7.7 Role and Use of Refinement

What Is Being Refined? Refinement is the main exploiter of vertical abstraction in formal development. It is important to consider what is being refined:

[2] We refer here to use of ADTs for elements of a specification. Of course an entire model-based Z specification can be viewed as an ADT, particularly when refining it.

1. The environment of the system being developed (particularly its interface with the system), or
2. The internals of the system (such as its architecture and data formats)

We think this distinction is important due to the different degrees of freedom typically involved. Significant parts of the environment are normally a given, or at least require negotiation with various stakeholders. However, there is often much more freedom to choose system internals. The former type of refinement tends to be dominated by requirements and/or systems engineering skills, whereas the latter relates more to architecture and/or software design.

Consequently, in our experience, different people/teams need to do the different refinements. Environment refinement is part of our specification process, whereas internal refinement is part of system development. Depending on the domain this can result in large specifications, like iFACTS, due to the amount of information to be negotiated and captured to use as a basis for development.

Role of Decomposition. To avoid too much detail at once, it seems to us that Event-B uses refinement to decompose and add detail [1,12]. This includes event decomposition, which allows an atomic event to be decomposed into a series of events. This could be used to specify iFACTS' radar update example (Sect. 7.4), as the listed steps appear to be a natural sequence. However, we see such decomposition as an internal refinement issue, best left to software architects and developers, not specifiers. There may be factors other than logical dependencies, such as performance, that affect how best to decompose the event.

Either way, event decomposition clearly supports internal refinement, such as an architecture of communicating subsystems implementing a system-level event. Event decomposition could also apply to environment refinement. However, when an environment interface is known in advance (e.g. it already exists) it is harder to justify the cost of constructing more abstract models, to be able to refine down to the already known interface, even if insights are gained (such as discussed in [7]) through the abstraction.

Where Is Refinement Beneficial? Our experience is that the major sources of error are things like significant domain misunderstandings and missing stakeholders. (For example, see the e-commerce security case study in [11].) Our impression is that formal refinement methods have limited value in eliminating these.

Conversely, formal refinement clearly assists critical property preservation, such as Tokeneer security requirements. Although we routinely do *informal* (especially internal) refinement, formal refinement has not been found cost-effective on our projects so far. This has been due to (1) the large effort required for formal proof of refinement steps, (2) not enough benefit given (insights or issues found) from the typically small refinement steps that have been required for proof, and (3) the cost of maintaining several descriptions of the same system, which can be seen as redundant in the feature-oriented more than insight-oriented industrial setting.

Currently we are devising approaches to do bigger refinement steps, formally stated, but verified by dynamic means. With this we hope to gain meaningful insights during the process of formalising the refinement, as well as increase the productivity of the V&V process.

8 Conclusions and Future Work

We have discussed how scalability, to the kinds of systems we build, is a key discriminating factor between languages and tools. We are still learning the appropriate language mechanisms in the various ABZ languages to manage the amount and detail of information needed for some of our domains. We welcome the ABZ community's input on the structuring idioms discussed in this paper.

We will complete the evaluations of the possible languages and toolsets (see Sect. 6.3). A comparison table will be produced to show the assessment of each against the evaluation criteria. This will drive the selection for our specification solution. Case studies will then be written using this solution, which will be published – as with all key SECT-AIR project outputs – in appropriate fora.

Acknowledgements. This work was made possible by the ATI SECT-AIR project. SECT-AIR is a UK collaborative Aerospace Technology Institute (ATI) research project with the twin aims of reducing aerospace software costs and timescales. SECT-AIR is co-funded by Innovate UK. We would also like to thank our Altran colleagues and John Colley for their valuable input.

References

1. Abrial, J.-R., Hallerstede, S.: Refinement, decomposition, and instantiation of discrete models: application to event-B. Fundam. Inform. **77**(1–2), 1–28 (2007)
2. Altran. Tokeneer project release (2008). https://www.adacore.com/tokeneer. Accessed 13 Dec 2017
3. Astill, J.: Precision instruments. Airport Focus International, September 2012. http://airportfocusinternational.com/precision-instruments/. Accessed 20 Dec 2017
4. Barnes, J., Chapman, R., Johnson, R., Widmaier, J., Cooper, D., Everett, B.: Engineering the Tokeneer enclave protection software. In: 1st IEEE International Symposium on Secure Software Engineering, March 2006
5. Barnes, J.: SPARK: The Proven Approach to High Integrity Software, 3rd edn. Altran, Paris (2012)
6. Bennett, M.: SECT-AIR: a UK initiative to reduce aerospace software cost. Presentation at Verification Futures 2017 Conference, April 2017. http://www.testandverification.com/conferences/verification-futures/vf2017-europe/vf2017-sect-air/. Accessed 3 Jan 2017
7. Butler, M.J.: Mastering system analysis and design through abstraction and refinement. In: Broy, M., Peled, D.A., Kalus, G. (eds.) Engineering Dependable Software Systems. NATO Science for Peace and Security Series, D: Information and Communication Security, vol. 34, pp. 49–78. IOS Press (2013)

8. Chrissis, M.B., Konrad, M., Shrum, S.: CMMI Guidelines for Process Integration and Product Improvement. Addison-Wesley Longman Publishing Co. Inc., Boston (2003)
9. Fürst, A., Hoang, T.S., Basin, D.A., Sato, N., Miyazaki, K.: Large-scale system development using abstract data types and refinement. Sci. Comput. Program. **131**, 59–75 (2016)
10. Hall, A.: What does industry need from formal specification techniques? In: Proceedings of 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques, pp. 2–7 (1998)
11. Hammond, J., Rawlings, R., Hall, A.: Will it work? In: Proceedings of the Fifth IEEE International Symposium on Requirements Engineering, RE 2001, pp. 102-. IEEE Computer Society, Washington, DC (2001)
12. Hoang, T.S., Iliasov, A., Silva, R., Wei, W.: A survey on event-B decomposition. ECEASST **46**, 1–15 (2011)
13. Jackson, M.: Aspects of abstraction in software development. Softw. Syst. Model. **11**(4), 495–511 (2012)
14. King, S., Hammond, J., Chapman, R., Pryor, A.: Is proof more cost-effective than testing? IEEE Trans. Softw. Eng. **26**(8), 675–686 (2000)
15. King, T.: Formalising British rail's signalling rules. In: Naftalin, M., Denvir, T., Bertran, M. (eds.) FME 1994. LNCS, vol. 873, pp. 45–54. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58555-9_86
16. Kossak, F., Mashkoor, A.: How to select the suitable formal method for an industrial application: a survey. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) ABZ 2016. LNCS, vol. 9675, pp. 213–228. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33600-8_13
17. Rolfe, M.: How technology is transforming air traffic management. NATS Blog, Jul 2013. https://nats.aero/blog/2013/07/how-technology-is-transforming-air-traffic-management/. Accessed 20 Dec 2017

Distributed Adaptive Systems
Theory, Specification, Reasoning

Klaus-Dieter Schewe[1(✉)], Flavio Ferrarotti[2], Loredana Tec[3], and Qing Wang[4]

[1] Laboratory for Client-Centric Cloud Computing, Linz, Austria
kdschewe@acm.org
[2] Software Competence Center Hagenberg, Hagenberg, Austria
flavio.ferrarotti@scch.at
[3] FAW GmbH, Hagenberg, Austria
loredana.tec@gmail.com
[4] The Australian National University, Canberra, Australia
qing.wang@anu.edu.au

Abstract. A *distributed system* can be characterised by autonomously acting agents, where each agent executes its own program, uses shared resources and communicates with the others, but otherwise is totally oblivious to the behaviour of the other agents. In a distributed *adaptive* system agents may change their programs, and enter or leave the collection at any time thereby changing the behaviour of the overall system. This article first develops a language-independent axiomatic definition of distributed adaptive systems and then presents *concurrent reflective Abstract State Machines* (crASMs), an abstract machine model for their specification. It can be proven that any distributed adaptive system as stipulated by the axiomatisation can be step-by-step simulated by a crASM. Proofs about crASMs can be grounded in a *multiple-step logic*, which extends known complete one-step logics for deterministic and non-deterministic ASMs.

1 Introduction

A *distributed system* can be characterised by autonomous agents, where each agent executes its own program, uses shared resources and communicates with the other agents, but otherwise is totally oblivious to the behaviour of others. Thus, the asynchronous concurrent execution of the agents' programs is the most important characteristic. There are numerous models of concurrency in the literature or implemented in current hard- and software systems and underlying distributed algorithms, specification and programming languages: distributed algorithms [19], process algebras [16,22,23,27], actor models [15], trace theory [20,41], Petri nets [24,25,40], etc.

The research reported in this paper was partially supported by the **Austrian Science Fund (FWF)[I2420-N31]** for the project: *Higher-Order Logics and Structures*.

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 16–30, 2018.
https://doi.org/10.1007/978-3-319-91271-4_2

Abstract State Machines (ASMs) have been used since their introduction to model sequential and concurrent systems (see [7, Chaps. 6, 9] for references), but only recently the theory counterpart of the celebrated sequential ASM thesis [13] has been discovered [4]. This *concurrent ASM thesis* provides an axiomatic characterisation of concurrency based on the intuitive understanding of computations of multiple autonomous agents, which execute each a sequential process, run asynchronously, each with its own clock, and interact with (and know of) each other only via reading/writing values of designated locations. It shows that concurrent algorithms are captured by *concurrent Abstract State Machines*, i.e. families of agents each equipped with a sequential ASM, the semantics of which is defined by *concurrent ASM runs*, which overcome the limitations of Gurevich's distributed ASM runs [12] and generalise Lamport's sequentially consistent runs [18]. This consitutes a behavioural theory that provides a foundation for concurrent sequential algorithms and their rigorous specification, refinement and verification. Exploiting the behavioural theory of unbounded parallelism [1,2,8] the concurrent ASM thesis extends naturally to families of agents, each executing a parallel algorithm.

Adaptive systems have attracted a lot of interest in research, in particular in connection with systems of (cyber-physical) systems [26], biologically-inspired systems [39] or observer/controller architectures [36,38]. Adaptivity refers to the ability of a system to change its own behaviour. In [9] a behavioural theory for reflective, sequential algorithms was proven. The integration of the theories of unbounded parallelism, reflection and concurrency (a proof was sketched in [33]) provides a uniform behavioural theory for distributed adaptive systems.

Besides rigorous design and stepwise refinement-based development verification is a key concern for rigorous systems development. All rigorous methods are supported by appropriate logics such as the logic for Event-B [35], the logic for ASMs [37], and the logic for TLA$^+$ [21]. These logics support primarily the verification of properties for a single machine step exploiting also concepts from dynamic logic [14], though extensions in temporal logic (see e.g. [28] for ASMs and [17] for TLA$^+$) have also been investigated to enable the reasoning about complete runs. In order to reason about distributed adaptive systems we investigate an extension of the one-step logic for ASMs to capture concurrency and reflection.

This logic has been extended in [10] to deal with non-deterministic ASMs, for which the unsolved problem of non-determinism and the handling of multi-set functions for synchronisation had to be solved. This was further streamlined, partly corrected and extended to concurrent ASMs in [11] based on the observation that concurrent runs can be mimicked by non-deterministic ASMs. To obtain full reasoning power, we will show how to move from a one-step logic to a multiple-step logic, as a single step of an agent in a concurrent system may correspond to multiple steps of the whole concurrent system. Furthermore, we will substitute the extra-logical rules that are used in the ASM logic by variables that are to be interpreted in a state, but yield rules, by means of which we can capture reflection.

In this paper we first revisit the behavioural theory of distributed adaptive systems. In Sect. 2 we first emphasise the motivation leading to the language-independent axiomatisation. In Sect. 3 we briefly present *concurrent reflective Abstract State Machines*, which capture distributed adaptive system. For the proofs of the behavioural theory we refer to the literature [1,4,8,9,13,33]. Section 4 is then dedicated to the development of a logic for concurrent reflective ASMs, where we first consider concurrency [11], and then integrate reflection into the logic. We conclude with an outlook on further research directions in Sect. 5.

2 Axiomatic Definition of Distributed Adaptive Systems

In this section we provide a language-independent, axiomatic definition of distributed adaptive systems, which integrates the postulates for parallel [8], reflective [9] and concurrent algorithms [4]. Naturally, in a distributed system we combine different (parallel, reflective) algorithms, each associated with a locality (which we may consider as an abstraction from physical processors).

Postulate 1 (Distribution Postulate). A *distributed adaptive system* (DAS) is given by a set $\mathcal{A}$ of agents a, each equipped with a parallel, reflective algorithm $alg(a)$. Furthermore, there is a set $\mathbb{L}$ of localities and assignment $loc : \mathcal{A} \to \mathbb{L}$.

We write $\mathcal{D} = \{(a, alg(a)) \mid a \in \mathcal{A}\}$ for a DAS $\mathcal{D}$. For our further discussion the localities $\mathbb{L}$ will be of minor importance, as we emphasise behaviour only. The most important aspect is that the algorithms $alg(a)$ work together asynchronously using shared locations and messages, but otherwise are oblivious to each other. In the sequential we will develop an axiomatisation for parallel, reflective algorithms and concretise the meaning of asynchronous, concurrent behaviour.

2.1 Sequential Time

Ignoring non-determinism, sequential or parallel algorithms operate in sequential time, i.e. they operate on a set $\mathcal{S}$ of states together with a subset $\mathcal{I}$ of initial states, such that an algorithm proceeds by means of a *transition function* $\mathcal{S} \to \mathcal{S}$, which map states $S \in \mathcal{S}$ to successor states $\tau(S)$. This defines the notion of a *run* as sequences of states $S_0, S_1, S_2, \ldots$ with $S_0 \in \mathcal{I}$ and $S_{i+1} = \tau(S_i)$. Algorithms are called *behaviourally equivalent* iff they have exactly the same runs.

Extending this to include reflection, i.e. the ability of an algorithm to modify its own behaviour, we may think of pairs (S_i, P_j) comprising a state S_i and a parallel, reflective algorithm P_j, so we obtain transitions $\tau : (S_i, P_i) \mapsto (S_{i+1}, P_{i+1})$. The algorithm P_i defines a state transition τ_{P_i} that can be applied to state (S_i, P_i), but we may also consider just the restriction to proper states S_i for which we use the notation $P|_{S_i}$. As observed in [9] we can capture the state-algorithm pairs by an extension of the states themselves, so for sequential time

the only extension we need to emphasise that the extension represents an algorithm, there is a unique algorithm in initial states, and the transition in state (S, P) is defined by P.

Postulate 2 (Sequential Time Postulate). A parallel, reflective algorithm $\mathcal{A}$ consists of

- a non-empty set $\mathcal{S}_{ext}$ of *extended states*, such that each $\hat{S} \in \mathcal{S}_{ext}$ can be written as a pair (S, P) with a *proper state* S and a representation of a (parallel, reflective) algorithm P,
- a non-empty subset $\mathcal{I} \subseteq \mathcal{S}_{ext}$ of *initial states* such that for all $(S, P), (S', P') \in \mathcal{I}$ the algorithms P and P' are behaviourally equivalent, and
- a one-step transformation function $\tau : \mathcal{S}_{ext} \to \mathcal{S}_{ext}$ such that $\tau(S, P) = \tau_P(S, P)$ holds for all extended states.

We preserve the notion of behavioural equivalence for algorithms with the same runs, and in addition call two algorithms P and P' *essentially equivalent* iff their restrictions $P|_S$ and $P'|_S$ to proper states are behaviourally equivalent.

2.2 Abstract States

For sequential and parallel algorithms states are meta-finite Tarski structures defined over a fixed signature Σ, i.e. a set of function symbols, by means of interpretation in a base set B. States, initial states and transitions are closed under isomorphisms. As extended states include (an encoding of) a parallel, reflective algorithm, we cannot simply require the set of states and the one step transformation function to be closed under isomorphisms, as that would interfere with the concept of behavioural equivalence.

Therefore, we say that two states (S, P) and (S', P') are *essentially isomorphic* if S and S' are isomorphic first-order structures of some vocabulary Σ and $P|_\Sigma$, $P'|_\Sigma$ are behaviourally equivalent.

Postulate 3 (Abstract State Postulate). For a parallel, reflective algorithm $\mathcal{A}$ there exist signatures $\Sigma \subseteq \Sigma_{ext}$ such that

- extended states of $\mathcal{A}$ are structures of signature Σ_{ext},
- for every extended state (S, P) of $\mathcal{A}$ the proper state S is a structure of signature Σ, and the algorithm P is represented as a structure of signature $\Sigma_{wt} = \Sigma_{ext} - \Sigma$,
- the one-step transformation τ of a RSA $\mathcal{A}$ does not change the base set of any extended state,
- the sets $\mathcal{S}_{ext}$ and $\mathcal{I}$ of extended states and initial states, respectively, are closed under essential isomorphisms, and
- if two extended states (S, P) and (S', P') are essentially isomorphic via an isomorphism ζ from S to S', then $\tau(S, P)$ and $\tau(S', P')$ are also essentially isomorphic via ζ.

The sequential time and abstract state postulates together allow us to define the notion of update set. If f is an n-ary function symbol in the signature Σ_{ext}, and $\bar{b} = (b_1, \ldots, b_n)$ is an n-tuple of values from some base set B, then the pair $(f, \bar{b})$ is called a *location*. A pair (ℓ, b) with a location ℓ and a value $b \in B$ is called an *update*, and an *update set* is a set Δ of updates. For later use, an *update multiset* is a multiset of updates. An update set Δ is *consistent* iff $(\ell, b) \in \Delta \wedge (\ell, b') \in \Delta \Rightarrow b = b'$ holds for all locations ℓ and all values $b, b' \in B$.

If S is an (extended) state defined over the base set B, and Δ is an update set with values in B, then $S + \Delta$ denotes another state, where the value at a location ℓ is defined by

$$val_{S+\Delta}(\ell) = \begin{cases} b & \text{if } (\ell, b) \in \Delta \\ val_S(\ell) & \text{else} \end{cases}$$

In addition, for an inconsistent Δ we set $S + \Delta = S$. Then it is easy to see that for each (extended) state S there exists a unique, consistent update set such that $tau(S) = S + \Delta(S)$ holds. Consequently, each run defines to a sequence of update sets.

Note that the postulates do not impose any restriction on the size of update sets. Algorithms that simultaneously update many locations are captured in the same way as algorithms with minimal update sets containing only a single update.

2.3 Concurrency

Returning to DAS we may abstract from details of how the agents interact by assuming that there are certain shared locations that can be updated by several agents, that is while each algorithm $alg(a)$ has its own extended signature $\Sigma_{a,ext}$, any function symbol f appearing in at least two of these signatures is considered to be *shared*. Locations defined by f are then shared locations. As emphasised in [4] this covers also interaction through messages, for which we may assume mailboxes as shared locations. A mailing subsystem can be left abstract or explicitly described by some agent(s).

To perform a step in some state S, an agent a computes an update set $\Delta_a(S)$. Actually, while S is defined over the union of signatures of all agents, the update set $\Delta_a(S)$ is built on top of the substructure defined by the signature of $alg(a)$, and consequently, $\Delta_a(S)$ only contains updates to locations accessible by $alg(a)$. The fact that the agents act asynchronously means that while an agent executes its step, other agents may already complete a step that has started earlier or later than the current step of agent a. This leads to the following postulate capturing the asynchronous concurrent behaviour of a DAS.

Postulate 4 (Concurrency Postulate). A DAS $\mathcal{D} = \{(a, alg(a)) \mid a \in \mathcal{A}\}$ defines *concurrent $\mathcal{D}$-runs* $S_0, S_1, \ldots$ starting in some initial state S_0, such that each state S_n $(n \geq 0)$ yields a next state S_{n+1} by a finite set $\mathcal{A}_n$ of agents

simultaneously completing the execution of their current $alg(a)$-step they had started in some preceding state S_j ($j \leq n$ depending on a), i.e.

$$S_{n+1} = S_n + \bigcup_{a \in \mathcal{A}_n} \Delta_a(S_j).$$

Note that by taking the union of update sets we emphasise simultaneous updates in the same way as state transitions by a single algorithm permits update sets of arbitrary size. The concept of interleaving is covered by the special case, where the sets $\mathcal{A}_n$ of agents interacting in state S_n is restricted to contain only a single agent.

2.4 Bounded Exploration

An algorithm must have a finite description, so whatever is needed to determine the update set in some state must be contained in this finite description. For sequential algorithms it is thus obvious that in every step only finitely many locations can be evaluated in a state S to determine the update set $\Delta(S)$. In the behavioural theory of sequential algorithms [13] it is therefore postulated that there is a fixed, finite set of ground terms W, called *bounded exploration witness*, such that whenever two states S, S' coincide on W, the update sets $\Delta(S)$ and $\Delta(S')$ are equal.

For parallel algorithms there may be an arbitrary number of parallel branches in a computation step—these branches were called *proclets* in [1]—and their number may depend on the state, but nonetheless the finite description of the algorithm must contain the means to determine the update set. It has therefore been concluded that instead of ground terms a bounded exploration witness W for a parallel algorithm should comprise multiset comprehension terms. When a multiset comprehension term is evaluated in a state, it will yield a multiset of arbitrary size, and each combination of elements of the multisets interpreting terms in W should correspond to a proclet. This justifies the bounded exploration postulate in [8].

Regarding reflective algorithms the problem is that in general we must expect that each algorithm P_i represented in state (S_i, P_i) has its own bounded exploration witness W_i. However, from the construction of W_i in [8] we know that W_i is somehow contained in the finite representation of P_i. So there must exist a finite set of terms W such that its interpretation in an extended state yields both values and terms, and the latter represent W_i. Consequently, terms over the subsignature Σ must be allowed as values in an extended base set B_{ext}, and then the interpretation of W and of its interpretation (removing non-logical constants such as keywords by using an operator *raise*) in an extended state suffice to determine the update set in that state. This will lead to our *bounded exploration postulate* for reflective algorithms. Thus, we need an extension of the notion of *strong coincidence* over a set of multiset comprehension terms.

If (S, P) and (S', P') are states of signature Σ_{ext}, W_{st} is a set of multiset comprehension terms over subsignature Σ and W_{wt} is a set of multiset

comprehension terms over signature $\Sigma_{ext} - \Sigma$, then (S, P) and (S', P') *strongly coincide* over $W_{st} \cup W_{wt}$ iff the following holds:

- For every $t \in W_{st}$ we have $val_{(S,P)}(t) = val_{(S',P')}(t)$.
- For every $t \in W_{wt}$ we have $val_{(S,P)}(t) = val_{(S',P')}(t)$ and $val_{(S,P)}(raise(t)) = val_{(S',P')}(raise(t))$, where $raise(t)$ denotes the interpretation of t as a term of signature Σ.

Postulate 5 (Bounded Exploration Postulate). For every parallel, reflective algorithm $\mathcal{A}$ of signature Σ_{ext} there is a finite set W_{st} of multiset comprehension terms over signature Σ and a finite set W_{wt} of multiset comprehension terms over signature $\Sigma_{ext} - \Sigma$ such that $\Delta(S, P) = \Delta(S', P')$ holds, whenever extended states (S, P) and (S', P') of $\mathcal{A}$ strongly coincide on $W_{st} \cup W_{wt}$.

If a set of multiset comprehension terms $W = W_{st} \cup W_{wt}$ satisfies the reflective bounded exploration postulate, we call it a *bounded exploration witness* for the algorithm $\mathcal{A}$.

2.5 Background

Each computation uses some background [3], which is usually left implicit for sequential algorithms [13]. In this case it contains the reserve of values not used in a current state, but available to be added to the active domain in any state transition, truth values and their connectives, and the value `undef` used to capture partial functions. When dealing with parallel algorithms we have to add at least a pairing constructor and a multiset constructor together with necessary operators on tuples and multisets. Also reflection requires some background, as it must be possible to refer to tuples to capture terms that represent algorithms, which may comprise extra-logical constants such as keywords.

Most important, reflection requires the presence of a *raise* function that takes values in the extended domain that are terms and removes all extra-logical elements, so that $raise(t)$ can be interpreted to yield a value in the domain.

Postulate 6 (Background Postulate). A distributed adaptive system $\mathcal{D}$ is associated with a background class $\mathcal{K}$ comprising at least a binary *tuple constructor* and a *multiset constructor* of unbounded arity, and a background signature Σ_B comprising at least the following function symbols (all static except `reserve`):

- nullary function symbols `true`, `false`, `undef` and $\oslash$,
- unary function symbols `reserve`, `atomic`, `Boole`, $\neg$, `first`, `second`, $\{\!\{\cdot\}\!\}$, $\biguplus$ and `AsSet`,
- binary function symbols $=$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, $\uplus$ and $(\,,\,)$, and
- *raise* mapping terms over the extended signature Σ_{ext} to terms over Σ.

The background class must also provide the means for the representation of algorithms by terms, but the postulate leaves open, which representation is used. If a representation using relations is used, this can be subsumed by the extended signature. If instead tree structures are exploited (as emphasised in [31]), then a sophisticated tree background structures including operators on trees and hedges as in [34] is required. For a more detailed discussion of the background see [8, Sect. 3].

To summarise, a *distributed adaptive system* (DAS) is a system $\mathcal{D}$ satisfying the distribution, concurrency and background postulates such that, if $\mathcal{D} = \{(a, alg(a)) \mid a \in \mathcal{A}\}$, then fir every $a \in \mathcal{A}$ $alg(a)$ satisfies the sequential time, abstract state and background postulates.

3 Concurrent Reflective ASMs

We define a *concurrent reflective* ASM (crASM) as a family $\{\mathcal{M}_a\}_{a\in\mathcal{A}}$ of reflective, parallel ASMs indexed by a set of agents $\mathcal{A}$ together with a common background structure to deal with the self-representation of an ASM, for which we exploit the tree algebra from [34] (for more details see [31]). We further let the reserve contain infinitely many function names of arity $r \geq 0$ and the background satisfy the minimum requirements as set out by the background postulate.

Each reflective parallel ASM $\mathcal{M}_a$ has a signature $\Sigma_{a,ext} = \Sigma_a \cup \{self_a\}$ with a nullary function symbol $self_a$, which is used to store the signature and rule of the machine. As usual function symbols in Σ_a have an arity r; they are static, dynamic or derived, and dynamic function symbols may be controlled, monitored or shared [7]. We may also distinguish between function symbols for finite and algorithmic parts, and for bridge functions, respectively, to emphasise meta-finite structures.

Each reflective parallel ASM $\mathcal{M}_a$ has a rule, for which we adopt common ASM rules with extensions for partial updates [30], communication [6] as well as location operators for synchronisation [29]. Rules are composed inductively using the following constructors:

assignment. $f(t_1, \ldots, t_{ar_f}) := t_0$ with terms t_i built over $\Sigma_{a,ext}$,
partial update. $f(t_1, \ldots, t_{ar_f}) \Leftarrow^{op} t_0$ with terms t_i and an operator op,
branching. `IF` φ `THEN` r_+ `ELSE` r_-,
parallel composition. `FORALL` x `WITH` $\varphi(x)$ $r(x)$,
bounded parallel composition. $r_1 \ldots r_n$,
sequence. $r_1; \ldots; r_n$,
let. `LET` $x = t$ `IN` $r(x)$,
location operator. `USE` $loc(t) = \rho$ `IN` r,
send. `SEND`(⟨message⟩, from:⟨sender⟩, to:⟨receiver⟩),
receive. `RECEIVE`(⟨message⟩, from:⟨sender⟩, to:⟨receiver⟩), and
consume. `CONSUME`(⟨message⟩, from:⟨sender⟩, to:⟨receiver⟩).

Each rule, when applied in an extended state S yields an update set $\Delta(S)$. For assignments, branching, let, bounded and unbounded parallelism and sequence the definition of $\Delta(S)$ is defined in [7]. The communication rules send, receive and consume maintain updates to mailboxes [6]: send places a message in the outbox of the sender, receive takes a message in the inbox of the receiver and manipulates it, and consume deletes a message in an inbox. For this in- and outmailboxes must be part of the signature. A partial update $f(t_1, \ldots, t_{ar_f}) \Leftarrow^{op} t_0$ evaluates the values of $f(t_1, \ldots, t_{ar_f})$ and t_0 in state S, say that these are b_S and b_0. However, instead of assigning directly $op(b_S, b_0)$ to the location $(f, (val_S(t_1), \ldots, val_S(t_{ar_f})))$, all these partial updates at this location are combined into a single update at this location [30]. Finally, the use-rule takes an update multiset produced by r and applies the multiset function ρ to all terms matching t, which reduces the update multiset to an update set [29].

With respect to concurrency the semantics of a crASM is then easily defined by *concurrent ASM runs* as defined in [4], i.e. we have

$$S_{n+1} = S_n + \bigcup_{a \in \mathcal{A}_n} \Delta(\mathcal{M}_a, S_{lastRead(a,n)}) \ ,$$

where $S_{lastRead(a,n)}$ denotes the state in which a performed its reads of all monitored and shared locations it uses for the current step (so that $lastRead(a, n) \leq n$).

The main extension to concurrent ASMs is that the individual machines $\mathcal{M}_a$ are now reflective, i.e. in each step the rule to be applied is the one stored in $self_a$, and this location may be updated by the rule like any other location. Note that the definition allows the variable $self_a$ to be itself shared. This permits the modification of $\mathcal{M}_a$ by a different agent. In particular, it enables a complete separation of agents for monitoring and adaptation.

4 A Logic for Concurrent Reflective ASMs

To support logical inferences on distributed adaptive systems we can exploit that they are captured by crASMs. It suffices to develop a logic for crASMs, which is what this section will address.

4.1 A Logic for Non-deterministic ASMs

In [10] we developed a complete logic for database ASMs. Different to the logic for ASMs developed by Stärk and Nanchen [37] the logic makes explicit use of meta-finite states, solves the problem of non-determinism, and adds multiset-based synchronisation terms. We then observed in [8] that meta-finite states were also present in parallel ASMs, so the logic actually captures non-deterministic parallel ASMs [11].

Let $\Upsilon = \Upsilon_f \cup \Upsilon_a \cup \mathcal{F}_b$ be a signature comprising function symbols for the finite and algorithmic parts, and for bridge functions, respectively. Fix a countable set $\mathcal{X}_f$ of first-order variables, denoted with standard lowercase letters $x, y, z, \ldots$, that range over the primary finite part of the states (i.e. the finite set B_f). The set $\mathcal{T}_{\Upsilon,\mathcal{X}_f}$ of first-order terms of vocabulary Υ is defined as usual in meta-finite model theory [10]. That is, $\mathcal{T}_{\Upsilon,\mathcal{X}_f}$ is constituted by the set $\mathcal{T}_f$ of *state terms* and the set $\mathcal{T}_a$ of *algorithmic terms*. The set of terms $\mathcal{T}_f$ is the closure of the set $\mathcal{X}_f$ of variables under the application of function symbols in Υ_f. The set of algorithmic terms $\mathcal{T}_a$ is defined inductively: If $t_1, \ldots, t_n$ are terms in $\mathcal{T}_f$ and f is an n-ary bridge function symbol in $\mathcal{F}_b$, then $f(t_1, \ldots, t_n)$ is an algorithmic term in $\mathcal{T}_a$; if $t_1, \ldots, t_n$ are algorithmic terms in $\mathcal{T}_a$ and f is an n-ary function symbol in Υ_a, then $f(t_1, \ldots, t_n)$ is an algorithmic term in $\mathcal{T}_a$; nothing else is an algorithmic term in $\mathcal{T}_a$.

If S is a meta-finite state of signature Υ, then a *valuation* or *variable assignment* ζ is a function that assigns to every variable in $\mathcal{X}_f$ a value in the base set of the finite part B_f of S. The value $val_{S,\zeta}(t)$ of a term $t \in \mathcal{T}_{\Upsilon,\mathcal{X}_f}$ in the state S under the valuation ζ is defined as usual in first-order logic. The *first-order logic of meta-finite states* is defined as the first-order logic with equality which is built up from equations between terms in $\mathcal{T}_{\Upsilon,\mathcal{X}_f}$ by using the standard connectives and first-order quantifiers. Its semantics is defined in the standard way. The truth value of a first-order formula of meta-finite states φ in S under the valuation ζ is denoted as $[\![\varphi]\!]_{S,\zeta}$.

Without loss of generality, a variable assignment ζ as previously defined for first-order variables that range over B_f, can be extended to first-order variables that range over B_a as well as to second-order variables that range over finite sets. We use $fr(t)$ to denote the set of (both first-order and second-order) free variables occurring in t.

The *set of terms in the logic for non-deterministic ASMs* is constituted by the set $\mathcal{T}_f$ and the set $\mathcal{T}_a$ of *algorithmic terms* expressed as follows:

- $x \in \mathcal{T}_f$ for $x \in \mathcal{X}_f$ and $fr(x) = \{x\}$;
- $\mathtt{x} \in \mathcal{T}_a$ for $\mathtt{x} \in \mathcal{X}_a$ and $fr(\mathtt{x}) = \{\mathtt{x}\}$;
- $f(t) \in \mathcal{T}_f$ for $f \in \Upsilon_f$, $t \in \mathcal{T}_f$ and $fr(f(t)) = fr(t)$;
- $f(t) \in \mathcal{T}_a$ for $f \in \mathcal{F}_b$, $t \in \mathcal{T}_f$ and $fr(f(t)) = fr(t)$;
- $f(t) \in \mathcal{T}_a$ for $f \in \Upsilon_a$, $t \in \mathcal{T}_a$ and $fr(f(t)) = fr(t)$;
- $\rho_x(t \mid \varphi(x, \bar{y})) \in \mathcal{T}_a$ for a location operator $\rho \in \Lambda$, a formula $\varphi(x, \bar{y})$ of the logic (see the definition below), x a variable in $\mathcal{X}_f$, $\bar{y}$ a tuple of arbitrary variables, and $t \in \mathcal{T}_a$.

In the last line we require $fr(t) \subseteq fr(\varphi(x, \bar{y})) = \{x_i \mid x_i = x \text{ or } x_i \text{ appears in } \bar{y}\}$ and $fr(\rho_x(t|\varphi(x, \bar{y}))) = fr(\varphi(x, \bar{y})) - \{x\}$.

We use the notion *ρ-term* for a term $\rho_x(t|\varphi(x, \bar{y}))$ and *pure term* for a term that does not contain ρ-terms, i.e. a term that does not contain any formulae. ρ-terms are built upon formulae; on the other hand they can also be used for constructing formulae.

The *formulae of the logic for non-deterministic ASMs* are those generated by the following grammar:

$$\begin{aligned}\varphi, \psi ::= {} & s = t \mid s_a = t_a \mid \neg\varphi \mid \varphi \wedge \psi \mid \forall x(\varphi) \mid \forall \mathtt{x}(\varphi) \mid \forall M(\varphi) \\ & \mid \forall X(\varphi) \mid \forall \mathcal{X}(\varphi) \mid \forall \ddot{X}(\varphi) \mid \forall \ddot{\mathcal{X}}(\varphi) \mid \forall F(\varphi) \mid \forall G(\varphi) \\ & \mid \mathrm{upd}(r, X) \mid \mathrm{upm}(r, \ddot{X}) \mid M(s, t_a) \mid X(f, t, t_0) \\ & \mid \mathcal{X}(f, t, t_0, s) \mid \ddot{X}(f, t, t_0, t_a) \mid \ddot{\mathcal{X}}(f, t, t_0, t_a, s) \\ & \mid F(f, t, t_0, t_a, t', t'_0, t'_a, s) \\ & \mid G(f, t, t_0, t_a, t', t'_0, t'_a, s_a) \mid [X]\varphi\end{aligned}$$

where s, t and t' denote terms in $\mathcal{T}_f$, s_a, t_a and t'_a denote terms in $\mathcal{T}_a$, $x \in \mathcal{X}_f$ and $\mathtt{x} \in \mathcal{X}_a$ denote first-order variables, M, X, $\mathcal{X}$, $\ddot{X}$, $\ddot{\mathcal{X}}$, F and G denote second-order variables, r is an ASM rule, f is a dynamic function symbol in $\Upsilon_f \cup \mathcal{F}_b$, and t_0 and t'_0 denote terms in $\mathcal{T}_f$ or $\mathcal{T}_a$ depending on whether f is in Υ_f or $\mathcal{F}_b$, respectively.

In the logic, disjunction $\vee$, implication $\rightarrow$, and existential quantification $\exists$ are defined as abbreviations in the usual way. $\forall M(\varphi)$, $\forall X(\varphi)$, $\forall \mathcal{X}(\varphi)$, $\forall \ddot{X}(\varphi)$, $\forall \ddot{\mathcal{X}}(\varphi)$, $\forall F(\varphi)$ and $\forall G(\varphi)$ are second-order formulae in which M, X, $\mathcal{X}$, $\ddot{X}$, $\ddot{\mathcal{X}}$, F and G range over finite relations.

When applying forall and parallel rules, updates yielded by parallel computations may be identical. Thus, we need the multiset semantics for describing a collection of possible identical updates. This leads to the inclusion of $\mathrm{upm}(r, \ddot{X})$ and $\ddot{X}(f, t, t_0, t_a)$ in the logic. $\mathrm{upd}(r, X)$ and $\mathrm{upm}(r, \ddot{X})$ respectively state that a finite update set represented by X and a finite update multiset represented by $\ddot{X}$ are generated by a rule r. $X(f, t, t_0)$ describes that an update (f, t, t_0) belongs to the update set represented by X, while $\ddot{X}(f, t, t_0, t_a)$ describes that an update (f, t, t_0) occurs at least once in the update multiset represented by $\ddot{X}$. If (f, t, t_0) occurs n-times in the update multiset represented by $\ddot{X}$, then there are n distinct $a_1, \ldots, a_n \in B_a$ such that $(f, t, t_0, a_i) \in \ddot{X}$ for every $1 \leq i \leq n$ and $(f, t, t_0, a_j) \notin \ddot{X}$ for every a_j other than $a_1, \ldots, a_n$. We use $[X]\varphi$ to express the evaluation of φ over a state after executing the update set represented by X on the current state. The second-order variables $\mathcal{X}$ and $\ddot{\mathcal{X}}$ are used to keep track of the parallel branches that produce the update sets and multisets, respectively. Finally, we use M to denote binary second-order variables which are used to represent the finite multisets in the semantic interpretation of ρ-terms, and F and G to denote second-order variables which encode bijections between update multisets.

A formula of the logic is *pure* if it does *not* contain any ρ-term and is generated by the following restricted grammar:

$$\varphi, \psi ::= s = t \mid s_a = t_a \mid \neg\varphi \mid \varphi \wedge \psi \mid \forall x(\varphi) \mid \forall \mathtt{x}(\varphi)$$

As defined before the formulae occurring in conditional, forall and choice rules are pure formulae of this logic. A formula or a term is *static*, if it does not contain any dynamic function symbol. In [10] a proof system for this logic

was developed, for which soundness and completeness with respect to Henkin semantics was proven.

4.2 Reasoning About Reflection

The logic for non-deterministic ASMs sketeched above does not yet handle reflection, which concerns rules r in the logic. In the logic above the main rule r is given as part of the specification and treated as extra-logical constant, while in a reflective ASM the main rule is the value in a location *self*. Consequently, we have $val_S(self) = r_S$, i.e. the interpretation of the term *self* in a state S yields the rule that is to be applied in S. This has to be taken into account for formulae of the form $\mathrm{upd}(r, X)$ and $\mathrm{upm}(r, \ddot{X})$. For a single machine step this change is rather irrelevant, as in a reflective ASM the main rule does not change within a single step. Thus, we have to take multiple steps into account. For these introduce two additional predicates r-upd and r-upm with the following informal meaning:

- $\text{r-upd}(n, X)$ means that n steps of the reflective ASM yield the update set X, where in each step the actual value of *self* is used.
- $\text{r-upm}(n, X)$ means that n steps of the reflective ASM yield the update multiset X.

In the light of the axioms definition of $\mathrm{upd}(r, X)$ and $\mathrm{upm}(r, \ddot{X})$ for sequence rules we can inductively define axioms for r-upd and r-upm. Clearly, we have $\text{r-upd}(1, X) \leftrightarrow \mathrm{upd}(self, X)$. Analogously, define $\text{r-upm}(1, X) \leftrightarrow \mathrm{upm}(self, \ddot{X})$. Then we further define

$$\begin{aligned}\text{r-upd}(n+1, X) \leftrightarrow\ & \big(\text{r-upd}(1, X) \wedge \neg\text{conUSet}(X)\big) \vee \\ & \big(\exists Y_1 Y_2(\text{r-upd}(1, Y_1) \wedge \text{conUSet}(Y_1) \wedge [Y_1]\text{r-upd}(n, Y_2) \wedge \\ & \bigwedge_{f \in \mathcal{F}_{dyn}} \forall xy(X(f, x, y) \leftrightarrow ((Y_1(f, x, y) \wedge \forall z(\neg Y_2(f, x, z))) \vee Y_2(f, x, y))))\big)\end{aligned}$$

as well as

$$\begin{aligned}\mathrm{upm}(n+1, \ddot{X}) \leftrightarrow\ & \Big(\text{r-upm}(1, \ddot{X}) \wedge \\ & \forall X\Big(\bigwedge_{f \in \mathcal{F}_{dyn}} \forall x_1 x_2(X(f, x_1, x_2) \leftrightarrow \exists \mathtt{x}_3(\ddot{X}(f, x_1, x_2, \mathtt{x}_3))) \wedge \neg\text{conUSet}(X)\Big)\Big) \vee \\ & \Big(\exists \ddot{Y}_1 \ddot{Y}_2\Big(\text{r-upm}(1, \ddot{Y}_1) \wedge \forall Y_1\Big(\bigwedge_{f \in \mathcal{F}_{dyn}} \forall x_1 x_2(Y_1(f, x_1, x_2) \leftrightarrow \\ & \qquad \exists \mathtt{x}_3(\ddot{Y}_1(f, x_1, x_2, \mathtt{x}_3))) \wedge \text{conUSet}(Y_1) \wedge [Y_1]\text{r-upm}(n, \ddot{Y}_2)\Big) \wedge \\ & \bigwedge_{f \in \mathcal{F}_{dyn}} \forall x_1 x_2 \mathtt{x}_3\big(\ddot{X}(f, x_1, x_2, \mathtt{x}_3) \leftrightarrow (\ddot{Y}_2(f, x_1, x_2, \mathtt{x}_3) \vee \\ & \qquad (\ddot{Y}_1(f, x_1, x_2, \mathtt{x}_3) \wedge \forall y_2 \mathtt{y}_3(\neg \ddot{Y}_2(f, x_1, y_2, \mathtt{y}_3))))\big)\Big)\end{aligned}$$

4.3 Reasoning About Concurrent Reflective ASMs

Finally, in order to capture also concurrency we make a very simple, but also powerful observation that a concurrent ASM can always be mimicked by a non-deterministic ASM. For each agent a replace its rule r by

IF ctl = idle **THEN CHOOSE** r
OR local(r) $\parallel$ ctl := active **ENDIF**
IF ctl = active **THEN CHOOSE** skip
OR final(r) $\parallel$ ctl := idle **ENDIF**

In an initial state the "control-state" location ctl is set to idle. If this is the case the agent executes either immediately its rule or executes a local version of it, i.e. all updates will be written to a local copy. In the second case the control-state becomes active. If the control-state is active, the agent may either do nothing or finalise the execution by copying all updates to the shared locations and returning to an idle control state. In doing so, the multi-step logic sketched above for reflective, non-deterministic ASMs can be used to reason about concurrent, reflective ASMs. Details concerning this are subject to ongoing research.

5 Further Directions

In this paper we first presented a behavioural theory for distributed adaptive systems (DAS), which integrates the corresponding theories of (synchronous) parallel algorithms [8], reflective algorithms [9] and concurrent algorithms [4]. DAS are captured by concurrent, reflective ASMs (crASMs), which we presented in more detail. The theory lays the foundations for rigorous development using crASMs, but due to the independence of the axiomatisation of any particular language it is not restricted to the context of ASMs. The second part of the paper presented a logic for crASMs that is based on a one-step logic for non-deterministic ASMs [10,11]. Concurrency can be mimicked by non-determinism, and reflection can be added by considering a multi-step extension, in which the so-far extra-logical rules can be replaced by rule terms that are subject to interpretation. This logic enables to formally reason about DAS and to rigorously verify desirabe properties.

We envision that in general distributed adaptive systems it will be desirable to capture also non-determinism or preferably randomised behaviour. The theory requires further extensions in this direction, which are subject of our current research (see [32] for first steps in this direction). Furthermore, for rigorous development extensions to the refinement method for ASMs [5] will be necessary. These problems are addressed in ongoing research.

References

1. Blass, A., Gurevich, Y.: Abstract State Machines capture parallel algorithms. ACM Trans. Comput. Log. **4**(4), 578–651 (2003)
2. Blass, A., Gurevich, Y.: Abstract State Machines capture parallel algorithms: correction and extension. ACM Trans. Comput. Log. **9**(3), 19:1–19:32 (2008)
3. Blass, A., Gurevich, Y.: Background of computation. Bull. EATCS **92**, 82–114 (2007)
4. Börger, E., Schewe, K.D.: Concurrent Abstract State Machines. Acta Informatica **53**(5), 469–492 (2016)
5. Börger, E.: The ASM refinement method. Formal Asp. Comp. **15**(2–3), 237–257 (2003)
6. Börger, E., Schewe, K.D.: Communication in Abstract State Machines. J. Univ. Comp. Sci. **23**(2), 129–145 (2017)
7. Börger, E., Stärk, R.: Abstract State Machines. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-642-18216-7
8. Ferrarotti, F., Schewe, K.D., Tec, L., Wang, Q.: A new thesis concerning synchronised parallel computing - simplified parallel ASM thesis. Theor. Comput. Sci. **649**, 25–53 (2016)
9. Ferrarotti, F., Schewe, K.-D., Tec, L.: A behavioural theory for reflective sequential algorithms. In: Petrenko, A.K., Voronkov, A. (eds.) PSI 2017. LNCS, vol. 10742, pp. 117–131. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-74313-4_10
10. Ferrarotti, F., Schewe, K.D., Tec, L., Wang, Q.: A complete logic for database Abstract State Machines. Log. J. IGPL **25**(5), 700–740 (2017)
11. Ferrarotti, F., Schewe, K.D., Tec, L., Wang, Q.: A unifying logic for non-deterministic, parallel and concurrent Abstract State Machines. Ann. Math. Artif. Intell. (2018, to appear)
12. Gurevich, Y.: Evolving algebras 1993: Lipari guide. In: Specification and Validation Methods, pp. 9–36. Oxford University Press (1995)
13. Gurevich, Y.: Sequential Abstract State Machines capture sequential algorithms. ACM Trans. Comput. Log. **1**(1), 77–111 (2000)
14. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press, Cambridge (2000)
15. Hewitt, C.: What is computation? Actor model versus Turing's model. In: Zenil, H. (ed.) A Computable Universe: Understanding Computation and Exploring Nature as Computation. World Scientific Publishing, Singapore (2012)
16. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Upper Saddle River (1985)
17. Kröger, F., Merz, S.: Temporal Logic and State Systems. Texts in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68635-4
18. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. **28**(9), 690–691 (1979)
19. Lynch, N.: Distributed Algorithms. Morgan Kaufmann, Burlington (1996)
20. Mazurkiewicz, A.: Introduction to trace theory. In: Diekert, V., Rozenberg, G. (eds.) The Book of Traces, pp. 3–67. World Scientific, Singapore (1995)
21. Merz, S.: On the logic of TLA$^+$. Comput. Artif. Intell. **22**(3–4), 351–379 (2003)
22. Milner, R.: A Calculus of Communicating Systems. LNCS, vol. 92. Springer, Heidelberg (1980). https://doi.org/10.1007/3-540-10235-3
23. Milner, R.: Communicating and Mobile Systems – The Pi-Calculus. Cambridge University Press (1999). ISBN 978-0-521-65869-0

24. Peterson, J.L.: Petri Net Theory and the Modeling of Systems. Prentice-Hall, Upper Saddle River (1981)
25. Petri, C.A.: Kommunikation mit Automaten. Ph.D. thesis, Institut für Instrumentelle Mathematik der Universität Bonn (1962). schriften des IIM Nr. 2
26. Riccobene, E., Scandurra, P.: Towards ASM-based formal specification of self-adaptive systems. In: Ait Ameur, Y., Schewe, K.D. (eds.) ABZ 2014. LNCS, vol. 8477, pp. 204–209. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43652-3_17
27. Roscoe, A.: The Theory and Practice of Concurrency. Prentice-Hall, Upper Saddle River (1997)
28. Schellhorn, G., Tofan, B., Ernst, G., Pfähler, J., Reif, W.: RGITL: a temporal logic framework for compositional reasoning about interleaved programs. Ann. Math. Artif. Intell. **71**, 1–44 (2014)
29. Schewe, K.D., Wang, Q.: A customised ASM thesis for database transformations. Acta Cybern. **19**(4), 765–805 (2010)
30. Schewe, K.D., Wang, Q.: Partial updates in complex-value databases. In: Heimbürger, A., et al. (eds.) Information and Knowledge Bases XXII, Frontiers in Artificial Intelligence and Applications, vol. 225, pp. 37–56. IOS Press (2011)
31. Schewe, K.D.: Concurrent reflective Abstract State Machines. In: Jebelean, T., et al. (eds.) 19th Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2017). IEEE (2018, to appear)
32. Schewe, K.D., Ferrarotti, F., Tec, L., Wang, Q.: Towards a behavioural theory for random parallel computing. In: Beierle, C., Brewka, G., Thimm, M. (eds.) Computational Models of Rationality - Essays Dedicated to Gabriele Kern-Isberner on the Occasion of Her 60th Birthday, Tributes, vol. 29, pp. 365–373. College Publications (2016)
33. Schewe, K.D., Ferrarotti, F., Tec, L., Wang, Q., An, W.: Evolving concurrent systems: behavioural theory and logic. In: Proceedings of the Australasian Computer Science Week Multiconference, (ACSW 2017), pp. 77:1–77:10. ACM (2017)
34. Schewe, K.D., Wang, Q.: XML database transformations. J. UCS **16**(20), 3043–3072 (2010)
35. Schmalz, M.: Formalizing the Logic of Event-B. Ph.D. thesis, ETH Zürich (2012)
36. Seebach, H., Nafz, F., Steghöfer, J.P., Reif, W.: How to design and implement self-organising resource-flow systems. In: Müller-Schloer, C., Schmeck, H., Ungerer, T. (eds.) Organic Computing - A Paradigm Shift for Complex Systems. ASYS, vol. 1, pp. 145–161. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-0348-0130-0_9
37. Stärk, R.F., Nanchen, S.: A logic for Abstract State Machines. J. Univ. Comput. Sci. **7**(11), 980–1005 (2001)
38. Steghöfer, J.P.: Large-scale open self-organising systems: managing complexity with hierarchies, monitoring, adaptation, and principled design. Ph.D. thesis, University of Augsburg (2014)
39. Steghöfer, J.P., et al.: Combining PosoMAS method content with Scrum: agile software engineering for open self-organising systems. Scalable Comput.: Pract. Exp. **16**(4), 333–354 (2015)
40. The Petri nets bibliography, University of Hamburg. http://www.informatik.uni-hamburg.de/TGI/pnbib/index.html
41. Winskel, G., Nielsen, M.: Models for concurrency. In: Abramsky, S., Gabbay, D., Maibaum, T.S.E. (eds.) Handbook of Logic and the Foundations of Computer Science: Semantic Modelling, vol. 4, pp. 1–148. Oxford University Press, Oxford (1995)

On B and Event-B: Principles, Success and Challenges

Jean-Raymond Abrial(✉)

Marseille, France
jrabrial@neuf.fr

After more than 20 years since the publication of the book on B [1], and almost 10 years since the publication of the book on Event-B [2], the purpose of this short paper is to present some key points of these technologies. Note that this presentation might be incomplete as I am certainly not aware of many developments which have taken place around B and Event-B in recent years.

This paper is organised around four topics. Section 1 is devoted to the listing of the basic principles on which B and Event-B have been developed. Section 2 is devoted to differences and similarities between B and Event-B. Section 3 explains where B and Event-B are spread around the world. Finally, Sect. 4 contains various issues and challenges encountered by these technologies.

1 Basic Principles

In this section, I study the main principles forming the basis of B and Event-B, i.e. developing systems which are intended to be correct by construction, using classical logic and mathematical notations, and developing various tools. Note that many information about B can be found in this website [3], whereas many information about Event-B can be found in this website [4].

1.1 Being Correct by Construction

The main purpose of B and Event-B is to help engineers developing systems that will be *correct by construction*. It means that B and Event-B are not programming languages of any kind. They are modelling systems. In case of B, modelling and developing software systems and in case of Event-B, modelling and developing global complex systems, involving not only software, but also physical environments and even human users.

1.2 Using Refinement

In order to achieve gradually a correct by construction approach, it is fundamental to handle refinements. This means that a development is made of a series of steps starting at a very abstract level and aiming at a final concrete one. This is simply incorporating in B and Event-B a classical approach used in many other engineering disciplines.

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 31–35, 2018.
https://doi.org/10.1007/978-3-319-91271-4_3

1.3 Mathematical Notation

The correct by construction approach together with the usage of refinement imply that each constructing steps be guaranteed by some theorems to be successfully proved. The goal of such theorems is to ensure that each step is valid and does not depart from the previous one. For doing so, it is important that statements of these theorems be expressed using a very classical mathematical notation, i.e. that of predicate calculus and of typed set theory.

1.4 Tools for Developing Models

It was necessary to develop tools for analysing and checking models for B (Atelier B, developed and maintained by the software house Clearsy) and for Event-B (Rodin, developed and maintained by the software house Systerel). As a matter of fact, a pen and paper approach is not possible any more as systems are becoming very complex these days.

1.5 Tools for Generating Theorems

It is out of the question that human users of B and Event-B generate directly mathematical statements to be proved at each steps of a development. This is far too much error prone to leave this in the hands of human users. An important tool called the *Proof Obligation Generator* has thus been developed for that purpose for B (in Atelier B) and for Event-B (in Rodin). Such a tool has been strongly influenced by what had been developed before for VDM [5].

1.6 Tools for Proving

Once some mathematical statements have been generated by the Proof Obligation Generator, it is important to be able to prove them in a mechanical way. For doing this, some proving tools have been constructed for B (in Atelier B) and for Event-B (in Rodin). With such tools, both automatic and interactive proofs can be performed.

1.7 More Tools

Other tools were developed in Universities (Southampton, Duesseldorf, Turku) and Industries (Siemens Transport, Clearsy, Systerel). With such tools, it is possible to perform model checking, automatic refinement, model decomposition and structuring, data validation and various translations to classical programming languages, etc.

2 Comparing B and Event-B

The book on B [1] was first published in 1996, whereas that on Event-B [2]was published in 2010. Consequently, Event-B had been able to take advantage of issues encountered in the usage of B. Also note that Event-B has been strongly influenced by the development of *Action systems* [6]. Event-B contains some simplifications over B. This has allowed us to extend the usage of Event-B to the modelling of systems that are not restricted to software. In what follows, I explain differences and similarities between B and Event-B.

2.1 Differences

One of the main differences between B and Event-B concerns operations (in B) and events (in Event-B). Each operation in B is usually defined together with a *pre-condition* containing a predicate that must be true for the operation to be able to be *called*. On the other hand, each event in Event-B is usually defined together with a *guard* containing a predicate that must be true for the event to be able to *occur*.

This results in having both pre-conditions or guards being assumptions when doing a proof on an operation or on an event (e.g. invariant preservation proofs). So far thus, there are no differences between the two. However, both differs strongly when dealing with refinement: pre-conditions can be weakened only, whereas guards can be strengthened only. This possibility of guard strengthening is particularly important as it allows users to build models starting from very abstract cases down to more realistic ones.

In fact, proof obligations are far simpler for events than for operations. In the case of operations one has always two rules: one for pre-conditions and one for post-conditions. In the case of event, one has always a single rule.

Another important distinction between the two concerns parameters. In the case of an operation, such parameters cannot be refined, whereas event parameters can be freely refined (removed, added or modified).

Basic sets, constants and their properties are handled differently in B and Event-B. In B, basic sets, constants and their properties are defined in the *abstract machine* where operations are defined. In Event-B, sets, constants and their properties are defined in separate structures called *contexts*. This gives users more flexibilities in Event-B than in B.

Event-B does not contain any programming constructs such as conditionals, choices, sequencings or loops as B does. This greatly simplifies proof obligations generated for Event-B with comparison to those generated for B. This simplification is particularly important in the case of sequencing. In fact, all such constructs can be handled in Event-B by using events only. Of course, code generation is simpler in B because of the presence of such constructs. To do the same in Event-B one has to apply some specific rules.

2.2 Similarities

Both B and Event-B use the mathematical notation of predicate calculus and typed set theory. This means that proof obligations stated for both approaches can be handled by similar provers. In fact, the prover of Atelier B (called PP) is used successfully in the Rodin toolset. Other external provers (e.g. SMT provers) are used in both Atelier B and Rodin.

In both cases, there is a notion of *machine* acting as an encapsulation for operations (in B) or events (in Event-B). However there is an important distinction between the two. The list of input-output definitions of the operations of a B machine (its *signature*) is fixed once and for all, although it is not the case for the events of an Event-B machine. Notice that events have no output parameters. But there are more: such a list of events can be further extended with new events in a refinement. This very important feature has been borrowed from Action Systems [6]. It allows users to be very flexible in the gradual construction (with refinement) of an event system.

In fact such similarities allows one to use Event-B within the Atelier B tool which is used for B. Some proof obligations which are specific to Event-B have been developed for that purpose in B.

3 Spreading

3.1 Spreading in Industry

B is extensively developed in Industry by the software house Clearsy, claiming that more than 30% of its business is devoted to B. A very rich and well documented article [7] presents the development of B at Clearsy. The main activity is with train systems in many places around the world: North and South America, Europe, Asia. Some information about the industrial use of B can be found in the Clearsy web site [3].

3.2 Spreading in Academia

Event-B (more than B) is widely spread in Universities in Europe (France, United Kingdom, Finland, Germany, Spain), in America (Canada, Brazil, Columbia), in Asia (China, Japan), etc.

4 Challenges

One of the main challenge of these technologies is the poor spreading of B in Industry: as said in Sect. 3.1, B is essentially used in train systems. There are clearly other industries where such a formal modelling approach could be used successfully, e.g. energy, automotive, aeronautics, space, etc. However, people in charge there claim that it is not possible, essentially because it is too difficult to modify engineering approaches which have been established for many years. The paper cited above [7] makes a very fine analysis of these difficulties.

There has been no development where both Event-B and B are used one after the other. First Event-B for the system analysis, then B for the software development. I think it could be quite possible. However, most of the time, system engineers and software engineers have different cultures.

References

1. Abrial, J.-R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
2. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
3. Clearsy. www.clearsy.com/en
4. Southampton University. www.Event-B.org
5. Jones, C.B.: Systematic Software Development Using VDM, 2nd edn. Prentice Hall, Upper Saddle River (1986)
6. Back, R.-J., Kurki-Suonio, R.: Decentralization of process nets with centralized control. Distrib. Comput. **3**(2), 73–87 (1989)
7. Lecomte, T., Deharbe, D., Prun, E., Mottin, E.: Applying a formal method in industry: a 25-year trajectory. In: Cavalheiro, S., Fiadeiro, J. (eds.) SBMF 2017. LNCS, vol. 10623, pp. 70–87. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70848-5_6

Translation and Transformation

CASM-IR: Uniform ASM-Based Intermediate Representation for Model Specification, Execution, and Transformation

Philipp Paulweber(✉), Emmanuel Pescosta, and Uwe Zdun

Research Group Software Architecture, Faculty of Computer Science,
University of Vienna, Währingerstraße 29, 1090 Vienna, Austria
{philipp.paulweber,uwe.zdun}@univie.ac.at, epescosta@casm-lang.org

Abstract. The Abstract State Machine (ASM) theory is a well-known formal method, which can be used to specify arbitrary algorithms, applications or even whole systems. Over the past years, there have been many approaches to implement concrete ASM-based modeling and specification languages. All of those approaches define their type systems and operator semantics differently in their internal representation, which leads to undesired or unexpected behavior during the modeling, the execution, and code generation of such ASM specifications. In this paper, we present CASM-IR, an Intermediate Representation (IR), designed to aid ASM-based language engineering which is based on a well-formed ASM-based specification format. Moreover, CASM-IR is conceptualized from the ground up to ease the formalization of ASM-based analysis and transformation passes. The feasibility of CASM-IR solving the *uniform ASM representation problem* is depicted. Based on our CASM-IR implementation, we were able to integrate a front-end of our statically inferred Corinthian Abstract State Machine (CASM) modeling language.

Keywords: CASM · Type system · Instruction · Register machine
Abstract State Machine · Intermediate Representation
Modeling and specification language

1 Introduction

In 1995 the Abstract State Machine (ASM) theory has been described by Gurevich [1] as a formal method based on transition rules, states and algebraic functions. ASMs are used to describe formally the evolving of function states in a step-by-step manner. This also explains why ASM theory was formerly called *Evolving Algebra* [2]. Based on the ASM programming language model from Gurevich, several tools with Domain-Specific Languages (DSLs) were created to solve application-specific problems, which were summarized by Börger [3].

E. Pescosta—Member of CASM organization.

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 39–54, 2018.
https://doi.org/10.1007/978-3-319-91271-4_4

The diversity of ASM-based applications[1] is widespread, ranging from formal specification semantics of programming languages, such as those for Java by Stark et al. [4] or VHDL by Sasaki [5], compiler back-end verification by Lezuo [6], software run-time verification by Barnett and Schulte [7], software and hardware architecture modeling e.g. of Universal Plug and Play (UPnP) by Glässer and Veanes [8], or even Reduced Instruction Set Computing (RISC) designs by Huggins and Campenhout [9].

Despite this diversity in applications, over the past years, different ASM-based language dialect were created to cover single or multiple application specific problem domains. This might not be perceived as a problem, as many *language users* [10] like to choose among multiple language dialects. The problem however is that the *language engineers* [10] craft and design those languages according to the needs of the *language user* and bind their implementations to a specific execution environment technology, instead of generalizing the mathematical foundation of the ASM-based languages in an independent model representation. This, in turn, means that those languages are difficult to integrate with each other [11], cannot easily be based on a common execution environment technology, and establishing a common set of language tools is difficult.

Moreover, the binding to various execution environment technologies introduces undesired and unexpected behaviors, e.g. if the same algorithm so to say is specified with different ASM modeling languages and the model execution leads to different floating point values or depending on the Integer representation to different overflow states. To overcome this *uniform ASM representation problem* a clear, precise, and formal intermediate model has to be introduced, which has the ability to represent various ASM language constructs of different contexts.

The major advantages of such an approach are the generalization of ASM-related analyzes, optimization, and transformation capabilities – first envisaged by Lezuo et al. [12] – in one single uniform model. Furthermore, another benefit for existing ASM languages is to directly reuse the numeric as well as the – proposed by Lezuo [6] – symbolic execution of specified ASM models. A huge disadvantage in the perspective of a *language engineer* is to port existing ASM language implementations to such a uniform ASM model.

This paper focuses on the design, implementation, and integration of an ASM-based Intermediate Representation (IR) model named CASM Intermediate Representation (CASM-IR) to address the *uniform ASM representation problem*. The main contribution of this paper is the definition of a well-formed ASM-based IR model which is independent of language front-ends and provides a well-defined type system, operator and built-in semantics.

This work is organized as follows: In Sect. 2 we describe our research context and the motivation of this paper. In Sect. 3 we describe our CASM-IR model. Section 4 presents details about the current implementation and integration of the CASM-IR. Section 5 gives an overview of the related work regarding IR's of other ASM languages and tools. Finally, in Sect. 6 we conclude the paper and outline the future work.

[1] For ASM applications of various domains, see: http://web.eecs.umich.edu/gasm.

2 Motivation

The broader context of our research is the creation of a modern state-of-the-art ASM modeling language implementation named the CASM, as well as transformation and deployment of CASM specifications to executable artifacts[2]. The primary application context of this work is the specification of embedded systems in a formal way. However, in the context of CASM, we not merely focus on specific application contexts like embedded systems, but rather aim to describe and specify arbitrary software and/or hardware applications. This overall idea is not new, but our approach to achieve this goal of generic transformations is different from a language engineering perspective, because we set our ASM-based IR into the center of the front-end language development. Other ASM language approaches, which are described in Sect. 5, do not, because they implement a forward directed transformation from ASM to the desired target language like C or C++. The transformation of ASM source specifications to specific target languages is by no means trivial. It involves the mapping of a mathematical-based specification model to a real executable program, which for itself resides in a specific execution environment.

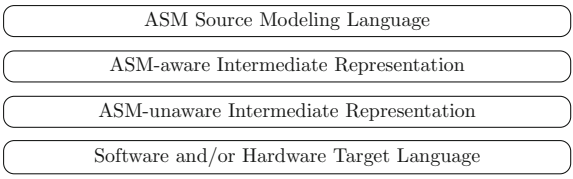


Fig. 1. CASM system abstraction layers

To overcome this complex transformation, we proposed and followed a model-based transformation approach in our earlier work [13], which defines four abstraction layers (illustrated in Fig. 1). At the top resides the *ASM Source Modeling Language* layer that includes besides the language grammar definition the lexer, parser, type inference, type checker, and Abstract Syntax Tree (AST) representation. A parsed input specification gets translated to the next layer, the *ASM-aware Intermediate Representation* layer. At this abstraction layer the CASM-IR, proposed in this paper, is defined. It allows us to analyze, transform and optimize the input specification for ASM related properties.

The CASM-IR gets further transformed in the next layer called *ASM-unaware Intermediate Representation*. At this abstraction layer the transformed specification has no longer any knowledge about the semantics or behavior of ASMs. Therefore it can be analyzed, transformed and optimized for traditional properties like execution speed or program size. In the final layer of Fig. 1, the *ASM-unaware Intermediate Representation* is mapped to various *Software*

[2] For CASM project website, see: http://casm-lang.org.

and/or Hardware Target Languages. Those CASM system abstraction layers describe a full transformation of an ASM specification to its desired target language. Due to the proposed layered structure, it is possible to only use a sub-set of the full functionality as well. For example, AST-based execution is used to recursively walk over the in-memory AST representation and interpret the input specification as it was parsed. In the context of AST-based execution, *language engineers* can (re)use the type system from the ASM-aware IR layer and can rely on its defined behavior and semantics. Therefore the CASM-IR can be used not only for transformation and code generation purposes, but also for AST-based interpreter applications. Furthermore, besides ASM-based languages, this proposed IR and its functionality could be used for other functional programming languages as well for their function definitions, type relations, numeric and symbolic computations. The approach to address the *uniform ASM representation problem* – introduced and described in Sect. 1 – with the CASM-IR raises several concerns regarding its existence and usefulness. First of all, the effort to investigate into such an IR design arises from the fact that accordingly to the state-of-the-art and to our knowledge no comparable IR for ASM languages with the focus on well-formed, reusable, retargetable, and optimizable ASM specifications exist yet. Second, as presented by Lezuo et al. [12], the optimization potential is huge of ASM languages regarding redundancy eliminations, but still not covered and addressed by any ASM language implementation in a unified manner.

3 CASM-IR

This section describes our ASM-based IR design that can be (re)used for designing and building ASM-based and other possible functional related specification languages. Before we go into the details of the model and the format of this IR, we first outline the composition of our CASM system [13]. Figure 2 depicts a more detailed overview of the sketched abstraction layers from Sect. 2 (see Fig. 1). A parsed ASM source – in our case the CASM language – gets translated to an AST representation and necessary type information gets inferred. In order to do so, the CASM-IR – depicted as Model (IR) – needs to provide type information for all possible operators and their type relations, which a language front-end can use, to implement a type inference pass. Furthermore, the CASM-IR model provides the ability to directly implement AST-based interpreter applications on top of it, because language front-ends can access the implemented run-time of the IR to evaluate expressions and terms.

If the execution shall be done directly using the IR model itself, a language front-end just has to perform a model-to-model transformation from its AST-based representation to an instance of this IR model. At this point the IR can optimize the specification for ASM-related properties fully decoupled from the original input specification in form of an AST representation. Some optimization properties were proposed by Lezuo et al. [12]. Furthermore, then the IR instance can be executed by the run-time implementation of the IR model.

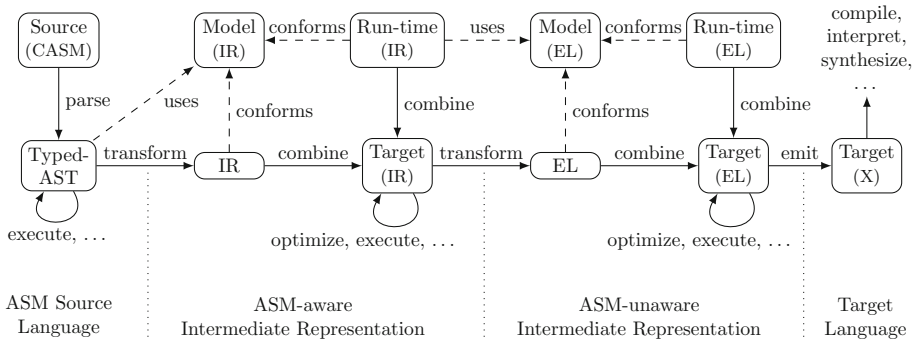


Fig. 2. CASM system design (high-level overview)

For further processing (code generation) of the specification to a specific programming target language, the IR instance can be transformed into an Emitting Language (EL) model, as proposed in our earlier work [13]. Details about the EL model are out of the scope of this paper.

3.1 Motivating Example

To better understand the solving of the research question regarding the *uniform ASM representation* problem that CASM-IR deals with, we describe a small ASM specification and point out the issues, which are addressed by the CASM-IR design and implementation. Listing 1.1 on Page 6 depicts a valid (high-level) CASM specification of a modeled *swap* algorithm[3]. It defines a rule `swap` (Line 6) and two nullary functions `x` (Line 3) and `y` (Line 4) of result type integer. The `init` (Line 1) defines a single execution agent with starting top-level rule `swap`. Rule `swap` defines a parallel block rule (Line 7-11) and three update rules. The first two update rules (Line 8-9) are producing updates to swap the function values from `x` and `y`. In the last update rule (Line 10), the ASM *program* function gets updated with an undefined value, which results into a termination of the specification, because the single execution agent top-level rule gets set to an undefined value and therefore the ASM execution concludes the model execution.

To get a feel for the expressed *swap* algorithm ASM specification in other ASM languages, we depict three further examples of the same algorithm modeled in *AsmL* [14] (Listing 1.2), *CoreASM* [15] (Listing 1.3), and *Asmeta* [16] (Listing 1.4). Even in this small specification, several behaviors and definitions are implicit and slightly different in the various ASM languages. E.g. the used function `program` (Listing 1.1 at Line 10) is not an explicitly defined function in this valid CASM specification, because this function definition is hidden from the *language user* and it gets implicitly defined, because it depends on an agent type domain. The type relation of this function would be a projection of the

[3] For CASM concrete syntax description, see: http://casm-lang.org/syntax.

```
1  CASM init swap
2
3  function x : -> Integer
4  function y : -> Integer
5
6  rule swap =
7  {
8    x := y
9    y := x
10   program( self ) := undef
11 }
```

Listing 1.1. Swap Example (*CASM)*

```
1  var x as Integer
2  var y as Integer
3
4  swap()
5    x := y
6    y := x
7
8  Main()
9      swap()
10   step
11     // terminates after this step
```

Listing 1.2. Swap Example (*AsmL)*

```
1  CoreASM swap
2  use StandardPlugins
3  init swap
4
5  function x : -> Integer
6  function y : -> Integer
7
8  rule swap =
9    par
10     x := y
11     y := x
12     program( self ) := undef
13   endpar
```

Listing 1.3. Swap Example (*CoreASM)*

```
1  asm swap
2  import ../STDL/StandardLibrary
3
4  signature:
5    dynamic controlled x : Integer
6    dynamic controlled y : Integer
7
8  definitions:
9    main rule swap =
10     par
11       x := y
12       y := x
13     endpar
```

Listing 1.4. Swap Example (*Asmeta)*

current agent type domain to a stored top-level rule, which is similar in the *CoreASM* specification (Listing 1.3 at Line 12). Furthermore, the initialization of this `program` function to the rule `swap` is implicit as well. In CASM and *CoreASM* this is achieved by setting the underlying agent through the `init` definition (Listing 1.1 at Line 1, and Listing 1.3 at Line 3). Similar behavior can be achieved in *Asmeta* by setting a certain rule to a `main` rule (Listing 1.4 at Line 9) or in *AsmL* which forces the uses to define a `Main()` rule (Listing 1.2 at Line 8) which controls the computation directly. Moreover, it can be observed that the *swap* examples of CASM and *CoreASM* explicitly define the termination of the specification whereas the *swap* examples written in *AsmL* and *Asmeta* do not.

In order to implement e.g. an AST-based interpreter to execute this specifications a *language engineer* would have to implement a run-time kernel, which handles those implicit defined behaviors. Furthermore, if we think about optimizing such specifications, implicitly defined behaviors cannot be optimized and addressed by transformation passes in a generic way.

Generally speaking we have discovered two implicit behaviors – *initialization of functions* and *agent life cycle handling*. Latter is very important if we consider synchronous and asynchronous multi-agent ASM specifications. To express ASM specifications in a well-formed IR we present in the following sub-sections the definition of our CASM-IR model and its textual representation.

3.2 Types, Constants, and Functions

Due to the fact that every ASM-based language will eventually be executed by a real machine a term, expression or even a value will have a concrete type. Even Gurevich [2] suggested his ASM language definition lacks explicit typing, and it would be more practical to introduce such. Therefore, in the center of our CASM-IR model stands the type system with all of its possible type domains, which we will call from now on just types[4]. An overview is depicted in Fig. 3. We can observe that the type system defines very basic (*Primitive*) types like *Boolean* or *Integer* up to very abstract ones (*Template*) like *List* or *File*.

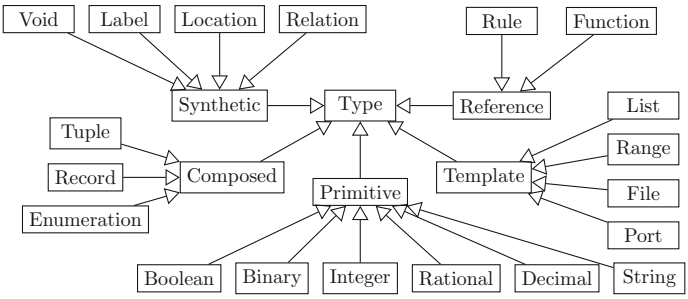


Fig. 3. CASM-IR type system (inheritance tree)

Notable to mention here in contrast to other ASM languages is that CASM-IR always tries to be as close as possible to the mathematical foundation of a type. This means e.g. the *Integer* representation is represented as an arbitrary precise Integer with range $]-\infty, \infty[$. There is even the possibility – similar to the Ada programming language – to restrict the type to a certain sub-range. Furthermore, CASM-IR introduces a *Binary* type which can be used to represent any binary bit-precise value with defined bit-size. Along with this type CASM-IR features a set of *Binary* built-in[5] arithmetic operations. In the implementation of Lezuo et al. [12] this type was indirectly specified with Integer types by limiting built-in operations over a predefined bit-size values and the language itself was not aware of these operations. Another novel type in CASM-IR compared to other languages is that it features a *Reference* type. All references to rules, functions, and derived functions have to be typed to ensure type safety for indirect calls. Due to the mathematical foundation of ASMs, all typed CASM-IR constants[6] can have besides the type-specified (domain) content, an undefined value. Furthermore, we directly include in CASM-IR the notion of symbolic values that enable a clear definition of numeric as well as symbolic execution, whereas the symbolic values are its own domain value as suggested by Lezuo [6].

[4] For CASM-IR type specification, see: http://casm-lang.org/ir/types.
[5] For CASM-IR built-in specification, see: http://casm-lang.org/ir/builtins.
[6] For CASM-IR constant specification, see: http://casm-lang.org/ir/constants.

```
1 ;; integer constant '123'
2 @c0 = i 123
3 ;; 'undefined' rule reference
4 ;;  constant of relation : -> Void
5 @c1 = r< -> v > undef
6 ;; function definition 'foo'
7 ;; with relation:
8 ;; Boolean * Rational -> Integer
9 @foo = < b * q -> i >
```

Listing 1.5. Constants and Functions

```
1 ;; enumeration type definition
2 bar = { A, B, C }
3 ;; setting agent type domain
4 ;; to enumeration type 'bar'
5 .agent = bar
6 ;; function definition 'program'
7 ;; with relation:
8 ;; bar -> RuleRef< -> Void >
9 @program = < bar -> r< -> v > >
```

Listing 1.6. Enum. and Agents

States are modeled through the function definitions[7]. As defined in [17] every ASM function has a name and an arbitrary type relation. By default every function – accordingly to the ASM definition – is undefined over its type relation domain and needs to be explicitly initialized in CASM-IR. Listing 1.5 on Page 8 depicts a constant `@c0` of type *Integer* and value 123, a constant `@c1` of type *Rule Reference* with relation $:\rightarrow Void$ and an *undefined* value, and a function `foo` with relation $: Boolean * Rational \rightarrow Integer$.

3.3 Agents, Rules, and Deriveds

ASM specifications can either be single or multi execution agent-based systems [1]. Therefore we provide a model instance to declare only the agent type domain that directly results in the desired behavior. For instance, if we would define the agent type domain to a *Boolean* type, we would define two operational agents. The agent type domain has an important role in the execution of all ASM specifications because starting from a defined agent rule the nested rules get called and so on. Furthermore, the defined agent domain is also used in a special internal *function* named `program` to store the current agent top-level rule as a rule reference. Listing 1.6 depicts how to set the model instance of the current agent type domain. In Line 2 an enumeration type `bar` gets defined with enumerators `A`, `B`, and `C`, and in Line 5 the agent type domain gets set to the type `bar`. Therefore we have specified in this example a multi-agent ASM with three agents. As already mentioned in Sect. 3.1 there is a special function named `program` that controls the execution of the agents in its kernel of ASM specifications which heavily depends on the agent type domain. Line 9 shows the corresponding `program` function definition with the agent type domain `bar`. The actual computation in ASMs is specified through transition rules. CASM-IR also has the notion of rules, but only for the *named rule* definitions. Other ASM rules like *Update*, *Conditional*, *Forall*, *Choose*, etc. are represented in CASM-IR through nested blocks and instructions (see Sect. 3.4).

Another important specification component in CASM-IR are *derived* functions or *deriveds* for short. It can be seen as a kind of typed macro to reuse *state-less* or *side-effect free* calculations. This means, that in *deriveds*, no state

[7] For CASM-IR function specification, see: http://casm-lang.org/ir/functions.

```
1 %r0 = ;; ... calculation which yields result of type 'i'
2 %r1 = location < i -> i> @foo, i %r0 ;; yields type 'loc'
3 %r2 = lookup loc %r1 ;; yields type 'i'
4 %r3 = ;; ... calculation which yields result of type 'i' and uses '%r2'
5 update loc %r1, i %r3 ;; produces an update to function 'foo'
```

Listing 1.7. Location-, Lookup-, and Update-Instruction

changes are allowed to be performed; ergo, no *Update* rules are allowed in derived function definitions.

3.4 Blocks, Instructions, and Registers

All basic expressions and state-modifying rules are represented in CASM-IR as *Instructions* in a Single Static Assignment (SSA) form. So produced results of instructions are stored in registers and the type is directly yielded from the specified instruction. This conceptual idea is borrowed from the Low Level Virtual Machine (LLVM) compiler IR design by Lattner and Adve [18]. So any instruction call can be specified by a resulting unique register name, an instruction name and possible instruction operands with explicit types. This also indicates that the CASM-IR follows a register machine design and implementation approach.

Basic ASM rules like *skip*, *choose*, or the definition of execution semantics (*fork* and *merge*) are represented as single instructions. Novel in CASM-IR is that it explicitly models the reading (*lookup*) and writing (*update*) of ASM function states by dedicated instructions. This allows to analyze and optimize CASM-IR specifications as suggested by Lezuo et al. [12]. A *location* instruction performs the function location calculation. How the location is calculated is not fixed and has to be decided in the run-time implementation. E.g. a common technique would be the calculation of a function location by a certain hashing algorithm. The *lookup* instruction determines at a certain point in the specification, which state value is assigned to a certain function depending on the nested parallel and sequential execution semantics. The argument needed to perform a lookup is a location constant. An *update* instruction produces a new *location* and *value* pair, which gets applied to the surrounding (local) function state also known as *pseudo state* [12]. Therefore, an update instruction needs, besides the exact calculated function location, a value operand.

Listing 1.7 depicts an example usage of the *location*, *lookup*, and *update* instruction. In Line 2 a *location* calculation is performed for the function `foo` which has accordingly to the type one Integer argument. At Line 3 the actual *lookup* of the function value is performed. And in Line 5, a new *update* is performed to the same location were the lookup was performed. Similar to traditional assembler languages, the CASM-IR includes a *call instruction* as well, but this call instruction is used for multiple invocation types. It is used to call specified rules, derived functions, and pre-defined built-ins either directly by its name or indirectly through a register value of a reference type. Besides the generic *call*

instruction there exist several instructions to perform intermediate calculations of arithmetic, logical, and comparison operations[8].

Multiple instructions are compound to a Statement Block (SB) whereas the execution semantics of the instructions is always sequential. Several blocks are grouped together and form an Execution Semantics Block (ESB) which can either have a *parallel* or *sequential* execution semantics. Additionally, every ESB contains, besides the sub-blocks, an *entry* and an *exit* SB, in which the actual execution semantics is specified by appropriate *fork* and *merge* instructions. Figure 4 on Page 10 depicts the composition of rules, the ESB and SB blocks as well as instructions.

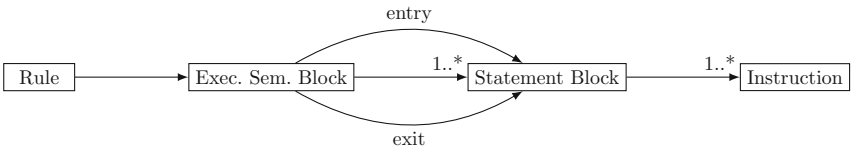


Fig. 4. CASM-IR rules, blocks, and instructions

3.5 Motivating Swap Example in CASM-IR

In this section we present an example output of the transformed motivating example `swap` CASM specification from Listing 1.1 to our CASM-IR. The performed model-to-model transformation is implemented in the CASM front-end (see Sect. 4). It shall summarize several of the presented concepts and sketch some optimization possibilities, which can be obtained through the representation of ASM specifications in the CASM-IR. Note that this presented transformed motivated example is valid for the other presented ASM `swap` specifications as well (Listing 1.2, Listing 1.3, and Listing 1.4).

Listing 1.8 on Page 10 visualizes a CASM-IR instance, where the missing definitions and implicit behaviors from Listing 1.1 are explicitly specified. In the transformed specification we can observe that first of all the agent type domain gets set to a enumeration type named `a` (Line 3) with the name `$` (Line 2). This means that the agent type domain consists of only one concrete value and hence we have a single execution agent ASM specification. Thereafter, a forward declaration of the rule `swap` is specified (Line 4) because the next listed constants (Line 5-8) contain the symbol of the `swap` rule to define a rule reference constant. Next, three functions are defined. The `program` function (Line 9) with the previous defined agent type domain that stores the ASM agent top-level rule reference. After that the functions `x` (Line 10) and `y` (Line 11) are defined accordingly to the originally input specification. Before the definition of the `swap` rule gets defined, the initialization of the ASM state has to be specified, which at least has to set the correct starting rule of the agents. Note that all

[8] For CASM-IR instr. specification, see: http://casm-lang.org/ir/instructions.

```
1  CASM-IR                                ;; CASM-IR specification header
2  a = { $ }                              ;; definition of enum. type 'a'
3  .agent = a                             ;; set agent type domain to type 'a'
4  @swap < -> v>                          ;; declaration of rule 'swap'
5  @c0 = r< -> v> @swap                   ;; 'swap' rule reference
6  @c1 = a $                              ;; agent constant of single agent
7  @c2 = r< -> v> undef                   ;; undefined rule reference
8  @c3 = a $                              ;; agent constant of single agent
9  @program = <a -> r< -> v>>             ;; 'program' function definition
10 @x = < -> i>                           ;; definition of function 'x'
11 @y = < -> i>                           ;; definition of function 'y'
12 @init -> v = {                         ;; definition of 'init' rule
13   lbl0: entry                          ;; ESB entry block of lbl0
14     fork par                           ;; fork instruction parallel
15
16   lbl1: %lbl0                          ;; SB lbl1 in ESB lbl0
17     %r0 = location <a -> r< -> v>> @program, a @c1
18     update loc %r0, r< -> v> @c0
19
20   exit: %lbl0                          ;; ESB exit block of lbl0
21     merge par                          ;; merge instruction parallel
22 }
23 @swap -> v = {                         ;; definition of 'swap' rule
24   lbl2: entry                          ;; ESB entry block of lbl2
25     fork par                           ;; fork instruction parallel
26
27   lbl3: %lbl2                          ;; SB lbl3 in ESB lbl2
28     %r1 = location < -> i> @y
29     %r2 = lookup loc %r1               ;; lookup of function 'y'
30     %r3 = location < -> i> @x
31     update loc %r3, i %r2              ;; update of function 'x'
32
33   lbl4: %lbl2                          ;; SB lbl4 in ESB lbl2
34     %r4 = location < -> i> @x
35     %r5 = lookup loc %r4               ;; lookup of function 'x'
36     %r6 = location < -> i> @y
37     update loc %r6, i %r5              ;; update of function 'y'
38
39   lbl5: %lbl2
40     %r7 = location <a -> r< -> v>> @program, a @c3
41     update loc %r7, r< -> v> @c2
42
43   exit: %lbl2                          ;; ESB exit block of lbl2
44     merge par                          ;; merge instruction parallel
45 }
```

Listing 1.8. Swap Example (*CASM-IR)*

function states in ASMs are by default undefined. Last but not least the rule `swap` gets defined. It contains a parallel execution semantics block with three trivial statements and several location, lookup and update instructions.

Regarding the optimization potential in this revised example we can detect several possible ASM-related optimizations. The most obvious one would be a hoisting optimization of redundant location calculations, because the location of nullary functions will always be the same. The calculation e.g. of the location of function `y` at register `%r1` (Line 28) could be moved up before the fork instruction of the entry section at `lbl2` (Line 24). And the location calculation of function `y` at the register `%r6` (Line 36) can be removed and all its uses can be replaced

by `%r1`. The same applies for the location of function `x` and register `%r3` and `%r4` (Line 30, Line 34).

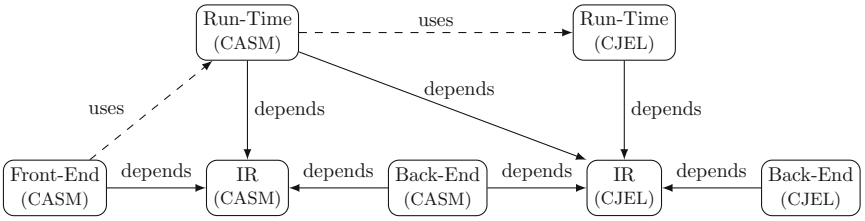


Fig. 5. CASM system implementation (library dependency graph)

4 Implementation and Integration

Figure 5 depicts the CASM system implementation libraries visualized as a library dependency graph. The CASM run-time and back-end libraries are based on corresponding CASM unaware Just-in-time Emitting Language (CJEL) libraries (situated one layer below the CASM libraries). The CJEL layer is not described in this paper. All libraries are implemented in C++11/14 standard.

The implementation of the CASM-IR model consists of two major base classes - *Type* and *Value*. The type system and type hierarchy is implemented according to the definition presented in Sect. 3.2. All other model instances are sub-classes of the *Value* class. This design approach was borrowed again from the LLVM compiler project where *everything is a value* [18]. Furthermore, every value has a type. The CASM-IR implementation provides a rich Application Programming Interface (API) to provide certain information to front-end implementations. To be more precise here, for every instruction and built-in, it is possible to fetch all defined type relations through an internal type map structure. This enables a clean separation between a front-end language definition and the IR internals.

Based on the CASM-IR, we have designed our CASM language front-end. Compared to the CASM language implementations from Lezuo et al. [12] the AST has resulted in a much simpler and clearer design then before, because all type, operator, and built-in design decisions were already made in the CASM-IR implementation. Therefore the AST only focuses on the input language itself. CASM is a statically strong inferred typed language. Hence, the difference between the front-end CASM input specification language and the CASM-IR model is that the front-end language requires a symbol resolver, type checker and type inference pass to fully type the parsed input specification AST representation. In the analyzer passes we use the provided API of the CASM-IR to query and check if certain types, built-ins, and operators exist. Furthermore, during type inference, the front-end can infer the correct type through the predefined type relations of the specified CASM-IR operators. E.g. if a type is not

possible to be inferred in the front-end, the possible types can be retrieved from the CASM-IR and used as helpful debugging information for language users.

Besides type inference and other analyzes done by the front-end implementation, the most important benefit of targeting the CASM-IR is that a language front-end engineer can directly call evaluation instrumentation functions of the CASM-IR to perform calculations of operator instructions and built-ins.

5 Related Work

One of the best-known ASM implementations is the *Asmeta*[9] tool-set with the *AsmetaL* language [16]. The core of *Asmeta* is designed and implemented using the Eclipse Modeling Framework (EMF) *Ecore* meta-model[10]. Based on the *Ecore* meta-model, the ASM language model of *Asmeta* is directly described as an instance (model). Therefore, the execution and precise calculation of the implemented ASM simulator is bound to the run-time implementation of the *Ecore* meta-model and its EMFs Java interface realizations.

Another notable ASM design and implementation is *CoreASM*[11] originally developed by Farahbod et al. [15]. The focus of CoreASM is to provide a flexible and extensible ASM implementation and to be as near as possible to the described ASM method by Börger [17]. CoreASM is implemented in Java and its IR and run-time is directly bound to the Java Virtual Machine (JVM). Microsoft research designed and implemented an ASM language named *AsmL*[12] [14]. AsmL is implemented and based to the `.NET` framework.

Besides CASM-IR, which solves a uniform ASM intermediate representation to be language front-end independent, Arcaini et al. [19] proposed a Unified Abstract State Machines (UASM) language syntax. Their approach is to unify the front-end ASM syntax representation and this is in the perspective of CASM-IR yet another ASM front-end input specification. Similar to the ASM language proposed by Anlauff [20], the eXtensible ASM (XASM) language[13], which compiles XASM specifications to C.

Lezuo and Krall [21] introduced in 2013 the CASM language. The origin of this language was that all the (publicly available) existing ASM tools were impracticable for industrial sized applications [22]. The tool-chain presented by Lezuo et al. [12] focuses like the other ASM designs only on the input specification itself, thus those research results were not directly usable by other ASM-based language frameworks. The latter motivated, as already stated in Sect. 2, to rethink the proposed ASM language engineering designs and implementations, leading to our model-based transformation approach [13] for the CASM language[14].

[9] For Asmeta project, see: http://asmeta.sourceforge.net.
[10] For EMF project, see: http://eclipse.org/modeling/emf.
[11] For CoreASM open-source project, see: http://github.com/coreasm.
[12] For AsmL documentation and project, see: http://asml.codeplex.com.
[13] For XASM documentation, see http://sourceforge.net/projects/xasm.
[14] For CASM open-source project, see: http://github.com/casm-lang.

Different representation and transformation approaches have been investigated in the *AsmGofer* language by Schmid [23], which is based on the programming language Gofer (similar to Haskell), and the *ASM Workbench* with the *ASM-SL* language introduced by Del Castillo [24], which is implemented in Standard ML. The *ASM-SL* has been explored further by Schmid [25] to represent and encode specifications in C++. The translation (compilation) scheme was limited to a *double buffering* concept and therefore unable to encode mixing sequential and parallel rules. CASM-IR solves this by using block-level nested *fork* and *merge* instructions to control the update-set behavior.

Another transformation scheme for ASMs was presented by Bonfanti et al. [26] to represent and encode *AsmetaL* specifications in C++ code targeting Arduino platforms. Their code generator directly converts the ASM specification to the desired target language and run-time environment. By targeting a different target run-time environment, platform, or architecture the encoded and implement ASM behavior would have to be re-implemented in every code generator.

Important to point out is that CASM-IR tries to establish a mid-end IR for ASM-based languages similar to the approach for classical programming language IR models such as GCCs *GENERIC* and *GIMPLE* by Merrill [27] or the LLVM IR by Lattner and Adve [18].

6 Conclusion

We have presented in this paper CASM-IR, a statically and strongly typed, well-formed ASM-based IR, to provide the ability for ASM-based language engineers to specify the internals of their ASM language in a well-defined representation model. Besides the type system, agent, functions, deriveds, rules, blocks, and instruction semantics, we discussed ASM properties, which are indirectly represented in ASM source languages and made explicitly and typed in the CASM-IR. There are several other issues regarding implicit behavior in ASM-based high-level languages we could point out, but it would go beyond of the scope of this paper. We have given a short overview of our implementation, corresponding libraries, and discussed the usefulness of our approach.

Regarding the CASM-IR itself, there is a lot of future work in the direction of the type system. The providing of types like *trees*, *sets*, *bags*, and so on, is still an open topic. We are already working on the implementation, formal definition and verification of ASM-related optimization transformations based on the gained knowledge from Lezuo et al. [12] for our CASM-IR. Another research direction we are working on is the byte-code representation of the CASM-IR. This would allow the implementation of very compact virtual machines for ASM-based specifications.

References

1. Gurevich, Y.: Evolving Algebras 1993: Lipari Guide - Specification and Validation Methods, pp. 9–36. Oxford University Press Inc., New York (1995)
2. Gurevich, Y.: Sequential abstract-state machines capture sequential algorithms. ACM Trans. Comput. Logic (TOCL) **1**(1), 77–111 (2000)
3. Börger, E.: The origins and the development of the ASM method for high level system design and analysis. J. Univ. Comput. Sci. **8**(1), 2–74 (2002)
4. Stärk, R.F., Schmid, J., Börger, E.: Java and the Java Virtual Machine: Definition, Verification, Validation. Springer, Heidelberg (2001). https://doi.org/10.1007/978-3-642-59495-3
5. Sasaki, H.: A formal semantics for Verilog-VHDL simulation interoperability by abstract state machine. In: Proceedings of the Conference on Design, Automation and Test in Europe, DATE 1999. ACM, New York (1999)
6. Lezuo, R.: Scalable translation validation; tools, techniques and framework. Ph.D. thesis, Wien Technische Universität Dissertation (2014)
7. Barnett, M., Schulte, W.: Spying on components: a runtime verification technique. In: Proceedings of the Workshop on Specification and Verification of Component-Based Systems, SAVCBS 2001, pp. 7–13 (2001)
8. Glässer, U., Veanes, M.: Universal plug and play machine models. In: Kleinjohann, B., Kim, K.H., Kleinjohann, L., Rettberg, A. (eds.) Design and Analysis of Distributed Embedded Systems. ITIFIP, vol. 91, pp. 21–30. Springer, Boston, MA (2002). https://doi.org/10.1007/978-0-387-35599-3_3
9. Huggins, J.K., Campenhout, D.V.: Specification and verification of pipelining in the ARM2 RISC microprocessor. ACM Trans. Des. Autom. Electron. Syst. (TODAES) **3**(4), 563–580 (1998)
10. Kleppe, A.G.: Software Language Engineering: Creating Domain-Specific Languages using Metamodels. Addisson-Wesley, Boston (2009)
11. Völter, M.: Generic tools, specific languages. Ph.D. thesis, Delft University of Technology, June 2014
12. Lezuo, R., Paulweber, P., Krall, A.: CASM - optimized compilation of abstract state machines. In: SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES), pp. 13–22. ACM (2014)
13. Paulweber, P., Zdun, U.: A model-based transformation approach to reuse and retarget CASM specifications. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) ABZ 2016. LNCS, vol. 9675, pp. 250–255. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33600-8_17
14. Gurevich, Y., Rossman, B., Schulte, W.: Semantic essence of AsmL: extended abstract. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2003. LNCS, vol. 3188, pp. 240–259. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30101-1_11
15. Farahbod, R., Gervasi, V., Glässer, U.: CoreASM: an extensible ASM execution engine. Fundam. Inf. **77**(1–2), 71–104 (2007)
16. Gargantini, A., Riccobene, E., Scandurra, P.: A metamodel-based language and a simulation engine for abstract state machines. J. Univ. Comput. Sci. **14**(12), 1949–1983 (2008)
17. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-642-18216-7

18. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: Code Generation and Optimization, pp. 75–86. IEEE (2004)
19. Arcaini, P., Bonfanti, S., Dausend, M., Gargantini, A., Mashkoor, A., Raschke, A., Riccobene, E., Scandurra, P., Stegmaier, M.: Unified Syntax for abstract state machines. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) ABZ 2016. LNCS, vol. 9675, pp. 231–236. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33600-8_14
20. Anlauff, M.: XASM- an extensible, component-based abstract state machines language. In: Gurevich, Y., Kutter, P.W., Odersky, M., Thiele, L. (eds.) ASM 2000. LNCS, vol. 1912, pp. 69–90. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44518-8_6
21. Lezuo, R., Krall, A.: Using the CASM language for simulator synthesis and model verification. In: Proceedings of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, p. 6. ACM (2013)
22. Lezuo, R., Barany, G., Krall, A.: CASM: implementing an abstract state machine based programming language. In: Software Engineering (Workshops), pp. 75–90 (2013)
23. Schmid, J.: Introduction to AsmGofer. http://www.tydo.de/AsmGofer (2001)
24. Del Castillo, G.: The ASM workbench: a tool environment for computer-aided analysis and validation of abstract state machine models. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 578–581. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45319-9_40
25. Schmid, J.: Compiling abstract state machines to C++. J. Univ. Comput. Sci. **7**(11), 1068–1087 (2001)
26. Bonfanti, S., Carissoni, M., Gargantini, A., Mashkoor, A.: `Asm2C++`: a tool for code generation from abstract state machines to arduino. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NFM 2017. LNCS, vol. 10227, pp. 295–301. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57288-8_21
27. Merrill, J.: GENERIC and GIMPLE: a new tree representation for entire functions. In: Proceedings of the 2003 GCC Developers' Summit, pp. 171–179 (2003)

Event-B Expression and Verification of Translation Rules Between SysML/KAOS Domain Models and B System Specifications

Steve Jeffrey Tueno Fotso[1,3(✉)], Amel Mammar[2], Régine Laleau[1], and Marc Frappier[3]

[1] Université Paris-Est Créteil, LACL, Créteil, France
steve.tuenofotso@univ-paris-est.fr, laleau@u-pec.fr
[2] Télécom SudParis, SAMOVAR-CNRS, Evry, France
amel.mammar@telecom-sudparis.eu
[3] Université de Sherbrooke, GRIL, Québec, Canada
Marc.Frappier@usherbrooke.ca

Abstract. This paper is about the extension of the *SysML/KAOS* requirements engineering method with domain models expressed as ontologies. More precisely, it concerns the translation of these ontologies into *B System* for system construction. The contributions of this paper are twofold. The first one is a formal semantics for the ontology modeling language. The second one is the formal definition of translation rules between ontologies and *B system* specifications in order to provide the structural part of the formal specification. These translation rules are modeled in *Event-B*. Their consistency and completeness are proved using *Rodin*. We show that they are structure preserving (two related elements within the source model remain related within the target model), by proving various isomorphisms between the ontology and the *B System* specification.

Keywords: *Event-B* · *B system* · Domain modeling · Ontologies *SysML/KAOS*

1 Introduction

Our study, part of the *FORMOSE* project [5], focuses on an approach for designing systems in critical areas such as railway or aeronautics. The development of such systems, in view of their complexity, requires several verifications and validation steps, more or less formal, with regard to the current regulations. In [17], rules have been defined in order to produce a formal specification from *SysML/KAOS* goal models [11,16]. Nevertheless, the generated specification did not contain the system state. This is why in [16], we have presented the use of

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 55–70, 2018.
https://doi.org/10.1007/978-3-319-91271-4_5

ontologies and *UML* class and object diagrams for domain properties representation; we have also introduced rules to derive the system state from these domain representations. Unfortunately, the proposed approach raised several concerns such as the use of several modeling formalisms for the representation of domain knowledge or the disregard of the variability aspect of domain models. In addition, the proposed rules were incomplete and informal. We have therefore proposed in [24] a language for domain knowledge representation through ontologies that meets the shortcomings of [16]. The language allows a high-level modeling of domain properties. This enables the expression of more precise and complete properties. In this paper, we propose rules for translating SysML/KAOS domain models into *B System* specifications. These rules have all been defined and the most relevant ones have been formally specified with *Event-B* [1] and verified with *Rodin* [8]. The formalisation activity is necessary to assess the quality of the SysML/KAOS domain metamodel and of the translation rules, given the criticality of application domains. The *Event-B* method has been chosen because it involves intuitive mathematical concepts and has a powerful refinement logic. It has also been chosen because it is supported by industrial-strength tools. This work contributes to define a formal semantics for the SysML/KAOS domain modeling language, through the definition of its metamodel and its associated constraints in the form of *Event-B* specifications. In the paper, we provide the formal definition of some translation rules, chosen because they are representative of our work and summarise the benefits and difficulties of their expression and verification with Rodin. The approach has been used to deal with the *Hybrid ERTMS/ETCS level 3* case study [12]. It has also been applied on the landing gear system case study [7] and on other case studies (see [28]). The presentation of the work done on the case studies is out of the scope of this paper, but we use an excerpt from the landing gear system case study to illustrate our work.
The remainder of this paper is structured as follows: Sect. 2 briefly describes the key concepts related to the study. This is followed by a presentation, in Sect. 3, of the specification in *Event-B*, of the *B System* and SysML/KAOS domain metamodels. In Sect. 4, we describe some representative translation rules and we provide their formal definition. Section 5 underlines the benefits of using the *Event-B* method for the expression and validation of rules and some challenges encountered. It ends with a positioning of our work with regard to the state of the art. Finally, Sect. 6 reports our conclusions and discusses future work.

2 Context

2.1 SysML/KAOS

SysML/KAOS [11,16] is a requirements engineering method which extends the *SysML* UML profile with a set of concepts from KAOS [15] allowing to represent functional and non-functional requirements. It combines the traceability features provided by *SysML* with goal expressiveness provided by *KAOS*. SysML/KAOS goal models allow the representation of requirements to be satisfied by the system

and of expectations with regard to the environment through a goal hierarchy. The hierarchy is built through a succession of refinements using different operators: *AND* and *OR*. An **AND refinement** decomposes a goal into subgoals, and all of them must be achieved to realise the parent goal. An **OR refinement** decomposes a goal into subgoals such that the achievement of only one of them is sufficient for the accomplishment of the parent goal. The formalisation of SysML/KAOS goal models is detailed in [17]. The proposed rules allow the generation of a formal model whose structure reflects the hierarchy of goal diagrams: one component is associated with each level of the goal hierarchy; this component defines one event for each goal. Proof obligations are defined to formalise the semantics of refinement links between goals.

In this paper, we use the landing gear system case study to illustrate some elements of our approach [7,28]. Figure 1 is an excerpt from its goal diagram focused on the purpose of landing gear expansion (**makeLGExtended**). To achieve it, the handle must be put down (**putHandleDown**) and landing gear sets must be extended (**makeLSExtended**).

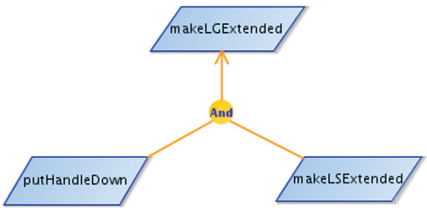


Fig. 1. Excerpt from the landing gear system goal diagram

```
domain model landing_gear_system_ref_0 {
concepts:
  concept LandingGear  {
    is variable: false

    individuals:
       LG1
  }
}
```

Fig. 2. Excerpt from the ontology associated with the root level of the goal model

2.2 Domain Modeling in SysML/KAOS

The SysML/KAOS domain modeling language [24,27] uses ontologies to represent domain models. It is based on *OWL* [21] and *PLIB* [19], two well-known and complementary ontology modeling languages. Figure 3 is an excerpt of its metamodel. The *parent* association represents the hierarchy of domain models. Each domain model corresponds to a refinement level in the SysML/KAOS goal model. A *concept* (instance of metaclass Concept) represents a collection of individuals with common properties. A *concept* can be declared *variable* (*isVariable* = *true*) when the set of its individuals can be updated by adding or deleting individuals. Otherwise, it is *constant* (*isVariable* = *false*). Figure 2 gives an excerpt from the domain model associated to the root level of the landing gear system goal model.

In the rest of this paper, *source* is used in place of SysML/KAOS domain model.

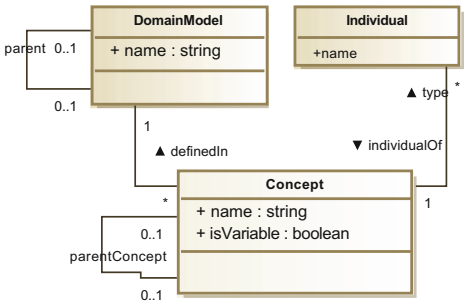


Fig. 3. Excerpt from the metamodel associated with the domain modeling language

2.3 Event-B and B System

Event-B [1] is an industrial-strength formal method for *system modeling*. It is used to incrementally construct a system specification, using refinement, and to prove useful properties. *B System* is an *Event-B* syntactic variant proposed by *ClearSy*, an industrial partner in the *FORMOSE* project [5], and supported by *Atelier B* [9]. *Event-B* and *B System* have the same semantics defined by proof obligations [1].

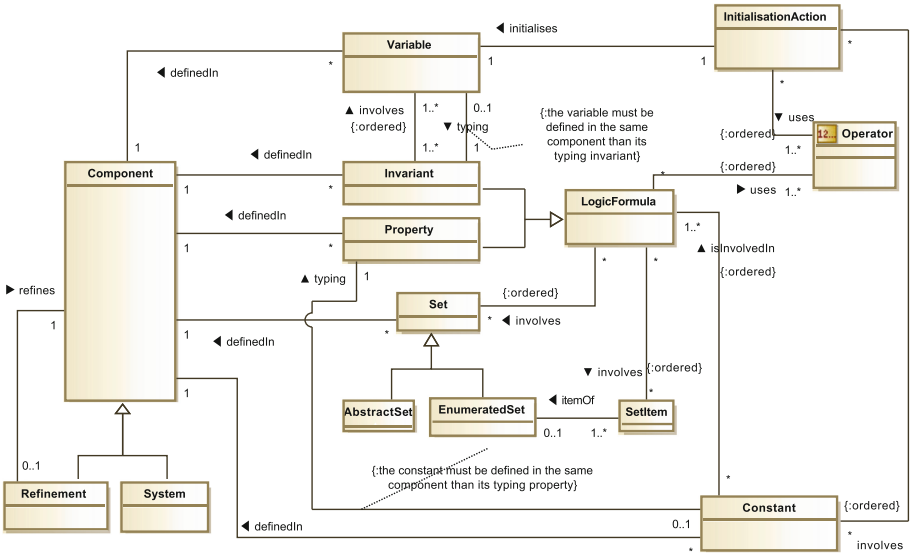


Fig. 4. Metamodel of the *B System* specification language

Figure 4 is a metamodel of the *B System* language restricted to concepts that are relevant to us. A *B System* specification consists of components (instances of

Component). Each component can be either a system or a refinement and it may define static or dynamic elements. A refinement is a component which refines another one in order to access the elements defined in it and to reuse them for new constructions. Constants, abstract and enumerated sets, and their properties, constitute the static part. The dynamic part includes the representation of the system state using variables constrained through invariants and initialised through initialisation actions. Properties and invariants can be categorised as instances of LogicFormula. Variables can be involved only in invariants. In our case, it is sufficient to consider that logic formulas are successions of operands in relation through operators. Thus, an instance of LogicFormula references its operators (instances of Operator) and its operands that may be instances of Variable, Constant, Set or SetItem. Operators include, but are not limited to[1], ***Inclusion_OP*** which is used to assert that the first operand is a subset of the second operand $((Inclusion\_OP, [op_1, op_2]) \Leftrightarrow op_1 \subseteq op_2)$ and ***Belonging_OP*** which is used to assert that the first operand is an element of the second operand $((Belonging\_OP, [op_1, op_2]) \Leftrightarrow op_1 \in op_2)$.

In the rest of this paper, *target* is used in place of *B System*.

3 Specification of Source and Target Metamodels in Event-B

As we have chosen *Event-B* to express and verify the translation rules between the source and target metamodels, the first step is to specify them in *Event-B*. This also allows us to formally define the semantics of SysML/KAOS domain models. Figure 5 represents the structure of the whole *Event-B* specification. This specification can only be split into two abstraction levels because all the translation rules use the class LogicFormula, except those related to the class DomainModel. The first machine, `Ontologies_BSystem_specs_translation`, contains the rules for the translation of instances of DomainModel into instances of Component. The other rules are defined in the machine `Ontologies_BSystem_specs_translation_ref_1`. We have defined static elements of the target metamodel in a context named `BSystem_Metamodel_Context` and static elements of

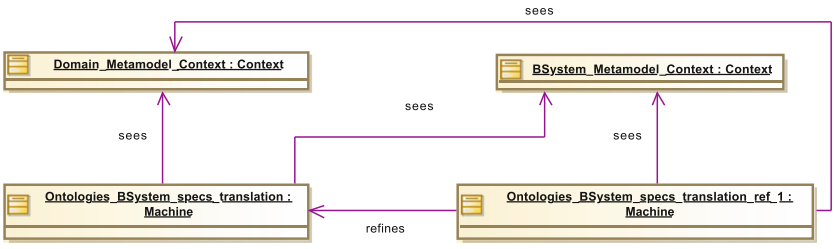


Fig. 5. Structure of the *Event-B* specification

[1] The full list can be found in [26].

the source metamodel in the one named `Domain_Metamodel_Context`. The two machines have access to the definitions of the contexts. For the sake of concision, we provide only an illustrative excerpt of these *Event-B* specifications. For instance, the model `Ontologies_BSystem_specs_translation_ref_1` contains more than a hundred variables, a hundred invariants and fifty events and it gives rise to a thousand proof obligations. The full version can be found in [25,26].

For the translation of some metamodel elements, we have followed the rules proposed in [14,22], such as: classes which are not subclasses give rise to abstract sets, each class gives rise to a variable typed as a subset and containing its instances and each association or property gives rise to a variable typed as a relation. For example, in the following specification, class DomainModel of the source metamodel and class Component of the target metamodel give rise to abstract sets representing all their possible instances. Variables are introduced and typed (`inv0_1, inv0_2` and `inv0_3`) to represent sets of defined instances.

CONTEXT Domain_Metamodel_Context
SETS DomainModel_Set
END

CONTEXT BSystem_Metamodel_Context
SETS Component_Set
END

MACHINE Ontologies_BSystem_specs_translation
VARIABLES Component System Refinement DomainModel
INVARIANT
inv0_1: $Component \subseteq Component\_Set$
inv0_2: $partition(Component, System, Refinement)$
inv0_3: $DomainModel \subseteq DomainModel\_Set$
END

UML enumerations are represented as *Event-B* enumerated sets. For example, in the following specification, defined in `BSystem_Metamodel_Context`, class Operator of the target metamodel is represented as an enumerated set containing the constants *Inclusion_OP* and *Belonging_OP*.

SETS Operator
CONSTANTS Inclusion_OP Belonging_OP
AXIOMS axiom1: $partition(Operator, \{Inclusion\_OP\}, \{Belonging\_OP\})$

Variables are also used to represent attributes and associations [14,22] such as the attribute isVariable of the class Concept in the source metamodel (`inv1_5`) and the association definedIn between the classes Constant and Component in the target metamodel (`inv1_7`). To avoid ambiguity, we have prefixed and suffixed each element name with that of the class to which it is attached (**e.g.** `Concept_isVariable` or `Constant_definedIn_Component`). Furthermore, for better readability of the specification, we have chosen to add *"s"* to the name of all *Event-B* relations for which an image is a set (**e.g.** `Constant_isInvolvedIn_LogicFormulas` or `Invariant_involves_Variables`).

MACHINE Ontologies_BSystem_specs_translation_ref_1
VARIABLES Concept_isVariable Constant_definedIn_Component Invariant_involves_Variables Constant_isInvolvedIn_LogicFormulas
INVARIANT
inv1_5: $Concept\_isVariable \in Concept \rightarrow BOOL$
inv1_7: $Constant\_definedIn\_Component \in Constant \rightarrow Component$
inv1_11: $Invariant\_involves\_Variables \in Invariant \rightarrow (\mathbb{N}_1 \nrightarrow Variable)$
inv1_12: $ran(union(ran(Invariant\_involves\_Variables))) = Variable$
inv1_13: $Constant\_isInvolvedIn\_LogicFormulas \in Constant \rightarrow \mathbb{P}_1(\mathbb{N}_1 \times LogicFormula)$
inv1_14: $\forall co \cdot (co \in Constant \Rightarrow ran(Constant\_isInvolvedIn\_LogicFormulas(co)) \cap Property \neq \emptyset)$
END

An association r from a class A to a class B to which the *ordered* constraint is attached is represented as a variable r typed through the invariant $r \in (A \rightarrow (\mathbb{N}_1 \nrightarrow B))$. This is for example the case of the association *Invariant_involves_Variables* of the target metamodel (`inv1_11`). If instances of B have the same sequence number, then the invariant becomes $r \in (A \rightarrow \mathbb{P}_1(\mathbb{N}_1 \times B))$. This is for example the case of the association *Constant_isInvolvedIn_LogicFormulas* of the target metamodel (`inv1_13`). Invariant `inv1_12` ensures that each variable is involved in at least one invariant and `inv1_14` ensures the same constraint for constants.

4 Translation Rules

4.1 Overview of Translation Rules

Table 1 summarises the translation rules. They are fully described in [26]. These rules cover the formalisation of all elements of the source metamodel, from domain models with or without parents to concepts with or without parents, including relations, individuals or attributes. It should be noted that o_x designates the result of the translation of x and that *abstract* is used for "without parent".

We are not interested in validating the transformation rules of predicates because both source and target metamodels express them using first-order logic notations. The translation of a predicate is a syntactic rewrite. The rules are outlined in [26].

The translation of the ontology of Fig. 2 produces the specification below:

SYSTEM	*landing_gear_system_ref_0*	***PROPERTIES***
SETS	*LandingGear*	$LG1 \in LandingGear \wedge LandingGear = \{LG1\}$
CONSTANTS	*LG1*	

The root domain model is translated into a system component named `landing_gear_system_ref_0` (line 1 of Table 1). The abstract set `LandingGear` appears because *LandingGear* is an instance of the class `Concept` (line 3). The individual `LG1` gives rise to a constant $LG1 \in LandingGear$ (line 8). The property $LandingGear = \{LG1\}$ translates the fact that the *isVariable* property of *LandingGear* is set to *false*.

Table 1. Summary of the translation rules

		Domain model		B system	
		Element	Constraint	Element	Constraint
1	Abstract domain model	DM	$DM \in$ **DomainModel** $DM \notin dom(DomainModel\_parent\_DomainModel)$	o_DM	$o\_DM \in$ **System**
2	Domain model with parent	DM PDM	$\{DM, PDM\} \subseteq$ **DomainModel** $DomainModel\_parent\_DomainModel(DM) = PDM$ PDM has already been translated	o_DM	$o\_DM \in$ **Refinement** $Refinement\_refines\_Component(o\_DM) = o\_PDM$
3	Abstract concept	CO	$CO \in$ **Concept** $CO \notin dom(Concept\_parentConcept\_Concept)$	o_CO	$o\_CO \in$ **AbstractSet**
4	Concept with parent	CO PCO	$\{CO, PCO\} \subseteq$ **Concept** $Concept\_parentConcept\_Concept(CO) = PCO$ PCO has already been translated	o_CO	$o\_CO \in$ **Constant** $o\_CO \subseteq o\_PCO$
5	Relation	RE CO1 CO2	$\{CO1, CO2\} \subseteq$ **Concept** $RE \in$ **Relation** $Relation\_domain\_Concept(RE) = CO1$ $Relation\_range\_Concept(RE) = CO2$ $CO1$ and $CO2$ have already been translated	o_RE	**IF** $(RE \mapsto FALSE) \in Relation\_isVariable$ **THEN** $o\_RE \in$ **Constant** **ELSE** $o\_RE \in$ **Variable** **END** $o\_RE \in o\_CO1 \leftrightarrow o\_CO2$[a]
6	Attribute	AT CO DS	$CO \in$ **Concept** $DS \in$ **DataSet** $AT \in$ **Attribute** $Attribute\_domain\_Concept(AT) = CO$ $Attribute\_range\_Concept(AT) = DS$ CO and DS have already been translated	o_AT	**IF** $(AT \mapsto FALSE) \in Attribute\_isVariable$ **THEN** $o\_AT \in$ **Constant** **ELSE** $o\_AT \in$ **Variable** **END** $o\_AT \in o\_CO \leftrightarrow o\_DS$[b]
7	Concept variability	CO	$CO \in$ **Concept** $Concept\_isVariable(CO) = TRUE$ CO has already been translated	X_CO	$X\_CO \in$ **Variable** $X\_CO \subseteq o\_CO$
8	Individual	Ind CO	$Ind \in$ **Individual** $CO \in$ **Concept** $Individual\_individualOf\_Concept(Ind) = CO$ (CO has already been translated)	o_Ind	$o\_Ind \in$ **Constant** $o\_Ind \in o\_CO$

[a] As usual, this relation becomes a *function*, an *injection*, ... according to the cardinalities of RE.
[b] Depending on attribute properties, this relation may become a partial or total function.

4.2 Event-B Specification of Translation Rules

The correspondence links between instances of a class A of the source metamodel and instances of a class B of the target metamodel are captured in a variable named $A\_corresp\_B$ typed by the invariant $A\_corresp\_B \in A \rightarrowtail B$. It is an injection because each instance, on both sides, must have at most one correspondence. The injection is partial because all the elements are not translated at the same time. Thus, it is possible that at an intermediate state of the system, there are elements not yet translated. For example, correspondence links between instances of Concept and instances of AbstractSet are captured as follows

INVARIANTS inv1_8: $Concept\_corresp\_AbstractSet \in Concept \rightarrowtail AbstractSet$

Translation rules have been modeled as *convergent* events, this guarantees that each rule will be triggered a finite number of times [1] (see Sect. 5.1). Each

event execution translates an element of the source into the target. Variants and event guards and type have been defined such that when the system reaches a state where no transition is possible (deadlock state), all translations are done (see Sect. 5.1). Up to fifty events have been specified. The rest of this section provides an overview of the specification of some of these events in order to illustrate the formalisation process and some of its benefits and difficulties. The full specification can be found in [25,26].

Translating a Domain Model with Parent (Line 2 of Table 1). The corresponding event is called *domain_model_with_parent_to_component*. It states that a domain model, associated with another one representing its parent, gives rise to a refinement component.

MACHINE Ontologies_BSystem_specs_translation
INVARIANT
inv0_6: $Refinement\_refines\_Component \in Refinement \rightarrowtail Component$
inv0_7: $\forall xx, px \cdot (\ (\ xx \in dom(DomainModel\_parent\_DomainModel) \ \wedge\ px = DomainModel\_parent\_DomainModel(xx) \wedge px \in dom(DomainModel\_corresp\_Component) \wedge xx \notin dom(DomainModel\_corresp\_Component)\ ) \Rightarrow DomainModel\_corresp\_Component(px) \notin ran(Refinement\_refines\_Component)\ )$
Event domain_model_with_parent_to_component ⟨convergent⟩ ≙
any DM PDM o_DM
where
grd0: $dom(DomainModel\_parent\_DomainModel) \setminus dom(DomainModel\_corresp\_Component) \neq \emptyset$
grd1: $DM \in dom(DomainModel\_parent\_DomainModel) \setminus dom(DomainModel\_corresp\_Component)$
grd2: $dom(DomainModel\_corresp\_Component) \neq \emptyset$
grd3: $PDM \in dom(DomainModel\_corresp\_Component)$
grd4: $DomainModel\_parent\_DomainModel(DM) = PDM$
grd5: $Component\_Set \setminus Component \neq \emptyset$
grd6: $o\_DM \in Component\_Set \setminus Component$
then
act1: $Refinement := Refinement \cup \{o\_DM\}$
act2: $Component := Component \cup \{o\_DM\}$
act3: $Refinement\_refines\_Component(o\_DM) := DomainModel\_corresp\_Component(PDM)$
act4: $DomainModel\_corresp\_Component(DM) := o\_DM$
END
END

The refinement component must be the one refining the component corresponding to the parent domain model. Guard `grd1` is the main guard of the event. It is used to ensure that the event will only handle instances of DomainModel with parent and only instances which have not yet been translated. It also guarantee that the event will be enabled until all these instances are translated. Action `act3` states that *o_DM* refines the correspondent of *PDM*. To discharge, for this event, the proof obligation related to the invariant `inv0_6`, it is necessary to guarantee that, given a domain model *m* not translated yet, and its parent *pm* that has been translated into component *o_pm*, then *o_pm* has no refinement yet. This constraint is encoded by invariant `inv0_7`.

Translating a Concept with Parent (Line 4 of Table 1). This rule leads to two events: the first one for when the parent concept corresponds to an abstract set (the parent concept does not have a parent: line 3 of Table 1) and the second

one for when the parent concept corresponds to a constant (the parent concept has a parent: line 4 of Table 1). Below is the specification of the first event[2].

Event concept_with_parent_to_constant_1 ⟨convergent⟩ ≙
any CO o_CO PCO o_lg o_PCO
where
grd1: $CO \in dom(Concept\_parentConcept\_Concept) \setminus dom(Concept\_corresp\_Constant)$
grd2: $PCO \in dom(Concept\_corresp\_AbstractSet)$
grd3: $Concept\_parentConcept\_Concept(CO) = PCO$
grd4: $Concept\_definedIn\_DomainModel(CO) \in dom(DomainModel\_corresp\_Component)$
grd5: $o\_CO \in Constant\_Set \setminus Constant$
grd6: $o\_lg \in LogicFormula\_Set \setminus LogicFormula$
grd7: $o\_PCO = Concept\_corresp\_AbstractSet(PCO)$
then
act1: $Constant := Constant \cup \{o\_CO\}$
act2: $Concept\_corresp\_Constant(CO) := o\_CO$
act3: $Constant\_definedIn\_Component(o\_CO) := DomainModel\_corresp\_Component($
$Concept\_definedIn\_DomainModel(CO))$
act4: $Property := Property \cup \{o\_lg\}$
act5: $LogicFormula := LogicFormula \cup \{o\_lg\}$
act6: $LogicFormula\_uses\_Operators(o\_lg) := \{1 \mapsto Inclusion\_OP\}$
act7: $Constant\_isInvolvedIn\_LogicFormulas(o\_CO) := \{1 \mapsto o\_lg\}$
act8: $LogicFormula\_involves\_Sets(o\_lg) := \{2 \mapsto o\_PCO\}$
act9: $Constant\_typing\_Property(o\_CO) := o\_lg$
END

The rule asserts that any concept, associated with another one, with the parentConcept association, gives rise to a constant, typed as a subset of the *B System* element corresponding to the parent concept. We use an instance of LogicFormula, named *o_lg*, to capture this constraint linking the concept and its parent correspondents (*o_CO* and *o_PCO*). Guard `grd2` constrains the parent correspondent to be an instance of AbstractSet. Guard `grd4` ensures that the event will not be triggered until the translation of the domain model containing the definition of the concept. Action `act3` ensures that *o_CO* is defined in the component corresponding to the domain model where *CO* is defined. Action `act6` defines the operator used by *o_lg*. Because the parent concept corresponds to an abstract set, *o_CO* is the only constant involved in *o_lg* (`act7`); *o_PCO*, the second operand, is a set (`act8`). Finally, action `act9` defines *o_lg* as the typing predicate of *o_CO*.

The specification of the second event (when the parent concept corresponds to a constant) is different from the specification of the first one in some points. The three least trivial differences appear at guard `grd2` and at actions `act7` and `act8`. Guard `grd2` constrains the parent correspondent to be an instance of Constant: $PCO \in dom(Concept\_corresp\_Constant)$. Thus, the first and the second operands involved in *o_lg* are constants:

act7: $Constant\_isInvolvedIn\_LogicFormulas := Constant\_isInvolvedIn\_LogicFormulas \leftarrow \{$
$(o\_CO \mapsto \{1 \mapsto o\_lg\}),$
$o\_PCO \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_PCO) \cup \{2 \mapsto o\_lg\}\}$

act8: $LogicFormula\_involves\_Sets(o\_lg) := \emptyset$

[2] Some guards and actions have been removed for the sake of concision.

This approach to modeling logic formulas allows us to capture all the information conveyed by the predicate which can then be used to make inferences and semantic analysis. It is especially useful when we deal with rules to propagate changes made to a generated *B System* specification back to the domain model (ie, propagate changes made to the target into the source). The study of these propagation rules will be the next step in our work.

5 Discussion and Experience

The rules that we propose allow the automatic translation of domain properties, modeled as ontologies, to *B System* specifications, in order to fill the gap between the system textual description and the formal specification. It is thus possible to benefit from all the advantages of a high-level modeling approach within the framework of the formal specification of systems: decoupling between formal specification handling difficulties and system modeling; better reusability and readability of models; strong traceability between the system structure and stakeholder needs. Applying the approach on case studies [28] allowed us to quickly build the refinement hierarchy of the system and to determine and express the safety invariants, without having to manipulate the formal specifications. Furthermore, it allows us to limit our formal specification to the perimeter defined by the expressed needs. This step also allowed us to enrich the domain modeling language expressiveness.

5.1 Benefits

Formally defining the SysML/KAOS domain modeling language, using *Event-B*, allowed us to completely fulfill the criteria for it to be an ontology modeling formalism [4]. Furthermore, formally defining the rules in *Event-B* and discharging the associated proof obligations allowed us to prove their consistency, to animate them using *ProB* and to reveal several constraints (guards and invariants) that were missing when designing the rules informally or when specifying the metamodels. For instance: (1) if an instance of Concept x, with parent px does not have a correspondent yet and if px does, then, the correspondent of px should not be refined by any instance of Component (`inv0_7` defined in`Ontologies_BSystem_specs_translation` and described in Sect. 4.2); (2) elements of an enumerated data set should have correspondents if and only if the enumerated data set does; (3) if a concept, given as the domain of an attribute (instance of Attribute), is variable, then the attribute must also be variable; the same constraint is needed for the domain and the range of a relation. In case of absence of this last constraint, it is possible to reach a state where an attribute maplet (instance of AttributeMaplet) is defined for a non-existing individual (because the individual has been dynamically removed). These constraints have been integrated in the SysML/KAOS domain modeling language in order to strengthen its semantics.

There are two essential properties that the specification of the rules must ensure and that we have proved using Rodin. The first one is that the rules

are isomorphisms and it guarantees that established links between elements of the ontologies are preserved between the corresponding elements in the *B System* specification and vice versa. To do this, we have introduced, for each link between elements, an invariant guaranteeing the preservation of the corresponding link between the correspondences and we have discharged the associated proof obligations. This leads to fifty invariants. For example, to ensure that for each domain model *pxx*, parent of *xx*, the correspondent of *xx* refines the correspondent of *pxx* and vice versa, we have defined the following invariants:

inv0_8: *For each domain model* **pxx**, *parent of a domain model* **xx**, *when* **xx** *and* **pxx** *will be translated, then the correspondence of* **xx** *will refine the correspondence of* **pxx** :

$$\begin{aligned}\forall xx, pxx \cdot (\ & (xx \in dom(DomainModel\_parent\_DomainModel) \\ & \wedge\ pxx = DomainModel\_parent\_DomainModel(xx) \\ & \wedge \{xx, pxx\} \subseteq dom(DomainModel\_corresp\_Component)) \\ \Rightarrow (& DomainModel\_corresp\_Component(xx) \in dom(Refinement\_refines\_Component) \\ & \wedge Refinement\_refines\_Component(DomainModel\_corresp\_Component(xx)) \\ & = DomainModel\_corresp\_Component(pxx))\ )\end{aligned}$$

Its dual version is defined by

inv0_9: *For each component* **o_xx**, *which refines a component* **o_pxx**, *if* **o_xx** *and* **o_pxx** *are introduced by translation rules, then the domain model corresponding to* **o_pxx** *is the parent of the domain model corresponding to* **o_xx**:

$$\begin{aligned}\forall o\_xx, o\_pxx \cdot (\ & (o\_xx \in dom(Refinement\_refines\_Component) \\ & \wedge\ o\_pxx = Refinement\_refines\_Component(o\_xx) \\ & \wedge \{o\_xx, o\_pxx\} \subseteq ran(DomainModel\_corresp\_Component)) \\ \Rightarrow (& DomainModel\_corresp\_Component^{-1}(o\_xx) \in dom(DomainModel\_parent\_DomainModel) \\ & \wedge DomainModel\_parent\_DomainModel(DomainModel\_corresp\_Component^{-1}(o\_xx)) = \\ & DomainModel\_corresp\_Component^{-1}(o\_pxx))\ )\end{aligned}$$

The second essential property is to demonstrate that the system will always reach a state where all translations have been established. To automatically prove it, we have introduced, within each machine, a *variant* defined as the difference between the set of elements to be translated and the set of elements already translated. Then, each event representing a translation rule has been marked as *convergent* and we have discharged the proof obligations ensuring that each of them decreases the *variant*. For each rule, the number of elements to be translated is defined and finite; since we are sure, regarding the event convergence, that each triggering of the rule translates an untranslated element, we are guaranteed that in a finite number of triggerings, all elements will be translated. For example, in the machine `Ontologies_BSystem_specs_translation` containing the definition of translation rules from domain models to *B System* components, the variant was defined as $DomainModel \setminus dom(DomainModel\_corresp\_Component)$. Thus, at the end of system execution, we will have $dom(DomainModel\_corresp\_Component) = DomainModel$, which will reflect the fact that each domain model has been translated into a component.

5.2 Challenges

There is no predefined type for ordered sets in *Event-B*. This problem led us to the definition of composition of functions in order to define relations on ordered sets. Moreover, because of the size of our model (about one hundred invariants and about fifty events for each machine), we noted a rather significant performance reduction of *Rodin* during some operations such as the execution of auto-tactics or proof replay on undischarged proof obligations that have to be done after each update in order to discharge all previously discharged proofs. Table 2 summarises the key characteristics of the Rodin project corresponding to the *Event-B* specification of metamodels and rules. The proof obligations have been discharged using the Rodin tool extended with *Atelier B provers* [20] and *SMT solvers* [23]. The automatic provers seemed least comfortable with functions ($\mapsto, \rightarrowtail, \rightarrow, \twoheadrightarrow$) and become almost useless when those operators are combined in definitions as for ordered associations ($r \in (A \rightarrow (\mathbb{N}_1 \nrightarrow B))$).

Table 2. Key Characteristics of the *Event-B* specification rodin project

Characteristics	Root level	First refinement level
Events	3	50
Invariants	11	98
Proof obligations (PO)	37	990
Automatically discharged POs	27	274 (86 for the *INITIALISATION* event)
Interactively discharged POs	10	**716** (Most used provers: *ML, PP, SMTs*)

5.3 Related Work

The study of correspondence links between domain models or ontologies and formal methods has been the subject of numerous works. The work presented in [6] is interested in describing entities, their mereology, their behaviours and their transformations. Rules are provided for the formalisation of these elements. On the other hand, our study is focused on the description of entities of a system application domain and their instances, of their constraints and of their attributes and associations. Moreover, our modeling is done through successive refinements and the translation rules integrate the refinement links between modules. In [29], an approach is proposed for the automatic extraction of domain knowledge, as *OWL* ontologies, from *Z/Object-Z (OZ)* models. A similar approach is proposed in [10], for the extraction of *DAML* ontologies from *Z* models. These approaches are interested in correspondence links between formal methods and ontologies, but their rules are restricted to the extraction of domain model elements from formal specifications. Furthermore, all elements extracted from a formal model are defined within a single ontology component, while in our approach, each ontology refinement level corresponds to a formal model component. Some rules for passing from an *OWL* ontology representing a domain

model to *Event-B* specifications are proposed in [2,3] and through a case study in [16]. The approaches in [2,3] require a manual transformation of the ontology before the possible application of translation rules to obtain the formal specifications. In [2], it is necessary to convert OWL ontologies into UML diagrams. In [3], the proposal requires the generation of a controlled English version of the *OWL* ontology which serves as the basis for the development of the *Event-B* specification. Furthermore, for this to be completed, the names of ontology elements must necessarily be expressed in English. Moreover, since the OWL formalism supports weak typing and multiple inheritance, the approaches define a unique *Event-B* abstract set named *Thing*. Thus, all sets, corresponding to OWL classes, are defined as subsets of *Thing*. Our formalism, on the other hand, imposes strong typing and simple inheritance; which makes it possible to translate some concepts into *Event-B* abstract sets. Several shortcomings are common to these approaches: the provided rules do not take into account the refinement links between model parts. Furthermore, they are provided in an informal way and they are not supported by tools. Finally, the approaches are only interested in static domain knowledge: they do not distinguish what gives rise to formal constants or variables.

Many studies have been done on the translation of UML diagrams into B specifications such as [14,22]. They inspired many of our rules, like those dealing with the translation of concepts (classes) and of attributes and relations (associations). But, our work differs from them because of the distinctions between ontologies and UML diagrams: within an ontology, concepts or classes and their instances are represented within the same model as well as the predicates defining domain constraints. Moreover, these studies are most often interested in the translation of model elements and not really in handling links between models. Finally, in the case of the SysML/KAOS domain modeling language, the variability properties (attributes characterising the belonging of an element to the static or dynamic knowledge) are first-class citizens, as well as association characteristics. As a result, they are explicitly represented.

6 Conclusion and Future Works

This paper proposes an *Event-B* formalisation of translation rules between domain ontologies and *B System* specifications. Their consistency has been proved through Rodin [8], which allowed us to prove some properties regarding rules such as isomorphisms and to determine some guards and invariants that were missed during their initial specification. The translation rules have been implemented within an open source tool [28], allowing the construction of domain models and the generation of the corresponding *B System* specifications. It is built through *Jetbrains Meta Programming System* [13], a tool to design domain specific languages using language-oriented programming. It has been used to apply the approach on three significant case studies [28]. Our work allows the complete extraction of the structural part of the system formal specification from domain models. We also extract the initialisation of system

variables. The specification obtained completes models resulting from the formalisation of SysML/KAOS goal diagrams. However, it remains necessary to manually provide the body of events, which can lead to updates on the structure of the system.

Work in progress is aimed at evaluating the impact of updates on formal specifications within domain models. We are also working on integrating the translation rules within the open-source platform *Openflexo* [18] which federates the various contributions of *FORMOSE* project partners [5] and which currently supports the construction of SysML/KAOS goal diagrams and domain models.

Acknowledgment. This work is carried out within the framework of the *FORMOSE* project [5] funded by the French National Research Agency (ANR). It is also partly supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

1. Abrial, J.: Modeling in Event-B - System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Alkhammash, E., Butler, M.J., Fathabadi, A.S., Cîrstea, C.: Building traceable Event-B models from requirements. Sci. Comput. Program. **111**, 318–338 (2015)
3. Alkhammash, Eman H.: Derivation of Event-B models from OWL ontologies. In: MATEC Web Conference, vol. 76, p. 04008 (2016)
4. Ameur, Y.A., Baron, M., Bellatreche, L., Jean, S., Sardet, E.: Ontologies in engineering: the OntoDB/OntoQL platform. Soft. Comput. **21**(2), 369–389 (2017)
5. ANR-14-CE28-0009: Formose ANR project (2017)
6. Bjørner, D., Eir, A.: Compositionality: ontology and mereology of domains. In: Dams, D., Hannemann, U., Steffen, M. (eds.) Concurrency, Compositionality, and Correctness. LNCS, vol. 5930, pp. 22–59. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11512-7_3
7. Boniol, F., Wiels, V.: The landing gear system case study. In: Boniol, F., Wiels, V., Ait Ameur, Y., Schewe, K.-D. (eds.) ABZ 2014. CCIS, vol. 433, pp. 1–18. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07512-9_1
8. Butler, M., Jones, C.B., Romanovsky, A., Troubitsyna, E. (eds.): Rigorous Development of Complex Fault-Tolerant Systems. LNCS, vol. 4157. Springer, Heidelberg (2006). https://doi.org/10.1007/11916246
9. ClearSy: Atelier B: B System (2014). http://clearsy.com/
10. Dong, J.S., Sun, J., Wang, H.: Z approach to semantic web. In: George, C., Miao, H. (eds.) ICFEM 2002. LNCS, vol. 2495, pp. 156–167. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36103-0_18
11. Gnaho, C., Semmak, F., Laleau, R.: Modeling the impact of non-functional requirements on functional requirements. In: Parsons, J., Chiu, D. (eds.) ER 2013. LNCS, vol. 8697, pp. 59–67. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-14139-8_8
12. Hoang, T.S., Butler, M., Reichl, K.: The Hybrid ERTMS/ETCS Level 3 Case Study, pp. 1–3. ABZ (2018)
13. JetBrains: JetBrains MPS (2017). https://www.jetbrains.com/mps/
14. Laleau, R., Mammar, A.: An overview of a method and its support tool for generating B specifications from UML notations, pp. 269–272. ICS (2000)

15. van Lamsweerde, A.: Requirements Engineering - From System Goals to UML Models to Software Specifications. Wiley, Hoboken (2009)
16. Mammar, A., Laleau, R.: On the use of domain and system knowledge modeling in goal-based Event-B specifications. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 325–339. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47166-2_23
17. Matoussi, A., Gervais, F., Laleau, R.: A goal-based approach to guide the design of an abstract Event-B specification. In: ICECCS 2011, pp. 139–148. IEEE Computer Society (2011)
18. OpenFlexo: OpenFlexo project (2015). http://www.openflexo.org
19. Pierra, G.: The PLIB ontology-based approach to data integration. In: Jacquart, R. (ed.) Building the Information Society. IIFIP, vol. 156, pp. 13–18. Springer, Boston, MA (2004). https://doi.org/10.1007/978-1-4020-8157-6_2
20. Project, D.: Rodin Atelier B Provers Plug-in (2017). https://www3.hhu.de/stups/handbook/rodin/current/html/atelier_b_provers.html
21. Sengupta, K., Hitzler, P.: Web ontology language (OWL). In: Encyclopedia of Social Network Analysis and Mining, pp. 2374–2378 (2014)
22. Snook, C., Butler, M.: UML-B: formal modeling and design aided by UML. ACM Trans. Softw. Eng. Methodol. **15**(1), 92–122 (2006)
23. SYSTEREL: Rodin SMT Solvers Plug-in (2017). http://wiki.event-b.org/index.php/SMT_Solvers_Plug-in
24. Tueno, S., Laleau, R., Mammar, A., Frappier, M.: Towards using ontologies for domain modeling within the SysML/KAOS approach. In: 25th IEEE International Requirements Engineering Conference on IEEE Proceedings of MoDRE Workshop (2017)
25. Tueno, S., Laleau, R., Mammar, A., Frappier, M.: Event-B Specification of Translation Rules (2017). https://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser/tree/master/SysMLKAOSDomainModelRules
26. Tueno, S., Laleau, R., Mammar, A., Frappier, M.: Formal Representation of SysML/KAOS Domain Models. ArXiv e-prints, cs.SE, 1712.07406, December 2017
27. Tueno, S., Laleau, R., Mammar, A., Frappier, M.: The SysML/KAOS Domain Modeling Approach. ArXiv e-prints, cs.SE, 1710.00903, September 2017
28. Tueno, S., Laleau, R., Mammar, A., Frappier, M.: The SysML/KAOS Domain Modeling Language (Tool and Case Studies) (2017). https://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser/tree/master
29. Wang, H.H., Damljanovic, D., Sun, J.: Enhanced semantic access to formal software models. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 237–252. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16901-4_17

A Translation from Alloy to B

Sebastian Krings[1(✉)], Joshua Schmidt[1], Carola Brings[1], Marc Frappier[2], and Michael Leuschel[1]

[1] Institut für Informatik, Universität Düsseldorf, Universitätsstr. 1, 40225 Düsseldorf, Germany
{krings,leuschel}@cs.uni-duesseldorf.de
[2] Université de Sherbrooke, Sherbrooke, Québec, Canada
marc.frappier@usherbrooke.ca

Abstract. In this paper, we introduce a translation of the specification language Alloy to classical B. Our translation closely follows the Alloy grammar, each construct is translated into a semantically equivalent component of the B language. In addition to basic Alloy constructs, our approach supports integers and orderings. The translation is fully automated by the tool "Alloy2B". We evaluate the usefulness by applying AtelierB and ProB to the translated models, and show benefits for proof and solving with integers and higher-order quantification.

1 Introduction

Both B [1] and Alloy [10] are specification languages based on first-order logic. The languages share several features, such as native support for integers, sets and relations as well as user-defined types. However, there are also considerable differences. For instance, one of B's key concepts is to encode state changes by means of transitions, effectively computing successor states featuring all variables. In contrast, Alloy allows to define orderings over certain types.

Another difference between Alloy and B is tool support, especially when it comes to available backends for constraint solving. For Alloy, the Alloy Analyzer [10] is used to compute models by translating Alloy predicates to SAT using Kodkod [30]. The most prominent constraint solver for B, ProB [16–18], however mainly relies on constraint logic programming [11]. In particular, it uses the CLP(FD) library of SICStus Prolog [2] and extends it to support constraints over infinite domains [12]. Additionally, ProB allows to use other backends, such as SMT solvers [13] or, again, Kodkod [28].

The different constraint solving techniques show different performance characteristics [29]. Certain predicates can be solved faster by using a particular backend or combination of backends; others cannot be handled by a particular solving technique at all. We thus suppose that a translation from Alloy models to B models serves different purposes:

– It provides Alloy users access to a set of new backends, and might enable constraint solving for Alloy models that can not be handled efficiently by the Alloy Analyzer,

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 71–86, 2018.
https://doi.org/10.1007/978-3-319-91271-4_6

Listing 1. Own grandpa (Alloy)

```
module SelfGrandpas
abstract sig Person {
    father : lone Man,
    mother : lone Woman
}
sig Man   extends Person {   wife : lone Woman    }
sig Woman extends Person {   husband : lone Man   }
```

Listing 2. Own grandpa (B - signatures)

```
MACHINE SelfGrandpas
SETS
    Person
CONSTANTS
    Man, Woman, father, mother, wife, husband
PROPERTIES
    father : Person +-> Man &
    mother : Person +-> Woman &
    Man <: Person &
    wife : Man +-> Woman &
    Woman <: Person &
    husband : Woman +-> Man &
    Man /\ Woman = {} &
    Man \/ Woman = Person
END
```

- it enables the application of the Atelier-B provers [3] to Alloy models,
- it enables the usage of ProB as a second toolchain to validate the results of the Alloy Analyzer,
- it provides new test cases and benchmarks to the B community and should aid in improving ProB,
- it helps communication between the Alloy and B communities.

Details about installing and using our translation can be found at: https://www3.hhu.de/stups/prob/index.php/Alloy

2 Translation Example

In the following section, we will introduce our translation on a simple Alloy model taken from [10]. The model is given in Listing 1 and Listing 3, the translation is given in Listing 2 and Listing 4. Our translation will only use the following concepts of a B machine:

1. Deferred sets, introducing new types for Alloy signatures in the SETS clause
2. Constants, introduced in the CONSTANTS clause,
3. Predicates about the constants and deferred sets in the PROPERTIES clause,
4. DEFINITIONS, aka B macros, to ease translating certain Alloy concepts,
5. B Operations for Alloy assertion checks.

In particular, our translation does not use variables, invariants or assertions.

2.1 Translating Signatures

We first concentrate on the translation of Alloy's signatures and fields in Listing 1 to B types. An overview of the signatures and fields can be found in Fig. 1.

In order to translate the Alloy module `SelfGrandpas`, we create a B machine with the same name. Inside, the basic signature `Person`, defined in line 2 of the Alloy model, is represented as a user-given set in line 3 of the B machine in Listing 2.[1] Deferred sets in B can have any size, just like signatures in Alloy. (In Sect. 3.3 we show how a limit on the size of the signature is translated.)

The signature features two fields, `father` and `mother`, each representing a relation of members of `Person` to members of `Man` and `Woman`. The keyword `lone` states that the relation is in fact a partial function, i.e., a 1-to-at-most-1 mapping. This can be encoded into B using a partial function, as created by the `+->` operator in lines 7 and 8 of Listing 2.

The extending `Man` and `Woman` are subsets of `Person`. As user-given sets in B are distinct, we introduce constants `Man` and `Woman` and assert the subset property in lines 9 and 11 of Listing 2. As above, the fields `wife` and `husband` are translated into partial functions in lines 10 and 12.

Since `Person` was declared `abstract`, two additional properties have to hold for the sub-signatures: each element of `Person` has to be in one of the sub-signatures and the two sub-signatures have to be disjoint. This partitioning of `Person` is encoded in B's set theory in lines 13 and 14 of Listing 2.

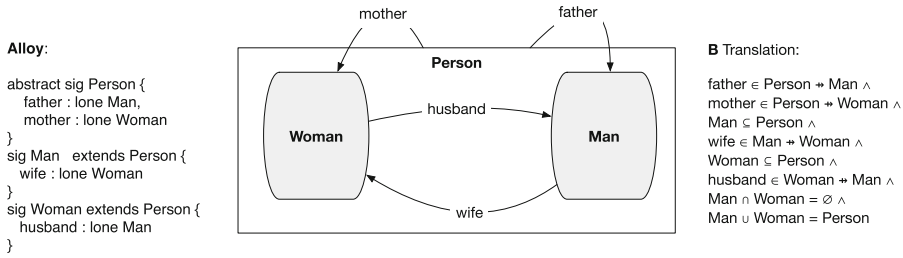


Fig. 1. Signatures and fields in the own grandpa model

[1] For the sake of readability, the example translation uses the same identifiers as the Alloy module. Of course, one has to ensure the translation is valid, e.g., identifiers do not collide with B's keywords.

Listing 3. Own grandpa (Alloy - facts and predicates)

```
1  ...
2  fact Terminology {   wife = ~husband   }
3  fact SocialConvention {
4      no wife & *(mother + father).mother
5      no husband & *(mother + father).father
6  }
7  fact Biology {
8      no p : Person | p in p.^(mother + father)
9  }
10 fun grandpas[p : Person] : set Person {
11     let parent = mother + father + father.wife + mother.
           husband
12     | p.parent.parent & Man
13 }
14 pred ownGrandpa[m : Man] {
15     m in grandpas[m]
16 }
17 run ownGrandpa for 4 Person
```

2.2 Translating Facts and Predicates

Alloy facts are added to the B machine's PROPERTIES clause. For example, the Alloy fact `Terminology` of Listing 3, stating that `wife` is the inverse of `husband`, can be encoded in B using the relational inverse, see line 11 of Listing 4.

The first fact in `SocialConvention` states that your wife cannot be your mother or the mother of any of your ancestors. The second fact asserts the same property for husband and father. Both can be translated directly as far as set union, intersection and closure computation are concerned. The dot join in this case is interpreted as the composition of the two relations, which is available in B as using the `;` operator. Other interpretations of the dot join operator will be discussed later. The `no` keyword enforcing the emptiness of a set is translated to equalities to the empty set in lines 12 and 13.

The Alloy fact `Biology`, stating that nobody can be its own ancestors, introduces a quantified local variable `p`. We translate the fact into a set comprehension, which again is able to introduce the variable. Again, `no` enforces emptiness of the set comprehension. Observe, that quantification in Alloy is over singleton sets only. More generally, we translate the quantification `no p : S | P` into $\{p|\{p\} \subseteq S \wedge P\} = \varnothing$.

The function definition `grandpas` and the predicate definition `ownGrandpa`, both with a parameter, are encoded as B definitions to allow their reuse throughout the model. `ownGrandpa` only includes the application of `grandpas` as well as a membership check and can thus be translated directly.

Listing 4. Own grandpa (B - facts and predicates)

```
MACHINE SelfGrandpas
...
DEFINITIONS
    parent == mother \/ father \/
              (father ; wife) \/ (mother ; husband);
    ownGrandpa(m) == {m} <: Man  &  ({m} <: grandpas(m));
    grandpas(p) == {tmp | {p} <: Person &
                          tmp : (parent[parent[{p}]] /\ Man)}
PROPERTIES
    ...
    wife = husband~ &
    wife /\ (closure((mother \/ father)) ; mother) = {} &
    husband /\ (closure((mother \/ father)) ; father) = {} &
    {p | {p} <: Person &
         {p} <: closure1((mother \/ father))[{p}]} = {} &
    card(Person) <= 4
OPERATIONS
    run_ownGrandpa = PRE #(m).(ownGrandpa(m)) THEN skip END
END
```

Translating `grandpas` however is not straightforward, as it includes a let expression, which is not available in B.[2] As an alternative to inlining, we again create a definition named `parent` in order to hold the value of the newly introduced variable. Note that this changes the scope in which the variable resides and might make renaming necessary to avoid conflicts. Furthermore, observe that there are no free variables in the definition of `parent`. Otherwise, those would be passed to the B definition as parameters. As `grandpas` returns a set of `Person`s, the definition again uses a set comprehension.

3 Translating Alloy to B

In this section, we will outline how to translate the components of an Alloy model into semantically equivalent B components. Each Alloy module is translated into a corresponding B machine. As of now, our translation supports all basic Alloy constructs, i.e., everything that is not inside a library. In particular, this includes integers and the corresponding operations. Regarding libraries, we support orderings, because they are closely related to state transitions in B. Further libraries will be considered in the future.

3.1 Signature Declarations

Since a signature declaration can be quite complex, let us start with the most simple one, omitting everything optional, i.e., we only add a named signature

[2] Let expressions are available in an extended version of B understood by ProB.

to the model. A signature has the properties of a set, containing atoms of the signature's type. For the translation to B, we will create a new deferred set for each signature.

Additionally, a signature can extend another signature by making use of either the `in` or the `extends` keyword. In this case, we set up a subset of an already existing set, i.e., for each sub-signature s extending base signature s_b we define a constant s and add $s \subseteq s_b$ to the `PROPERTIES` clause.

For the `extends` keyword, we ensure that extending signatures are pairwise disjoint by adding $s_1 \cap s_2 = \varnothing$ for each combination of extending signatures s_1, s_2 to the B machine's `PROPERTIES` clause.

Next, base signatures can be declared as `abstract`: Abstract signatures are used for the sole purpose of being extended by other signatures. They do not contain elements which are not also elements of other sets [10]. In B, this property can be modeled by adding the following constraint to the `PROPERTIES` section:

$$s_b = \bigcup_{s \text{ extends } s_b} s.$$

Alloy allows to state the cardinality of signatures by using one of the quantifiers `no` (empty), `lone` (at most one), `one` (exactly one), `some` (at least one) or `set` (any number). The quantifiers can be translated straightforwardly using cardinality constraints as well as existential and universal quantification.

An Alloy signature may contain a list of fields, i.e., relations defined over the signature's elements. Since B natively supports relations, the translation is straightforward - for a signature s with fields f_i, each mapping an element of s to s_i, we add a constant f_i and state that f_i is a relation between s and s_i by the B constraint $f_i \in s \leftrightarrow s_i$.

It is also possible to make use of quantifiers when declaring field variables: In this way we can decide on the number of elements that are mapped to. The default quantification for relations in Alloy is a 1-to-1 mapping (Alloy quantifier `one`) while in B it is an 1-to-n mapping (Alloy quantifier `set`). Therefore, if no quantifier is given in the Alloy model, the translation to B has to be adapted, i.e., we add the constraint $f_i \in s \rightarrow s_i$, stating that f is a total function.

The translation of the remaining quantifiers is analogous, e.g., the quantifier `lone` results in a partial function. In case of `set`, no additional property is needed, since it is the default of B. Alloy allows to provide additional constraints on signature elements together with the signature definition. However, aside of syntactical sugar, they do not differ from regular constraints stated via `fact` declarations and are thus not considered further in this article.

3.2 Fact, Function and Predicate Declaration

Alloy's `fact` declaration has an optional name and contains any number of expressions, which pose additional constraints to be added to the model. The translation of these expressions will be discussed in Sect. 3.4. The results are conjoined and added to the `PROPERTIES` section of our B model.

Alloy allows to declare functions and predicates for later reuse. As usual, a function declaration takes a name, a (possibly empty) list of parameters and a body containing the actual computation. Parameters are scoped and can only be referred to by the function itself. Furthermore, they are typed as subsets of an Alloy signature and can again be quantified to constrain the set sizes.[3]

Functions will be listed in the `DEFINITIONS` section of the B model, if the model contains at least one invocation. Each function is translated into a single definition with matching parameters, consisting of a set comprehension wrapping the actual body to account for the expected return type, e.g., the function declaration `fun f [p : S] : S { body }` is translated into the B definition $f(p) == \{x | p \in S \wedge \text{body}\}$. We include the translation of parameter types conjoined with the translated body.

Syntax and functionality of the predicate definition is slightly different. For the predicate to evaluate to true or false instead of computing a value, we omit the set comprehension.

3.3 Assertion Declaration and Run and Check Commands

In Alloy, assertions can be stated using the `assert` declaration. An assertion does not immediately enforce further constraints. Rather it can later be verified or falsified in a given variable scope, using the `run` and `check` commands. To do so, assertions are named and contain any number of predicates to be checked. The predicates are translated and added to the `DEFINITIONS` clause of the model once they are used inside a `run` or `check` command.

The `run` command instructs the Alloy Analyzer to search for variable states that satisfy the model's constraints. It can either refer to a named predicate introduced by one of the declarations above or include an explicit Alloy predicate. The `check` command is used to check an assertion.

We introduce an `operation` to the B machine for each `run` and `check` command having the translated instructions of the command as its precondition. The operation's substitution is a skip, i.e., we only test if the operation can be executed, without any effect on the model. If the translated model satisfies the predicate to be checked, its specific operation is enabled.

Together with the predicate to be checked, both `run` and `check` include a scope, used to control the search space. By default, the scope defines an upper bound for the cardinality of a signature. The size can be set to a fixed value by using the keyword `exactly`. We define the translated scope in the precondition of the corresponding operation. For instance, the command `run p for 3 S`, for a predicate p and an unordered signature S, results to $card(S) \leq 3$ in B.

To run the Alloy check with ProB one can either use model checking, i.e., try all possible ways to instantiate the constants of the B translation and examine whether the operation is covered, or use constraint-based checking, e.g., using the `cbc_sequence` command of ProB, which will send the operation's guard and the properties to ProB's constraint solver.

[3] Quantifiers are used for typing but do not enforce restrictions on possible models.

3.4 Expressions

Numbers, Identifiers and Blocks. The most basic expressions in Alloy are numbers, identifiers and blocks. Renaming aside, numbers and identifiers can simply be copied to the B machine.

Integer arithmetic is available both in Alloy and in B. Operators have direct counterparts and no involved translation is needed. However, the Alloy Analyzer's approach of translating to SAT is limited: bit width has to be restricted and overflows might occur. We discuss several implications in Sect. 6.1.

Two kinds of blocks can be used for grouping and to manage precedences: `(` *expr* `)` and `{` *expr** `}`, where the list of expressions in the second case is connected conjunctively. Both can directly be translated.

Operations. Aside of basic expressions mentioned above, Alloy expressions can be operations on expressions. As usual, the Alloy grammar distinguishes between unary, binary and comparison operators. For the sake of brevity, we will only discuss operators, that have no direct correspondence in B.

One such special case is the implication operation: *ifExpr* (`implies` | `=>`) *thenExpr* `else` *elseExpr*. B does not include a native if-then-else. However, we can achieve the same behavior using two implications: $ifExpr \Rightarrow thenExpr \wedge \neg ifExpr \Rightarrow elseExpr$.

The translation of the join operation is the most challenging one. Since all variables are tuples - either unary or binary ones - this operator can be used on any two variables (with the exception of two unary variables, which would always result in an empty set). Joining a field variable (binary tuples) with a signature variable (unary tuples), returns a set of unary tuples. Joining a field variable with another field variable, returns a set of binary tuples.

Unfortunately, there is no universal operator to achieve this behavior in B. Thus, we have a different translation for each of the three possibilities:

- Join of unary tuple e_u with binary tuple e_b is translated as the relational image $e_b[e_u]$,
- Join of binary tuple e_b with unary tuple e_u is translated as the relational image of the inverse binary tuple $(e_b{\sim})[e_u]$,
- Join of two binary tuples b_1, b_2 is translated as the sequential substitution $(b_1; b_2)$.

In order to select the correct translation for the join operation, we compute the arity of the expressions involved.

Universe and Identity. Alloy features two special constants: `univ`, referring to the set of all instances of all signatures and `ident`, the identity relation over the universe. Both are unavailable in B. To translate `univ`, we create a top-level set `UNIVERSE` and ensure that all base signatures implicitly extend it. This negatively impacts ProB's solving capabilities: without distinct sets for different signatures, techniques such as symmetry reduction cannot be applied as

efficiently. ProB's kernel becomes unable to reason on types and thus has to perform more involved case distinctions. In consequence, we only create the universal type if necessary. Translation can be avoided in several typical use cases, e.g., left and right joins with the universe can be translated into domain and range computation.

Using `UNIVERSE`, we could translate `ident` to `id(UNIVERSE)`. However, as we want to avoid the universal type as much as possible, we again chose a more specialized translation wherever possible, i.e., instead of translating into the identity over the universe, we rely on Alloy's type checking information and translate into a more restricted identity relation.

Let, Quantified Expressions and Set Comprehensions. As discussed in the introductory example in Sect. 2, classical B does not feature a let expression. This can either be resolved by using a definition as done in the example, inlining or by using the extended version of B understood by ProB.

Alloy features two types of quantified expressions, one that constrains the cardinality of sets and one that allows to introduce quantified variables. The first one uses the quantifiers introduced in Sect. 3.1 followed by a set expression, e.g., `no Number & Letter`, and is translated accordingly. In the second case, quantified variables are introduced and translated to B using set comprehensions as well as universal and existential quantification.

Both Alloy and B feature set comprehensions, consisting of local identifiers and a constraining predicate. Translation is straightforward, as only the predicate has to be translated according to the rules given above.

4 Translating Orderings

Alloy data types are universally based on relations. For instance, sets are unary relations while scalars are singleton sets. Signatures are not ordered by default. However, Alloy allows to declare a total order on signature elements.

Alloy offers the operations *first*, *last*, *next*, *prev*, $nexts(s)$ and $prevs(s)$ for element access on ordered signatures. For an ordered Signature s_o, $s_o/nexts(s)$ returns the set of all successors of $s \in s_o$.

Initially, we translated ordered signatures to B sequences. Sequences are ordered sets of couples whose domains are finite and enumerated from $1..n$, where n is the number of elements. Usually, we translate an Alloy signature to a deferred set in B having the same name as described in Sect. 3.1. The ordered signature can then be represented by a sequence of type s_o, i.e., a set of couples of integer and s_o. B directly offers the operations *first*, *last*, *next*, *prev* for sequences while $nexts(s)$ and $prevs(s)$ can be implemented using set comprehensions.

However, ProB's performance on predicates involving sequences can be lacking when compared to (sets of) integers. In consequence, we tried a different

translation: The scope of a signature is defined within the run or check statement of an Alloy model. Assuming the ordered signature s_o has size $k \in \mathbb{N}$, we translate it to an interval $1, \ldots, k$ in B.

However, we have to consider that ordered signatures can interact, e.g., when computing the union. In consequence, we ensure that ordered signatures are distinct by translating them into disjoint intervals.

Besides that, ordered signatures might interact with unordered ones in Alloy. We then have to define the unordered signature as a set of integer too to avoid type errors in B. To do so, we check an Alloy model for interactions between ordered and unordered signatures prior to the translation.

Using integer intervals, we can define the operations provided by Alloy orderings using set comprehensions. For *first* and *last* we memorize the bounds of each defined set in B which are constant values. We then define $s_o/next$ and $s_o/nexts(s)$ (and *prev* and $prevs(s)$ analogously) for a signature s_o as

$$next(s) == \{x | x = s + 1 \wedge x \in s_o\} \qquad nexts(s) == \{x | x > s \wedge x \in s_o\}.$$

5 Empirical Evaluation

To validate the correctness of our translation we have applied it to a variety of mathematical laws and have checked that ProB does not find counter examples to those laws on the translated B models. In this section we will give a brief empirical evaluation, comparing the Alloy Analyzer and ProB applied to Alloy models. Since the Alloy Analyzer translates models to SAT, we assume it to be efficient for mostly relational models. However, for integers SAT encoding is often inefficient, e.g., one has to encode arithmetic using binary adders. ProB on the other hand has native support for integers, hopefully leading to better performance for arithmetic calculations. In contrast, relations often cause a combinatorial explosion, which results in a weaker performance compared to the Alloy Analyzer.

To explore both extremes, we chose two different models: First, we translate an Alloy model of the river crossing puzzle, a type of transport puzzle with the goal to carry several objects from one river bank to another. There are constraints defining which objects are safe to be left alone, e.g., a fox can not be left alone with a chicken. The model uses an ordered signature for states.

Second, we translate a model of the n queens problem. Here, the goal is to place n queens on a $n*n$ chess board without two queens threatening each other. The chess board is represented as tuples of row and column, encoded as integers.

Benchmarks were run on an Intel Core I7-7700HQ CPU (2.8 GHz) and 32 GB of RAM. We use the median time of five independent checks where the runtime of the Alloy analyzer includes generating the conjunctive normal form.

For the river crossing puzzle, the Alloy analyzer finds a solution in 10 ms. The translated model is valid, yet ProB fails to find a solution in <5 min.

The B model defines three relations, two of which have an ordered signature for a domain. Using a total function instead of a relation improves performance:

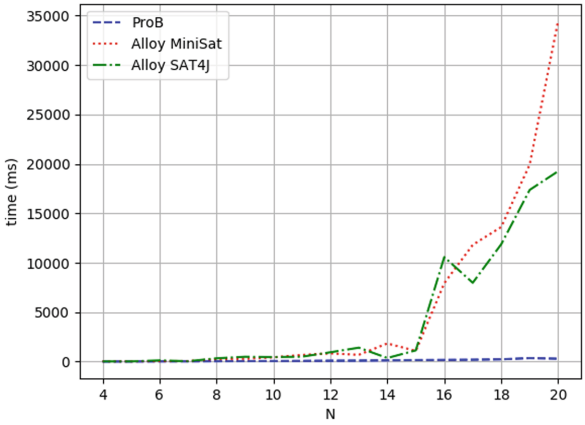


Fig. 2. Find a single solution for the n queens puzzle with varying N

ProB now finds a solution in ∼7 s. After rewriting the model in idiomatic B style by hand, ProB can solve it in about 80 ms. However, this translation is a manual optimization using background knowledge and cannot simply be generalized. An exact opposite to our translation is [28], which uses the Alloy Analyzer's Kodkod API [30] to translate B to SAT. When we use this backend within ProB, the unmodified Alloy translation is solved in about 0.3 s. Note that in recent work [15], we have shown that an integration of the Alloy and ProB backend can be very useful for complex constraint satisfaction problems.

We evaluated the n queens model for $n \in 4..20$ using ProB and the Alloy analyzer with the MiniSat and SAT4J backend. The evaluation in Fig. 2 shows that ProB is the fastest solver for the chosen model. The runtime of the Alloy analyzer gets worse when increasing the bit-width for $n \geq 8$ and $n \geq 16$.

6 Improvements over Existing Alloy Tools

Even though our translation cannot always compete with the Alloy Analyzer as we have demonstrated in Sect. 5, it provides several interesting improvements and applications.

6.1 Integers

Mathematically speaking, integers in Alloy are unsound due to overflows. In contrast, ProB has multi-precision integers without overflows[4]. Using [24] the Alloy Analyzer can detect models with overflows, but to our knowledge cannot detect where an overflow has prevented a model being found. For this purpose, an alternative to translation is to use an SMT-based backend, e.g., [7,31] or [22].

[4] CLP(FD) overflows are caught and handled by custom implementation.

For example, for the following model Alloy 4.2 finds a counter example, while ProB correctly determines that no counter example exists. If overflows are permitted (the default), the Alloy Analyzer finds a counter example for the first formula. If overflows are forbidden, no counter example is detected by the Alloy Analyzer for the first formula, but then a counter example is found for the second one. With higher integer ranges the translation fails.

```
open util/integer
abstract sig setX { }
one sig V {
  SS:   setX -> setX
}
assert Bug {
 #(V.SS)>1 implies #(V.SS->V.SS) >3
 #(V.SS->V.SS)=0 iff no V.SS
}
check Bug for 3 setX, 7 int // for 8 int Translation capacity exceeded
```

6.2 Higher-Order Quantification

The universal quantification below, using the same signatures as in Sect. 6.1 above, causes an error[5], while ProB can check the validity of this assertion. An extension of Alloy called Alloy* [25] might be able to handle this example. In future, we would like to investigate translating Alloy* models to B.

```
assert HO {
   V.SS + V.SS = V.SS
   all xx : V.SS | (xx in V.TT implies xx in V.SS & V.TT)
}
check HO for 3 setX
```

6.3 Proof

Finally, our translation to B also makes it possible to apply its provers, such as [3]. One could thus try and develop a proof assistant for Alloy, similar to the work pursued in [32] by a translation to Key.

In the example below, we can prove the assertion using AtelierB's prover for any scope, by applying it to the translated B machine. We check that the move predicate preserves the invariant `src+dst=Object`.

```
sig Object {}
sig Vars {
    src,dst : Object
}
```

[5] Analysis cannot be performed since it requires higher-order quantification that could not be skolemized.

```
pred move (v, v': Vars, n: Object) {
    v.src+v.dst = Object
    n in v.src
    v'.src = v.src - n
    v'.dst = v.dst + n
    }
assert add_preserves_inv {
    all v, v': Vars, n: Object |
         move [v,v',n] implies  v'.src+v'.dst = Object
}
check add_preserves_inv for 3
```

Note that our translation does not (yet) generate an idiomatic B encoding, with `move` as a B operation and `src+dst=Object` as an invariant: it generates a check operation encoding the predicate `add_preserves_inv` with universal quantification. Below we show the B machine we have input into AtelierB. It was obtained by pretty-printing from ProB, and putting the negated guard of the `add_preserves_inv` into an assertion (so that AtelierB generates the desired proof obligation).

```
MACHINE alloytranslation
SETS /* deferred */
  Object_; Vars_
CONCRETE_CONSTANTS
  src_Vars, dst_Vars
PROPERTIES
    src_Vars : Vars_ --> Object_
  & dst_Vars : Vars_ --> Object_
ASSERTIONS
  !(v_,v__,n_).(v_ : Vars_ & v__ : Vars_ & n_ : Object_
   =>
   (src_Vars[{v_}] \/ dst_Vars[{v_}] = Object_ &
    v_ |-> n_ : src_Vars &
    src_Vars[{v__}] = src_Vars[{v_}] - {n_} &
    dst_Vars[{v__}] = dst_Vars[{v_}] \/ {n_}
    =>
    src_Vars[{v__}] \/ dst_Vars[{v__}] = Object_)
   )
END
```

7 Related Work, Future Work and Conclusions

Translations to Alloy directly have been pursued from B [21,23] and also Z [20]. Other formal languages have previously been translated to B as well [8,27]. A comparison between TLA+ and Alloy can be found in [19].

The original paper [9] (notably Fig. 2 in [9]) provides a semantics of the kernel of Alloy in terms of logical and set-theoretic operators. Our translation rules can

be seen as an alternate specification of this semantics, using the B operators and also using B quantification.

While our translation of orderings, as given in Sect. 4, allows to translate arbitrary Alloy models, the resulting B machine is often suboptimal for ProB's solving kernel. To improve performance, we want to investigate a translation into a (bounded or explicit) model checking rather than a constraint problem. In particular, we intend to translate predicates over states and their successors into B operations. While this is not possible in general, e.g., in the presence of predicates relating more than two states, it would allow us to use symbolic model checking algorithms [14] to find solutions. [26] presents an imperative extension of Alloy, i.e, making a step towards B and its operations. In a similar fashion, [5,6] extended Alloy with actions or bounded model checking [4]. It would be interesting to extend our translation and produce idiomatic B machines with B operations from such Alloy models.

In summary, we have presented an automatic translation of Alloy to B, which provides an alternative semantic definition of Alloy, enables proof and constraint solving tools of B to be applied, and can serve as a vehicle of communication between the Alloy and B community.

References

1. Abrial, J.-R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, New York (1996)
2. Carlsson, M., Ottosson, G., Carlson, B.: An open-ended finite domain constraint solver. In: Glaser, H., Hartel, P., Kuchen, H. (eds.) PLILP 1997. LNCS, vol. 1292, pp. 191–206. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0033845
3. ClearSy: Atelier B, User and Reference Manuals. Aix-en-Provence, France (2009). http://www.atelierb.eu/
4. Cunha, A.: Bounded model checking of temporal formulas with Alloy. In: Ait Ameur, Y., Schewe, K.D. (eds.) ABZ 2014. LNCS, vol. 8477, pp. 303–308. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43652-3_29
5. Frias, M.F., Galeotti, J.P., Pombo, C.L., Aguirre, N.: DynAlloy: upgrading alloy with actions. In: Proceedings of the ICSE, pp. 442–451 (2005)
6. Frias, M.F., Pombo, C.L., Galeotti, J.P., Aguirre, N.: Efficient analysis of DynAlloy specifications. ACM Trans. Softw. Eng. Methodol. **17**(1), 4:1–4:34 (2007)
7. Ghazi, A.A.E., Taghdiri, M.: Analyzing alloy formulas using an SMT solver: a case study. CoRR, abs/1505.00672 (2015)
8. Hansen, D., Leuschel, M.: Translating TLA$^+$ to B for validation with ProB. In: Derrick, J., Gnesi, S., Latella, D., Treharne, H. (eds.) IFM 2012. LNCS, vol. 7321, pp. 24–38. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30729-4_3
9. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. **11**, 256–290 (2002)
10. Jackson, D.: Software Abstractions: Logic, Language and Analysis. MIT Press, Cambridge (2006)
11. Jaffar, J., Michaylov, S.: Methodology and implementation of a CLP system. In: Proceedings ICLP, pp. 196–218. MIT Press (1987)

12. Krings, S., Leuschel, M.: Constraint logic programming over infinite domains with an application to proof. In: Proceedings of WLP. Electronic Proceedings in Theoretical Computer Science, EPTCS, vol. 234 (2016)
13. Krings, S., Leuschel, M.: SMT solvers for validation of B and Event-B models. In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 361–375. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33693-0_23
14. Krings, S., Leuschel, M.: Proof assisted bounded and unbounded symbolic model checking of software and system models. Sci. Comput. Program. **158**, 41–63 (2017)
15. Krings, S., Leuschel, M., Körner, P., Hallerstede, S., Hasanagić, M.: Three is a crowd: SAT, SMT and CLP on a chessboard. In: Calimeri, F., Hamlen, K., Leone, N. (eds.) PADL 2018. LNCS, vol. 10702, pp. 63–79. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73305-0_5
16. Leuschel, M., Bendisposto, J., Dobrikov, I., Krings, S., Plagge, D.: From animation to data validation: the ProB constraint solver 10 years on. In: Boulanger, J.-L. (ed.) Formal Methods Applied to Complex Systems: Implementation of the B Method, pp. 427–446. Wiley ISTE, Hoboken (2014)
17. Leuschel, M., Butler, M.: ProB: a model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45236-2_46
18. Leuschel, M., Butler, M.: ProB: an automated analysis toolset for the B method. Int. J. Softw. Tools Technol. Transf. **10**(2), 185–203 (2008)
19. Macedo, N., Cunha, A.: Alloy meets TLA+: an exploratory study. CoRR, abs/1603.03599 (2016)
20. Malik, P., Groves, L., Lenihan, C.: Translating Z to Alloy. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 377–390. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11811-1_28
21. Matos, P.J., Marques-Silva, J.: Model checking Event-B by encoding into Alloy. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, p. 346. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87603-8_34
22. Meng, B., Reynolds, A., Tinelli, C., Barrett, C.: Relational constraint solving in SMT. In: de Moura, L. (ed.) CADE 2017. LNCS, vol. 10395, pp. 148–165. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63046-5_10
23. Mikhailov, L., Butler, M.: An approach to combining B and Alloy. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) ZB 2002. LNCS, vol. 2272, pp. 140–161. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45648-1_8
24. Milicevic, A., Jackson, D.: Preventing arithmetic overflows in Alloy. Sci. Comput. Program. **94**, 203–216 (2014)
25. Milicevic, A., Near, J.P., Kang, E., Jackson, D.: Alloy*: a general-purpose higher-order relational constraint solver. In: Formal Methods in System Design, January 2017
26. Near, J.P., Jackson, D.: An imperative extension to Alloy. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 118–131. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11811-1_10
27. Plagge, D., Leuschel, M.: Validating Z specifications using the ProB animator and model checker. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 480–500. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73210-5_25
28. Plagge, D., Leuschel, M.: Validating B,Z and TLA$^+$ using ProB and Kodkod. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 372–386. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_31

29. Sülflow, A., Kühne, U., Wille, R., Große, D., Drechsler, R.: Evaluation of SAT-like proof techniques for formal verification of word-level circuits. In: Proceedings IEEE WRTLT, Beijing, China. IEEE Computer Society Press, October 2007
30. Torlak, E., Jackson, D.: Kodkod: a relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_49
31. Torlak, E., Taghdiri, M., Dennis, G., Near, J.P.: Applications and extensions of Alloy: past, present and future. Math. Struct. Comput. Sci. **23**(4), 915–933 (2013)
32. Ulbrich, M., Geilmann, U., El Ghazi, A.A., Taghdiri, M.: A proof assistant for Alloy specifications. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 422–436. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_29

Analysis and Tests

Extracting Symbolic Transitions from TLA$^+$ Specifications

Jure Kukovec[1], Thanh-Hai Tran[1], and Igor Konnov[1,2(✉)]

[1] TU Wien (Vienna University of Technology), Vienna, Austria
{jkukovec,tran,konnov}@forsyte.at
[2] University of Lorraine, CNRS, Inria, LORIA, 54000 Nancy, France
igor.konnov@inria.fr

Abstract. In TLA$^+$, a system specification is written as a logical formula that restricts the system behavior. As a logic, TLA$^+$ does not have assignments and other imperative statements that are used by model checkers to compute the successor states of a system state. Model checkers compute successors either explicitly — by evaluating program statements — or symbolically — by translating program statements to an SMT formula and checking its satisfiability. To efficiently enumerate the successors, TLA's model checker TLC introduces side effects. For instance, an equality $x' = e$ is interpreted as an assignment of e to the yet unbound variable x.

Inspired by TLC, we introduce an automatic technique for discovering expressions in TLA$^+$ formulas such as $x' = e$ and $x' \in \{e_1, \ldots, e_k\}$ that can be provably used as assignments. In contrast to TLC, our technique does not explicitly evaluate expressions, but it reduces the problem of finding assignments to the satisfiability of an SMT formula. Hence, we give a way to slice a TLA$^+$ formula in symbolic transitions, which can be used as an input to a symbolic model checker. Our prototype implementation successfully extracts symbolic transitions from a few TLA$^+$ benchmarks.

1 Introduction

TLA is a general language introduced by Leslie Lamport for specifying temporal behavior of computer systems. It was later extended to TLA$^+$ [18], which provides the user with a concrete syntax for writing expressions over sets, functions, integers, sequences, etc. TLA$^+$ does not fix a model of computation, and thus it found applications in the design of concurrent and distributed systems, e.g., see [2,12,22–24].

A specification alone brings almost no guarantees of system correctness. As it is an untyped language, TLA$^+$ allows for expressions such as $1 \cup \{2\}$, which

Supported by the Vienna Science and Technology Fund (WWTF) through project APALACHE (ICT15-103) and the Austrian Science Fund (FWF) through Doctoral College LogiCS (W1255-N23).

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 89–104, 2018.
https://doi.org/10.1007/978-3-319-91271-4_7

MODULE *prodcons*

VARIABLE S, *empty*

$Init \triangleq S = \{\} \land empty = \text{TRUE}$

$Produce \triangleq \land empty' = \text{FALSE}$
$\land \exists X \in \text{SUBSET } \{\text{“A”}, \text{“B”}, \text{“Z”}, \text{“1”}, \text{“8”}\} : S' = S \cup \{X\}$

$Consume \triangleq \neg empty \land S' \in \text{SUBSET } S \land empty' = (S' = \{\})$

$Next \triangleq Produce \lor Consume$

Fig. 1. A simple producer-consumer

are considered ill-typed in statically-typed programming languages. To formally prove specification properties such as safety and liveness, one can use TLAPS — a proof system for TLA$^+$ [8]. Although progress towards proof automation was made in the last years [20], writing formal proofs is still a challenging task [23,24].

On the other side of the spectrum are model checkers that require little user effort to run. Indeed, TLA$^+$ users debug their specifications with TLC [26]. Beyond simple debugging, TLC found serious bugs in specifications of distributed algorithms [23]. Although TLC contains remarkable engineering solutions, its core techniques *enumerate* reachable states and inevitably suffer from state explosion.

Instead of enumerating states, software model checkers run SAT and SMT solvers in the background to reason about computations symbolically. To name a few, CBMC [15] and CPAChecker [3] implement bounded model checking [4] and CEGAR [9]. Domain-specific tools ByMC and Cubicle prove properties of parameterized distributed algorithms with SMT [10,14].

A simple example in Fig. 1 illustrates the problems that one faces when developing a symbolic model checker for TLA$^+$. In this example, we model two processes: *Producer* that inserts a subset of $\{\text{“}A\text{”}, \text{“}B\text{”}, \text{“}Z\text{”}, \text{“1”}, \text{“8”}\}$ into the set S, and *Consumer* that removes from S its arbitrary subset. The system is initialized with the operator *Init*. A system transition is specified with the operator *Next* that is defined via a disjunction of operators *Produce* and *Consume*. Both Producer and Consumer maintain the state invariant $empty \Leftrightarrow (S = \emptyset)$. We notice the following challenges for a symbolic approach:

1. The specification does not have types. This is not a problem for TLC, since it constructs states on the fly and hence dynamically computes types. In the symbolic case, one can use type synthesis [20] or the untyped SMT encoding [21].
2. Direct translation of *Next* to SMT would produce a *monolithic* formula, e.g., it would not analyze *Produce* and *Consume* as independent actions. This is in sharp contrast to translation of imperative programs, in which variable assignments allow a model checker to focus only on the local state changes.

In this paper, we focus on the second problem. Our motivation comes from the observation on how TLC computes the successors of a given state [18, Chap. 14]. Instead of precomputing all potential successors — which would be

anyway impossible without types — and evaluating *Next* on them, TLC explores subformulas of *Next*. The essential exploration rules are: (1) Disjunctions and conjunctions are evaluated from left to right, (2) an equality $x' = e$ assigns the value of e to x' if x' is yet unbound, (3) if an unbound variable appears on the right-hand side of an assignment or in a non-assignment expression, TLC terminates with an error, and (4) operands of a disjunction assign values to the variables independently. In more detail, rule (4) means that whenever a disjunction $A \vee B$ is evaluated and x' is assigned a value in A, this value does not propagate to B; moreover, x' must be assigned a value in B.

In our example, TLC evaluates the actions *Produce* and *Consume* independently and assigns variables as prescribed by these formulas. As TLC is explicit, for each state, it produces at most 2^{2^5} successors in *Produce* as well as in *Consume*.

We introduce a technique to statically label expressions in a TLA$^+$ formula ϕ as assignments to the variables from a set V', while fulfilling the following:

1. For purely Boolean formulas, if one transforms ϕ into an equivalent formula $\bigvee_{1 \leq i \leq k} D_i$ in disjunctive normal form (DNF), then every disjunct D_i has *exactly one* assignment per variable from V'.
2. The assignments adhere the following partial order: if $x' \in V'$ is assigned a value in expression e, that uses a variable $y' \in V'$, then the assignment to y' precedes the assignment to x'.
3. In general, we formalize the above idea with the notion of a branch.

As expected, the following sequence of expressions is given as assignments in our example: (1) $empty' = \text{TRUE}$, (2) $S' = S \cup \{X\}$, (3) $S' \in \text{SUBSET}\, S$, and (4) $empty' = (S' = \emptyset)$. Using this sequence, our technique constructs two symbolic transitions that are equivalent to the actions *Produce* and *Consume*.

In general, finding assignments and slicing a formula into symbolic transitions is not as easy as in our example, because of quantifiers and IF-THEN-ELSE complicating matters. In this paper, we present our solution, demonstrate its soundness and report on preliminary experiments.

2 Abstract Syntax α-TLA$^+$

TLA$^+$ has rich syntax [18], which cannot be defined in this paper. To focus only on the expressions that are essential for finding assignments in a formula, we define abstract syntax for TLA$^+$ formulas below. In our syntax, the essential operators such as conjunctions and disjunctions are included explicitly, while the other non-essential operators are hidden under the star expression $\star$.

We assume predefined three infinite sets:

- A set $\mathcal{L}$ of *labels*. We use notation ℓ_i to refer to its elements, for $i \in \mathbb{N}$.
- A set *Vars*$'$ of *primed variables* that are decorated with prime, e.g., x' and a'.
- A set *Bound* of *bound variables*, which are used by quantifiers.

The abstract syntax α-TLA$^+$ is defined in terms of the following grammar:

$$
\begin{aligned}
\mathit{expr} ::= {} & \mathit{ex}_\alpha \mid \ell :: \text{FALSE} \\
& \mid \ell :: v' \in \mathit{ex}_\alpha \mid \ell :: \mathit{expr} \wedge \cdots \wedge \mathit{expr} \mid \ell :: \mathit{expr} \vee \cdots \vee \mathit{expr} \\
& \mid \ell :: \exists x \in \mathit{ex}_\alpha : \ \mathit{expr} \mid \ell :: \text{IF} \ \ \mathit{ex}_\alpha \ \ \text{THEN} \ \ \mathit{expr} \ \ \text{ELSE} \ \ \mathit{expr} \\
\mathit{ex}_\alpha ::= {} & \ell :: \star(v', \ldots, v') \\
\ell ::= {} & \text{ a unique label from the set } \mathcal{L} \\
v' ::= {} & \text{ a variable name from the set } \mathit{Vars}' \\
x ::= {} & \text{ a variable name from the set } \mathit{Bound}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Next} \triangleq \ell_1 :: \Big(& \ell_2 :: (\ell_3 :: \mathit{empty}' \in \ell_4 :: \star \wedge \ell_5 :: \exists X \in \ell_6 :: \star : \ell_7 :: S' \in \ell_8 :: \star) \\
& \vee \ell_9 :: (\ell_{10} :: \star \wedge \ell_{11} :: S' \in \ell_{12} :: \star \wedge \ell_{13} :: \mathit{empty}' \in \ell_{14} :: \star(S')) \Big)
\end{aligned}
$$

Fig. 2. The *Next* operator of producer-consumer in α-TLA$^+$

A few comments on the syntax and its relation to TLA$^+$ expressions are in order. We require every expression to carry a unique label $\ell_i \in \mathcal{L}$. Although this is not a requirement in TLA$^+$, it is easy to decorate every expression with a unique label. The expressions of the form $\ell :: v' \in \mathit{expr}$ are of ultimate interest to us, as they are treated as assignment candidates. Under certain conditions, they can be used to assign to v' a value from the set represented by the expression *expr*. Perhaps somewhat unexpectedly, expressions such as $v' = e$ and UNCHANGED$\langle v_1, \ldots, v_k \rangle$ are not included in our syntax. To keep the syntax minimal, we represent them with $\ell :: v' \in \mathit{expr}$. Indeed, these expressions can be rewritten in an equivalent form: $v' = e$ as $v' \in \{e\}$, and UNCHANGED$\langle v_1, \ldots, v_k \rangle$ as $v'_1 \in \{v_1\} \wedge \cdots \wedge v'_k \in \{v_k\}$. Every non-essential TLA$^+$ expression e is presented in the abstract form $\ell :: \star(v'_1, \ldots, v'_k)$, where $v'_1, \ldots, v'_k$ are the names of the primed variables that appear in e. When no primed variable appears in an expression, we omit parenthesis and write $\ell :: \star$. TLA$^+$ expressions often refer to user-defined operators, which are not present in our abstract syntax. We simply assume that all non-recursive user-defined operators have been expanded, that is, recursively replaced with their bodies. All uses of recursive operators are hidden under $\star$; hence, recursive operator definitions are ignored when searching for assignment candidates.

It should be now straightforward to see how one could translate a TLA$^+$ expression to our abstract syntax. We write $\alpha(e)$ to denote the expression in α-TLA$^+$, that represents an expression e in the complete TLA$^+$ syntax. With γ we denote the reverse translation from α-TLA$^+$ to TLA$^+$ that has the property

that $\gamma(\alpha(e)) = e$. Figure 2 shows the abstract expression $\alpha(\mathit{Next})$ of the operator *Next* defined in Fig. 1.

Discussions. Notice that α-TLA$^+$ is missing several fundamental constructs permitted in TLA$^+$, such as CASE expressions, universal quantifiers, and negations. They are all abstracted to $\star$. The primary purpose of α-TLA$^+$ is to allow us to determine whether a given expression containing set inclusion — or equality — can be used as an assignment. If such an expression occurs under a universal quantifier, it is not clear which value should be used for an assignment. Hence, we abstract the expressions under universal quantifiers. For similar reason, we abstract the expressions under negation. The latter is consistent with TLC, which produces an error when given, for example, $\mathit{Next} == \neg(x' = 1)$. Finally, we abstract CASE, due to its semantics, which is defined in terms of the CHOOSE operator [18, Ch. 6]. In practice, there are no potential assignments under CASE in the standard TLA$^+$ examples.

3 Preliminary Definitions

Every TLA$^+$ specification declares a certain finite set of variables, which may appear in the formulas contained therein. Let ϕ be an α-TLA$^+$ expression. We assume, for the purposes of our analysis, that ϕ is associated with some finite set $\mathit{Vars}'(\phi)$, which is a subset of Vars', containing all of the variables that appear in ϕ (and possibly additional ones). This is the set of variables declared by the specification in which $\gamma(\phi)$ appears.

Since the labels are unique, we write $\mathrm{lab}(\ell :: \psi)$ to refer to the expression label ℓ and $\mathrm{expr}(\ell)$ to refer to the expression that is labeled with ℓ. We refer to the set of all subexpressions of ϕ by $\mathrm{Sub}(\phi)$. See [16] for a formal definition.

The set $\mathrm{Sub}(\phi)$ allows us to reason about terms that appear inside an expression ϕ, at some unknown/irrelevant depth. We will often refer to the set of all labels appearing in ϕ, that is, $\mathrm{Labs}(\phi) = \{\mathrm{lab}(\psi) \mid \psi \in \mathrm{Sub}(\phi)\}$.

Of special interest to us are *assignment candidates*, i.e., expressions of the form $\ell :: v' \in \phi_1$. Given a variable $v' \in \mathit{Vars}'(\phi)$ and an α-TLA$^+$ expression ϕ, we write $\mathrm{cand}(v', \phi)$ to mean the set of labels that belong to assignment candidates for v' in subexpressions of ϕ. More formally, $\mathrm{cand}(v', \phi)$ is $\{\ell \mid (\ell :: v' \in \psi) \in \mathrm{Sub}(\phi)\}$. An exhaustive definition is included in [16]. We use the notation $\mathrm{cand}(\phi)$ to mean $\bigcup_{v' \in \mathit{Vars}'(\phi)} \mathrm{cand}(v', \phi)$.

Finally, we assign to each label ℓ in $\mathrm{Labs}(\phi)$ a set $\mathrm{frozen}_\phi(\ell) \subseteq \mathit{Vars}'(\phi)$. Intuitively, if a variable v' is in $\mathrm{frozen}_\phi(\ell)$, then no expression of the form $\hat{\ell} :: v' \in \psi$ can be treated as an assignment inside $\mathrm{expr}(\ell)$. Formally, for every $\ell \in \mathrm{Labs}(\phi)$ the set $\mathrm{frozen}_\phi(\ell)$ is defined as the minimal set satisfying all the constraints in Table 1.

Table 1. The constraints on frozen_ϕ

α-TLA$^+$ expression ϕ	Constraints on frozen_ϕ
$\ell :: \star(v'_1, \ldots, v'_k)$	$\{v'_1, \ldots, v'_k\} \subseteq \text{frozen}_\phi(\ell)$
$\ell :: v' \in \phi_1$	$\text{frozen}_\phi(\ell) = \text{frozen}_\phi(\text{lab}(\phi_1))$
$\ell :: \bigwedge_{i=1}^{s} \phi_i$ or $\ell :: \bigvee_{i=1}^{s} \phi_i$	$\text{frozen}_\phi(\ell) \subseteq \text{frozen}_\phi(\text{lab}(\phi_i))$ for $i \in \{1, \ldots, s\}$
$\ell :: \exists x \in \phi_1 : \ \phi_2$	$\text{frozen}_\phi(\ell) \subseteq \text{frozen}_\phi(\text{lab}(\phi_1)) \subseteq \text{frozen}_\phi(\text{lab}(\phi_2))$
$\ell ::$ IF ϕ_1 THEN ϕ_2 ELSE ϕ_3	$\text{frozen}_\phi(\ell) \subseteq \text{frozen}_\phi(\text{lab}(\phi_1))$
	$\text{frozen}_\phi(\text{lab}(\phi_1)) \subseteq \text{frozen}_\phi(\text{lab}(\phi_i))$ for $i = 2, 3$

The sets frozen_ϕ naturally lead to the dependency relations $\lhd_{v'}$ on $\text{Labs}(\phi)$, where $v' \in \mathit{Vars}'(\phi)$. We will use $\ell_1 \lhd_{v'} \ell_2$ to mean that ℓ_1 is an assignment candidate for v', which also belongs to the frozen set of ℓ_2. Formally:

$$\ell_1 \lhd_{v'} \ell_2 \iff \ell_1 \in \text{cand}(v', \phi) \wedge v' \in \text{frozen}_\phi(\ell_2)$$

Intuitively, if $\ell_1 \lhd_{v'} \ell_2$ we want to make sure that $\text{expr}(\ell_1)$ is evaluated before $\text{expr}(\ell_2)$, if possible.

Example 1. Let us look at the following α-TLA$^+$ expression:

$$\ell_1 :: [\exists i \in [\ell_2 :: \star(y')] : \ell_3 :: x' \in [\ell_4 :: \star]]$$

Take the subexpression $\ell_3 :: x' \in [\ell_4 :: \star]$, which we name ψ . By solving the constraints for $\text{frozen}_\psi(\ell_3)$ we conclude that $\text{frozen}_\psi(\ell_3) = \emptyset$. However, if we take the additional constraints for $\text{frozen}_\phi(\ell_3)$ into consideration, the empty set no longer satisfies all of them, specifically, it does not satisfy the condition imposed by the existential quantifier in ℓ_1. The additional requirement $\{y'\} \subseteq \text{frozen}_\phi(\ell_3)$ implies that $\text{frozen}_\phi(\ell_3) = \{y'\}$. This corresponds to the intuition that expressions under a quantifier, like ψ, implicitly depend on the bound variable and the expressions used to define it, which is $\text{expr}(\ell_2)$ in our example. $\lhd$

4 Formalizing Symbolic Assignments

As TLC evaluates formulas in a left-to-right order, there is a very clear notion of an assignment; the first occurrence of an expression $v' \in S$ is interpreted as an assignment to v'. In our work, we want to *statically* find expressions that can safely be used as assignments. If we were only dealing with Boolean formulas, we could transform the original TLA$^+$ formula to DNF, $\bigvee_{i=1}^{s} D_i$, and treat each D_i independently. However, we also need to find assignments, which may be nested under existential quantifiers. To transfer our intuition about DNF to the general case we first introduce a transformation boolForm, that captures the Boolean structure of the formula. Then, we introduce branches and assignment strategies to formalize the notion of assignments in the symbolic case.

Table 2. The definition of boolForm(ϕ)

α-TLA$^+$ expression ϕ	boolForm(ϕ)
ℓ :: FALSE or ℓ :: $\star\,(v'_1, \ldots, v'_k)$ or $\ell :: v' \in \phi_1$	b_ℓ
$\ell :: \bigwedge_{i=1}^{s} \phi_i$	$\bigwedge_{i=1}^{s}$ boolForm(ϕ_i)
$\ell :: \bigvee_{i=1}^{s} \phi_i$	$\bigvee_{i=1}^{s}$ boolForm(ϕ_i)
$\ell :: \exists x \in \phi_1 :\ \phi_2$	boolForm(ϕ_2)
ℓ :: IF ϕ_1 THEN ϕ_2 ELSE ϕ_3	boolForm(ϕ_2) $\vee$ boolForm(ϕ_3)

Boolean Structure of a Formula and Branches. The transformation boolForm maps an α-TLA$^+$ expression to a Boolean formula over variables from $\{b_\ell \mid \ell \in \mathcal{L}\}$. The definition of boolForm can be found in Table 2. As boolForm(ϕ) is a formula in Boolean logic, a model of boolForm(ϕ) is a mapping from $\{b_\ell \mid \ell \in \mathcal{L}\}$ to $\mathbb{B} = \{\text{true}, \text{false}\}$. Take $S \subseteq \mathcal{L}$. The set S naturally defines a model induced by S, denoted $\mathcal{M}[S]$, by the requirement that $\mathcal{M}[S] \vDash b_\ell$ if and only if $\ell \in S$.

The boolForm transformation allows us to formulate the central notion of a branch: A set $Br \subseteq \mathcal{L}$ is called a *branch* of ϕ if the following constraints hold:

(a) The set Br induces a model of boolForm(ϕ), i.e., $\mathcal{M}[Br] \vDash$ boolForm(ϕ), and
(b) The model $\mathcal{M}[Br]$ is minimal, that is, $\mathcal{M}[S] \nvDash$ boolForm(ϕ) for every $S \subset Br$.

Then, Branches(ϕ) is the set of all branches of ϕ.

Example 2. Let us look the α-TLA$^+$ expression ϕ given by

$$\ell_1 :: [[\ell_2 :: x' \in \star] \wedge [\ell_3 :: [[\ell_4 :: x' \in \star] \vee [\ell_5 :: x' \in \star]]]]$$

We know that boolForm(ϕ) $= b_{\ell_2} \wedge (b_{\ell_4} \vee b_{\ell_5})$. The set $S = \{\ell_2, \ell_4, \ell_5\}$ induces a model of boolForm(ϕ), but it is not a branch of ϕ because $\mathcal{M}[S]$ is not a minimal model. It is easy to see that ϕ has two branches $Br_1 = \{\ell_2, \ell_4\}$, and $Br_2 = \{\ell_2, \ell_5\}$. Therefore, we see that Branches(ϕ) $= \{Br_1, Br_2\}$. ◁

As our goal is to reason about the side-effects of variable assignments, the remainder of this section looks at how we can achieve this with the help of branches.

Assignment Strategies. We want to statically mark some expressions as assignments, that is, pick a set $A \subseteq$ Labs(ϕ). Below, we formulate the critical properties we require from such a set, which we will later call an assignment strategy.

Most obviously, we want to consider only assignment candidates:

Definition 1. *A set $H \subseteq$ Labs(ϕ) is* homogeneous *if all the labels in H are assignment candidates. Formally, $H \subseteq$ cand(ϕ).*

If we choose an arbitrary homogeneous set H, it might lack assignments on some branches or have multiple assignments to the same variable on others. Formally, we say that H has a *covering index* $d \in \mathbb{N}_0$ if there is a branch $Br \in \text{Branches}(\phi)$ and a variable $v' \in Vars'(\phi)$ for which $d = |Br \cap H \cap \text{cand}(v', \phi)|$. Now we define sets, that cover all branches with assignments:

Definition 2. *A homogeneous set C is a* covering *of ϕ, if it does not have* 0 *as a covering index. It is a* minimal covering *of ϕ, if it only has* 1 *as a covering index.*

Consider the TLA$^+$ formula $x' = y' \wedge y' = 2x'$. Its corresponding α-TLA$^+$ expression $\ell_0 :: (\ell_1 :: x' \in \ell_2 :: \star(y') \wedge \ell_3 :: y' \in \ell_4 :: \star(x'))$ has a minimal covering $\{\ell_1, \ell_3\}$. However, there is no way to order the assignments to x' and y'. To detect such cases, we define acyclic sets:

Definition 3. *A homogeneous set A is* acyclic*, if there is a strict total order $\prec_A$ on A, with the following property: For every variable $v' \in V$, every branch $Br \in$ Branches and every pair of labels ℓ_i and ℓ_j in $A \cap Br$ the relation $\ell_i \lhd_{v'} \ell_j$ implies $\ell_i \prec_A \ell_j$.*

Having defined homogeneous, minimal covering, and acyclic sets, we can formulate the notion of an *assignment strategy.*

Definition 4. *Let ϕ be an α-TLA$^+$ expression. A set $A \subseteq \mathcal{L}$ is an assignment strategy for ϕ, if it is an acyclic minimal covering.*

Static Assignment Problem. Given an α-TLA$^+$ expression ϕ, our goal is to find an assignment strategy, or prove that none exists.

5 Finding Assignment Strategies with SMT

For a given α-TLA$^+$ expression ϕ, we construct an SMT formula $\theta(\phi)$, that encodes the properties of assignment strategies. Technically, $\theta(\phi)$ is defined as $\theta_H(\phi) \wedge \theta_C(\phi) \wedge \theta_A(\phi)$, and consists of:

1. A Boolean formula $\theta_H(\phi)$, that encodes homogeneity.
2. A Boolean formula $\theta_C(\phi)$, that encodes the minimal covering property.
3. A formula $\theta_A(\phi)$, that encodes acyclicity. This formula requires the theories of linear integer arithmetic and uninterpreted functions (*QF_UFLIA*).

In the following, Propositions 1, 3, and 4 formally establish the relation between ϕ and its three SMT counterparts. Together, the propositions allows us to prove the following theorem:

Theorem 1. *For every α-TLA$^+$ formula ϕ and $A \subseteq$ Labs(ϕ), it holds that $\mathcal{M}[A] \vDash \theta(\phi)$ if and only if A is an assignment strategy for ϕ.*

Table 3. The definition of $\delta_{v'}(\phi)$

α-TLA$^+$ expression ϕ	$\delta_{v'}(\phi)$
$\ell ::$ FALSE or $\ell :: \star(v'_1, \ldots, v'_k)$	false
$\ell :: w' \in \phi_1$	$\begin{cases} b_\ell & ; w' = v' \\ \text{false} & ; \text{otherwise} \end{cases}$
$\ell :: \bigwedge_{i=1}^{s} \phi_i$	$\bigvee_{i=1}^{s} \delta_{v'}(\phi_i)$
$\ell :: \bigvee_{i=1}^{s} \phi_i$	$\bigwedge_{i=1}^{s} \delta_{v'}(\phi_i)$
$\ell :: \exists x \in \phi_1 : \phi_2$	$\delta_{v'}(\phi_2)$
$\ell ::$ IF ϕ_1 THEN ϕ_2 ELSE ϕ_3	$\delta_{v'}(\phi_2) \wedge \delta_{v'}(\phi_3)$

5.1 Homogeneous Sets

We introduce a Boolean formula, whose models are exactly those induced by homogeneous sets. To this end, take the set of labels corresponding to expressions that are not assignment candidates, $\mathcal{N}(\phi)$, given by $\mathcal{N}(\phi) := \text{Labs}(\phi) \setminus \text{cand}(\phi)$. Then, we define the following:

$$\theta_H(\phi) := \bigwedge_{\ell \in \mathcal{N}(\phi)} \neg b_\ell$$

Proposition 1. *For every α-TLA$^+$ expression ϕ and $A \subseteq \text{Labs}(\phi)$, it holds that $\mathcal{M}[A] \vDash \theta_H(\phi)$ if and only if A is homogeneous.*

5.2 Minimal Covering Sets

Next we construct a Boolean formula $\theta^*_C(\phi)$, whose models are exactly those induced by covering sets. To this end, we define, for each $v' \in \mathit{Vars}'(\phi)$, the transformation $\delta_{v'}$ as shown in Table 3. Intuitively, $\delta_{v'}(\phi)$ is satisfiable exactly when there is an assignment to v' on every branch of ϕ. We then define

$$\theta^*_C(\phi) := \bigwedge_{v' \in \mathit{Vars}'(\phi)} \delta_{v'}(\phi)$$

Formally, the following proposition holds:

Proposition 2. *For every α-TLA$^+$ expression ϕ and $A \subseteq \text{Labs}(\phi)$, it holds that $\mathcal{M}[A] \vDash \theta_H(\phi) \wedge \theta^*_C(\phi)$ if and only if A is a covering set for ϕ.*

It is easy to restrict coverings to the minimal coverings. To do this, we define the set of collocated labels, denoted $\text{Colloc}(\phi)$, as

$$\text{Colloc}(\phi) := \{(\ell_1, \ell_2) \in \mathcal{L}^2 \mid \exists Br \in \text{Branches}(\phi) \,.\, \{\ell_1, \ell_2\} \subseteq Br\}$$

We can use this set to reason about minimal coverings: A minimal covering may contain, per variable, no more than one label from each pair of collocated assignments to that variable. We describe these labels by using the sets $\mathrm{Colloc}_{v'}(\phi) := \mathrm{Colloc}(\phi) \cap \mathrm{cand}(v', \phi)^2$ and

$$\mathrm{Colloc}_{Vars'}(\phi) := \bigcup_{v' \in Vars'(\phi)} \mathrm{Colloc}_{v'}(\phi)$$

Then, the following SMT formula, in addition to $\theta^*_C(\phi)$, helps us find minimal covering sets:

$$\theta^{\exists!}(\phi) := \bigwedge_{\substack{(i,j) \in \mathrm{Colloc}_{Vars'}(\phi) \\ i<j}} \neg(b_i \wedge b_j)$$

We denote by $\theta_C(\phi)$ the formula $\theta^*_C(\phi) \wedge \theta^{\exists!}(\phi)$.

Proposition 3. *For every α-TLA$^+$ expression ϕ and $A \subseteq \mathrm{Labs}(\phi)$, it holds that $\mathcal{M}[A] \vDash \theta_H(\phi) \wedge \theta_C(\phi)$ if and only if A is a minimal covering set for ϕ.*

5.3 Acyclic Assignments

The last step is reasoning about acyclicity. Recall that, for $\ell_1, \ell_2 \in \mathcal{L}$, the relation $\ell_1 \lhd_{v'} \ell_2$ holds if and only if $\ell_1 \in \mathrm{cand}(v', \phi) \wedge v' \in \mathrm{frozen}_\phi(\ell_2)$. It is prudent to see that $\lhd_{v'}$ is not, in general, a strict total order (possibly not even irreflexive). However, the acyclicity property states that we can find a strict total order, which agrees with all relations $\lhd_{v'}$, on all branches.

Take $\mathrm{Colloc}_\lhd(\phi)$ to be the filtering of $\mathrm{Colloc}(\phi)$ by the relations $\lhd_{v'}$, i.e. the set $\{(i,j) \in \mathrm{Colloc}(\phi) \cap \mathrm{cand}(\phi)^2 \mid \exists v' \in Vars'(\phi)\ .\ i \lhd_{v'} j\}$. The SMT formula describing acyclicity is as follows:

$$\theta^*_A(\phi) := \bigwedge_{(i,j) \in \mathrm{Colloc}_\lhd(\phi)} b_i \wedge b_j \Rightarrow R(i) < R(j)$$

where R is an uninterpreted $\mathcal{L} \to \mathbb{N}$ function, capturing assignment order. In practice, we take $\mathcal{L} = \mathbb{N}$. Unlike the previous formulas, $\theta^*_A(\phi)$ extends beyond Boolean logic, requiring both linear integer arithmetic and uninterpreted functions. Thus, a model for $\theta^*_A(\phi)$ is a pair (M, r), where M models the Boolean part of the formula, i.e. assigns truth values to each b_i, and $r\colon \mathbb{N} \to \mathbb{N}$ is the interpretation of R.

To simplify the analysis, we force R to be injective, when it is restricted to $\mathrm{Labs}(\phi)$. Otherwise we could always construct an injective function from R, which respects the required inequalities. The formula we therefore consider is as follows:

$$\theta_A(\phi) := \theta^*_A(\phi) \wedge \bigwedge_{\substack{\ell_1, \ell_j \in \mathrm{Labs}(\phi) \\ \ell_i < \ell_j}} R(\ell_i) \neq R(\ell_j)$$

Proposition 4. *For every α-TLA$^+$ expression ϕ and $A \subseteq \mathrm{Labs}(\phi)$, there is a function $r\colon \mathbb{N} \to \mathbb{N}$, for which $(\mathcal{M}[A], r) \vDash \theta_H(\phi) \wedge \theta_A(\phi)$ if and only if A is acyclic.*

6 Soundness of Our Approach

In this section, we show the relation between assignment strategies and the original TLA$^+$ formulas. To this end, we introduce the notion of a slice. Together, branches allow us to rewrite a TLA$^+$ formula into an equivalent disjunction of slices.

In TLA$^+$, there are two kinds of variables: rigid variables that correspond to the variables declared with CONSTANT, and flexible variables whose values change during the course of an execution. Primed versions of the variables exist only for flexible variables and are used in transition formulas. Transition formulas in TLA$^+$ are first-order terms and formulas with flexible variables (unprimed and primed ones). We give the necessary definitions of TLA$^+$ semantics, whereas details can be found in [19]. An interpretation $\mathcal{I}$ defines a universe $|\mathcal{I}|$ of values and interprets each function symbol by a function and each predicate symbol by a relation. A state s is a mapping from unprimed flexible variables to values, and a state s' is a similar mapping for primed variables. A valuation ξ is a mapping from rigid variables to values. Given an interpretation $\mathcal{I}$, a pair of states (s, s'), and a valuation ξ, the semantics of a TLA$^+$ transition formula E is the standard predicate logic semantics of E with respect to the extended valuation of s, s', ξ. With these definitions, $M = (\mathcal{I}, \xi, s, s')$ is a model for E, if E is equivalent to true under M. Let ϕ be a formula and $S \subseteq \mathcal{L}$. We define ϕ *sliced by* S, denoted $\text{slice}(\phi, S)$ in Table 4.

Table 4. The definition of $\text{slice}(\phi, S)$

α-TLA$^+$ formula ϕ	$\text{slice}(\phi, S)$
$\ell :: \text{FALSE}$	$\ell :: \text{FALSE}$
$\ell :: \star(v_1', \ldots, v_1')$ or $\ell :: v' \in \phi_1$	$\begin{cases} \phi & ; \ell \in S \\ \ell :: \text{FALSE} & ; \text{otherwise} \end{cases}$
$\ell :: \bigwedge_{i=1}^{s} \phi_i$	$\ell :: \bigwedge_{i=1}^{s} \text{slice}(\phi_i, S)$
$\ell :: \bigvee_{i=1}^{s} \phi_i$	$\ell :: \bigvee_{i=1}^{s} \text{slice}(\phi_i, S)$
$\ell :: \exists x \in \phi_1 : \phi_2$	$\ell :: \exists x \in \phi_1 : \text{slice}(\phi_2, S)$
$\ell :: \text{IF } \phi_1 \text{ THEN } \phi_2 \text{ ELSE } \phi_3$	$\ell :: \text{IF } \phi_1 \text{ THEN } \text{slice}(\phi_2, S) \text{ ELSE } \text{slice}(\phi_3, S)$

Below, we show that the branches and slices induced by them naturally decompose a TLA$^+$ formula. Let ϕ be an α-TLA$^+$ expression and $\gamma(\phi)$ its corresponding TLA$^+$ formula. Then, the following holds:

Proposition 5. *Let ϕ be an α-TLA$^+$ expression and $M = (\mathcal{I}, \xi, s, s')$ a model of the TLA$^+$ formula $\gamma(\phi)$. There exists a branch Br of ϕ such that M is also a model of $\gamma(\text{slice}(\phi, Br))$.*

Proposition 6. *Let ϕ be an α-TLA$^+$ expression and $M = (\mathcal{I}, \xi, s, s')$ a model of the TLA$^+$ formula $\gamma(\text{slice}(\phi, Br))$. Then, M is also a model of $\gamma(\phi)$.*

Proposition 7. *Let ϕ be an α-TLA$^+$ expression. For every $S, T \subseteq \text{Labs}(\phi)$, every model M of the* TLA$^+$ *formula $\gamma(\text{slice}(\phi, S))$, is also a model of $\gamma(\text{slice}(\phi, S \cup T))$.*

It is easy to see that Proposition 7 does not hold in the other direction. For instance, take the empty set as S and $\text{Labs}(\phi)$ as T. This implies the following:

$$\gamma(\text{slice}(\phi, S)) = \text{FALSE} \text{ and } \text{slice}(\phi, S \cup T) = \phi.$$

Obviously, FALSE cannot have a model, regardless of whether $\gamma(\phi)$ has one or not.

Since Propositions 5 and 6 hold, it would suffice to consider the set $\text{Branches}(\phi)$, together with an assignment strategy, to generate symbolic transitions. However, it is often the case that, for two distinct branches Br_1 and Br_2, the same assignments in A are chosen, that is, the intersections $Br_1 \cap A$ and $Br_2 \cap A$ are the same. We show that one can reduce the number of considered symbolic transitions, by analyzing how various branches intersect A.

An assignment strategy A naturally defines an equivalence relation $\sim_A$ on $\text{Branches}(\phi)$, given by $Br_1 \sim_A Br_2$ if and only if $Br_1 \cap A = Br_2 \cap A$. We use the notation $[Br]_A$ to refer to the equivalence class of Br by $\sim_A$, that is, the set $\{X \in \text{Branches}(\phi) \mid Br \sim_A X\}$.

Definition 5. *Let ϕ be an α-TLA$^+$ expression, A an assignment strategy for ϕ and Br a branch of ϕ. Using $X = [Br]_A$ and $Y = \bigcup_{Z \in X} Z$, we define the symbolic transition generated by Br and A to be* $\text{slice}(\phi, Y)$.

Example 3. Let us look Example 2 again. The formula ϕ has two assignment strategies $A_1 = \{\ell_2\}$, and $A_2 = \{\ell_4, \ell_5\}$. If the first assignment strategy A_1 is chosen, we have that $Br_1 \cap A_1 = Br_2 \cap A_1 = \{\ell_2\}$. This implies that Br_1 and Br_2 are in the same equivalence class, or $Br_1 \sim_{A_1} Br_2$. Therefore, we have only one symbolic transition which is exactly ϕ. However, if A_2 is selected, branches Br_1 and Br_2 are not equivalent because $Br_1 \cap A_2 = \{\ell_4\}$ and $Br_2 \cap A_2 = \{\ell_5\}$. Therefore, we have two symbolic transitions:

$$T_1 = \ell_1 :: [[\ell_2 :: x' \in \star] \wedge [\ell_3 :: [[\ell_4 :: x' \in \star] \vee \ell_5 :: \text{FALSE}]]]$$
$$T_2 = \ell_1 :: [[\ell_2 :: x' \in \star] \wedge [\ell_3 :: [\ell_4 :: \text{FALSE} \vee [\ell_5 :: x' \in \star]]]]$$

The first assignment strategy A_1 seems to be better than A_2 because A_1 generates fewer symbolic transitions than A_2. However, in this paper, we do not introduce any metric, by which we could compare assignment strategies. In the implementation, we use any strategy found by the SMT solver. ◁

The equivalence relation $\sim_A$ allows us to define a counterpart to Proposition 7:

Proposition 8. *Let ϕ be an α-TLA$^+$ expression. For any selection $Br_1, \ldots, Br_k$ from the branches of ϕ, the following holds: If there exists a model M of the formula $\gamma(\text{slice}(\phi, Br_1 \cup \cdots \cup Br_k))$, then M must be a model of*

$\gamma(\text{slice}(\phi, Br))$, *for some branch* $Br \in \text{Branches}(\phi)$. *Additionally, if there is an assignment strategy* A *for* ϕ, *such that* $Br_1, \ldots, Br_k$ *all belong to the same equivalence class* $[B]_A$, *then* M *must be a model of* $\gamma(\text{slice}(\phi, Br))$, *for some branch* $Br \in [B]_A$.

The following result allows us to use symbolic transitions, not individual branches:

Theorem 2. *Let* ϕ *be an* α-TLA$^+$ *expression and* A *an assignment strategy for* ϕ. *There is a model* M *of the* TLA$^+$ *formula* $\gamma(\phi)$ *if and only if there exists a* $Br \in \text{Branches}(\phi)$, *such that* M *is a model of* $\gamma(\psi)$, *where* ψ *is the symbolic transition generated by* Br *and* A.

7 Preliminary Experiments and Potential Applications

Implementation and Evaluation. Based on the theory presented in this paper, we have implemented a procedure to find assignment strategies and their corresponding symbolic transitions from TLA$^+$ specifications, or report that none exist. It uses Z3 as the background SMT solver.

We have chose specifications both from publicly available sources, e.g. EWD840 and Paxos from [1], and from a collection of algorithms we have encoded in TLA$^+$ ourselves. For each specification, we focus on the *Next* formula. We report the number of subexpressions in $\alpha(Next)$, that is, $|\,\text{Sub}(\alpha(Next))|$, the number of assignments in the strategy found by our procedure, the number of symbolic transitions computed and the total runtime. The results are presented in Table 5. Note that the results for the specification in Fig. 1 are as expected; all assignment candidates must be part of the strategy and we find two symbolic transitions corresponding to *Produce* and *Consume*. We also see that the number of symbolic transitions is generally much smaller than the number of transitions an explicit-state model checker would find, as even simple specifications, like in Fig. 1, would generate numerous transitions in explicit state model checking, but only two symbolic transitions.

Table 5. Experimental results

Specification	#Subexpressions	Size of strategy	#Symbolic transitions	Time (ms)
aba [6]	86	48	8	271
nbacg [13]	126	82	13	205
EWD840 [11]	47	16	4	25
prodcons (Fig. 1)	12	4	2	19
Paxos [17]	60	16	4	29
nbac [25]	47	15	14	26
bcastFolklore [7]	41	17	4	28

Applications. We illustrate an application of our technique for bounded model checking [4] by the means of the example in Fig. 3. In this example, three processes pass a unique token in one direction, with the goal of computing the largest process identifier.

Our technique extracts three symbolic transitions T_1, T_2, and T_3, each T_i being equivalent to $P(i) \wedge id' = id$ for $1 \leq i \leq 3$. As common in bounded model checking, with $[\![F]\!]_{i,i+1}$ we denote the SMT encoding of a transition by action F from an ith to an $(i+1)$-th state. For instance, $[\![Next]\!]_{0,1}$ and $[\![T_3]\!]_{0,1}$ encode the transitions from the state 0 to the state 1 by *Next* and T_3. Likewise, $[\![Init]\!]_0$ encodes SMT constraints by *Init* on the initial states. One can use the SMT encodings introduced in [20,21].

MODULE *max*

EXTENDS *Naturals*
VARIABLE tok, max, id
$Init \triangleq tok = 1 \wedge id \in [1..3 \to Nat] \wedge max = 0$
$P(i) \triangleq tok = i \wedge tok' = 1 + i \,\%\, 3 \wedge max' = \text{IF } id[i] > max \text{ THEN } id[i] \text{ ELSE } max$
$Next \triangleq (P(1) \vee P(2) \vee P(3)) \wedge id' = id$

Fig. 3. A distributed maximum computation in a ring of three processes in TLA$^+$

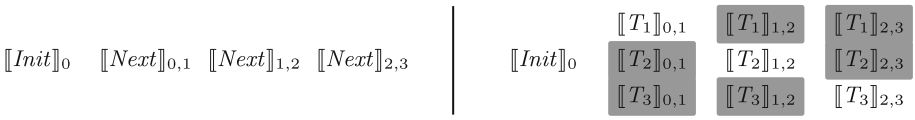


Fig. 4. SMT formulas that are constructed when checking the executions up to length 4: using the action *Next* (left), and using symbolic transitions (right). The gray formulas are excluded from the SMT context during the exploration.

Figure 4 shows the SMT formulas that are constructed by a bounded model checker when exploring executions up to length 4. (For the sake of space, we omit the formulas that check property violation.) On one hand, the monolithic encoding that uses only *Next* has to keep all the formulas in the SMT context. On the other hand, by incrementally checking satisfiability of the SMT context, the model checker can discover that some formulas — for example, $[\![T_2]\!]_{0,1}$ and $[\![T_3]\!]_{1,2}$ — lead to unsatisfiability and prune them from the SMT context. Similar approach improves efficiency of bounded model checking `C` programs [5, Chap. 16], hence, we expect it to be effective for the verification of TLA$^+$ specifications too.

8 Conclusions

We have introduced a technique to compute symbolic transitions of a TLA$^+$ specification by finding expressions that can be interpreted as assignments.

Importantly, we designed the technique with soundness in mind. Detailed proofs can be found in the report [16]. We believe that our results can be used as a first preprocessing step when developing a symbolic model checker or a type checker for TLA$^+$.

As in the case of TLC, one can find TLA$^+$ specifications, for which no assignment strategy exists. However, TLA$^+$ users are systematically checking their specifications with TLC, in order to find simple errors. Hence, most of the benchmarks already come in a form compatible with TLC. Thus, we expect our approach to also work in practice. Based on these ideas, we are currently developing a symbolic model checker for TLA$^+$.

Acknowledgments. We are grateful to Stephan Merz for insightful discussions on semantics of TLA$^+$. We thank anonymous reviewers for their helpful suggestions.

References

1. A collection of TLA$^+$ specifications. https://github.com/tlaplus/Examples/. Accessed 21 Oct 2017
2. Azmy, N., Merz, S., Weidenbach, C.: A rigorous correctness proof for pastry. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) ABZ 2016. LNCS, vol. 9675, pp. 86–101. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33600-8_5
3. Beyer, D., Keremoglu, M.E.: CPAchecker: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_16
4. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49059-0_14
5. Biere, A., Heule, M., van Maaren, H.: Handbook of Satisfiability, vol. 185. IOS press, Amsterdam (2009)
6. Bracha, G., Toueg, S.: Asynchronous consensus and broadcast protocols. J. ACM **32**(4), 824–840 (1985)
7. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM **43**(2), 225–267 (1996)
8. Chaudhuri, K., Doligez, D., Lamport, L., Merz, S.: The TLA$^+$ proof system: building a heterogeneous verification platform. In: Cavalcanti, A., Deharbe, D., Gaudel, M.-C., Woodcock, J. (eds.) ICTAC 2010. LNCS, vol. 6255, p. 44. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14808-8_3
9. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003)
10. Conchon, S., Goel, A., Krstić, S., Mebsout, A., Zaïdi, F.: Cubicle: a parallel SMT-based model checker for parameterized systems. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 718–724. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_55
11. Dijkstra, E.W., Feijen, W.H., Van Gasteren, A.M.: Derivation of a termination detection algorithm for distributed computations. In: Broy, M. (ed.) Control Flow and Data Flow: concepts of distributed programming, pp. 507–512. Springer, Heidelberg (1986). https://doi.org/10.1007/978-3-642-82921-5_13

12. Gafni, E., Lamport, L.: Disk paxos. Distrib. Comput. **16**(1), 1–20 (2003)
13. Guerraoui, R.: On the hardness of failure-sensitive agreement problems. Inf. Process. Lett. **79**(2), 99–104 (2001)
14. Konnov, I., Lazić, M., Veith, H., Widder, J.: A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In: POPL, pp. 719–734 (2017)
15. Kroening, D., Tautschnig, M.: CBMC – C bounded model checker. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 389–391. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_26
16. Kukovec, J., Tran, T.H., Konnov, I.: Extracting symbolic transitions from TLA$^+$ specifications (technical report 2018). http://forsyte.at/wp-content/uploads/abz2018_full.pdf. Accessed 7 Feb 2018
17. Lamport, L.: The part-time parliament. ACM TCS **16**(2), 133–169 (1998)
18. Lamport, L.: Specifying systems: the TLA$^+$ language and tools for hardware and software engineers. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
19. Merz, S.: The specification language TLA$^+$. In: Bjørner, D., Henson, M.C. (eds.) Logics of specification languages. Monographs in Theoretical Computer Science (An EATCS Series), pp. 401–451. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-74107-7_8
20. Merz, S., Vanzetto, H.: Automatic verification of TLA$^+$ proof obligations with SMT solvers. In: Bjørner, N., Voronkov, A. (eds.) LPAR 2012. LNCS, vol. 7180, pp. 289–303. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28717-6_23
21. Merz, S., Vanzetto, H.: Harnessing SMT solvers for TLA$^+$ proofs. ECEASST, **53** (2012). https://hal.inria.fr/hal-00760579
22. Moraru, I., Andersen, D.G., Kaminsky, M.: There is more consensus in Egalitarian parliaments. In: SOSP, pp. 358–372. ACM (2013)
23. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon web services uses formal methods. Commun. ACM **58**(4), 66–73 (2015)
24. Ongaro, D.: Consensus: bridging theory and practice. Ph.D. thesis, Stanford University (2014)
25. Raynal, M.: A case study of agreement problems in distributed systems: non-blocking atomic commitment. In: HASE, pp. 209–214 (1997)
26. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA$^+$ specifications. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 54–66. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48153-2_6

Systematic Generation of Non-equivalent Expressions for Relational Algebra

Kaiyuan Wang[1(✉)], Allison Sullivan[1], Manos Koukoutos[2], Darko Marinov[3], and Sarfraz Khurshid[1]

[1] University of Texas at Austin, Austin, USA
{kaiyuanw,allisonksullivan,khurshid}@utexas.edu
[2] École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
emmanouil.koukoutos@epfl.ch
[3] University of Illinois at Urbana-Champaign, Urbana, USA
marinov@illinois.edu

Abstract. Relational algebra forms the semantic foundation in multiple domains, e.g., Alloy models, OCL constraints, UML metamodels, and SQL queries. Synthesis and repair techniques in such domains require an efficient procedure to generate (non-equivalent) expressions subject to relational constraints, e.g., the types of sets and relations, their cardinality, size of expressions, maximum arity of the intermediate expressions, etc. This paper introduces the first generator for relational expressions that are non-equivalent with respect to the semantics of relational algebra. We present the algorithms that define our generator, its embodiment based on the Alloy tool-set, and an experimental evaluation to show the effectiveness of its non-equivalent generation for a variety of problems with relational constraints.

1 Introduction

Relational algebra forms the semantic foundation in multiple domains, e.g., Alloy models [16], OCL constraints [39], UML metamodels [42], and SQL queries [25]. Developing program synthesis [6,14,22,36,37,46] or program repair [10,20,23,24,40,57] methods in such domains requires an efficient technique to generate (non-equivalent) expressions subject to relational constraints, e.g., the types of sets and relations, their cardinality, size of expressions, maximum arity of the intermediate expressions, etc.

While syntactically different expressions can be generated by simple standard bottom-up [54] or top-down [6] grammar-based generation techniques, generating expressions that way can produce an infeasibly large number of expressions even for relatively small expression sizes. (In this paper, we measure the size of an expression by the number of AST nodes in that expression.) For example, for just one binary relation $r \subseteq S \times S$ on some set S and expression size up to 7, there are 17109 syntactically different expressions that can be built using the operators from Alloy [16]: 5 standard binary operators (relational join, Cartesian

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 105–120, 2018.
https://doi.org/10.1007/978-3-319-91271-4_8

product, set union, intersection, difference) and 3 unary operators (transpose, transitive closure, reflexive transitive closure). Reducing this number of expressions requires reasoning about semantic equivalences in relational algebra. Some such equivalences are well-known, e.g., associativity and commutativity (AC) for set union and intersection. While AC rules reduce the number of expressions to 11191 in this example, there are actually only 771 non-equivalent expressions. We need more advanced equivalences to prune out equivalent expressions.

We introduce RexGen, the first generator for semantically non-equivalent relational expressions. We present the algorithms that define our generator, its embodiment based on the Alloy tool-set, and an experimental evaluation to show the effectiveness of its non-equivalent generation for a variety of problems with relational constraints.

Our choice of Alloy is driven by its foundation in relational, first-order logic [16], its focus on analyzability, and its wide application in various domains, e.g., software design [7,17], analysis [4,9,18], testing [31], and security [21]. Alloy's tool-set includes an automatic analysis tool [53] for checking the satisfiability of formulas written in Alloy using off-the-shelf propositional satisfiability (SAT) solvers. The analyzer performs *scope-bounded* analysis, which checks the properties within a given *scope*, i.e., bound on the universe of discourse. While the Alloy analyzer could be used to check semantic equivalences of all expressions during generation, it results in an impractically slow generation.

Our key insight is that the Alloy analyzer enables a systematic method for creating and evolving an *optimized* generator for non-equivalent relational expressions. Our method first uses the analyzer (with its expensive, semantic equivalence checks) to discover likely equivalences of expressions that already get generated. We then generalize and validate these likely equivalences using manual reasoning, and incorporate them in the expression generator as *equivalence rules*. These rules directly prune equivalent expressions based on quick, (mostly) *syntactic* checks without expensive, semantic equivalence checks.

RexGen offers three automatic pruning modes for bottom-up generation of relational expressions. One mode, *static* pruning, directly prunes from generation many equivalent expressions based on a fixed suite of equivalence rules, which include well-known equivalences and also dozens more that we discovered using the Alloy analyzer. Another mode, *dynamic* pruning, uses the analyzer during generation to prune equivalent expressions incrementally by comparing each new expression to a representative from each equivalence class formed thus far, while forming new equivalence classes as needed. The third mode, *modulo-instance* pruning, allows the user to provide AUnit *test* valuations [50,51], and prunes an expression if it is equivalent to some generated expression with respect to all given test valuations (even if not equivalent for some other valuations [3]).

We perform an experimental evaluation of RexGen using expression generation problems derived from 12 Alloy models. We evaluate the number of expressions that RexGen generates and the time that RexGen takes to generate those expressions for each problem under different settings. The experimental results show that static pruning offers the best trade-off, creating mostly *semantically*

different expressions, substantially reducing the number of expressions from simple grammar-based generation, while not increasing the generation time—in fact, often having smaller generation time than not using any equivalence pruning rules. In comparison, using only AC rules, as done by some state-of-the-art systems for expression generation [23] (albeit not for relational expressions, so we added appropriate extensions for comparison), generates a larger number of expressions, while not substantially reducing the time. Using dynamic pruning removes all equivalent expressions w.r.t. the scope but takes substantially more time. Finally, pruning equivalences based on a relatively small but diverse suite of test valuations works similarly to dynamic pruning.

This paper makes the following contributions:

Problem: We are the first to study the problem of expression generation for relational algebra.

Optimizations: We introduce a suite of equivalence pruning rules for relational expressions to improve the efficacy of expression generation.

Experiments: We present an experimental evaluation based on problems derived from 12 Alloy models; the results show that RexGen with static pruning offers a promising approach for generating non-equivalent relational expressions.

2 Example

We next present an example model to motivate relational expression generation and introduce the basic concepts of our approach. Consider this small but illustrative Alloy model of directed trees, adapted from a recent paper [32]:

```
sig Node { edges: set Node }
pred Acyclic { no iden & ^edges }
pred Injective { edges.~edges in iden }
pred Connected { (Node -> Node) in ^(edges + ~edges) }
pred isDirectedTree { Acyclic and Injective and Connected }
run isDirectedTree for 4 Node
```

The model declares a set (called *signature* in Alloy) of nodes with a *field* called `edges` that is a binary relation of type `Node`×`Node`. The keyword `set` declares an arbitrary relation; Alloy also has keywords `one` and `lone` to constrain the relation to be a total or partial function, respectively. The *predicate* (`pred`) is a named formula that can be invoked elsewhere. The conjunction of `Acyclic`, `Injective`, and `Connected` would precisely represent directed trees. The binary operator `&` is set intersection; `+` is set union; `in` is subset; `.` is relational join (and relational image); and `->` is Cartesian product. The (prefix) unary operator `^` is transitive closure, and `~` is transpose; Alloy also has *reflexive* transitive closure (`*`). The keyword `iden` represents the identity relation. The formula `no E` for expression `E` constrains `E` to be the empty set. The `run` command runs a given formula, and presents an instance of the given formula if the formula is

satisfiable. The scope of 4 instructs the analyzer to create an instance with at most 4 nodes.

To illustrate expression generation using our approach, consider the signature declaration in this model, which introduces one set (`Node`) and one binary relation (`edges`). Given those declarations, a user may want to generate various expressions, e.g., in synthesis or repair tasks. For example, many Alloy beginners write `pred Acyclic' { all n: Node | n !in n.~edges }` and may want to know if there is a semantically equivalent formula without any quantified variables (as in `Acyclic`). In that case, the user may want to systematically try {`UO E`} where `UO` represents any unary operator (`no`, `some`, `lone`, `one`) and `E` represents any valid expression, such that the formula {`UO E`} is equivalent to `Acyclic'`.

Assume we set the maximum size of any generated expression to 5, which suffices to generate even the largest relational expressions in this particular model. RexGen generates 581 expressions with no pruning, 438 with AC pruning (i.e., associativity and commutativity), 116 with static pruning, 105 with dynamic pruning, and 102 with modulo-instance pruning (for 14 tests). The generation time is largest for dynamic pruning, which uses Alloy analyzer to check each equivalence and takes 2.8 s; in all other cases, no constraint solving is used, and the generation time is <1 s. The following shows some of the equivalences discovered with dynamic pruning (where `univ` denotes the universe of discourse, which is equal to `Node` in the example model):

```
Node->Node = univ->univ                | (~edges)&(^edges) = (~edges)&(*edges)
(~edges).Node = Node.edges             | *((^edges)-edges) = *((*edges)-edges)
edges.(Node.edges) = edges.Node        | ^(edges.(^edges)) = edges.(^edges)
```

To illustrate generation of larger expressions, consider size 7. RexGen generates 17109 expressions with no pruning, 11191 with AC pruning, 1464 with static pruning, 771 with dynamic pruning, and 691 with modulo-instance pruning (for 14 tests). The generation time for dynamic pruning increases to 82.3 s, for modulo-instance pruning increases to 1.7 s, and for the other techniques remains <1 s. Thus, for this example, static pruning reduces the number of expressions by 86.9% over AC pruning while taking a similar amount of time; dynamic pruning reduces the number by 47.3% over static pruning but takes much longer due to many SAT calls. Moreover, modulo-instance pruning creates a similar number of expressions as dynamic pruning, which indicates the diversity of the tests, but takes less time due to not making SAT calls.

3 RexGen Framework

We next present our ***Re****lational* ***Ex****pression* ***Gen****erator* (RexGen) approach for generating non-equivalent relational expressions. We first describe the technique input and then the expression generation techniques.

3.1 Technique Input

RexGen takes as input (1) a number of sets (signatures), relations (fields), and variables declared in an Alloy model (in the context in which the expressions

should be generated), (2) a limit on the size of generated expressions, (3) optionally a target arity of expressions to generate, and (4) optionally a number of test valuations, i.e., values for the input sets and relations (but not for the bound variables). RexGen generates expressions using the following grammar:

$$\begin{aligned} expr &::= expr\ binOp\ expr \mid expr^* \mid expr^+ \mid expr^{-1} \mid terminal \\ binOp &::= \cup \mid \cap \mid \setminus \mid \times \mid \bowtie \\ terminal &::= set \mid relation \mid variable \end{aligned}$$

The grammar captures a subset of syntactically possible Alloy expressions, which cover a large space of candidate expressions likely to be intended by Alloy users. For example, we do not consider rarely used Alloy operators such as domain restriction (`<:`). We use standard notation of relational algebra: $\cup$ is set union, $\cap$ is set intersection, $\setminus$ is set difference, $\times$ is Cartesian product, $\bowtie$ is the relational join; e^*, e^+, e^{-1} denote the reflexive transitive closure, transitive closure, and transpose of e, respectively. Additionally we use the empty set $\emptyset$, the universal set $univ$, and the identity $iden = \{(x, x) | x \in univ\}$.

To systematically generate expressions, RexGen limits: (1) the size of expressions and (2) the maximum arity of expressions. There are different ways to define expression size; we consider the number of AST nodes in the expression: $size(terminal) = 1, size(e_1\ binOp\ e_2) = size(e_1) + size(e_2) + 1, size(expr^{unOp}) = size(expr) + 1$.

3.2 Generating Expressions

We next describe how RexGen enumerates expressions within the given limits. In the spirit of synthesis tools [3], enumeration works bottom-up, starting from *terminal* expressions (sets, relations, and variables given as inputs) and then iteratively combining smaller expressions to generate larger ones.

Our key contribution is pruning that aggressively removes expressions to increase the efficiency of the generation and/or reduce the number of generated expressions. The goal of pruning is to eliminate expressions that are semantically equivalent with previously generated expressions. Pruning has three modes: *static*, *dynamic*, and *modulo* pruning.

Expression Generation Algorithm. The generation algorithm maintains a list of expressions, $exprs[arity]$, indexed by the arity. The list maintains a total order among expressions of the same arity; we use $ind(e)$ to denote the index of the expression e in the list, and some pruning rules use this index.

The lists are instantiated with the terminal expressions (i.e., sets, relations, and variables declared in the model), based on their arity. The size of these expressions is 1. Then, until a limit is reached, the algorithm iteratively increases size and combines every operator and every combination of expressions of appropriate smaller sizes to generate expressions of the larger size. Each generated expression is then added to $exprs$ if it is (1) within the limits given for the generation, (2) well typed in Alloy, and (3) not pruned by the current pruning mode.

Note that, by construction, expressions in *exprs* are syntactically different. The rest of this section explains in detail well typedness and the three pruning modes.

Well Typedness. RexGen tracks type information for generated expressions, typically using the default Alloy type system, which includes subset/subtyping and union types [16]. However, for some expressions, RexGen tracks a more precise type than the default type system. The main reason is the semantics of reflexive transitive closure (*). In Alloy, reflexive transitive closure is a superset of the identity relation for the union of all sets ($univ$) and thus has type $univ \times univ$. For example, if a model has two sets, $Node$ and $Value$, and a relation, $edges$, of type $Node \times Node$, then $edges^*$ is not of type $Node \times Node$ but $univ \times univ$, where $univ = Node \cup Value$. However, this type is too broad; it allows for arbitrary applications of other operators and makes expression generation intractable, producing expressions that are not intended in practical use.

For example, consider the expression $a^* + b$, where a has type $A \times A$. Intuitively, we want to allow only expressions of type $A \times A$ for b; however, we cannot track this precisely if we allow a^* to have type $univ \times univ$. On the other hand, we cannot consider a^* to have type $A \times A$ because that would make a^* a subset of $A \times A$, causing the static pruning to incorrectly prune expressions like $a^* + A \times A$. Therefore, RexGen conceptually uses a special type system to type intermediate generated expressions, but uses Alloy type for static pruning.

Static Pruning. Static pruning removes expressions that are known to be semantically equivalent with other generated expressions. This pruning considers equivalence with respect to *all* possible valuations not only given test valuations. To prune equivalent expressions, we derive a comprehensive suite of equivalence rules specific to relational algebra. Other generation systems [36] use similar pruning rules for other domains, but our work is the first to provide rules specific to relational algebra.

Table 1 presents the static pruning rules of RexGen. The first column gives the pattern of equivalent expressions that the rule intends to eliminate. RexGen prunes the expression whose syntactic shape is the left-hand side of the equivalence. The second column specifies the condition for pruning. Note that almost all rules use only syntactic information or type (and arity) information for the involved expressions, which makes the rules easily checkable. An exception are a few rules that check the subset property between two sets/relations; because subset is a semantic property and not easily checkable, we approximate it conservatively, as shown in Table 2. Another exception is the rule for commutativity. To avoid generating both $a\ op\ b$ and $b\ op\ a$, where op is a commutative operation, we use the total order defined for each arity by $exprs$: we prune the expression with $ind(a) > ind(b)$, where $ind(e)$ is the index of e in the list $exprs$.

Dynamic Pruning. Dynamic pruning removes equivalent expressions by using the Alloy analyzer to check whether an expression is equivalent to another one already generated. Unlike static pruning, dynamic pruning considers (1) all signature/field constraints (e.g., that a relation must be a function) and (2) bound variables in the scope of the generated expression. To our knowledge, no previous

Table 1. Static pruning rules

Equivalence (lhs = rhs)	Condition if needed; otherwise `true`
$a\ op\ (b\ op\ c) = (a\ op\ b)\ op\ c$	op associative
$a\ op\ b = b\ op\ a$	op commutative and $ind(a) > ind(b)$
$a \cup b = b$ and $b \cup a = b$	$[\![a]\!] \subseteq [\![b]\!]$
– Similar for $\cap$ and $\supseteq$	
$a \setminus b = \emptyset$	$[\![a]\!] \subseteq [\![b]\!]$
$a \cup b = c \cup b$	$\exists c.a \cong c \cup b$ or $a \cong b \cup c$ or $a \cong c \setminus b$
$a \cup b = b$	$\exists c.a \cong c \cap b$ or $a \cong b \cap c$ or $a \cong b \setminus c$
– Also symmetrically	– where $\cong$ is syntactic pattern matching
$(a\ op_1\ b)\ op_2\ (a\ op_1\ c) = a\ op_1\ (b\ op_2\ c)$	$op_1 \in \{\bowtie, \times, \cap\}, op_2 \in \{\cup, \cap\}$
– Similar for $(a\ op_1\ b)\ op_2\ (c\ op_1\ b)$	
$a^{-1}\ op\ b^{-1} = (a\ op\ b)^{-1}$	$op \in \{\cup, \cap, \setminus, \bowtie\}$
$\bigcup e_i = \bigcup_{i \neq j} e_i$,	$e_j \cong e_k$ for some $j \neq k$
– Similar for $\bigcap$	
$a \setminus (b \cup c) = (a \setminus b) \setminus c$	
$a \setminus (a \cap b) = a \setminus b$	
– Similar for $a \setminus (b \cap a)$	
$a \setminus (a \setminus b) = a \cap b$	
$a \setminus (b \setminus a) = a$	
$(a \cup b) \setminus a = b \setminus a$	
$(a\ op\ b) \setminus (a\ op\ c) = a\ op\ (b \setminus c)$	$op \in \{\times, \cap\}$
$(a \cap b) \setminus c = a \cap (b \setminus c)$	
$a \bowtie (a \times b) = b$	$card(a) \geq 1$
– Similar for $(b \times a) \bowtie a$	
$a \bowtie b^{-1} = b \bowtie a$	$arity(a) = 1$
$A \bowtie b^* = A$	$b : A \times A$
– Similar for $b^* \bowtie A$	– where $b : A \times A$ means that b has type $A \times A$
$A \bowtie b^+ = A \bowtie b$	$b : A \times A$
$b^+ \bowtie A = b \bowtie A$	$b : A \times A$
$b \bowtie b^* = b^+$	
– Similar for $b^* \bowtie b$	
$a^{*+} = a^*$	
– Similar for a^{+*}	
$a^{-1-1} = a$	
$a^{*-1} = a^{-1*}$	
$a^{+-1} = a^{-1+}$	
$(a\ op\ b^{-1})^{-1} = a^{-1}\ op\ b$	$op \in \{\cup, \cap, \setminus, \bowtie\}$
$(a \times b)^+ = a \times b$	
$a \bowtie (b \times c) = (a \bowtie b) \times c$	$arity(a) + arity(b) > 2$
– Similar for $(a \times b) \bowtie c$	
$b^{-1} \bowtie a = a \bowtie b$	$arity(a) = 1$
$a^+ \bowtie a = a \bowtie a^+$	
$a^* \bowtie a^* = a^*$	
$a^* \bowtie a^+ = a^+$	
– Similar for $a^+ \bowtie a^*$	
$a^+ \bowtie a^+ = a \bowtie a^+$	
$(a \setminus b) \bowtie (b \times c) = \emptyset$	
– Also symmetrically	
$a \bowtie ((b \setminus a) \times c) = \emptyset$	
– Also symmetrically	
$A \bowtie (A \times b) = b$	$arity(A) = 1$
– Similar for $(b \times A) \bowtie A$	$b : B_1 \times \ldots \times A \times \ldots \times B_n$ for some $B_i = A$

Table 2. Syntactic approximation for $a \subseteq b$. $\cong$ means syntactic match.

1. $b \cong A$, $a : A$	5. $a \cong b \setminus c$
2. $a \cong b$	6. $a \cong c^+, b \cong c^*$
3. $b \cong a \cup c$ or $b \cong c \cup a$	7. $a \cong c \bowtie c \bowtie \ldots \bowtie c, b \cong c^+$ or $b \cong c^*$
4. $a \cong b \cap c$ or $a \cong c \cap b$	8. $a \cong c \times c$, $b \cong d^*$, a has cardinality 1, c has arity 1

work handles variables locally bound by a quantifier in the scope of the generated expression.

For a new expression, E, and a previously generated expression, E', RexGen creates a new Alloy model that includes all signature/field declarations from the RexGen input plus `check { all` v_1`:` D_1 `| ... | all` v_n`:` D_n `|` E `=` E' `}`, where $v_1 \ldots v_n$ are variables used in the two expressions (except for sigs/fields from the model) and $D_1 \ldots D_n$ are their corresponding domains. For example, if E is `n. ~edges` and E' is `Node.*edges`, then the equivalence checking command is `check { all n: Node | n. ~edges = Node.*edges }`. This check is issued for every previously generated expression in *exprs* until either the new expression is found equivalent to some previously generated one, or the new expression is found not equivalent to any previously generated one and is thus added to *exprs*. Dynamic pruning can be applied to all expressions for every arity or only expressions of the target arity.

Modulo Pruning. Modulo pruning [54] removes equivalent expressions based on their values for the user-given valuations of the input test suite. Specifically, modulo pruning builds equivalence classes of expressions by grouping together all expressions that *evaluate* to the same value across all test valuations, and keeping only one expression per equivalence class.

Modulo pruning determines an expression's equivalence class without constraint *solving*, by utilizing the `Evaluator` feature of the Alloy Analyzer to perform constraint *checking*. The `Evaluator` takes as input an Alloy instance and an Alloy expression, and returns the concrete value of the expression for the given instance. For a new expression E, modulo pruning evaluates E for every test valuation in the suite, building a map of E's concrete values. If E contains any free variable(s), modulo pruning evaluates E for each element in the variable's domain, or more generally, for the cross product of domain elements if E contains multiple variables. If E's concrete-value map matches a previous expression, then E is pruned out; otherwise, E is kept. Modulo pruning only determines equivalence based on the user-given test suite, not guaranteeing equivalence across all instances in scope as dynamic pruning does.

4 Experimental Evaluation

We next present our experimental evaluation of RexGen. We use 12 diverse Alloy models for evaluation (Sect. 4.1). We evaluate the number of expressions RexGen generates and the time it takes for each model under different settings (Sect. 4.2).

4.1 Evaluation Models

We evaluate RexGen using 12 models comprised of a wide variety of example, educational, and "real-world" specifications. Address book (*addr*), Dijkstra mutual exclusion algorithm (*dijkstra*), farmer crossing-river puzzle (*farmer*), Halmos handshake problem (*hshake*), and genealogy (*gene*) are from the Alloy's distribution examples. Bad employee (*bempl*), colored tree (*ctree*), directed tree (*dtree*), and grade book (*grade*) are Alloy translations of access-control specifications used to evaluate existing scenario-finding work [32,43]. Binary tree (*btree*) constrains the graph to be a binary tree. Propositional resolution (*resfm*) is from Torlak et al. [52]. Singly linked list (*sll*) models acyclic lists.

Table 3 shows the basic information of these models. *Model* is the name. *#AST* is the number of AST nodes in each model. *#Sig* is the number of signatures declared in each model. *#Rel* is the number of relations declared in each model. For each model, we find all identifiers in scope, including signatures, relations, and bound variables, for the *largest* expression (w.r.t. our measure of size). *#Var* is the number of all identifiers in scope to generate expressions. In our experiment, we first find the expression with the largest size in each model and then use all sigs, relations, and variables in the scope of that expression to generate more expressions. *#PrimVar* is the number of primary variables when we run an empty command (`run {}`) without test-specific constraints; it represents the basic complexity of signature declarations and constraints that always hold in each model. *#Test* is the number of tests; we use the same number of tests for each model so that the results do not depend on the number of tests. We chose the number of tests based on the *sll* model, where we create tests such that modulo pruning generates the same number of expressions of size 4 as dynamic pruning for this model. We iteratively add tests until modulo pruning and dynamic pruning create the same set of expressions. In the end, we obtain 14 tests for *sll* and use the same number of tests for other models.

Our experiments are performed on a MacBook Pro running OS X El Capitan with 2.5 GHz Intel Core i7-4870HQ and 16 GB of RAM.

4.2 RexGen Results

Table 4 shows the performance of RexGen across different expression pruning environments: *No Pr.* uses no pruning rules, *AC Pr.* uses just associativity and commutativity pruning rules, *Static Pr.* uses all static pruning rules, *Dynamic Pr.* uses dynamic pruning, and *Modulo Pr.* uses modulo-instance pruning. Note that both dynamic pruning and modulo-instance pruning are applied on expressions after they are pruned by static pruning. Column *Problem* shows the Alloy model and the corresponding size used for generation. For each pruning environment, *#expr* shows the number of expressions generated and *time* shows the time duration in milliseconds to generate all expressions, with a time-out of one hour. The number of generated expressions shown in the table is for expressions of all arities up to 3.

Table 3. Basic information of models used to evaluate RexGen

Model	#AST	#Sig	#Rel	#Var	#PrimVar	#Test
addr	114	4	2	8	45	14
bempl	46	6	3	11	38	14
btree	53	2	2	6	24	14
ctree	71	4	2	8	18	14
dijkstra	385	3	1	10	57	14
dtree	49	1	1	2	12	14
farmer	169	6	3	14	24	14
gene	139	5	2	8	20	14
grade	64	5	4	11	48	14
hshake	127	3	2	6	19	14
resfm	285	8	7	19	101	14
sll	33	2	2	5	15	14

Expression generation using *No Pr.*, *AC Pr.*, and *Static Pr.* is fast, taking at most 7.9 s (*farmer* and size 7 using *No Pr.*), but frequently finishing in under a second. Accordingly, both *AC Pr.* and *Static Pr.* have negligible overhead. However, the number of expressions generated can vary greatly, as seen in Table 4. *No Pr.* generates all possible expressions and provides a means of measuring the effectiveness of different pruning environments. Compared to *No Pr.*, *AC Pr.* reduces the number of expressions generated by 8.7–54.7%, while *Static Pr.* reduces the number of expressions generated by 36.6–91.4%. Compared directly, *Static Pr.* generates 28.4–86.9% fewer expressions than *AC Pr.*. In other words, *Static Pr.*'s additional pruning rules highlight that associativity and commutativity are not strong enough to prune relational expressions on their own. Moreover, Table 4 shows that the pruning rules for *AC Pr.* and *Static Pr.* reduce the space of possible expressions by a large enough degree that both techniques often finish faster than *No Pr.*, despite the time they spend on applying equivalence rules to check expressions. Although *Static Pr.* has 40 more rules than *AC Pr.*, the difference in runtime between *AC Pr.* and *Static Pr.* is often less than a second. Therefore, *Static Pr.*'s rules are inexpensive to run but effective at reducing the number of generated expressions.

We can analyze expressions to prune out more equivalences. *Dynamic Pr.* further prunes expressions generated by *Static Pr.*; *Dynamic Pr.* is motivated by using Alloy to find all equivalences (within a given scope), thus capturing equivalences which cannot be captured by generic static pruning rules. As expected, *Dynamic Pr.* reduces the number of expressions from *Static Pr.*, by 3.6–71.4%. *Dynamic Pr.* gives the minimum number of non-equivalent expressions for each model, showing the lower bound of what *Static Pr.* could achieve.

Table 4. RexGen performance. Times are in ms. ⊥ indicates a timeout (>1 hour).

Problem		No Pr.		AC Pr.		Static Pr.		Dynamic Pr.		Modulo Pr.	
		#expr	time	#expr	time	#expr	time	#expr	time	#expr	time
addr	4	231	2	199	4	129	25	118	1259	108	279
	5	3984	18	2374	19	1335	56	823	15869	600	1396
	6	7913	27	5563	29	2034	64	1193	19359	900	1879
	7	139971	204	65346	131	24839	189	7116	635296	3546	7902
bempl	4	427	5	377	6	261	29	246	1939	237	343
	5	7027	27	4369	25	2463	64	1708	25098	1424	2999
	6	15396	50	11144	41	4096	80	2588	43840	2198	3814
	7	254843	363	128706	274	47747	296	15363	1555983	10309	29174
btree	4	415	4	355	6	223	29	215	6247	196	484
	5	3264	18	2391	18	1032	51	915	62153	740	1920
	6	17956	42	12919	41	4553	93	3424	999892	2227	6221
	7	139882	204	88578	148	25031	195	⊥	⊥	8505	26140
ctree	4	369	4	327	6	202	27	185	2773	144	754
	5	4625	21	3031	21	1446	59	996	28674	737	5282
	6	14315	37	10707	38	3473	79	2143	192314	1169	9584
	7	168181	221	93805	175	27660	213	⊥	⊥	5530	60968
dijkstra	4	287	2	251	4	140	26	135	2235	133	264
	5	4661	19	2763	20	1397	53	1097	20544	1069	2185
	6	9939	30	7159	39	2175	60	1637	36446	1552	3083
	7	138703	213	65991	139	17976	180	7007	670275	5704	17820
dtree	4	111	1	95	3	40	21	38	680	37	144
	5	581	4	438	6	116	30	105	2809	102	401
	6	2957	15	2130	14	376	39	250	11247	234	841
	7	17109	40	11191	34	1464	61	771	82268	691	1686
farmer	4	1077	8	939	8	654	39	619	28327	454	695
	5	41007	73	24322	53	16969	141	⊥	⊥	5116	8992
	6	96607	140	68468	124	33097	215	⊥	⊥	9247	22555
	7	3666499	7942	1661501	4581	923952	3985	⊥	⊥	80553	2156722
gene	4	641	5	551	6	376	32	348	10853	242	916
	5	12055	31	7653	29	4597	83	3228	632913	1675	8614
	6	42897	76	30703	64	14621	145	⊥	⊥	4324	20490
	7	763031	1998	393015	1150	177920	665	⊥	⊥	26222	326804
grade	4	421	4	373	6	267	30	244	2570	229	447
	5	6533	25	4168	25	2342	65	1496	28995	1105	2450
	6	14930	45	11033	42	4141	89	2321	52542	1740	3446
	7	234482	373	122012	258	45312	311	13166	1858565	7300	19416
hshake	4	471	3	403	5	260	31	244	8543	173	1131
	5	5625	20	3805	22	1936	61	1505	164478	1020	8180
	6	25523	51	18319	46	7640	112	5378	2570827	3031	21775
	7	286661	355	163874	247	58505	318	⊥	⊥	16149	222019
resfm	4	1030	8	940	12	652	38	625	10051	510	767
	5	18692	50	12250	42	7337	99	5705	336197	3626	5634
	6	47128	111	35984	94	13384	146	9406	938031	5026	9076
	7	822434	2107	449935	653	175337	845	⊥	⊥	24997	181425
sll	4	209	2	183	4	104	25	98	1468	98	283
	5	1549	10	1100	13	397	40	331	6868	330	987
	6	6267	25	4694	25	1203	58	808	37988	803	2593
	7	45527	86	28622	64	5463	95	2712	429769	2671	9391

Modulo Pr. also filters expressions generated by *Static Pr.*; specifically, *Modulo Pr.* reduces the expressions from *Static Pr.* by 5.0–91.3%. *Dynamic Pr.* can be viewed as *Modulo Pr.* if the input test suite covered all instances in scope. However, since we use only 14 tests per model for *Modulo Pr.*, *Modulo Pr.* may even prune expressions that are semantically non-equivalent up to a given scope but equivalent over all 14 tests. For example, *Modulo Pr.* prunes 10 more size 4 expressions and 223 more size 5 expressions for *addr* compared to *Dynamic Pr.*. Therefore, as expected, *Modulo Pr.* can reduce the number of generated expressions compared to *Dynamic Pr.*, by as much as 50.2% (*addr*), or *Modulo Pr.* can generate the same number of expressions (*sll* and size 4) but 5.2× faster. The trade-off is that, while *Dynamic Pr.* is guaranteed to not prune expressions that are semantically non-equivalent within a given scope, it is slower than *Modulo Pr.*; *Dynamic Pr.* times out on 7 different problems, while *Modulo Pr.* frequently finishes in under a minute, with the longest runtime being 2156.7 s. While *Modulo Pr.* provides a practical, lighter-weight alternative to *Dynamic Pr.*, *Modulo Pr.* still has a high overhead over *Static Pr.*. For instance, for *farmer* and size 7, *Static Pr.* can generate expressions in 4.0 s, while *Modulo Pr.* needs 2156.7 s to finish.

In our experiment, applying *Dynamic Pr.* or *Modulo Pr.* on expressions generated with *No Pr.* or *AC Pr.* takes significantly longer. *Static Pr.*'s ability to significantly reduce the number of generated expressions, with a negligible overhead, makes *Static Pr.* the recommended approach for relational expression generation (even when considering more advanced pruning techniques like *Dynamic Pr.* or *Modulo Pr.*). To check that our static pruning rules are correct, we ran dynamic pruning on expressions generated using *AC Pr.* and *Static Pr.*: we found that the numbers of non-equivalent expressions generated after dynamic pruning for both *AC Pr.* and *Static Pr.* are exactly the same, which indicates that *Static Pr.* does not incorrectly prune any non-equivalent expression.

5 Related Work

Enumeration Algorithms include bottom-up enumeration [3,54], used by RexGen, and top-down enumeration [6]. EuSolver [3] has been one of the most prominent solvers in Syntax-Guided Synthesis (SyGuS) competitions. FlashMeta [38] uses version-space algebra to concisely represent a large number of programs. Neither EuSolver nor FlashMeta focus on relational expressions, which can generate a large number of equivalent expressions. Our work proposes a number of pruning rules that substantially reduce the number of equivalent expressions, thus providing basis for practical synthesis with relational expressions.

Search Space Pruning of expression generation is important because search spaces for any realistic programming language quickly become intractable. Pruning techniques include indistinguishability of expressions modulo a set of inputs [3,54] and partial evaluation of incomplete expressions [6]. Knowledge about operator properties has also been used to explore equivalent expressions, either

after expression generation [36] or by applying an automated transformation to the grammar which represents candidate programs [23]. However, most techniques have only been explored in the domains of integers, booleans, and abstract data types, all of which have less comprehensive sets of equivalence rules than our work with the domains of sets and relations.

Applications of Expression Generation are quite common. For example, program synthesis has attracted attention for a few decades [28], and researchers have applied it in a variety of domains [5,6,8,12,22,27]. Program sketching [46] is another example, which demonstrated the opportunities to apply modern solver technology to the synthesis problem, and introduced the counter-example guided inductive synthesis paradigm to program synthesis. Sketch requires the user to provide generators of expressions for expression holes [2,14,19,45]. While most work on sketching is in the context of synthesis, SketchRep [13] applies sketching to the problem of program repair [10,20,24,40,57], i.e., correcting faulty lines of code. Synthesis from examples, the inspiration behind test valuations, has also been extensively studied [1,35]. Notably, synthesis from examples has been successfully employed in commercial products [11]. EdSketch [14] introduced an optimized backtracking search for completing Java sketches using test executions for pruning. SketchFix [15] used EdSketch as the backend synthesis engine for program repair. EdSynth [58] builds on EdSketch and synthesizes method sequences for given sketches that may contain conditional branches. SyPet [5] introduced a novel use of Petri nets in synthesizing straightline sequences of method invocations for complex APIs using tests. The key enabler of all of the above applications is efficient expression generation; ours is the first work that addresses generation for relational algebra.

Alloy is a well studied lightweight modeling approach that has been applied in various domains, including software design [29,30], networking [41], and security [26,34]. This paper is the first to study expression generation for Alloy and more generally for relational algebra. Our work leverages the AUnit [48,51] approach for writing tests for Alloy models. Various approaches assist Alloy users to build their models correctly, e.g., by improving scenario exploration [32,33], supporting state modeling [7,17,18,31,49], highlighting UNSAT cores [44,52,53], and creating tests [50,55]. RexGen provides a novel basis of a synthesis or sketching engine for Alloy in particular and relational logic in general [47,56].

6 Conclusions

We introduced RexGen, the first generator for non-equivalent relational expressions. We presented a set of equivalence rules for relational expressions, used them for pruning in our generator, embodied the generator based on the Alloy tool-set, and presented an experimental evaluation of the effectiveness of our non-equivalent generation for a variety of problems with relational constraints. RexGen provides the key step to address the broader problems of synthesis and repair of declarative models in Alloy. Our companion paper on ASketch [56]

shows how to use the generated expressions to synthesize Alloy models from sketches. We hope our work inspires the development of a broader tool-set to support software models and eventually leads to more reliable software systems.

Acknowledgements. We thank Viktor Kuncak for his comments on this work. Manos Koukoutos is supported in part by the European Research Council (ERC) project Implicit Programming. This material is based upon work partially supported by the US National Science Foundation under Grant Nos. CCF-1409423, CCF-1421503, CNS-1646305, CCF-1718903, and CNS-1740916.

References

1. Albarghouthi, A., Gulwani, S., Kincaid, Z.: Recursive program synthesis. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 934–950. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_67
2. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: FMCAD (2013)
3. Alur, R., Radhakrishna, A., Udupa, A.: Scaling enumerative program synthesis via divide and conquer. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 319–336. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_18
4. Dennis, G., Chang, F.S., Jackson, D.: Modular verification of code with SAT. In: ISSTA (2006)
5. Feng, Y., Martins, R., Wang, Y., Dillig, I., Reps, T.W.: Component-based synthesis for complex APIs. In: POPL (2017)
6. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. In: PLDI (2015)
7. Frias, M.F., Galeotti, J.P., Pombo, C.G.L., Aguirre, N.M.: DynAlloy: upgrading Alloy with actions. In: ICSE (2005)
8. Galenson, J., Reames, P., Bodik, R., Hartmann, B., Sen, K.: CodeHint: dynamic and interactive synthesis of code snippets. In: ICSE (2014)
9. Galeotti, J.P., Rosner, N., Pombo, C.G.L., Frias, M.F.: TACO: efficient SAT-based bounded verification using symmetry breaking and tight bounds. TSE **39**, 1283–1307 (2013)
10. Gopinath, D., Malik, M.Z., Khurshid, S.: Specification-based program repair using SAT. In: TACAS (2011)
11. Gulwani, S., Hernández-Orallo, J., Kitzelmann, E., Muggleton, S.H., Schmid, U., Zorn, B.: Inductive programming meets the real world. CACM **58**(11), 90–99 (2015)
12. Gvero, T., Kuncak, V., Piskac, R.: Interactive synthesis of code snippets. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 418–423. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_33
13. Hua, J., Khurshid, S.: A sketching-based approach for debugging using test cases. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 463–478. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_29
14. Hua, J., Khurshid, S.: EdSketch: execution-driven sketching for Java. In: SPIN (2017)

15. Hua, J., Zhang, M., Wang, K., Khurshid, S.: Towards practical program repair with on-demand candidate generation. In: ICSE (2018)
16. Jackson, D.: Alloy: a lightweight object modelling notation. TSE **11**, 256–290 (2002)
17. Jackson, D., Fekete, A.: Lightweight analysis of object interactions. In: Kobayashi, N., Pierce, B.C. (eds.) TACS 2001. LNCS, vol. 2215, pp. 492–513. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45500-0_25
18. Jackson, D., Vaziri, M.: Finding bugs with a constraint solver. In: ISSTA (2000)
19. Jeon, J., Qiu, X., Foster, J.S., Solar-Lezama, A.: JSketch: sketching for Java. In: FSE (2015)
20. Jobstmann, B., Griesmayer, A., Bloem, R.: Program repair as a game. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 226–238. Springer, Heidelberg (2005). https://doi.org/10.1007/11513988_23
21. Kang, E., Milicevic, A., Jackson, D.: Multi-representational security analysis. In: FSE (2016)
22. Kneuss, E., Kuraj, I., Kuncak, V., Suter, P.: Synthesis modulo recursive functions. In: OOPSLA (2013)
23. Koukoutos, M., Kneuss, E., Kuncak, V.: An update on deductive synthesis and repair in the leon tool. In: SYNT Workshop (2016)
24. Long, F., Rinard, M.: Staged program repair with condition synthesis. In: FSE (2015)
25. Maier, D.: Theory of Relational Databases. Computer Science Press, Rockville (1983)
26. Maldonado-Lopez, F.A., Chavarriaga, J., Donoso, Y.: Detecting network policy conflicts using alloy. In: Ait Ameur, Y., Schewe, K.D. (eds.) ABZ 2014. LNCS, vol. 8477, pp. 314–317. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43652-3_31
27. Mandelin, D., Xu, L., Bodík, R., Kimelman, D.: Jungloid mining: helping to navigate the API jungle. In: PLDI (2005)
28. Manna, Z., Waldinger, R.: Toward automatic program synthesis. CACM **14**(3), 151–165 (1971)
29. Maoz, S., Ringert, J.O., Rumpe, B.: CD2Alloy: class diagrams analysis using Alloy revisited. In: MODELS (2011)
30. Maoz, S., Ringert, J.O., Rumpe, B.: CDDiff: semantic differencing for class diagrams. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 230–254. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22655-7_12
31. Marinov, D., Khurshid, S.: TestEra: a novel framework for automated testing of Java programs. In: ASE (2001)
32. Nelson, T., Danas, N., Dougherty, D.J., Krishnamurthi, S.: The power of "why" and "why not": enriching scenario exploration with provenance. In: FSE (2017)
33. Nelson, T., Saghafi, S., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Aluminum: principled scenario exploration through minimality. In: ICSE (2013)
34. Nelson, T., Barratt, C., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: The Margrave tool for firewall analysis. In: LISA (2010)
35. Pei, Y., Furia, C.A., Nordio, M., Meyer, B.: Automated program repair in an integrated development environment. In: ICSE (2015)
36. Perelman, D., Gulwani, S., Grossman, D., Provost, P.: Test-driven synthesis. In: PLDI (2014)
37. Polikarpova, N., Kuraj, I., Solar-Lezama, A.: Program synthesis from polymorphic refinement types. In: PLDI (2016)

38. Polozov, O., Gulwani, S.: FlashMeta: a framework for inductive program synthesis. In: OOPSLA (2015)
39. Richters, M., Gogolla, M.: OCL: syntax, semantics, and tools. In: Clark, T., Warmer, J. (eds.) Object Modeling with the OCL. LNCS, vol. 2263, pp. 42–68. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45669-4_4
40. Rothenberg, B.-C., Grumberg, O.: Sound and complete mutation-based program repair. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 593–611. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_36
41. Ruchansky, N., Proserpio, D.: A (not) NICE way to verify the Openflow switch specification: formal modelling of the Openflow switch using Alloy. In: SIGCOMM (2013)
42. Rumbaugh, J., Jacobson, I., Booch, G.: Unified Modeling Language Reference Manual, 2nd edn. Pearson Education, London (2004)
43. Saghafi, S., Danas, R., Dougherty, D.J.: Exploring theories with a model-finding assistant. In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 434–449. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_30
44. Shlyakhter, I., Seater, R., Jackson, D., Sridharan, M., Taghdiri, M.: Debugging overconstrained declarative models using unsatisfiable cores. In: ASE (2003)
45. Singh, R., Solar-Lezama, A.: Synthesizing data structure manipulations from storyboards. In: FSE (2011)
46. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V.: Combinatorial sketching for finite programs. In: ASPLOS (2006)
47. Sullivan, A.: Automated testing and sketching of Alloy models. Ph.D. thesis, University of Texas at Austin (2017)
48. Sullivan, A., Wang, K., Khurshid, S.: AUnit: a test automation tool for Alloy. In: ICST (2018)
49. Sullivan, A., Wang, K., Khurshid, S., Marinov, D.: Evaluating state modeling techniques in alloy. In: SQAMIA (2017)
50. Sullivan, A., Wang, K., Zaeem, R.N., Khurshid, S.: Automated test generation and mutation testing for Alloy. In: ICST (2017)
51. Sullivan, A., Zaeem, R.N., Khurshid, S., Marinov, D.: Towards a test automation framework for alloy. In: SPIN (2014)
52. Torlak, E., Chang, F.S.-H., Jackson, D.: Finding minimal unsatisfiable cores of declarative specifications. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 326–341. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68237-0_23
53. Torlak, E., Jackson, D.: Kodkod: a relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_49
54. Le, V., Afshari, M., Su, Z.: Compiler validation via equivalence modulo inputs. In: PLDI, vol. 49, no. 6, pp. 216–226 (2014)
55. Wang, K., Sullivan, A., Khurshid, S.: MuAlloy: a mutation testing framework for alloy. In: ICSE (2018)
56. Wang, K., Sullivan, A., Marinov, D., Khurshid, S.: Solver-based sketching Alloy models using test valuations. In: ABZ (2018)
57. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In: ICSE (2009)
58. Yang, Z., Hua, J., Wang, K., Khurshid, S.: Test execution driven synthesis of API sequences with conditionals and loops. In: ICST (2018)

Solver-Based Sketching of Alloy Models Using Test Valuations

Kaiyuan Wang[1(✉)], Allison Sullivan[1], Darko Marinov[2], and Sarfraz Khurshid[1]

[1] University of Texas at Austin, Austin, USA
{kaiyuanw,allisonksullivan,khurshid}@utexas.edu
[2] University of Illinois at Urbana-Champaign, Urbana, USA
marinov@illinois.edu

Abstract. We introduce *ASketch*, the first framework for sketching models in the Alloy language. The Alloy Analyzer is a SAT-based constraint solver that allows users to create *valuations* for relations with respect to given constraints and bound on the universe of discourse. Alloy users routinely use the valuations to validate their models: enumerate some valuations and inspect them to detect underconstraints or overconstraints. Our key insight is that valid and invalid valuations enable *sketching* Alloy models where the user writes a *partial* model with *holes* and provides some valuations, and the sketching infrastructure completes the model by synthesizing Alloy fragments for the holes.

ASketch offers the following extensions to Alloy: (1) it expands the Alloy grammar, allowing users to write holes in an Alloy model; (2) it can parse regular expressions and automatically generate pools of matching fragments to replace the holes; (3) it includes a solver-based technique that encodes the model with holes, the fragments for each hole, and the expected valuations to a *meta-model* which completes the holes when solved. Experimental results show that *ASketch* works well for different Alloy models with various number of holes, providing a promising approach to bring the success of traditional program sketching for imperative and functional programs to declarative, relational logic.

1 Introduction

Building software models plays an important role in building reliable systems. Alloy [11] is a well-known, relation-based modeling language that has been used in academic and industrial settings [8,12,22,45]. Alloy has a SAT-based analyzer that performs automatic analysis over a user-defined *scope*, i.e., bound on the universe of discourse. Specifically, the analyzer finds *instances*, i.e., valuations for relations in the model such that the formulas in the model evaluate to true. The analyzer can also find *counterexamples* that refute properties of interest; an instance for the negation of the property formula serves as a counterexample. While Alloy's expressive notation allows succinct formulation of complex properties, reasoning about the correctness of Alloy formulas, e.g., in the presence

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 121–136, 2018.
https://doi.org/10.1007/978-3-319-91271-4_9

of quantification and transitive closure, requires much care. Because Alloy models are effectively logical constraints, they can have two basic kinds of faults: *overconstraints* that rule out valid valuations and *underconstraints* that permit invalid valuations.

We introduce the first approach for *sketching* Alloy models, where the user does not need to write complete models. Instead, the user writes a *partial* model with *holes* and also provides (1) some regular expressions that encode possible fragments for each hole and (2) some valid and invalid valuations that serve as test cases [35,38] for the desired model. Our key insight is that these test valuations enable sketching Alloy models, where the sketching framework completes the partial model with respect to the given fragments and valuations.

Our sketching framework, called *ASketch*, focuses on sketching several constructs of Alloy models, including relational expressions, logical operators, and quantifiers. Given a partial model and the corresponding test valuations, *ASketch* first parses the user-provided regular expressions and generates pools of matching fragments that can replace the holes. Then, *ASketch* systematically explores the resulting search space of candidate Alloy models, to find a model that satisfies all test valuations. Specifically, *ASketch* uses constraint *solving* to explore the space of candidate models by creating one Alloy *meta-model* that encodes the model to sketch along with the fragments for holes and test valuations all at once. The meta-model effectively encodes multiple Alloy models, i.e., all models from the entire candidate space. Finally, *ASketch* uses the Alloy Analyzer to find solutions that can fill in the holes.

We perform an experimental evaluation of *ASketch* using 24 sketches derived from 5 core Alloy models. Experimental results show that *ASketch* can complete sketches that can simultaneously have up to 3 expression holes and 3 non-expression holes. To highlight the complexity of the underlying problem, one example sketch, *BinaryTree* with 6 holes, has a search space of over 4 billion candidate Alloy models (3 expression holes with 400 expression fragments each and 3 non-expression holes with 4 fragments each). *ASketch* finds a solution Alloy model (w.r.t. 16 test valuations) in 12 min, and the Alloy meta-model generated by *ASketch* creates a SAT problem with 1,378 primary variables and 1,188,735 clauses.

While *ASketch* introduces a new technique for writing Alloy models in general, a particular application that we envision for *ASketch* is for *education* about Alloy and more broadly, software modeling using relational specifications. Our experience with beginner Alloy users shows that they often struggle to make their formulas "just right". They have a general idea for a formula skeleton, and they can tell whether certain instances should or should not satisfy a formula, but they still make mistakes that overconstrain or underconstrain their models. We expect that beginners could greatly benefit from an iterative methodology where the user could start from some skeleton formula with holes, use *ASketch* to complete the formula, obtain some valuations, label them as valid or invalid, and repeatedly iterate until getting all (and only) the valuations that the user expects. In fact, our evaluation subjects are inspired by the example models that beginners often struggle with.

This paper makes the following contributions:

Idea: We introduce the idea of sketching Alloy models using test valuations.

***ASketch*:** We introduce a technique for completing Alloy sketches based on constraint solving.

Experiments: We present an experimental evaluation with small but intricate Alloy formulas; the results show that *ASketch* introduces a promising approach for sketching Alloy models.

2 Example

To illustrate our *ASketch* approach, consider the following partial Alloy model for an acyclic singly linked list:

```
one sig List { header: lone Node }   sig Node { link: lone Node }
pred Acyclic() { \Q,q\ n: Node | n \CO,co\ \E,e\ => n \CO,co\ \E,e\ }
q := {| all|no|some|lone|one |}
co := {| =|in|!=|!in |}
e := {| (List.header|n).(~?)(*|^)link |}
```

The *signature* (`sig`) declaration introduces a set of atoms and a user-defined type. A signature may declare *fields*, i.e., relations. `List` declares a set of list atoms; `one` makes the set *singleton*, i.e., have exactly 1 atom, which represents the list we are modeling. The field `header` declares a binary relation of type `List` × `Node`; `lone` declares `header` to be a *partial* function, i.e., each `List` atom maps to at most one `Node` atom. `Node` declares a set of nodes and introduces the field `link`, which is a partial function of type `Node` × `Node`. The predicate (`pred`) `Acyclic` introduces a named formula (which may have parameters).

The body of the predicate is a formula *sketch* with three different kinds of holes: `\Q,q\` (quantifier hole), `\CO,co\` (comparison operator hole), and `\E,e\` (expression hole). For the sake of illustrative example, we create several holes of different kinds (potentially more than a user would actually create), and we explicitly list all potential fragments for each hole. Each hole states the syntactic kind of the hole followed by an identifier, e.g., `E` followed by `e`. Each identifier refers to a regular expression (within `{| ... |}`, following [30]), e.g., `e` refers to `(List.header|n).(~?)(*|~)link`, which encodes a set of eight Alloy expressions in this example, including expressions `List.header.*link` and `n.~link`. *ASketch* extends the Alloy grammar [39] with these holes. The variable `n` is introduced by the quantifier (to be sketched) and is of type `Node`; the operator `=>` denotes logical implication.

The goal is to fill in the holes such that the formula constrains the nodes in the list to form an acyclic structure. Figure 1 graphically illustrates four test valuations for the model. Three valuations—$T0, T1$, and $T3$—are valid with respect to the expected acyclicity constraint. One valuation, $T2$, is invalid. Note that $T3$ is valid although $N1$ links to itself: $N1$ is not in the list, and the formula we are sketching should constrain only the nodes that are in the list, i.e., reachable from the `header`.

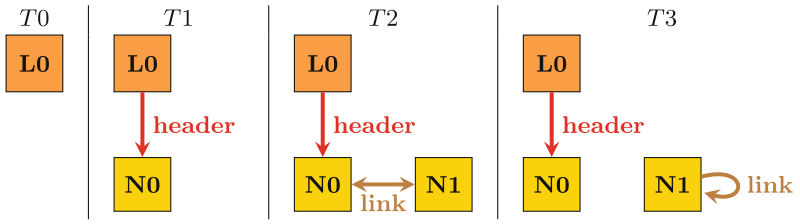


Fig. 1. Four test valuations shown graphically: $T0, T1$, and $T3$ are valid for the expected acyclicity; $T2$ is invalid. $L0$ is the list atom; $N0$ and $N1$ are node atoms.

The user can provide the test valuations simply as Alloy predicates. For example, the following represent test valuations $T0$ and $T2$ from Fig. 1:

```
pred Test0() {
  some L0: List {
    List = L0 and no header and no Node and no link and Acyclic[] }}
pred Test2() {
  some L0: List | some disj N0, N1: Node {
    List = L0 and header = L0->N0 and Node = N0+N1 and link = N0->N1 + N1->N0 and !Acyclic[] }}
```

The predicate `Test0` uses an existentially quantified (`some`) formula to assign a value to the `List` set. Using the Alloy keyword `no`, `Test0` declares the other signatures and relations to be empty. The predicate invocation `Acyclic[]` labels the valuation as *valid* for the expected acyclicity constraint. The predicate `Test2` uses existentially quantified formulas to assign values to the `List` and `Node` sets. The keyword `disj` requires the variables in the declaration to represent disjoint sets (i.e., unique nodes), the operator `->` denotes Cartesian product, the operator `+` denotes set union, and the predicate invocation `!Acyclic[]` labels the valuation as *invalid* for the expected acyclicity constraint.

Consider using *ASketch* to complete all five holes. Two are expression holes `\E,e\` with the same given regular expression assigned for the fragment space, and each expression hole has eight syntactically different expression fragments. Alloy also allows five quantifiers for `\Q,q\` (`all`, `no`, `some`, `lone`, and `one`) and four comparison operators for `\CO,co\` (`=`, `in`, `!=`, and `!in`). In total, there are $5 \times 4 \times 8 \times 4 \times 8 = 5,120$ candidate Alloy models. For our example, we use 8 test valuations to obtain the expected solutions (4 shown in Fig. 1 plus 4 more). To complete the sketch, *ASketch* takes less than 1 s when solving the entire Alloy meta-model that encodes all 5,120 models and 8 valuations at once. Here is a solution *ASketch* finds:

```
all n: Node | n in List.header.*link => n !in n.^link
```

The Alloy keyword `in` represents the subset, and `!` denotes logical negation. The operator `*` denotes reflexive transitive closure, and `^` denotes transitive closure. The expression `List.header.*link` represents the set of all nodes reachable from the list's header (following *zero* or more traversals of the field `link`). The expression `n.^link` represents the set of all nodes reachable from `n` (following *one* or more traversals of the field `link`). Thus, this universally quantified

formula states that for any node that is in the list, the node is not reachable from itself, which correctly characterizes our expected acyclicity constraint.

3 *ASketch* Framework

We next present the *ASketch* grammar for Alloy models with holes and describe how *ASketch* determines which fragments complete the sketch to produce an Alloy model that satisfies all the given test valuations.

3.1 Input Language

The input to *ASketch* is an Alloy model with holes. For lack of space, we do not show the full grammar for *ASketch*'s input language, but it effectively extends the Alloy grammar with new syntactic constructs that represent holes. The current Alloy grammar is available at http://alloy.csail.mit.edu/alloy/documentation/alloy4-grammar.txt; we follow an older exposition [11] that included the semantics of the kernel Alloy language. Consider this part of the *ASketch* grammar:

```
quant ::= "all" | "no" | "some" | "lone" | "one" | "\Q," identifier "\"
expr ::= "*"expr | expr "+" expr | ... | "\E," identifier "\"
compareOp ::= "=" | "in" | "!=" | "!in" | "\CO," identifier "\"
formula ::= quant v ":" type "|" formula | ...
regExDecl ::= identifier ":=" "{|" regex "|}"
regex ::= nonSpecial | regex "?" | "(" regex ")" | regex regex | regex "|" regex
```

We extend `quant` so the quantifier can be a hole `\Q,i\` where `Q` indicates the quantifier hole kind and `i` is an identifier that maps to a regular expression via `regExDecl`. The `expr` options include the expressions from Alloy, formed with unary (e.g., `*`) or binary operators (e.g., `+`), and we add a hole (`\E,i\`) that can replace an entire expression. Comparison operators include all operators from Alloy and also a hole `\CO,i\`. The `formula` options include the Alloy first-order logic formulas. `regExDecl` has the form `i:={|e|}` where `i` is referred from a hole and `e` is a regular expression. We follow the design of popular sketching system [13,30,32] that include a few regular expression operators: options (`e?`), concatenations (`e1 e2`), and choices (`e1 | e2`). `nonSpecial` is any character that Alloy supports except for `?`, `(`, `)`, and `|`; to use those, requires escaping them as `\(`, `\)`, and `\|`. Finally, *ASketch* generates all possible fragments that match `e` using a standard backtracking algorithm [20]. *ASketch* supports all fragments for non-expression holes, as shown in Table 1. Our current implementation requires

Table 1. Supported fragments for non-recursively defined holes

Sketch kind	Hole	Candidates	Sketch kind	Hole	Candidates
Quantifier	\Q\	all, no, some, lone, one	Unary operator formula	\UOF\	!, ␣
Logical operator	\LO\	||, & &, <=>, =>	Unary operator expression	\UOE\	~, *, ^
Compare operator	\CO\	=, in, !=, !in	Binary Operator	\BO\	&, +, −
Unary operator	\UO\	no, some, lone, one			

an explicit regular expression for every hole, although a default could be set up such that non-expression holes implicitly get all possible fragments without listing them explicitly.

3.2 Solver-Based Sketching

ASketch reduces the sketching problem to a constraint-solving problem in the Alloy language itself, which is then solved by the Alloy Analyzer. Effectively, *ASketch* generates one *meta-model* in Alloy that encodes multiple potential solutions (i.e., concrete models) to the sketch. To represent the fragments for each hole, two constructs are added to the meta-model: (1) an Alloy *atom* that names a specific fragment for the hole, and (2) constraints that characterize the semantics of the different fragments for the sketch.

Because *ASketch* uses the Alloy tool-set itself to encode Alloy expressions and formulas, their semantics need not be explicitly modeled in Alloy; rather, they just need to be stated—indeed, the Alloy tool-set understands the semantics of Alloy. Therefore, we can use a shallow embedding of Alloy fragments in the model. Specifically, to represent the expression fragments, *ASketch* creates new Alloy *functions*, i.e., parameterized expressions. To represent the operator fragments, *ASketch* creates new Alloy *predicates*, i.e., parameterized formulas. Moreover, to encode multiple given test valuations in the same meta-model, *ASketch parameterizes* formulas with respect to user-defined relations, which are extracted out of their declaring *signatures* and added as new parameters. Our encoding allows constraining the model with respect to *all* valuation constraints at once—without causing an unnecessary increase in the number of propositional variables in the resulting SAT formula and without requiring higher-order solving [22].

We use the linked-list example from Sect. 2 to describe how *ASketch* sketches the body of a predicate and completes five holes of three kinds—quantifiers (`\Q,q\`), comparison operators (`\CO,co\`), and expressions (`\E,e\`). *ASketch* uses the following steps to create an Alloy meta-model whose solutions complete the sketch: (1) parameterize Alloy construct (Sect. 3.2.1); (2) create Alloy meta constructs to encode holes (Sect. 3.2.2); (3) translate test valuations to facts (Sect. 3.2.3); and (4) invoke the Alloy Analyzer to complete the holes (Sect. 3.2.4).

3.2.1 Parameterize Alloy Constructs

In the first step, *ASketch* parameterizes all predicates, functions, and facts. To parameterize an Alloy fact, *ASketch* first converts it to a semantically equivalent predicate. Without loss of generality, we only present how *ASketch* parameterizes predicates. The goal is to allow *multiple* test valuations to be encoded *in the same meta-model*. *ASketch* constructs a meta-model which includes (1) all signature declarations from the partial model, but *without* any of the declared relations, and (2) all predicates. Moreover, all predicates in the meta-model get additional parameters: one new parameter per signature and one new parameter

per field; parameters that represent signatures have fresh variable names generated, whereas those that represent fields use the same names as in the partial model. In the body of the predicates, any reference to a declared signature is replaced by the corresponding fresh variable name.

For our acyclic linked-list example from Sect. 2, we get the following:

```
one sig List {}   sig Node {}
pred Acyclic(ls: one List, header: List -> Node, ns: set Node, link: Node -> Node) {
  \Q,q\ n: ns | n \CO,co\ \E,e\ => n \CO,co\ \E,e\ }
```

3.2.2 Create Alloy Meta Constructs to Encode Holes

ASketch creates Alloy meta constructs that encode concrete values for every hole in Alloy predicates. We present how to encode only quantifier holes, comparison operator holes, and expression holes in Alloy predicates. The algorithm takes as inputs a mapping from expression holes to the corresponding expression fragments and a mapping from holes to *all Alloy variables* (sigs, fields, predicate parameters, `let`-bound variables, and quantified variables) in scope of the holes. The algorithm iterates over each Alloy predicate in the meta-model and updates the predicate body by recursively replacing *ASketch* holes with predicate/function calls, and creating and adding the predicate/function declarations to the meta-model. Note that any reference to a declared signature in the generated predicate/function is replaced by the corresponding fresh variable name as described in Sect. 3.2.1, e.g., `List` with `ls`.

After this step, *ASketch* constructs the following meta-model (note that the two comparison operator holes share the same operator fragments, and the two expression holes share the same expression fragments):

```
pred Acyclic(ls: one List, header: List -> Node, ns: set Node, link: Node -> Node) {
  q1[RQ1, ls, header, ns, link] }
abstract sig Q {}   one sig RQ1 in Q {}
one sig Q_All, Q_No, Q_Some, Q_Lone, Q_One extends Q {}
pred q1(h: Q, ls: one List, header: List -> Node, ns: set Node, link: Node -> Node) {
  h = Q_All => all n: ns | co2[RCO2, n, expr3[RE3, ls, header, ns, link, n]] =>
    co2[RCO4, n, expr3[RE5, ls, header, ns, link, n]]
  h = Q_No => no n: ns | co2[RCO2, n, expr3[RE3, ls, header, ns, link, n]] =>
    co2[RCO4, n, expr3[RE5, ls, header, ns, link, n]]
  ... }
abstract sig CO {}   one sig RCO2 in CO {}   one sig RCO4 in CO {}
one sig CO_Eq, CO_In, CO_NEq, CO_NIn extends CO {}
pred co2(h: CO, e1, e2: set univ) {
  h = CO_Eq => e1 = e2
  h = CO_In => e1 in e2
  ... }
abstract sig E3 {}   one sig RE3 in E3 {}   one sig RE5 in E3 {}
one sig E3_1, E3_2, E3_3, E3_4, E3_5, E3_6, E3_7, E3_8 extends E3 {}
fun expr3(h: E3, ls: one List, header: List -> Node,
          ns: set Node, link: Node -> Node, n: one Node): univ {
  (h = E3_1 => ls.header.*link else
  (h = E3_2 => n.^link else
  ... else none)) }
```

For quantifier holes, *ASketch* creates a unique *abstract* sig `Q` and declares 5 disjoint singleton sigs that represent all possible values for the hole (`all`, `no`, `some`, `lone`, and `one`). For each quantifier hole, *ASketch* translates the quantified

formula to a predicate call. The predicate has the following parameters: (1) one parameter of the new abstract sig type that allows evaluating the predicate to one of the 5 quantifiers; and (2) one parameter for each variable in scope: signatures and fields from the original model, and optionally, predicate parameters, `let`-bound variables, and/or quantified variables in case of nested quantified formulas. The corresponding predicate declaration, `q1` in our example, is added to the meta-model. The predicate body is a conjunction of implications that model different quantified formulas corresponding to the hole. *ASketch* also introduces a result sig, `RQ1` in our example, that will obtain one of the 5 values (`Q_All`, `Q_No`, `Q_Some`, `Q_Lone` and `Q_One`) to represent the quantifier to fill in the hole.

For comparison operator holes, *ASketch* creates a unique *abstract* sig `CO` and declares 4 disjoint singleton sigs that represent all possible values for the hole (`=`, `in`, `!=`, and `!in`). Unlike for quantifier holes where each hole requires a new predicate, all comparison operator holes (of the same arity) can be encoded using a single predicate if they share the same set of fragments. *ASketch* creates a predicate, `co2` in our example, which encodes a formula that contains a comparison operator. The predicate contains 3 parameters: (1) one parameter of the new abstract sig type that allows evaluating the predicate to one of the 4 comparison operators (`CO_Eq`, `CO_In`, `CO_NEq`, and `CO_NIn`); (2) left operand; and (3) right operand. For each comparison operator hole, *ASketch* introduces a result sig, `RCO2` and `RCO4` in our example, similar as for quantifier holes. (*ASketch* treats the other non-expression holes similar to comparison operator holes, but we do not present details due to space limits.)

To model values of expression holes, *ASketch* creates one new *abstract* sig, `E3` in our example, for all holes that share the same set of expression fragments and declares k singleton sigs that partition the new sig, where k is the number of expression fragments for the corresponding expression hole, 8 in our example. *ASketch* also introduces result sigs, `RE3` and `RE5` in our example, that will obtain one of the k values to represent which fragment fills the hole. Next, *ASketch* creates an Alloy function that can select from these choices. The function has these parameters: (1) one parameter of the new abstract sig type that allows evaluating the function to one of the expression fragments based on the invocation context; and (2) one parameter for each Alloy variable in scope. The function body is a nested if-then-else expression where exactly one choice is true for any invocation, and the function evaluates to the value of the expression fragment corresponding to that choice.

3.2.3 Express Test Valuations as Facts

To complete the sketch with respect to the given test valuations (labeled as valid or invalid), *ASketch* automatically translates the test valuations (expressed as predicates in Sect. 2) to *facts*, which forces any solution that is created (in the final meta-model) to conform to all given valuations. Because valuations from different tests may contradict one another, *ASketch* uses Alloy's `let` construct to introduce the necessary names for sets and relations that are assigned values. Then, *ASketch* passes these sets and relations to the parameterized predicates

(described in Sect. 3.2.1) so that the final sketched model satisfies all the tests at once. For example, `Test0` from Sect. 2 becomes the following fact:

```
fact Test0 {
  some L0: List {
    let ls = L0 | let header = none->none | let ns = none | let links = none->none |
      Acyclic[ls, header, ns, links] }}
```

3.2.4 Invoke Alloy Analyzer to Complete Holes

The final meta-model consists of all pieces generated in Sects. 3.2.1, 3.2.2, and 3.2.3. *ASketch* invokes the Alloy Analyzer to execute an empty `run` command (`run {}`) on the final meta-model. The analyzer searches for possible valuations of the result `R` sigs so that they conform to all tests. In our example, `RQ1` evaluates to `Q_All`, `RCO2` to `CO_In`, `RE3` to `E3_1`, `RCO4` to `CO_NIn`, and `RE5` to `E3_2`. Finally, *ASketch* maps result values to the corresponding Alloy fragments and reports concrete values of all holes to the user, e.g., ⟨`all, in, List.header.*link, !in, n.^link`⟩ in our example. The completed, sketched model becomes this:

```
one sig List { header: lone Node }   sig Node { link: lone Node }
pred Acyclic() { all n: Node | n in List.header.*link => n !in n.^link }
```

Our example used only 8 expressions, but realistic *ASketch* models may have hundreds of expressions, which results in much larger meta-models. Our experiments show that the above encoding technique still works relatively well even for a large number of expressions. It also works much better than all other meta-model encoding techniques we tried.

4 Experimental Evaluation

We next present our experimental evaluation of *ASketch*. We use five small but intricate Alloy problems to derive 24 sketching models for evaluation (Sect. 4.1). We evaluate how much time *ASketch* takes to find complete Alloy models that satisfy all test valuations (Sect. 4.2).

4.1 Sketching Problems

We use 24 sketches derived from five core Alloy models: *LinkedList* from Sect. 2, *BinaryTree* models the acyclicity constraint of a binary tree, *Contains* checks whether a list contains an element, *Remove* models removing an element from a list, and *Dijkstra* models Dijkstra's mutual exclusion algorithm.

For each core model, we picked one predicate to create several sketches by increasing the total number of holes in the body of the predicate, from left to right. This process enables us to systematically create model variants to explore how the number of holes affects our techniques. For example, for *LinkedList*, we identified 3 non-expression holes and 2 expression holes in the `Acyclic` predicate and produced these 5 variants:

```
\Q,q\ n: Node | n in List.header.*link => n !in n.^link          // LinkedList 1H
\Q,q\ n: Node | n \CO,co\ List.header.*link => n !in n.^link     // LinkedList 2H
\Q,q\ n: Node | n \CO,co\ \E,e\ => n !in n.^link                 // LinkedList 3H
\Q,q\ n: Node | n \CO,co\ \E,e\ => n \CO,co\ n.^link             // LinkedList 4H
\Q,q\ n: Node | n \CO,co\ \E,e\ => n \CO,co\ \E,e\               // LinkedList 5H
```

Our experiments are performed on a MacBook Pro running OS X El Capitan with 2.5 GHz Intel Core i7-4870HQ and 16 GB of RAM.

4.2 *ASketch* Results

Table 2 shows the results of *ASketch* for various sketching problems. The column *Model* shows the model variants for each core model; columns $\#N$ and $\#E$ show the number of non-expression holes and expression holes, respectively; the column *Search Space* shows the number of fragments combinations for all holes; and the columns *#Primary Vars*, *#Clauses*, and *Solving Time* show the number of primary variables, clauses, and solving time in seconds for the meta model, respectively. The *Search Space* is computed as the product of the number of fragments for each hole in the model. For example, if the *LinkedList* model with 5 holes has 1 quantifier hole with 5 fragments, 2 comparison operator holes with 4 fragments each, and 2 expression holes with 400 fragments each, then the sketching problem has a search space of $5 \times 4^2 \times 400^2 = 12,800,000 \cong 1.3\text{e}7$.

The columns $50, 100, 200, 300$, and 400 show the number of expression fragments in the experiment, e.g., 50 means that we use 50 syntactically different expressions for each expression hole in the model variant. We generate regular expressions for expression holes using RexGen[1] [43] such that two properties hold. First, the set of expressions contains the expected solutions. Second, the larger set of expressions contains all expressions from the smaller set, e.g., the set of 100 expressions includes the set of 50 expressions and adds 50 more. We ensure the first property as follows. Suppose we have H expression holes and E expected expressions to fill the holes. We run RexGen to get X expressions and exclude E expected expressions from X expressions. Next, we run *ASketch* to find all solutions w.r.t. the test valuations and exclude any expression in the solutions that is non-equivalent to any of the E expected expressions. The idea is to remove all expressions that could lead to a solution that passes all tests but is incorrect. Then, to form a set of expressions with size Y (where Y is $50, 100, 200, 300$, or 400), we sample the remaining expressions to obtain $Y - E$ expressions, and add the E expected expressions back.

Dijkstra has two expression holes with different variables in scope, so each expression hole uses a different set of expression fragments (but with the same number of expressions). Expression holes for each of *LinkedList*, *BinaryTree*, *Contains*, and *Remove* share the same set of expression fragments. In the experiments, we use 16 test valuations for each core model, and all model variants of the

[1] Note that RexGen can work in the mode where it prunes out equivalent expression fragments. We do not use that mode because we want to generate a large number of expression fragments for our experiments. All expressions that we generate are syntactically different but some may be semantically equivalent.

Table 2. *ASketch* results for finding a solution. Times are in seconds.

Model		#N	#E	Search Space					#Primary Vars					#Clauses					Solving Time				
				50	100	200	300	400	50	100	200	300	400	50	100	200	300	400	50	100	200	300	400
LinkedList	1H	1	0	5	5	5	5	5	138	138	138	138	138	6170	6170	6170	6170	6170	0.1	0.2	0.1	0.1	0.1
	2H	2	0	20	20	20	20	20	142	142	142	142	142	7397	7397	7397	7397	7397	0.2	0.2	0.2	0.2	0.2
	3H	2	1	1e3	2e3	4e3	6e3	8e3	192	242	342	442	542	6.3e4	1.1e5	2.3e5	3.6e5	5.1e5	2.8	4.5	88.6	267.9	141.0
	4H	3	1	4e3	8e3	1.6e4	2.4e4	3.2e4	196	246	346	446	546	6.5e4	1.2e5	2.4e5	3.8e5	5.3e5	3.2	5.2	91.1	286.6	150.1
	5H	3	2	2e5	8e5	3.2e6	7.2e6	1.3e7	246	346	546	746	946	1.1e5	2.1e5	4.4e5	7e5	1e6	6.7	32.5	252.9	574.6	759.1
BinaryTree	1H	1	0	4	4	4	4	4	170	170	170	170	170	8e3	8e3	8e3	8e3	8e3	0.1	0.1	0.1	0.1	0.1
	2H	1	1	2e2	4e2	8e2	1200	1600	220	270	370	470	570	6e4	1.1e5	2.2e5	3.5e5	4.8e5	0.6	1.1	3.1	5.7	5.6
	3H	2	1	8e2	1600	3200	4800	6400	224	274	374	474	574	6.1e4	1.1e5	2.2e5	3.5e5	4.9e5	0.5	1.4	2.9	3.6	12.3
	4H	2	2	4e4	1.6e5	6.4e5	1.4e6	2.6e6	274	374	574	774	974	8.6e4	1.7e5	3.6e5	5.8e5	8.4e5	1.4	5.6	27.6	36.9	265.7
	5H	3	2	1.6e5	6.4e5	2.6e6	5.8e6	1e7	278	378	578	778	978	8.6e4	1.7e5	3.6e5	5.8e5	8.4e5	1.8	7.0	30.3	47.5	127.4
	6H	3	3	8e6	6.4e7	5.1e8	1.7e9	4.1e9	328	478	778	1078	1378	1.1e5	2.3e5	4.9e5	8.1e5	1.2e6	1.4	37.7	138.7	515.1	709.3
Contains	1H	0	1	50	1e2	2e2	3e2	4e2	312	362	462	562	662	3.3e4	6.1e4	1.3e5	2.2e5	3.2e5	0.4	0.7	3.2	7.4	7.6
	2H	1	1	2e2	4e2	8e2	1200	1600	316	366	466	566	666	5.4e4	1e5	2.1e5	3.2e5	4.6e5	0.7	1.7	9.4	25.1	24.9
	3H	1	2	1e4	4e4	1.6e5	3.6e5	6.4e5	366	466	666	866	1066	7.7e4	1.5e5	3.3e5	5.4e5	7.9e5	2.0	15.6	103.3	397.0	1925.2
RemoveC	1H	0	1	50	1e2	2e2	3e2	4e2	267	317	417	517	617	4e4	7.3e4	1.5e5	2.5e5	3.6e5	0.2	0.5	1.1	2.2	4.4
	2H	1	1	150	3e2	6e2	9e2	1200	270	320	420	520	620	4.1e4	7.4e4	1.5e5	2.5e5	3.6e5	0.3	0.7	1.3	2.8	5.3
	3H	1	2	7500	3e4	1.2e5	2.7e5	4.8e5	320	420	620	820	1020	7.2e4	1.4e5	3e5	5e5	7.3e5	0.7	2.0	3.6	16.7	75.9
	4H	1	3	3.8e5	3e6	2.4e7	8.1e7	1.9e8	370	520	820	1120	1420	1e5	2.1e5	4.5e5	7.5e5	1.1e6	7.0	62.2	61.6	6903.8	19579.8
Dijkstra	1H	1	0	4	4	4	4	4	370	370	370	370	370	9348	9348	9348	9348	9348	0.1	0.1	0.0	0.0	0.0
	2H	1	1	2e2	4e2	8e2	1200	1600	420	470	570	670	770	9.2e4	1.4e5	2.6e5	3.9e5	5.4e5	1.1	2.1	4.8	24.2	21.9
	3H	2	1	1e3	2e3	4e3	6e3	8e3	404	454	554	654	754	9.2e4	1.4e5	2.6e5	3.9e5	5.4e5	1.1	2.2	4.9	21.6	22.1
	4H	3	1	5e3	1e4	2e4	3e4	4e4	409	459	559	659	759	9.4e4	1.5e5	2.6e5	4e5	5.4e5	1.9	4.2	6.0	40.9	41.3
	5H	4	1	2e4	4e4	8e4	1.2e5	1.6e5	413	463	563	663	763	9.8e4	1.5e5	2.7e5	4e5	5.5e5	1.7	2.9	14.3	26.9	46.9
	6H	4	2	1e6	4e6	1.6e7	3.6e7	6.4e7	463	563	763	963	1163	2.7e5	5.4e5	1.1e6	1.7e6	2.3e6	26.8	80.6	1053.1	4542.8	6535.0

same core model share the same test suite. All experiment settings, with various fragments and test valuations, yield solutions that are semantically equivalent to the correct solutions.

If a sketch has no expression hole, then increasing the number of the expression fragments does not increase the search space, primary variables, or clauses in the generated meta-model. For example, *BinaryTree* model with 1 hole has only a comparison operator hole, and the search space (4), the number of primary variables (170), and clauses (7,957) remain unchanged as the number of expression fragments increases. If the sketch has expression holes, then the search space, primary variables, and clauses increase when we use more expression fragments. In our experiment, the search space goes up to 4.1e9 (*BinaryTree*), the number of primary variables goes up to 1420 (*Remove*), and the number of clauses goes up to 2.3e6 (*Dijkstra*). Overall these numbers show that the sketching problems are non-trivial.

The solving time depends on various factors, including the number of primary variables and clauses, the size of each clause, the complexity of the expression fragments, the search strategy of the SAT solver, etc. In general, the solving time increases with the size of the search space and the number of holes. However, there are exceptions. For example, in *LinkedList* with 4 holes, the solving time decreases as the size of expression fragments grows from 300 to 400. The reason is that multiple expression fragments are correct and equivalent. We cannot control how the Alloy Analyzer generates CNF clauses from the meta-model, so some solutions are found sooner than the others even if we increase the search space. Another exception is when *BinaryTree* goes from 4 holes to 5 holes using 400 expression fragments. Again, the solving time decreases as the number of holes increases. The reasons are that (1) adding an operator hole does not increase the number of primary variables or clauses by much; (2) it can make the sketching problem easier to solve as more equivalent correct solutions can be found; and (3) the Alloy Analyzer encodes the problem such that the solver is able to find the solution fast. Overall, *ASketch*'s encoding is relatively efficient and works well for large search spaces.

5 Related Work

We introduce the first approach to sketching Alloy models. Program sketching [1, 13, 28–33] is a form of program synthesis, which is a mature yet active research topic [2, 5–7, 9, 17, 19, 21, 25, 28]. Researchers have proposed program synthesis techniques for a number of languages, including synthesis of logic programs, e.g., using inductive synthesis based on positive and negative examples [3]. However, prior work has not addressed the complexity of synthesis in the presence of quantifiers, transitive closure, relational operators, and more generally, formulas that express structurally complex properties, which are the focus of our work.

The Sketch system [30] takes as input a partial program in the Java-like Sketch language, and uses SAT and inductive synthesis in a counterexample-guided loop. Sketch requires users to provide generators for expression fragments

for expression holes. The JSketch tool translates Java to Sketch to allow sketching Java programs [13]. Some tools focus on specific kinds of programs to sketch, such as PSketch for concurrent data structures [32].

Previous work on program synthesis has also used user-provided tests, albeit for imperative code, to guide synthesis. SyPet [5] introduced a novel use of Petri nets in synthesizing sequences of method invocations for complex APIs using tests. EdSketch [10] and EdSynth [44] introduced an optimized backtracking search for completing Java sketches using test executions for pruning. Test-Driven Synthesis iteratively builds a C# program such that it satisfies all tests [26]. Component-based synthesis builds programs by combining components from given libraries, e.g., work in this line used I/O oracles to synthesize loop-free programs [14].

Our approach also shares the spirit of storyboard programming, which uses user-provided graphical representations of data structures to synthesize imperative code that performs desired data structure manipulations based on the insight that it can be easier and more intuitive for a user to provide concrete data structure manipulations than to write the code [29]. Our test valuations make use of a similar insight.

An approach for creating Alloy models using instances was introduced by aDeryaft [15] in the spirit of Daikon [4] that uses a collection of known properties to check which hold with respect to given inputs. Alchemy [18] defined a translation to database update operations and integrity constraints. AUnit [37,38] recently defined the concepts of test case, test execution, and model coverage for unit testing of Alloy models in the spirit of popular xUnit frameworks for imperative languages. AUnit has also enabled the adoption of other traditional imperative testing infrastructures to Alloy such as mutation testing [37,42]. The test valuations that ASketch uses in the context of synthesis follow AUnit's definition of a test case. *ASketch*'s solver-based approach for sketching also inspired a way to model state and state transitions in Alloy [36].

While this paper focuses on sketching for Alloy, one of the earliest approaches for helping Alloy users build their models correctly was based on identifying unsatisfiable cores in overconstrained models [27,40,41], which aids in automated debugging. More recent work introduced different strategies for scenario exploration for better understanding of the properties modeled [23,24].

6 Conclusions and Future Work

We introduced *ASketch*, the first approach for sketching Alloy models. Given a model with holes and some (valid and invalid) valuations for the desired model, *ASketch* completes the given model with respect to the valuations. *ASketch* performs two key steps: it generates a pool of fragments (e.g., expressions) for each hole from user-provided regular expressions, and it creates a meta-model to explore the resulting space of candidate (completed) models to find a model that conforms to the valuations. An experimental evaluation using a suite of sketches shows that *ASketch* introduces a promising approach for sketching Alloy models.

ASketch brings the spirit of traditional program sketching [1,10,13,16,28–33]—often regarded as the breakthrough approach in program synthesis for imperative and functional programs during the last decade—to a declarative, relational logic. We hope *ASketch* serves as a sound basis for a highly effective methodology for synthesizing Alloy models, which ultimately increases the use of analyzable models and leads to better software.

Future work can build on *ASketch* for solving other problems, such as automated debugging of faulty Alloy models. To illustrate, consider a model that is erroneously overconstrained. To repair it, first identify its unsat core using SAT to localize likely faulty expressions or formulas, and then create a sketch and complete it using *ASketch*. Future work can also evaluate the usability of *ASketch* via a user study; as common in sketching [30], we start first from the algorithmic foundations for sketching and leave actual user evaluations for later. An alternative to *ASketch*, which is a *solver-based* technique, is to employ an *enumeration-based* technique [34]; future work can rigorously compare the two techniques and combine them for a likely more effective synergistic approach.

Acknowledgements. We thank Manos Koukoutos and Viktor Kuncak for their comments on this work. This material is based upon work partially supported by the National Science Foundation under Grant Nos. CCF-1409423, CCF-1421503, CNS-1646305, CCF-1718903, and CNS-1740916.

References

1. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: FMCAD (2013)
2. Bodík, R., Jobstmann, B.: Algorithmic program synthesis: introduction. STTT **15**, 397–411 (2013)
3. Deville, Y., Lau, K.K.: Logic program synthesis. J. Logic Program. **19–20**, 321–350 (1994)
4. Ernst, M.D.: Dynamically discovering likely program invariants. Ph.D. thesis, University of Washington Department of Computer Science and Engineering (2000)
5. Feng, Y., Martins, R., Wang, Y., Dillig, I., Reps, T.W.: Component-based synthesis for complex APIs. In: POPL (2017)
6. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. In: PLDI (2015)
7. Galenson, J., Reames, P., Bodik, R., Hartmann, B., Sen, K.: CodeHint: dynamic and interactive synthesis of code snippets. In: ICSE (2014)
8. Galeotti, J.P., Rosner, N., Pombo, C.G.L., Frias, M.F.: TACO: efficient SAT-based bounded verification using symmetry breaking and tight bounds. TSE **39**, 1283–1307 (2013)
9. Gvero, T., Kuncak, V., Piskac, R.: Interactive synthesis of code snippets. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 418–423. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_33
10. Hua, J., Khurshid, S.: EdSketch: Execution-driven sketching for Java. In: SPIN (2017)

11. Jackson, D.: Alloy: a lightweight object modelling notation. TSE **11**, 256–290 (2002)
12. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press, Cambridge (2006)
13. Jeon, J., Qiu, X., Foster, J.S., Solar-Lezama, A.: JSketch: sketching for Java. In: FSE (2015)
14. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: ICSE (2010)
15. Khurshid, S., Malik, M.Z., Uzuncaova, E.: An automated approach for writing Alloy specifications using instances. In: ISoLA (2006)
16. Kneuss, E., Koukoutos, M., Kuncak, V.: Deductive program repair. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 217–233. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_13
17. Kneuss, E., Kuraj, I., Kuncak, V., Suter, P.: Synthesis modulo recursive functions. In: OOPSLA (2013)
18. Krishnamurthi, S., Fisler, K., Dougherty, D.J., Yoo, D.: Alchemy: transmuting base Alloy specifications into implementations. In: FSE (2008)
19. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Complete functional synthesis. In: PLDI (2010)
20. Larson, E., Kirk, A.: Generating evil test strings for regular expressions. In: ICST (2016)
21. Mandelin, D., Xu, L., Bodík, R., Kimelman, D.: Jungloid mining: helping to navigate the API jungle (2005)
22. Milicevic, A., Near, J.P., Kang, E., Jackson, D.: Alloy*: a general-purpose higher-order relational constraint solver. In: ICSE (2015)
23. Nelson, T., Danas, N., Dougherty, D.J., Krishnamurthi, S.: The power of "why" and "why not": enriching scenario exploration with provenance. In: FSE (2017)
24. Nelson, T., Saghafi, S., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Aluminum: principled scenario exploration through minimality. In: ICSE (2013)
25. Osera, P.M., Zdancewic, S.: Type-and-example-directed program synthesis. In: PLDI (2015)
26. Perelman, D., Gulwani, S., Grossman, D., Provost, P.: Test-driven synthesis. In: PLDI (2014)
27. Shlyakhter, I., Seater, R., Jackson, D., Sridharan, M., Taghdiri, M.: Debugging overconstrained declarative models using unsatisfiable cores. In: ASE (2003)
28. Singh, R., Gulwani, S.: Predicting a correct program in programming by example. In: CAV (2015)
29. Singh, R., Solar-Lezama, A.: Synthesizing data structure manipulations from storyboards. In: FSE (2011)
30. Solar-Lezama, A.: Program synthesis by sketching. Ph.D. thesis, University of California, Berkeley (2008)
31. Solar-Lezama, A., Arnold, G., Tancau, L., Bodik, R., Saraswat, V., Seshia, S.: Sketching stencils. In: PLDI (2007)
32. Solar-Lezama, A., Jones, C.G., Bodik, R.: Sketching concurrent data structures. In: PLDI (2008)
33. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V.: Combinatorial sketching for finite programs. In: ASPLOS (2006)
34. Sullivan, A.: Automated testing and sketching of Alloy models. Ph.D. thesis, University of Texas at Austin (2017)
35. Sullivan, A., Wang, K., Khurshid, S.: AUnit: a test automation tool for Alloy. In: ICST (2018)

36. Sullivan, A., Wang, K., Khurshid, S., Marinov, D.: Evaluating state modeling techniques in Alloy. In: SQAMIA (2017)
37. Sullivan, A., Wang, K., Zaeem, R.N., Khurshid, S.: Automated test generation and mutation testing for Alloy. In: ICST (2017)
38. Sullivan, A., Zaeem, R.N., Khurshid, S., Marinov, D.: Towards a test automation framework for Alloy. In: SPIN (2014)
39. Alloy Team: http://alloy.mit.edu/alloy/documentation/alloy4-grammar.txt
40. Torlak, E., Chang, F.S.-H., Jackson, D.: Finding minimal unsatisfiable cores of declarative specifications. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 326–341. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68237-0_23
41. Torlak, E., Jackson, D.: Kodkod: a relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_49
42. Wang, K., Sullivan, A., Khurshid, S.: MuAlloy: a mutation testing framework for Alloy. In: ICSE (2018)
43. Wang, K., Sullivan, A., Koukoutos, M., Marinov, D., Khurshid, S.: Systematic generation of non-equivalent expressions for relational algebra. In: ABZ (2018)
44. Yang, Z., Hua, J., Wang, K., Khurshid, S.: Test execution driven synthesis of API sequences with conditionals and loops. In: ICST (2018)
45. Zave, P.: Using lightweight modeling to understand chord. SIGCOMM Comput. Commun. Rev. **42**, 49–57 (2012)

Reals and Hybrid Systems

Abstract State Machines with Exact Real Arithmetic

Christoph Beierle[1] and Klaus-Dieter Schewe[2(✉)]

[1] FernUniversität in Hagen, Hagen, Germany
christoph.beierle@fernuni-hagen.de
[2] Christian-Doppler Laboratory for Client-Centric Cloud Computing, Linz, Austria
kdschewe@acm.org

Abstract. *Type-2 Theory of Effectivity* is a well established theory of computability on infinite strings, which in this paper is exploited to define a data type *Real* as part of the background structure of Abstract State Machines. Real numbers are represented by rapidly converging Cauchy sequences, on top of which standard operations such as addition, multiplication, division, exponentials, trigonometric functions, etc. can be defined. In this way exact computation with real numbers is enabled. Output can be generated at any degree of precision by exploring only sufficiently long prefixes of the representing Cauchy sequences.

1 Introduction

Many computations including the bulk of algorithms in numerical mathematics require the use of real arithmetic, but in general, functions on real numbers are not computable in the classical Turing sense, aka *type-1 computability*. The common approach in numerical mathematics (see e.g. [17]) is to nonetheless specify algorithms using common operations on real numbers, as if these were exactly computable, and to investigate in depth the propagation of errors, if real numbers are replaced by floating point numbers. The alternative is to deal with exact representations of real numbers by Dedekind cuts, Cauchy sequences, nested intervals, etc. to enable exact real arithmetic. There are various approaches and implementations in this direction (see e.g. the survey in [10]). Early work in [19] and in [2] exploit continued fractions and nested intervals, respectively. In [7], implementations in the programming languages Python, C and C++ are described, similarly in [16] for C++. In [3], the integration of exact real arithmetic into the functional programming language Curry is handled, and in [14] the validation of algorithms for exact real arithmetics is investigated. In [15], the semantics of sequential languages with integrated exact real arithmetic is studied. Exact real arithmetics is also supported by theorem provers such as Coq, HOL and PVS.

A precise theoretical foundation of computations with exact real arithmetic is provided by *type-2 theory of effectivity* (TTE) developed by Weihrauch for computable analysis [20]. In a nutshell, TTE is a theory of computability on infinite

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 139–154, 2018.
https://doi.org/10.1007/978-3-319-91271-4_10

strings over an arbitrary alphabet. Prefixes of an input string (or of several input strings) are used to compute output strings, and while processing longer prefixes the already computed outputs persist, i.e. the final output string(s) result from concatenation. This can well be realised using non-termination Turing machines that scan infinite strings on input tapes and continuously produce output on separate output tapes [18]. Using TTE, several numerical problems have been proven to be computable, e.g. TTE has been applied for Jordan decomposition [21] and for computation of specific derivatives [12]. In Sect. 2 we briefly describe some aspects of TTE as we use them.

For many application problems, in particular in connection with scientific computations and hybrid systems, it would be desirable to support specification, refinement and reasoning with exact real arithmetic in common rigorous methods. As emphasised in numerical mathematics [17] the correctness of an algorithm using real numbers requires not only the exploitation of an axiomatic theory of real numbers, as for instance provided for B and Event-B by the RODIN theory plug-in [1], but also reasoning about the precision of the computation, as every implementation can only lead to approximate results. Instead of reasoning about the propagation of errors for a fixed finite representation of real numbers, TTE enables computations up to any desired precision.

In this paper we investigate the integration of exact real arithmetic into Abstract State Machines (ASMs) [6]. Naturally, this requires to precisely define a data type *Real* together with common operations in the background structure of an ASM [4]. This will be done in Sects. 3 and 4. We follow the principle idea employed in [3,13] using a representation of real numbers by rapidly converging Cauchy sequences. Common operations on real numbers such as addition, multiplication, division, exponentials, trigonometric functions, etc. can be defined on Cauchy sequences [20]. As equality of real numbers is not exactly TTE computable and thus cannot be decided, we use a data type *MultiBool* (corresponding to the Curry type Fuzzybool in [3,13]) as functions assigning truth values to rational numbers. The meaning is that for a given precision $r \in \mathbb{Q}$ the equality check yields the associated truth value. This supports the modelling of non-deterministic functions, called multi-functions in [20], which may yield different single values, a set of values or no value at all, e.g. in the case of a non-terminating computation. Multi-functions are an essential ingredient of TTE. In Sect. 5 we show how the datatype *Real* can be used in ASMs focussing on a simple numerical algorithm. This includes a discussion of the handling of streams representing the rapidly converging Cauchy sequences with algorithms that exploit prefixes of these streams, i.e. the algorithm is executed as often as required to obtain a desired precision for the final result. In Sect. 6 we conclude with a brief summary and outlook emphasising among others how to support the integration of exact real arithmetics in common ASM tools such as ASMeta [9] and CoreASM [8].

In [11] Gurevich, Leinders and van den Bussche showed how to capture TTE in general in ASMs. Similar to Turing machine-based computations, arbitrary input streams are considered with the extension that elements in the streams

can stem from arbitrary domains defined in a background structure [4]. A key result is that abstract computable stream queries are exactly defined by continuous functions on streams with respect to a topology defined by generalised Cantor metric considering strings to be closer the longer they coincide on prefixes. Furthermore, all such queries, i.e. the type-2 computations, are captured by stream ASMs. This can be combined with the *Real* background structure we develop in this paper. In case the input Cauchy sequences result from input streams, the exact output will be computed in a streaming way. Hence, while in principle the exact computation of an infinite output stream from an infinite input stream will run forever, the decisive property of TTE guarantees that the ASM will effectively compute the resulting number up to any desired precision in finite time.

2 Type-2 Theory of Effectivity

Type-1 computability theory considers (partial) functions $f : \Sigma^* \to \Sigma^*$ on finite words over an arbitrary alphabet Σ. One of the many equivalent ways to characterise type-1 computable functions is by means of Turing machines. Computations over arbitrary sets such as rational numbers, arrays, or trees require an encoding[1] of elements of these sets by finite words. This notion of computability cannot be applied to functions on real numbers, as these cannot be represented by finite words. The type-2 theory of effectivity extends type-1 computability by taking (possibly partial) functions $f : \Sigma^\omega \to \Sigma^\omega$ on infinite words into account. A computable function is given by a *type-2 machine* transforming infinite sequences into infinite sequences [20].

Such a type-2 machine is a Turing machine M with k *one-way*, read-only input tapes, finitely many, two-way work tapes, and a single *one-way*, write-only output tape. The partial function f_M computed by M is specified by the following two cases for values $y_1, \ldots, y_k \in \Sigma^* \cup \Sigma^\omega$ on the input tapes:

(1) $f_M(y_1, \ldots, y_k) = y_0 \in \Sigma^*$, if M stops with y_0 on the output tape;
(2) $f_M(y_1, \ldots, y_k) = y_0 \in \Sigma^\omega$, if M does not stop and writes y_0 onto the output tape.

Of course, infinite computations cannot be finished in reality, but finite computations on *finite initial parts* of inputs producing *finite initial parts* of outputs can be realised up to any precision. Increasing the precision of a computation requires to extend the initial part of the input that is used for the computation, and to produce a longer output.

It is well known that decimal representations of real numbers cannot be used for exact real arithmetic with TTE. Instead, we may use *rapidly converging*

[1] As remarked in [5] there is no known way to encode structures such as trees or graphs by finite words such that isomorphisms are preserved. ASMs, however, deal directly with structures as states that are preserved under isomorphisms.

Cauchy sequences[2] to represent real numbers, i.e. sequences $r_0, r_1, r_2, \ldots$ of rational numbers $r_i \in \mathbb{Q}$ satisfying $\mid r_k - r_i \mid \leq 2^{-k}$ for all non-negative integers $i, k \in \mathbb{N}$ with $k < i$. Note that this condition implies $\mid r_k - x \mid \leq 2^{-(k+1)}$ for the limit $x = \lim_{i \to \infty} r_i$ of the Cauchy sequence.

Example 1. Let $r_0, r_1, r_2, \ldots$ and $r'_0, r'_1, r'_2, \ldots$ be two rapidly converging Cauchy sequences representing x and x', respectively. Then for all i, k with $k < i$ we obtain

$$\mid r_{k+1} + r'_{k+1} - r_{i+1} - r'_{i+1} \mid \leq \mid r_{k+1} - r_{i+1} \mid + \mid r'_{k+1} - r'_{i+1} \mid \leq 2 \cdot 2^{-(k+1)} = 2^{-k}.$$

Thus, $r_1 + r'_1, r_2 + r'_2, r_3 + r'_3, \ldots$ is a rapidly converging Cauchy sequence representing $x + x'$.

For the definition of addition in Example 1 we had to drop the first element $r_0 + r'_0$ to guarantee the required rapid convergence property. We say that in this case a *look-ahead* of 1 is required for the result. Different values for such a look-ahead are needed for other operations on $\mathbb{R}$ [20]. While the look-ahead for addition is a constant, the look-ahead in general depends on the given arguments.

Example 2. Given the two Cauchy sequences as in Example 1, then $r_m \cdot r'_m$, $r_{m+1} \cdot r'_{m+1}$, $r_{m+2} \cdot r'_{m+2}, \ldots$ is a Cauchy sequence representing the product $x \cdot x'$, where the look-ahead is determined as the smallest number $m \in \mathbb{N}$ such that $|r_0| + 2 \leq 2^{m-1}$ and $|r'_0| + 2 \leq 2^{m-1}$ hold, or equivalently

$$\left(\frac{1}{2}\right)^m \leq \frac{1}{2 \cdot (\max\{|r_0|, |r'_0|\} + 2)}. \tag{1}$$

Here we have (for $k < i$)

$$\begin{aligned} |r_{k+m} \cdot r'_{k+m} - r_{i+m} \cdot r'_{i+m}| = |(r_{k+m} - r_{i+m}) \cdot ((r'_{k+m} - r'_0) + r'_0) + \\ (r'_{k+m} - r'_{i+m}) \cdot ((r_{i+m} - r_0) + r_0)| \leq 2 \cdot 2^{-(k+m)} \cdot 2^{m-1} = 2^{-k}. \end{aligned}$$

In the case of multiplication in Example 2 the look-ahead is not a constant, but it only depends on the first elements of the two given Cauchy sequences. We will later see examples, where the determination of the look-ahead is significantly more complicated.

When comparing two real numbers one has to take into account that the relations $=, \leq, <$ between two real numbers are not exactly computable. Many other relations and functions on the real numbers are likewise not computable [18]. Therefore, TTE employs the crucial notion of a *multi-function* that permits sets of values as results. For instance, the equality on real numbers corresponds to a multi-function $eq : \mathbb{R} \times \mathbb{R} \rightrightarrows \{true, false\}$, which may yield *true*, *false*, both or no result at all. The equality check has to be based on the rational numbers in the Cauchy sequences, so even if the comparison of the k-th elements in the

[2] Note that the condition of rapid convergence is essential here, whereas just using Cauchy sequences would not be sufficient [20].

sequence yields equality, the knowledge that the following elements are close is insufficient for a certain decision, so both truth values are possible. However, if the comparison of the k-th elements in the sequence yields inequality, then this may suffice for a certain decision. In general, a multi-function $f : A \rightrightarrows B$ is interpreted as a partial function $f : A \to 2^B$ into the powerset.

In the sequel we will present a high-level specification as ASM background structure of exact real arithmetic based on TTE and rapidly converging Cauchy sequences. Our choice of using Cauchy sequences is motivated by the observation that this approach is quite intuitive and close to the underlying concepts. For instance, it uses rational numbers at the core of the representation, and functions on real numbers can often be defined via componentwise application of the corresponding function on rational numbers, as illustrated in Examples 1 and 2 above for addition and multiplication. In addition, for many real-valued functions there are well-established definitions via limits of sequences of function applications on rational numbers.

3 An Abstract View on the Data Type Real

We start with introducing a new data type Real. As the rational numbers are a proper subset of the real numbers, there is a function embedding rational numbers into reals

$$realq : Rat \to Real$$

where *Rat* is the type for rational numbers. For illustrating basic computation functions on real numbers, we will consider for instance

$$\begin{array}{ll} add : Real \times Real \rightrightarrows Real & dvd : Real \times Real \rightrightarrows Real \\ neg : Real \rightrightarrows Real & power : Int \times Real \rightrightarrows Real \\ mul : Real \times Real \rightrightarrows Real & exp : Real \rightrightarrows Real \end{array}$$

realising addition, additive inverse, multiplication, division, power, and the exponential function. Further prominent examples of functions on real numbers that are available in the ASM background structure are the transcendental functions like logarithm and the trigonometric functions.

The crucial requirement for multi-functions as non-deterministic functions available in the ASM background structure is that for any arbitrary desired precision that can be given as an additional parameter, the correct result is among the returned results. In order to add a precision parameter when modelling e.g. TTE's multi-function $eq : \mathbb{R} \times \mathbb{R} \rightrightarrows \{true, false\}$, we introduce a new data type *MultiBool* (corresponding to the type Fuzzybool in [3,13]) taking a rational number as precision parameter into account:

$$\textbf{datatype } MultiBool = MBool(Rat \rightrightarrows Bool)$$

The equality relation on real numbers is then modelled by

$$eq : Real \times Real \rightrightarrows MultiBool$$

where for $eq(x, y) = MBool(f)$ the function f is a non-deterministic function mapping rational numbers to Booleans. Evaluating $eq(x, y)$ with respect to precision r is done by the non-deterministic function:

$$selBool : Rat \times MultiBool \rightrightarrows Bool$$
$$selBool(r, MBool(f)) = f(r)$$

It will always be guaranteed that for any r, the correct result is among the results returned by $selBool(r, eq(x, y))$; thus, if $selBool(r, eq(x, y)$ returns a unique result, then this result is the unique correct result. Furthermore, if x and y are Cauchy sequences representing different real numbers $\tilde{x}$ and $\tilde{y}$, then there is a precision $r \in \mathbb{Q}$ with $0 < r < |\tilde{x} - \tilde{y}|$ such that $selBool(r, eq(x, y)$ returns the unique result *false*. If x and y represent the same real number, then $selBool(r, eq(x, y)$ may return both *true* and *false* for any precision $r > 0$. Note that this does not mean that two reals can be both equal and unequal at the same time; it is just not possible to refine *selBool* to a deterministic function since at least in the TTE framework, equality on $\mathbb{R}$ is undecidable, not just when using Cauchy sequences, but for any representation [20, Theorem 4.1.16]. Similarly, the functions

$$lt : Real \times Real \rightrightarrows MultiBool$$
$$leq : Real \times Real \rightrightarrows MultiBool$$
$$isPositive : Real \rightrightarrows MultiBool$$
$$isZero : Real \rightrightarrows MultiBool$$

realise the predicates *less than*, *less or equal*, *is a positive number*, and *is zero* on $\mathbb{R}$ that are also not exactly computable.

4 Function Definitions in ASM Background Structure

In this section we show how the operations listed in the previous section are defined in the background structure. We emphasise the basic operations and predicates on real numbers, and then choose the exponential function as a more sophisticated example.

4.1 Auxiliary Types and Functions

Rational Numbers. We introduce the datatype *Rat* representing rational numbers as quotients of integers such that the denominator is positive. The function *ratn* embeds integers into the rational numbers, and *ratf* normalizes a pair of integers to an element of *Rat* with a positive dominator.

$$\textbf{datatype } Rat = Rat(Int, Nat)$$

$$ratn : Int \rightarrow Rat$$
$$ratn(n) = Rat(n, 1)$$

$$ratf : Int \times Int \rightarrow Rat$$
$$ratf(n, d) = \textbf{if } d > 0 \textbf{ then } Rat(n, d) \textbf{ elsif } d < 0 \textbf{ then } Rat(-n, -d)$$

Addition, subtraction, multiplication, division, and additive and multiplicative inverse on *Rat* are given by the functions *addRat*, *subRat*, *mulRat*, *dvdRat*, *negRat*, and *invRat* defined as expected, e.g.:

$$\begin{aligned} &\mathit{mulRat} : \mathit{Rat} \times \mathit{Rat} \to \mathit{Rat} \\ &\mathit{mulRat}(\mathit{Rat}(n1, d1), \mathit{Rat}(n2, d2)) = \mathit{Rat}(n1 * n2, d1 * d2) \end{aligned}$$

$$\begin{aligned} &\mathit{invRat} : \mathit{Rat} \to \mathit{Rat} \\ &\mathit{invRat}(\mathit{Rat}(n, d)) = \textbf{ if } n \neq 0 \textbf{ then } \mathit{ratf}(d, n) \end{aligned}$$

For comparing rational numbers, we employ the standard mathematical notion and use $=$, $<$, and $\leq$. Furthermore, for convenience, we may also use the standard notations for functions on rational numbers, allowing us to write, e.g., $\frac{x}{2+y}$ instead of $\mathit{mulRat}(x, \mathit{addRat}(2, y))$. Note that as expected, *invRat* is a partial function because $\mathit{invRat}(\mathit{Rat}(n, d))$ is undefined for $n = 0$.

For computing the look-ahead of functions defined on *Real* (cf. Sect. 2) we will use the auxiliary function *minexp*:

$$\begin{aligned} &\mathit{minexp} : \mathit{Rat} \times \mathit{Rat} \to \mathit{Nat} \\ &\mathit{minexp}(x, b) = \textbf{ if } 1 \leq x \textbf{ then } 0 \textbf{ else } \mathit{minexp}(\mathit{dvd}(x, b), b) + 1 \end{aligned}$$

The function call $\mathit{minexp}(x, b)$ returns the smallest natural number n such that $x \geq b^n$ holds. For instance, $\mathit{minexp}(\mathit{Rat}(1, 1000), \mathit{Rat}(1, 2)) = 10$ since $1/1000 \not\geq (1/2)^9$ and $1/1000 \geq (1/2)^{10}$.

MultiBool. The data type *MultiBool* and the function *selBool* have already been given in Sect. 3. In addition, we need logical operators:

$$\begin{aligned} &\mathit{andMB} : \mathit{MultiBool} \times \mathit{MultiBool} \rightrightarrows \mathit{MultiBool} \\ &\mathit{andMB}(a, b) = \mathit{MBool}(\lambda r.(\mathit{selBool}(r, a) \wedge (\mathit{selBool}(r, b)))) \end{aligned}$$

$$\begin{aligned} &\mathit{orMB} : \mathit{MultiBool} \times \mathit{MultiBool} \rightrightarrows \mathit{MultiBool} \\ &\mathit{orMB}(a, b) = \mathit{MBool}(\lambda r.(\mathit{selBool}(r, a) \vee (\mathit{selBool}(r, b)))) \end{aligned}$$

$$\begin{aligned} &\mathit{notMB} : \mathit{MultiBool} \rightrightarrows \mathit{MultiBool} \\ &\mathit{notMB}(a) = \mathit{MBool}(\lambda r.\neg(\mathit{selBool}(r, a))) \end{aligned}$$

Thus, these operators on *MultiBool* are defined to correspond to the usual definitions. While $\mathit{selBool}(r, b)$ might return *true*, *false*, or both Boolean values, the correct value is always among the returned results; note that for any precision r, this property is preserved by the logical operations on *MultiBool*, e.g., if b is a conjunction like $\mathit{andMB}(b1, b2)$.

Intervals. Intervals of rational numbers are used for checking the required precision when computing and comparing real numbers and thus provide a basis for realising not exactly computable functions. An interval is valid only if its lower bound is less or equal to its upper bound.

$$\textbf{datatype } \mathit{Interval} = \mathit{Interval}(\mathit{Rat}, \mathit{Rat})$$

$$\begin{aligned} &\mathit{isValid} : \mathit{Interval} \to \mathit{Bool} \\ &\mathit{isValid}(\mathit{Interval}(a, b)) = a \leq b \end{aligned}$$

The computation of relations on real numbers will be reduced to three non-deterministic functions on intervals. The function *maybeZero* yields *true* if 0 is in the interval, and it yields *false* if the interval contains a number that is not equal to 0. The function *maybePos* yields *true* if the given interval contains a positive number, and it yields *false* if the interval contains a number that is not positive. The function *maybeNeg* yields *true* if the given interval contains a negative number, and yields *false* if the interval contains a non-negative number.

$$
\begin{aligned}
&maybeZero : Interval \rightrightarrows Bool\\
&maybeZero(Interval(a,b)) = \textbf{ if } a \leq 0 \wedge 0 \leq b \textbf{ then } true\\
&maybeZero(Interval(a,b)) = \textbf{ if } a < 0 \vee 0 < b \textbf{ then } false\\
&maybePos : Interval \rightrightarrows Bool\\
&maybePos(Interval(a,b)) = \textbf{ if } 0 < b \textbf{ then } true\\
&maybePos(Interval(a,b)) = \textbf{ if } a \leq 0 \textbf{ then } false\\
&maybeNeg : Interval \rightrightarrows Bool\\
&maybeNeg(Interval(a,b)) = \textbf{ if } a < 0 \textbf{ then } true\\
&maybeNeg(Interval(a,b)) = \textbf{ if } 0 \leq b \textbf{ then } false
\end{aligned}
$$

Note that these functions are indeed proper multi-functions. For instance, while $maybeZero(Interval(\frac{1}{4}, \frac{1}{2}))$ yields the unique value *false*, $maybeZero(Interval(0, \frac{1}{2}))$ yields both *true* and *false*. Also $maybePos(Interval(0, \frac{1}{2}))$ yields both *true* and *false*, while $maybeNeg(Interval(0, \frac{1}{2}))$ yields the unique value *false*. For $I = Interval(\frac{-1}{2}, \frac{1}{2}))$, the three function calls $maybeZero(I)$, $maybePos(I)$, and $maybeNeg(I)$ all yield both *true* and *false*.

4.2 Representing Real Numbers and Basic Functions on Them

The approach of representing real numbers as Cauchy sequences is achieved by the datatype *Real*:

$$\textbf{datatype } Real = Cauchy(Int \rightarrow Rat)$$

The embedding of *Rat* into *Real* is then given by:

$$
\begin{aligned}
&realq : Rat \rightrightarrows Real\\
&realq(a) = Cauchy(\lambda r.a)
\end{aligned}
$$

For addition, we employ componentwise operation, observing that the look-ahead is 1 (cf. Sect. 2); subtraction and additive inverse are also easily defined:

$$
\begin{aligned}
&add : Real \times Real \rightrightarrows Real\\
&add(a,b) = Cauchy(\lambda k.get(a, m+1) + get(b, m+1))\\
&sub : Real \times Real \rightrightarrows Real\\
&sub(a,b) = add(a, neg(b))\\
&neg : Real \rightrightarrows Real\\
&neg(a) = Cauchy(\lambda k. - get(a,k))\\
&get : Real \times Int \rightrightarrows Rat\\
&get(Cauchy(f), k) = f(k)
\end{aligned}
$$

Using the auxiliary function *minexp* (Sect. 4.1), the computation of the look-ahead m for multiplication *mul* as achieved by the auxiliary function *lahmul* that mirrors exactly the condition given by (1):

$$
\begin{aligned}
&\mathit{mul} : \mathit{Real} \times \mathit{Real} \rightrightarrows \mathit{Real}\\
&\mathit{mul}(a,b) = \textbf{let } x = \mathit{max}(\mathit{abs}(\mathit{get}(a,0))(\mathit{abs}(\mathit{get}(b,0)))) \textbf{ in}\\
&\qquad\qquad \textbf{let } m = \mathit{lahmul}(x) \textbf{ in } \mathit{Cauchy}(\lambda k.\mathit{get}(a,k+m) \times \mathit{get}(b,k+m))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{lahmul} : \mathit{Rat} \to \mathit{Nat}\\
&\mathit{lahmul}(x) = \mathit{minexp}(\tfrac{1}{2\times(x+2)}, \tfrac{1}{2})
\end{aligned}
$$

When computing the multiplicative inverse of a real number, we can use the function for the multiplicative inverse on rational numbers, but there is a particular subtlety to be taken into account. In the function definition for rational numbers, we can easily check that $x \neq 0$ for an argument x, and leave the function application undefined for $x = 0$; cf. the definition of *invRat* in Sect. 4.1. However, for a real value $x \in \mathbb{R}$, the relation $x \neq 0$ is not exactly computable. Before applying *invRat* to the elements of a Cauchy sequence representing x, we therefore have to determine an appropriate look-ahead m:

$$
\begin{aligned}
&\mathit{inv} : \mathit{Real} \rightrightarrows \mathit{Real}\\
&\mathit{inv}(a) = \textbf{let } m = \mathit{lahinv}(a) \textbf{ in } \mathit{Cauchy}(\lambda k.\tfrac{1}{\mathit{get}(a,k+m)})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{lahinv} : \mathit{Real} \to \mathit{Nat}\\
&\mathit{lahinv}(a) = \textbf{let } h = \lambda z,k\,.\,\textbf{if } z \leq \mathit{get}(a,k) \textbf{ then } k \textbf{ else } h(\tfrac{z}{2}, k+1)\\
&\qquad\qquad \textbf{in } 2 \times h(3,0)
\end{aligned}
$$

Thus, if *inv* is applied to $r_0, r_1, r_2, \ldots$, the look-ahead for *inv* is $m = 2n$ where n is the smallest natural number n such that $3 \times 2^{-n} \leq |r_n|$ holds (cf. [20, Theorem 4.3.2]). Note that also *inv* is a partial function: If for $x \in \mathbb{R}$ represented by $r_0, r_1, r_2, \ldots$ the relationship $x \neq 0$ holds, then the condition $z \leq \mathit{get}(a,k)$ in the recursive calls of the auxiliary function h will eventually be true, causing the call of *lahinv* as well as the call of *inv* to terminate and to yield the correct result. If $x = 0$ holds, then the call of *inv* will not terminate since equality on reals is undecidable [20].

Using *inv*, it is now easy to define division on *Real* as ${}^{a}/_{b} = ab^{-1}$ holds:

$$
\begin{aligned}
&\mathit{dvd} : \mathit{Real} \times \mathit{Real} \rightrightarrows \mathit{Real}\\
&\mathit{dvd}(a,b) = \mathit{mul}(a, \mathit{inv}(b))
\end{aligned}
$$

The basic relations on $\mathbb{R}$ discussed in Sect. 3 are implemented by reducing them to the question of checking whether a number is 0 or positive:

$$
\begin{aligned}
&\mathit{eq} : \mathit{Real} \times \mathit{Real} \rightrightarrows \mathit{MultiBool}\\
&\mathit{eq}(x,y) = \mathit{isZero}(\mathit{sub}(y,x))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{lt} : \mathit{Real} \times \mathit{Real} \rightrightarrows \mathit{MultiBool}\\
&\mathit{lt}(x,y) = \mathit{isPositive}(\mathit{sub}(y,x))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{leq} : \mathit{Real} \times \mathit{Real} \rightrightarrows \mathit{MultiBool}\\
&\mathit{leq}(x,y) = \mathit{notMB}(\mathit{isPositive}(\mathit{sub}(x,y)))
\end{aligned}
$$

The two functions *isZero* and *isPositive* are in turn reduced to the corresponding functions on intervals. To do so, the resulting object of type *MultiBool* contains a function that for any precision uses an interval realising this precision with respect to the given x of type *Real*.

$$
\begin{aligned}
&\mathit{isPositive} : \mathit{Real} \rightrightarrows \mathit{MultiBool}\\
&\mathit{isPositive}(x) = \mathit{MBool}(\lambda r.\mathit{maybePos}(\mathit{toInterval}(r,x)))\\
&\mathit{isZero} : \mathit{Real} \rightrightarrows \mathit{MultiBool}\\
&\mathit{isZero}(x) = \mathit{MBool}(\lambda r.\mathit{maybeZero}(\mathit{toInterval}(r,x)))\\
&\mathit{isNegative} : \mathit{Real} \rightrightarrows \mathit{MultiBool}\\
&\mathit{isNegative}(x) = \mathit{MBool}(\lambda r.\mathit{maybeNeg}(\mathit{toInterval}(r,x)))
\end{aligned}
$$

Given numbers r of type *Rat* and x of type *Real*, the auxiliary function *toInterval* determines an interval containing the real number in $\mathbb{R}$ represented by x and approximating that number with precision r.

$$
\begin{aligned}
&\mathit{toInterval} : \mathit{Rat} \times \mathit{Real} \rightrightarrows \mathit{Interval}\\
&\mathit{toInterval}(r,x) = \textbf{let } y = \mathit{approx}(\tfrac{r}{2},x) \textbf{ in } \mathit{Interval}(y-\tfrac{r}{2},\, y+\tfrac{r}{2})\\
&\mathit{approx} : \mathit{Rat} \times \mathit{Real} \rightrightarrows \mathit{Rat}\\
&\mathit{approx}(r,x) = \mathit{get}(x, \mathit{prec}(r))\\
&\mathit{prec} : \mathit{Rat} \to \mathit{Int}\\
&\mathit{prec}(r) = \textbf{ if } 0 < r \textbf{ then } \mathit{minexp}(r, \tfrac{1}{2})
\end{aligned}
$$

The approximation given by *approx* determines the smallest natural number n such that $(1/2)^n \leq r$ holds in order to determine the position in the Cauchy sequence to be used for obtaining the bounds of the interval.

4.3 Further Functions

On the basis of the data type *Real* providing a representation for the real numbers in the sense that any given precision can be obtained, together with the set of background functions defined so far, further functions on real numbers can be defined. As a core function providing the key for defining many other functions on $\mathbb{R}$ we will present the realization of the exponential function e^x. For this, we will use the power function x^n which we define first.

Similarly as for addition or multiplication, computing the nth power x^n of a real number x and an integer n can be done by componentwise computations on the representing Cauchy sequence. For negative exponents, the equality $x^{-n} = (1/x)^n$ is used, and for $n > 1$, the required look-ahead is $(n-1)$-times the look-ahead for multiplication of x with itself:

$$
\begin{aligned}
&\mathit{power} : \mathit{Int} \times \mathit{Real} \rightrightarrows \mathit{Real}\\
&\mathit{power}(n,x) = \textbf{ if } n = 0 \textbf{ then } \mathit{realq}(\mathit{ratn}(1))\\
&\mathit{power}(n,x) = \textbf{ if } n = 1 \textbf{ then } x\\
&\mathit{power}(n,x) = \textbf{ if } n < 0 \textbf{ then } \mathit{power}(-n, \mathit{inv}(x))\\
&\mathit{power}(n,x) = \textbf{ if } n > 0 \textbf{ then let } \mathit{lah} = (n-1) * \mathit{lahmul}(\mathit{abs}(\mathit{get}(x,0))\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in } \mathit{Cauchy}(\lambda k.\mathit{power}(n, \mathit{get}(x, \mathit{lah}+k)))
\end{aligned}
$$

In many cases, real numbers or functions on real numbers are defined using a power series. For instance, for the exponential function we have $e^x = \sum_{k=0}^{\infty} \frac{1}{k!}\, x^k$. In general, a function given by a power series $\sum_{k=0}^{\infty} a_k\ x^k$ is well-defined for any $|x| < R_K$ where $R_K = (\limsup_{n\to\infty} \sqrt[n]{|a_n|})^{-1}$ is the radius of convergence. The computation of the power series for $x \in \mathbb{R}$ can be approximated by a finite sum $a_0 y^0 + \ldots a_N y^N$ with $y \in \mathbb{Q}$. The error of this approximation depends on the coefficients a_k, on the number of summands N, and on $|x - y|$. In order to eliminate the dependence on the a_k, we will require $|a_k| \leq 1$ for all $k \geq 0$. This restriction is satisfied for many power series expressions defining functions like exp, sin, cos, etc. Furthermore, because it implies $R_K \geq 1$, for any $r \in \mathbb{Q}$ with $0 < r < 1$, the power series converges for all $x \in \mathbb{R}$ with $|x| \leq r$. If a is a sequence $(a_k)_{k\geq 0}$ of rational numbers with $|a_k| \leq 1$ and r is a rational number with $0 < r < 1$, then a call $powerser(a, r, x)$ of the auxiliary function

$$powerser : (Int \to Rat) \times Rat \times Real \rightrightarrows Real$$

computes the power series for $\tilde{x}$ if x represents $\tilde{x} \in \mathbb{R}$ and $|\tilde{x}| \leq r$. The argument r is used in the estimation of the error of the approximation and in the determination of the number of summands needed in order to meet the rapid convergence condition (cf. [20, Chap. 4.3], [14]); a smaller r means fewer summands, but also a smaller function domain. Thus, by choosing $r = 1/2$, the function

$$\begin{aligned} &exp1 : Real \rightrightarrows Real \\ &exp1(x) = powerser((\lambda k.\tfrac{1}{k!}),\ \tfrac{1}{2},\ x) \end{aligned}$$

can be used for computing the exponential function for any values $|x| \leq 1/2$. A straightforward method to extend the computation of e^x to values $|x| > 1/2$ which is also used in e.g. [14] is to exploit the property $e^x = \left(e^{\frac{x}{m}}\right)^m$ and to choose m to be an integer such that $|x/m| \leq 1/2$ This allows us to use *exp1* for the inner and *power* for the outer exponentiation in order to obtain a full realization of the exponential function *exp* given by the following:

$$\begin{aligned} &exp : Real \rightrightarrows Real \\ &exp(x) = \textbf{let}\ m = 2 \times estimate(abs(x)) + 1 \\ &\qquad\qquad \textbf{in}\ power(m, exp1(mul(\tfrac{1}{m}, x))) \\ &estimate : Real \rightrightarrows Int \\ &estimate(x) = floor(get(x, 2) + \tfrac{1}{2}) \\ &abs : Real \rightrightarrows Real \\ &abs(x) = Cauchy(\lambda k\,.\,|get(x, k)|) \end{aligned}$$

where for a rational number x, the auxiliary function $floor : Rat \to Int$ returns the largest integer less or equal to x. Having available the exponential function on real numbers, further functions can defined using *exp*, e.g. a general root function, the trigonometric functions, and many others.

To give a further illustration that the functions realzing exact real arithmetic in the framework of TTE are typically multi-functions, assume that *sqrt2* is of type *Real*, denoting $\sqrt{2}$. Assume further that we have a binary function *dec* that takes an element x of type *Real* and a natural number k and returns the value of x as a string containing k decimal places, such that the returned decimal representation is correct with precision 10^{-k}, but without any rounding being applied; these functions are also available in the Curry implementation described in [3,13]. Then the function call $dec(sqrt2, 20)$ will give the unique result 1.41421356237309504880. On the other hand, the decimal representation obtained by *dec* for the multiplication of *sqrt2* with itself and for $k = 20$ yields two results, 1.99999999999999999999 and 2.00000000000000000000, which are finite prefixes of the exactly two different infinite decimal representations of the real value 2, illustrating that $dec : Real \times Nat \rightrightarrows String$ is a multi-function.

5 Computations with Exact Real Arithmetics

In this section we briefly show how to use the background structure defined in the previous sections in an ASM. We use a simple algorithm for the solution of an inhomogeneous set of linear equations as example. Basically, the use of the data type *Real* together with its operations is applied in the same way as an other data type such as *Int* or *Rat*. We simplify the presentation for functions on *Real*, writing, e.g., $A \cdot B$ for $mul(A, B)$. We will discuss a streaming interpretation of the algorithm, i.e., input data is assumed to be represented by Cauchy sequences that can be read one element after the other from input streams.

The signature of our ASM comprises function symbols A, k, B, r, C and D with arities $3, 0, 2, 0, 1$ and 2, respectively. k denotes an integer constant with the meaning that the system of linear equations comprises $k+1$ indeterminates. A and B represent the *Real*-valued input matrix and vector, for an integer counter r (initially 0) $A(i, j, r)$ is the entry in row i and column j in round r, and likewise $B(i, r)$ is the i'th entry in the righ-hand-side vector in round r. C represents a *Real*-valued vector taking a solution of the inhomogeneous system, while the first $k+1$ columns of D represent a basis for the corresponding homogeneous system (excluding zero vectors). The algorithm first creates an upper triangular matrix, then a diagonal matrix to keep the determination of C and D as simple as possible. The rule of the ASM is given as follows:

if $r < k$
then if $A(r, r, r) \neq 0$
  then
    forall i, j **with** $0 \leq i \leq k \wedge 0 \leq j \leq k$
      if $i \leq r \vee j \leq r - 1$
      then $A(i, j, r + 1) := A(i, j, r)$ $B(i, r + 1) := B(i, r)$
    forall i **with** $r + 1 \leq i \leq k$ $A(i, r, r + 1) := 0$
    forall i **with** $r + 1 \leq i \leq k$
      forall j **with** $r + 1 \leq j \leq k$

$$A(i,j,r+1) := A(i,j,r) - \frac{A(i,r,r)}{A(r,r,r)} \cdot A(r,j,r)$$
$$B(i,r+1) := B(i,r) - \frac{A(i,r,r)}{A(r,r,r)} \cdot B(r,r)$$
$$r := r+1$$

else **if** $\exists i.i > r \wedge A(i,r,r) \neq 0$
then choose i **with** $i > r \wedge A(i,r,r) \neq 0$
forall j **with** $r \leq j \leq k$
$A(i,j,r) := A(r,j,r)$ $A(r,j,r) := A(i,j,r)$
$B(i,r) := B(r,r)$ $B(r,r) := B(i,r)$
else $r := r+1$
forall i,j **with** $0 \leq i \leq k \wedge 0 \leq j \leq k$
$A(i,j,r+1) := A(i,j,r)$ $B(i,r+1) := B(i,r)$

if $k \leq r < 2k$
then if $A(2k-r,2k-r,r) \neq 0$
then forall i,j **with** $0 \leq i \leq k \wedge 0 \leq j \leq k$
if $i > 2k-r \vee i \geq j \vee j > 2k-r$
then $A(i,j,r+1) := A(i,j,r)$ $B(i,r+1) := B(i,r)$
forall i **with** $0 \leq i \leq 2k-1-r$ $A(i,2k-r,r+1) := 0$
forall i **with** $0 \leq i \leq 2k-1-r$
forall j **with** $i < j \leq 2k-1-r$
$$A(i,j,r+1) := A(i,j,r) - \frac{A(i,2k-r,r)}{A(2k-r,2k-r,r)} \cdot A(2k-r,j,r)$$
$$B(i,r+1) := B(i,r) - \frac{A(i,2k-r,r)}{A(2k-r,2k-r,r)} \cdot B(2k-r,r)$$
$r := r+1$
else **forall** i,j **with** $0 \leq i \leq k \wedge 0 \leq j \leq k$
$A(i,j,r+1) := A(i,j,r)$ $B(i,r+1) := B(i,r)$
$r := r+1$

if $r = 2k$
then forall i **with** $0 \leq i \leq k$
if $A(i,i,r) \neq 0$
then $C(i) := \frac{B(i,r)}{A(i,i,r)}$
forall j **with** $0 \leq j \leq k$ $D(j,i) := 0$
else **if** $B(i,r) = 0$ **then** $C(i) := 0$
forall j **with** $1 \leq i \leq k \wedge 1 \leq j \leq k$
if $j = i$ **then** $D(j,i) := 1$
if $j > i$ **then** $D(j,i) := 0$
if $j < i$ **then**
if $A(j,j,r) \neq 0$
then $D(j,i) := -\frac{\sum_{j'=j+1}^{i} A(j,j',r)}{A(j,j,r)}$
else $D(j,i) := 0$

In this specification the fact that the matrix and vector entries are real numbers remains transparent. For exact real arithmetic we can assume that for $r = 0$ we have input streams providing members of rapidly converging Cauchy sequences. Therefore, in order to be able to access still incoming members of the sequences, we used copies in every round r for the results to separate the incoming stream for the operations from the streams computed by the operations, where the look-ahead has to be taken into account. Combining the view of incoming streams with the algorithm above actually means that the algorithm runs arbitrarily many times computing as many members of the result Cauchy sequences as desired. In this way the computation adopts a two-dimensional structure, as for each i the algorithm will be executed on finite prefixes of the input to determine the ith elements in the Cauchy sequences result, and with increasing value of i the precision of the result increases. In principle this can run forever. However, the specification in the background enables to reason about the desired level of precision. Of course, the higher the precision is, the higher will also be the complexity of the algorithm. Thus, the background specification enables also the analysis, up to which level of precision a computation with the specified algorithm is tractable.

6 Conclusions and Further Work

Using the concept of background structure for Abstract State Machines, we specified a data type *Real* together with basic operations and predicates, which support exact real arithmetic based on TTE [20]. We exploited representations of real numbers by rapidly converging Cauchy sequences and the concept of multi-functions required for TTE-based exact real arithmetic. We further showed how to specify the important exponential function. This can be naturally extended to many other mathematical functions on real numbers such as logarithms, trigonometric functions and many more.

Such operations and functions are important for many algorithms in science and engineering, in particular, when arbitrarily high precision is required. We illustrated this for the specification of an ASM for the solution of systems of linear equations. We further sketched that the approach can be combined with streaming computations, where higher precision of the result can be obtained by longer prefixes of the input streams, i.e. the Cauchy sequences. However, the computation might yield more than one result, but the correct result will be among the obtained results. In doing so, reasoning of the precision of specified computations will be enabled, which is highly important for scientific computing. It opens a perspective for system verification that so far is largely neglected in the field of rigorous methods.

Exploiting the experience with implementations of TTE, e.g. the implementation in Curry [3], the integration of exact real arithmetic as defined in this paper into common ASM tools such as ASMeta [9] or Core-ASM [8] appears as a straightforward exercise. The TTE-based definition of the data type *Real* provides also an alternative for the support of real numbers in other rigorous methods, e.g. for the RODIN theory plug-in for the Event-B [1].

References

1. Abrial, J.R., Butler, M., Hallerstede, S., Leuschel, M., Schmalz, M., Voisin, L.: Proposals for mathematical extensions for Event-B. Technical report (2010). http://deploy-eprints.ecs.soton.ac.uk/216/
2. Aït Ameur, Y.: Refinement of rational end-points real numbers by means of floating-point numbers. Sci. Comput. Program. **33**(2), 133–162 (1999)
3. Beierle, C., Lelitko, U.: On a high-level approach to implementing exact real arithmetic in the functional logic programming language Curry. In: Hanus, M., Rocha, R. (eds.) WLP 2013. LNCS (LNAI), vol. 8439, pp. 48–64. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08909-6_4
4. Blass, A., Gurevich, Y.: Background of computation. Bull. EATCS **92**, 82–114 (2007)
5. Blass, A., Gurevich, Y., Shelah, S.: Choiceless polynomial time. Ann. Pure Appl. Logic **100**(1–3), 141–187 (1999)
6. Börger, E., Stärk, R.F.: Abstract State Machines. A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-642-18216-7
7. Briggs, K.: Implementing exact real arithmetic in Python, C++ and C. Theor. Comput. Sci. **351**(1), 74–81 (2006)
8. Farahbod, R., Gervasi, V., Glässer, U.: CoreASM: an extensible ASM execution engine. Fundamenta Informaticae **77**(1–2), 71–103 (2007)
9. Gargantini, A., Riccobene, E., Scandurra, P.: A metamodel-based language and a simulation engine for abstract state machines. J. Univ. Comput. Sci. **14**(12), 1949–1983 (2008)
10. Gowland, P., Lester, D.: A survey of exact arithmetic implementations. In: Blanck, J., Brattka, V., Hertling, P. (eds.) CCA 2000. LNCS, vol. 2064, pp. 30–47. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45335-0_3
11. Gurevich, Y., Leinders, D., Van den Bussche, J.: A theory of stream queries. In: Arenas, M., Schwartzbach, M.I. (eds.) DBPL 2007. LNCS, vol. 4797, pp. 153–168. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75987-4_11
12. Hoyrup, M., Rojas, C., Weihrauch, K.: Computability of the Radon-Nikodym derivative. Computability **1**(1), 3–13 (2012)
13. Lelitko, U.: Realisation of exact real arithmetic in the functional logic programming language Curry. Diploma thesis, Department of Computer Science, University of Hagen, Germany (2013, in German)
14. Lester, D.R., Gowland, P.: Using PVS to validate the algorithms of an exact arithmetic. Theor. Comput. Sci. **291**(2), 203–218 (2003)
15. Marcial-Romero, J.R., Hötzel Escardó, M.: Semantics of a sequential language for exact real-number computation. Theor. Comput. Sci. **379**(1–2), 120–141 (2007)
16. Müller, N.T.: The iRRAM: exact arithmetic in C++. In: Blanck, J., Brattka, V., Hertling, P. (eds.) CCA 2000. LNCS, vol. 2064, pp. 222–252. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45335-0_14
17. Scott, L.R.: Numerical Analysis. Princeton University Press, Princeton (2011)
18. Tavana, N., Weihrauch, K.: Turing machines on represented sets, a model of computation for analysis. Logical Methods Comput. Sci. **7**(2), 1–21 (2011)

19. Vuillemin, J.: Exact real computer arithmetic with continued fractions. IEEE Trans. Comput. **39**(8), 1087–1105 (1990)
20. Weihrauch, K.: Computable Analysis - An Introduction. Texts in Theoretical Computer Science - An EATCS Series. Springer, Heidelberg (2000). https://doi.org/10.1007/978-3-642-56999-9
21. Weihrauch, K., Jafarikhah, T.: Computable Jordan decomposition of linear continuous functionals on C[0;1]. Logical Methods Comput. Sci. **10**(3), 1–13 (2014)

Proof-Based Approach to Hybrid Systems Development: Dynamic Logic and Event-B

Guillaume Dupont(✉), Yamine Aït-Ameur, Marc Pantel, and Neeraj Kumar Singh

INPT-ENSEEIHT/IRIT, University of Toulouse, Toulouse, France
{guillaume.dupont,yamine,marc.pantel,nsingh}@enseeiht.fr

Abstract. The design of hybrid systems controllers requires one to handle both discrete and continuous functionalities in a single development framework. In this paper, we propose the design and verification of such controllers using a *correct-by-construction* approach. We use proof-based formal methods to model and verify the required safety properties of the given controllers. Both Event-B with Rodin, and hybrid programs and dynamic differential logic with KeYmaera are experimented on a common case study related to the modelling of a car controller. Finally, we discuss the lessons learnt from these experiments and draw the first steps towards a generic method for modelling hybrid systems in Event-B.

Keywords: Hybrid systems · Event-B · Hybrid programs
Differential dynamic logic · Proofs · Refinement

1 Introduction

Hybrid systems are present in many safety-critical applications. Thus, the formal verification of hybrid systems is a key issue in system engineering. Contrary to classical discrete systems, the formal specification and verification of hybrid systems require taking into account continuous features like differential equations to characterise plant behaviours and the appropriate logic and proof system. Several research works addressed the formal verification of hybrid systems [4]. Hybrid model-checking, proof based approaches, program analysis and simulation have been proposed. These approaches consider a hybrid system as the tight integration of discrete and concrete features defining models for controllers and for the plant to be controlled.

However, addressing formal verification at the model level allows for the abstraction of some implementation details, and in particular floating point arithmetic. The verification of the defined models and the synthesis of controllers guarantees the satisfaction of system requirements independently of any implementation. Only implementation requirements like floating point computation will remain to be addressed once code is obtained from the verified models.

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 155–170, 2018.
https://doi.org/10.1007/978-3-319-91271-4_11

Our paper deals with correct-by-construction approaches with refinement and proof-based techniques. We propose to handle the integration of continuous and discrete behaviours in Event-B [1] and in the Rodin [2] platform to develop hybrid systems models. This approach requires the modelling of continuous mathematical concepts which are not currently available in Event-B. For this purpose we use the *Theory* plug-in developed for Rodin [13] to define a theory for continuous functions and differential equations that we need to handle in our Event-B models.

In order to position our work, we select the approach of Platzer as a second formal development technique. This choice is motivated by the defined proof-based approach and the availability of a tool. Hence, we show how dynamic differential logic [15] and KeYmaeara [16] are set up for the same objective as ours. A case study of a car controller, borrowed from Platzer's work, is used to illustrate both approaches. As a second step, we present a generalisation of our approach with Event-B in order to reduce the effort needed for feasibility proofs during model instantiation.

This paper is organised as follows. The next section presents the case study we have chosen to illustrate our approach. Event-B and dynamic differential logic are summarised in Sect. 3 and their use for the development of the case study is described in Sect. 4. The methodological lessons learnt from these developments are discussed in Sect. 5. Finally, the last section is devoted to concluding remarks and a research agenda.

2 Case-Study

The chosen case study deals with a *stop sign controller* proposed by Quesel et al. in [17] (Sect. 5.3). It consists in modelling a car controller that automatically stops a car at a stop-sign (SP).

Behavioural Requirements. The car is modelled through its horizontal position and behaves according to three modes, each of which corresponds to a particular *acceleration*. The given accelerations are constant and, as a matter of simplification, wrap every potential physical phenomena (air and road friction for instance). These modes are defined as follows.

- **Accelerating:** the car increases its velocity. In this case, the associated acceleration, denoted by A, is positive.
- **Braking:** the velocity of the car decreases. In this case, the associated acceleration is $-B$, where B denotes the braking power (positive).
- **Stabilizing:** the velocity of the car does not change. In this case, the associated acceleration is 0.

The system is modelled by its position (p), velocity (v) and acceleration (a), which evolve according to the differential equation: $\dot{p} = v, \dot{v} = a$, where the dot stands for *time derivative*.

At initialisation, the system is in *stabilizing* mode. The car is given an arbitrary initial position and velocity denoted as p_0 and v_0, respectively.

Safety Requirements. The system shall observe two invariant properties.

- **SAF1.** The velocity of the car cannot be negative.
- **SAF2.** The position of the car never exceeds the stop sign position SP.

Note that the two safety requirements are of different nature. **SAF1** has a purely physical origin whereas **SAF2** is a behavioural system requirement.

3 Two Proof-Based Methods

To address the case study presented in Sect. 2 we considered two different approaches. A first one is based on differential logic ($\mathtt{d}\mathcal{L}$) and KeYmaera to express and prove an hybrid controller, and the second one uses Event-B and Rodin to express the system using events and invariants.

3.1 Hybrid-Programs/Dynamic Logic with KeYmaera

The seminal work of [15] led to the definition of a rigorous method to model controllers for hybrid systems integrating both continuous and discrete behaviours. The approach revolves around three components: hybrid programs to model system behaviours, differential dynamic logic $\mathtt{d}\mathcal{L}$ to specify properties and the KeYmaera tool that supports system behaviour specification and verification using a theorem prover for $\mathtt{d}\mathcal{L}$. Below we give the required information to understand the development conducted in this paper. More details can be found in the abundant bibliography published by the authors.

Modelling: Hybrid Programs. According to Platzer [15], hybrid programs (HP) define a program notation for hybrid systems. These HP offer a structural decomposition to support $\mathtt{d}\mathcal{L}$ reasoning. Additionally to classical programs, HP support the definition of variables that evolve along a differential equation. Some basic constructs of such programs are discrete assignments (`:=`), sequential composition (`;`), choice ($\cup$), state assertion or condition (`?H`), iteration (`*`) and continuous evolution of a continuous variable along differential equation (`x'=t & H`) in an evolution domain H (optional).

Property Specification and Verification: Differential Dynamic Logic. Differential dynamic logic $\mathtt{d}\mathcal{L}$ is a first order logic with built-in statements dealing with hybrid systems. Similarly to first order logic which supports reasoning on classical programs using weakest precondition or substitution calculi, $\mathtt{d}\mathcal{L}$ supports reasoning on hybrid programs. Operators of first order logic together with the modalities [] and $\langle\rangle$ are defined in $\mathtt{d}\mathcal{L}$. $[\alpha]\,\phi$ and $\langle\alpha\rangle\,\phi$ assert respectively that ϕ holds after all runs and after at least one run of the HP α.

For example, $Init \longrightarrow [plant]\,(req)$ defines an uncontrolled system where $plant$ is a HP, $Init$ is the $\mathtt{d}\mathcal{L}$ predicate characterising the initial state and req is a $\mathtt{d}\mathcal{L}$ predicate defining a safety property. $Init \longrightarrow [(ctrl; plant)^*]\,(req)$

defines another system where the HP is made of instantaneous control events *ctrl* sequentially composed with the *plant* HP with a possible modification of its behaviour. The definitions of *ctrl* and *plant* are built using the constructs of HP, and *req* is again a safety property. We will use this template to model our case study.

Tool: KeYmaera. KeYmaera [16] is the theorem prover associated with differential logic. It supports proof of properties of hybrid programs. Additionally to the classical proof rules associated to first order logic, KeYmaera implements a set of specific proof rules defined for $\mathtt{d}\mathcal{L}$, including differential invariants, differential auxiliary and ODE-related tactics. In particular, differential invariants give an induction proof principle on differential equations.

3.2 Event-B with Rodin

Event-B [1] is a *correct-by-construction* approach to design an abstract model and a series of refined models for developing any large and complex system.

Modelling: Event-B Machines. The Event-B language uses set theory and first order logic. It has two main components, *context* and *machine*, to characterise systems. A *context* describes the static structure of a system using *carrier sets* s, *constants* c, *axioms* $A(s, c)$ and *theorems* $T_c(s, c)$, and a *machine* describes the dynamic structure of a system using *variables* v, *invariants* $I(s, c, v)$, *theorems* $T_m(s, c, v)$, *variants* $V(s, c, v)$ and *events* evt. A list of events can be used to model possible system behaviour to modify the state variables by providing appropriate *guards* in a *machine*. A set of *invariants* and *theorems* can be used to represent relevant properties to check the correctness of the formalized behaviour. To define the convergence properties, *variants* can be used.

Refinement of Event-B Models. Refinement decomposes a model (thus a transition system) into another transition system containing more design decisions while moving from an abstract level to a less abstract one. It supports the modelling of a system gradually by introducing safety properties at various refinement levels. New variables and new events may be introduced. These refinements preserve the relation between the refining model and the refined one while introducing new events and variables to specify more concrete behaviour of the system. The defined abstract and concrete state variables are linked by introducing *gluing invariants*.

Property Verification: Proof Obligations (PO). To verify the correctness of an Event-B model (machine or refinement) the generated POs (issued from the calculus of substitutions) need to be proved. A proof system allows to prove the POs. The main proof obligations are listed in Table 1, in which the prime notation is used to denote the value of a variable after an event is triggered.

These POs require to demonstrate that the theorems hold, each event preserves the invariant (inductive), each event can be triggered (feasibility) and if a variant is declared, it shall decrease.

Table 1. Proof obligations

Theorems	$A(s,c) \Rightarrow T_c(s,c)$ $A(s,c) \wedge I(s,c,v) \Rightarrow T_m(s,c,v)$
Invariant preservation	$A(s,c) \wedge I(s,c,v) \wedge G(s,c,v,x) \wedge BA(s,c,v,x,v') \Rightarrow I(s,c,v')$
Event feasibility	$A(s,c) \wedge I(s,c,v) \wedge G(s,c,v,x) \Rightarrow \exists v'.BA(s,c,v,x,v')$
Variant progress	$A(s,c) \wedge I(s,c,v) \wedge G(s,c,v,x) \wedge BA(s,c,v,x,v') \Rightarrow V(s,c,v') < V(s,c,v)$

Regarding refinement, two more relevant proof obligations need to be discharged. First, the simulation PO to show that the new modified action in the refined event is not contradictory to the abstract action and the concrete event simulates the corresponding abstract event. Second, in the refined events, we can strengthen the abstract guards to specify more concrete conditions. More details on proof obligations can be found in [1].

Tool: Rodin Platform. Rodin [2] is an open source tool based on the Eclipse framework for developing Event-B models. It is a collection of different tools including project management, model development, refinement and proof assistance, model checking, and code generation.

The Theory Plug-In. A recent development of the Event-B language allows to extend it with theories [3] similar to algebraic specifications. In the Rodin Platform, this development is provided by the *Theory* plug-in [13]. In our work, we extend the theory of *Reals*, written by Abrial and Butler[1], for developing the required theories for modelling hybrid systems. In particular, all the relevant definitions, theorems and proof rules related to continuous functions, Ordinary Differential Equations (ODEs), Cauchy-Lipschitz conditions, etc. are defined in the developed theories.

4 Development of the Case Study

In this section, we describe how the approaches presented in Sect. 3 can be used to address the case study exposed in Sect. 2. As for the section presenting the tool, we first describe what has been done by Quesel et al. to design a solution using $\mathsf{d}\mathcal{L}$ and KeYmaera, and then we move on to how we dealt with this problem using Event-B and Rodin.

4.1 HP/dL/KeYmaera

Model. Table 2 shows the $\mathsf{d}\mathcal{L}$ formula (Eq. (1)) specifying the behaviour and requirements of the system described in the case study of Sect. 2. In Eq. (2), an initial condition is defined for velocity v, acceleration A and breaking power B. It also describes the *safe* condition which defines the safety envelope (or evolution domain) for the car regarding the stopping point SP.

[1] http://wiki.event-b.org/index.php/Theory_Plug-in#Standard_Library.

Equation (1) states that given the initial condition and after any run of non-deterministic iteration composing sequentially the controller (4) and the plant (5) hybrid programs, the safety requirement *req* (6) stating that the position is always ([ ]) before the stopping point SP. Finally two equations define controller and plant. Equation (4) models the control.

First, when the *safe* condition holds, the acceleration is unchanged, otherwise it is set to $-B$ for breaking. Second, Eq. (5) sets up the ODEs associated to the position and velocity with respect to the reachability of the stopping point SP.

Table 2. Hybrid program for the self-driven car

$init$	$\rightarrow [(ctrl; plant)^*]\,(req)$	(1)
$init$	$\equiv v \geq 0 \wedge A > 0 \wedge B > 0 \wedge safe$	(2)
$safe$	$\equiv p + \frac{v^2}{2B} < SP$	(3)
$ctrl$	$\equiv (?safe; a := A) \cup (?v = 0; a := 0) \cup (a := -B)$	(4)
$plant$	$\equiv (p' = v, v' = a \& v \geq 0 \wedge p + \frac{v^2}{2B} \leq SP)$	(5)
	$\cup (p' = v, v' = a \& v \geq 0 \wedge p + \frac{v^2}{2B} \geq SP)$	
req	$\equiv p \leq SP$	(6)

Property Verification. The hybrid program of Table 2 is given to the KeYmaera prover. The user must then prove the global formula (1), and the proof is conducted by applying inference rules in a natural deduction style. Similarly to other provers, several proof rules and tactics are available. Figure 1 shows an extract of the proof tree associated to the $\mathsf{d}\mathcal{L}$ formula $init \rightarrow [(ctrl; plant)^*]\,(req)$.

$$\dfrac{\dfrac{\text{QE}\ \dfrac{\cdots}{init \vdash J} \qquad \text{ODE}\ \dfrac{\cdots}{J \vdash [(ctrl; plant)]\,J} \qquad \text{QE}\ \dfrac{\cdots}{J \vdash req}}{\text{loop}\ \dfrac{}{init \vdash [(ctrl; plant)^*]\,(req)}}}{\rightarrow R\ \dfrac{init \vdash [(ctrl; plant)^*]\,(req)}{\vdash init \rightarrow [(ctrl; plant)^*]\,(req)}}$$

Fig. 1. Example of KeYmaera proof tree ($J \equiv v \geq 0 \wedge p + \frac{v^2}{2B} \leq SP$)

The power of the KeYmaera prover resides in the availability of several proof rules and tactics dealing with continuous aspects, ODEs and induction using differential equations (*loop* proof rule in Fig. 1).

4.2 Event-B/Rodin

Model. To build our Event-B model of the case study, our approach encodes a classical hybrid automaton [4,5] where transitions are *events* and states are simply stored as variables. Similar to the approach of [8], two types of variables are considered: (1) control variables (discrete) used for the controller (e.g. to record mode changes) and (2) variables (continuous) recording continuous state of the plant (e.g. to record the physical state of the plant). However, this is not enough to address all the complexity of hybrid systems. Namely, nothing is done to convey the "internal" evolution of the system (the time-step transitions), to handle the changes of the continuous variables with respect to time. So, additionally, we introduce events to reflect the overall continuous *progress* of the system.

Last, the core Event-B modelling language is not equipped with the formal material related to the definition of continuous mathematics required to model the physics of the controlled plant. To overcome this drawback, instead of redesigning a language, we used the so-called Event-B *theories*. Several theories have thus been defined and used for the development of the case study. The remainder of this section describes the whole Event-B development which can be downloaded from yamine.perso.enseeiht.fr/HS_eventb_models.pdf.

The *Derivative* Global Context to Manipulate Continuous Functions. Concept such as continuous functions, derivative, differential equations etc. required to model the physics of a plant are introduced in a generic context `Derivative` holding various (axiomatic) mathematical definitions. In particular, it gives the sets of continuous, once- and twice- differentiable functions as well as a basic "derivation" operator.

Beside that, we also defined a weak and simple version of the *Cauchy-Lipschitz theorem*, in order to be able to express the condition of existence of a solution to the given differential equation. Observe that the derivation operator D is used to define *time* derivation operation of the form $\frac{\mathrm{d}}{\mathrm{d}t}$.

CONTEXT Derivative
...
AXIOMS
axm1: $D \in ((\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R}))$ -- *derivative operator*
axm2: $\mathcal{C}^0(\mathbb{R}^+) \subset (\mathbb{R}^+ \to \mathbb{R})$ -- *continuous functions*
axm3: $\mathcal{D}^1(\mathbb{R}^+) \subset (\mathbb{R}^+ \to \mathbb{R})$ -- *once-differentiable functions*
axm4: $\mathcal{D}^2(\mathbb{R}^+) \subset (\mathbb{R}^+ \to \mathbb{R})$ -- *twice-differentiable functions*
axm5: $\mathcal{D}^1(\mathbb{R}^+) \subset \mathcal{C}^0(\mathbb{R}^+)$
axm6: $\mathcal{D}^2(\mathbb{R}^+) \subset \mathcal{D}^1(\mathbb{R}^+)$
...
cauchy_lipschitz:
$\forall f, t_0, x_0 \cdot f \in (\mathbb{R}^+ \to \mathbb{R}) \wedge f \in \mathcal{C}^0(\mathbb{R}^+) \wedge t_0 \in \mathbb{R}^+ \wedge x_0 \in \mathbb{R}$
$\Rightarrow \exists x \cdot x \in (\mathbb{R}^+ \to \mathbb{R}) \wedge D(x) = f \circ x \wedge x(t_0) = x_0$
...

The *Car_Context_C1* Context for Car Behaviours. It extends *Derivative* and declares the required concepts needed to build the Event-B model of the studied system. The constants defining the states of the controller (`acceleration`, `braking`, `stabilizing`, `nearing_stop` and `stopped`) are introduced together with A (acceleration), B (breaking power), v_0 (initial velocity) and SP (stopping point) used in the differential equations throughout the model.

This context also introduces the particular structure *Plant* for the characteristics of the plant (i.e.: the car). It is a 7-tuple with a differential equation (with its initial condition) for a (acceleration), v (velocity) and p (position) functions on time. They represent the state of the plant. k denotes the constant value of the acceleration, t_i, v_i and p_i represent the initial condition ($v(t_i) = v_i$ and $p(t_i) = p_i$). `axm10` defines the *Plant* structure, which holds every properties the elements of the model should satisfy with regards to the plant behaviour (type of the functions and differential equation). It also enforces **SAF2** (by indicating that whenever the velocity v becomes 0, the acceleration becomes 0 as well). Last, `CL_extension` entails the Cauchy-Lipschitz condition for this specific plant.

CONTEXT $Car\_Context\_C1$
EXTENDS $Derivative$
SETS $STATES$
CONSTANTS $B, A, v_0, SP, Plant$
AXIOMS
axm1: $partition(STATES, \{accelerating\}, \{braking\}, \{stabilizing\}, \{nearing\_stop\}, \{stopped\})$
axm2_9: $A \in \mathbb{R} \land B \in \mathbb{R} \land 0 < A \land 0 < B \land SP \in \mathbb{R} \land 0 < SP \land v_0 \in \mathbb{R} \land 0 \leq v_0$
axm10: $\forall a, v, p, k, t_i, v_i, p_i \cdot a \in \mathbb{R}^+ \to \mathbb{R} \land v \in \mathbb{R}^+ \to \mathbb{R} \land p \in \mathbb{R}^+ \to \mathbb{R} \land k \in \mathbb{R} \land t_i \in \mathbb{R}^+ \land v_i \in \mathbb{R} \land p_i \in \mathbb{R} \land (a, v, p, k, t_i, v_i, p_i) \in Plant \Leftrightarrow (v \in \mathcal{D}^1 \land p \in \mathcal{D}^2 \land D(v) = a \land D(p) = v \land v(t_i) = v_i \land p(t_i) = p_i \land (\exists t_0 \cdot t_0 \in \mathbb{R}^+ \land v(t_0) = 0 \Rightarrow (\forall t \cdot t \in \mathbb{R}^+ \Rightarrow ((t < t_0 \Rightarrow a(t) = k) \land (t \geq t_0 \Rightarrow a(t) = 0)))) \land (\forall t_0 \cdot t_0 \in \mathbb{R}^+ \land v(t_0) \neq 0 \Rightarrow (\forall t \cdot t \in \mathbb{R}^+ \Rightarrow a(t) = k)))$
CL_extension: $\forall k, t_i, v_i, p_i \cdot k \in \mathbb{R} \land t_i \in \mathbb{R}^+ \land v_i \in \mathbb{R} \land p_i \in \mathbb{R} \Rightarrow (\exists a, v, p \cdot a \in \mathbb{R}^+ \to \mathbb{R} \land v \in \mathbb{R}^+ \to \mathbb{R} \land p \in \mathbb{R}^+ \to \mathbb{R} \land (a, v, p, k, t_i, v_i, p_i) \in Plant)$
END

The *Plant* structure is handled as a whole in the Event-B model. From a methodological point of view, it shall be defined for each modelled plant.

$inv1\_2 : t \in \mathbb{R}^+ \land current\_state \in state$
$inv3\_5 : a \in \mathbb{R}^+ \to \mathbb{R} \land v \in \mathbb{R}^+ \to \mathbb{R} \land p \in \mathbb{R}^+ \to \mathbb{R}$
$inv6\_7 : \forall t0 \in \mathbb{R}^+ : p(t0) \geq 0 \land \forall t0 \in \mathbb{R}^+ : p(t0) \leq SP$
$inv8\_11 : v \in \mathcal{D}^1 \land p \in \mathcal{D}^2 \land D(v) = a \land D(p) = v$

Variables and Invariants. We use the relevant contexts and model the system's state with five variables. A read only variable t for time is introduced. *current_state* defines the current state of the controller, and the three variables are associated to the controlled plant (p position, v speed and a acceleration).

Initialisation. The initialisation event defines the initial state and the starting read time value as well as the initial values of each variables. `act3` defines, using the *Plant* structure, the initial conditions and the differential equation governing the evolution of the variables p, v and a. It sets the acceleration a to 0 and initialises the position and velocity with p_0 and v_0.

INITIALISATION $\widehat{=}$
THEN
act1 : $current\_state := stable$
act2 : $t := 0$
act3 : $a, v, p :| a' \in \mathbb{R}^+ \to \mathbb{R} \land v' \in \mathbb{R}^+ \to \mathbb{R} \land p' \in \mathbb{R}^+ \to \mathbb{R} \land (a', v', p', 0, t, v_0, p_0) \in Plant$
END

Progress $\widehat{=}$
THEN
act1 : $t :| t' \in \mathbb{R}^+ \land t < t'$
END

Time Handling: The *Progress* Event. In order to model the progress of time *independently* of any other event (i.e. for the other events, time can only be read), the `Progress` event is introduced. This event occurs continuously in parallel with the other model events. A before-after predicate describes strictly increasing time using a positive real variable t.

Behaviour of the Plant. The remainder of the model contains two categories of events: sensing and actuating events. Sensing events are split into command events (user or driver orders) and the actual sensing (coming from the environment through sensors) events. To keep the paper at a reasonable length, we only describe one event of each category. The whole model is available on yamine.perso.enseeiht.fr/HS_eventb_models.pdf.

Command-Sensing Events. The command-sensing events observe, through sensing, the state of the car (plant) and trigger state changes on the controller (state-transition system). For example, under the condition that the velocity is positive, the `ctrl_sense_usr_input_stabilize` records that the driver decided to stabilize her speed.

ctrl_sense_usr_input_stabilize $\widehat{=}$
WHERE
grd1 : $p(t) + \frac{v(t)^2}{2B} < SP$
THEN
act1 : $current_s tate := stabilizing$
END
ctrl_sense_usr_input_accel $\widehat{=}$ ...
ctrl_sense_usr_input_brake $\widehat{=}$ ...

ctrl_sense_near_stop $\widehat{=}$
WHERE
grd1 : $p(t) + \frac{v(t)^2}{2 \times B} \geq S$
THEN
act1 : $current\_state := nearing\_stop$
END
ctrl_sense_stopping $\widehat{=}$ ...

ctrl_actuate_stabilize $\widehat{=}$
WHERE
grd1 : $current\_state \in \{stabilizing, stopped\}$
THEN
act1 : $a, v, p : |a' \in \mathbb{R}^+ \rightarrow \mathbb{R} \wedge v' \in \mathbb{R}^+ \rightarrow \mathbb{R} \wedge$
$p' \in \mathbb{R}^+ \rightarrow \mathbb{R} \wedge$
$(a', v', p', 0, t, v(t), p(t)) \in Plant$
END
ctrl_actuate_accelerating $\widehat{=}$ ...
ctrl_actuate_brake $\widehat{=}$ ...

Control-Sensing Events. These events are triggered when information from the external environment (typically: coming from sensors) is available. For example, the `ctrl_sense_near_stop` event is triggered when the stop sign needs to be taken into account. The physics of the car provides the model with the relevant trigger condition, used as a guard (`grd1`).

Control-Actuating Events. Whenever the controller changes state, it sets the right actuation on the car (plant). Using a before-after predicate, it changes the differential equation characterizing the plant behaviour (car) to a new one (change of acceleration a), ensuring that the past behaviour is preserved. This change occurs at the current time and holds until the next actuation. For example, the `ctrl_actuate_stabilze` event modifies the variables a, v and p in `act1` when the controller is in either `stabilizing` or `stopped` mode.

Property Verification. The theory defined for continuous features and ODEs generated several proof obligations, in particular those related to the theorems and thus to the proof rules. Then, other proof obligations are generated from the Event-B model. Due to our extensive use of the theory plug-in, most of these proofs have numerous manual steps, particularly the ones related to the continuous features. Even simple proofs, such as well-definedness, need to access real type operators via interactive theorem instantiation.

Obtained Results. The Event-B development of Sect. 4.2 shows that it is possible to model both continuous and discrete behaviours, using an event-based modelling style, within Event-B on the Rodin platform. It also shows that Event-B can handle modelling of hybrid systems modelled by hybrid automata.

5 A Development Method for Hybrid Systems

Taking the development carried out in Sect. 4.2 one step further, we present, in this section, the methodological lessons learnt from this development.

5.1 The Approach

The development of Sect. 4.2 is a *direct* formalisation of the requirements presented in the case study. However, when observing how the events are defined in the Event-B machine, one can identify a set of methodological rules that help to produce such models in a systematic way.

Required Theories. First, the global set of axioms and theorems, CONTEXT `Derivative`, related to the manipulation of continuous functions (derivation, Lipshitz condition, etc.) is used for all the Event-B developments. Second, the specific context with all the concepts needed to model the specific case study shall be introduced. This context defines the control states of the mode automaton associated to the control together with the continuous functions associated to the plant to be controlled. It also defines the global structure representing both continuous and discrete elements manipulated by the behavioural model through variables and events (definition of the *Plant* 7-tuple structure). Regarding the case study developed in this paper, this context corresponds to the `Car_Context_C1` context.

$x_s : Ctrl\_State$
$x_p : Plant\_State$
INVARIANT
Inv : $Inv\_Exp(x_s, x_p)$

INITIALISATION
$x_s, x_p :| Init\_Pred(x_s, x_p, x'_p, x'_s)$

Modeling Hybrid Systems. The next steps concern the design of the model itself using an Event-B machine. First, state variables are declared. x_s and x_p are the variables associated to the controller and to the plant respectively. They shall conform to the invariant defined by the Inv_Exp predicate. They are initialised with initial conditions defined by the predicate $Init\_Pred$. Two categories of events are needed to handle sensing and actuation. They are defined by two templates. `CTRL_Sense` events for the sensing events (user commands or plant sensing) that may modify the state ($Exp\_Next\_for\_s$ before-after predicate) of the controller while `CTRL_Actuate` defines the actuation events to modify ($Exp\_Next\_for\_p$ before-after predicate) the plant behaviour.

EVENT CTRL_Sense
WHEN
grd : $Cond\_Exp\_p(x_p)$
THEN
act : $x_s :| Exp\_Next\_for\_s(x_s, x_p)$
END

EVENT CTRL_Actuate
WHEN
grd : $Cond\_Exp\_s(x_s)$
THEN
act : $x_p :| Exp\_Next\_for\_p(x_p, x_s)$
END

The steps described above have been followed, in Sect. 4.2, to develop the Event-B models of the case study.

5.2 Deep Modelling: Generalisation

As mentioned previously, the approach described in the previous section sets up some methodological steps without any restriction on the resulting development. Encoding the previous steps in a generic deep Event-B model makes it possible to introduce more oversight and rigour into the design of models for hybrid systems.

In this section, we present a generalisation of the previous approach. We propose a generic Event-B model that explicitly formalises the different methodological steps defined in the previous section. We also show that a particular system can be modelled by instantiating this generic model and supplying witnesses.

A Theory for ODEs. Continuous functions, ODEs together with their relevant properties are defined with an Event-B theory. Listing 1.1 shows an extract of the theory of ODEs with *ode* as a constructor for differential equations. The operators $solutionOf$, $Solvable$ and $CauchyLipschitzCondition$ define predicates that states respectively that a given function is a solution of an ODE (with initial conditions), that an ODE admits a solution and that the given equation fulfill a Cauchy-Lipschitz condition. The *bind* operator returns an expression that links generic plant variables to a pair of variables associated to a particular plant. It has been introduced to ease instantiation.

Listing 1.1. Elements of the differential equations theory

TYPE PARAMETERS E, F, G
DATATYPES
DE(F)
Constructors
$ode(f : \mathcal{P}(\mathbb{R}^+ \times F \times F),\ f_0 : F,\ t_0 : \mathbb{R}^+)$
OPERATORS
solutionOf <predicate> (eta : $\mathbb{R}^+ \to F$, eq : $DE(F)$)
WHEN eq $\equiv$ ode(f,f_0,t_0) **THEN**
$eta \in \mathbb{R}^+ \to F \land eta \in \mathcal{D}^1(\mathbb{R}^+, F) \land D(eta) = f(eta) \land eta(t_0) = f_0$
Solvable <predicate> (eq : $DE(F)$)$\exists x \cdot x \in (\mathbb{R}^+ \to F) \land (x$ solutionOf $eq)$
CauchyLipschitzCondition <predicate> (eq : $DE(F)$)
WHEN eq $\equiv$ ode(f,f_0,t_0) **THEN**
$f_0 \in F \land t_0 \in \mathbb{R}^+ \land f \in (\mathbb{R}^+ \times F \to F) \land f \in \mathcal{C}^0(\mathbb{R}^+ \times F, F) \land$
$(\forall t^* \cdot t^* \in \mathbb{R}^+ \Rightarrow lispchitzContinuous(F, F, (\lambda y \cdot y \in F \mid f(t^*, y)))$
bind <expression> (A:$\mathcal{P}(E)$, B:$\mathcal{P}(F)$, C:$\mathcal{P}(G)$, f_{ab}:$A \to B$, f_{ac}:$A \to C$)
$(\lambda x \cdot x \in A \mid (f_{ab}(x), g_{ac}(x)))$
...
AXIOMATIC DEFINITIONS
$\mathcal{C}^0$, $\mathcal{C}^n$, $\mathcal{D}^1$, $\mathcal{D}^n$,
...
lipschitzContinuous <predicate> (A : $\mathcal{P}(E)$, B : $\mathcal{P}(F)$, f_{ab} : $A \to B$)
...
THEOREMS
CauchyLipschitz: $\forall eq \cdot eq \in DE(F) \land CauchyLipschitzCondition(eq) \Rightarrow Solvable(eq)$

A Generic Model. The following model elements defined in `MACHINE System` gives a generalisation of the approach. We consider that the plant variable x_p

belongs to $S = \mathbb{R} \times \mathbb{R}$. It is indexed on time ($x_p \in \mathbb{R}^+ \rightarrow S$) and evolves according to any ODE e in the `actuate` event. The controller models state transitions belonging to the set of states `STATES` using the `Transition` event.

MACHINE System
...
INVARIANTS
inv1 : $t \in \mathbb{R}^+$
inv2 : $x_p \in \mathbb{R}^+ \rightarrow S$
EVENTS
INITIALISATION $\widehat{=}$
THEN
act1 : $t := 0$
act2 : $x_p :\in \mathbb{R}^+ \rightarrow S$
act3 :
$current\_state :\in STATES$
END
Progress $\widehat{=}$
THEN
act1 : $t : |t' \in \mathbb{R}^+ \wedge t' > t$
END

Actuate $\widehat{=}$
ANY e, s
WHERE
grd1 : $e \in DE(S)$
grd2 : $Solvable(e)$
grd3 : $s = current\_state$
THEN
act1 : $x_p : |x'_p \in \mathbb{R}^+ \rightarrow S \wedge$
$(x'_p$ solutionOf $e)$
END
Transition $\widehat{=}$
ANY s
WHERE
grd1 : $s \in STATES$
THEN
act1 : $current\_state := s$
END

Instantiation of the Generic Model: Two Steps. To get the specific model associated to the system corresponding to the case study of Sect. 2, two steps are required.

The first step consists of defining the Event-B context for the relevant information of the system by introducing acceleration, velocity etc. and the different ODEs describing plant evolution. It uses the constructors defined in the theory presented in Listing 1.1. The `CONTEXT Car_C0` shows such instantiation.

CONTEXT Car_C0
CONSTANTS $B, A, v_0, SP, Plant$
AXIOMS
axm1: $partition(STATES, \{stabilizing\}, \{braking\}, \{accelerating\},$
$\{nearing\_stop\}, \{stopped\}$
...
axm12: $f_{stable} \in \mathbb{R}^+ \times S \rightarrow S$
axm13: $f_{stable} = (\lambda t, (p, v) \cdot t \in \mathbb{R}^+ \wedge (p, v) \in S | (v, 0))$
...
axm16: $\forall t_0 \cdot t_0 \in \mathbb{R}^+ \Rightarrow lipschitzContinuous(S, S, (\lambda x \cdot f_{stable}(t_0, x)))$
...

The second step consists of supplying witnesses to the actuating and sensing events. The `MACHINE Car_M0` shows a witness for the plant variable x and v evolving according to an ODE for function f_{stabe}. It also shows an actuating event `ctrl_actuate_stabilize` where the actuation sets the plant variables to evolve according to the defined ODE.

MACHINE Car_M0
REFINES $ControlledSystem$
...
INVARIANTS
inv1 : $v \in \mathbb{R}^+ \rightarrow \mathbb{R}$
inv2 : $x \in \mathbb{R}^+ \rightarrow \mathbb{R}$
inv3 : $triggers \subseteq STATES$
inv4 : $x_p = bind(\mathbb{R}^+, \mathbb{R}, \mathbb{R}, v, x)$
EVENTS
INITIALISATION$\widehat{=}$
WITH
$x'_p : x'_p = bind(\mathbb{R}^+, \mathbb{R}, \mathbb{R}, v', x')$
THEN
act1 : $t := 0$
act2 : $current\_state := stabilizing$
act3 : $v, x :|$
$x' \in \mathbb{R}^+ \rightarrow \mathbb{R} \wedge v' \in \mathbb{R}^+ \rightarrow \mathbb{R} \wedge$
$bind(\mathbb{R}^+, \mathbb{R}, \mathbb{R}, v', x')$
solutionOf $ode(f_{stable}, (v_0, 0), 0)$
END

ctrl_actuate_stabilize $\widehat{=}$
REFINES $Actuate$
WHERE
grd1 : $current\_state \in \{stabilizing, stopped\}$
WITH
$e : e = ode(f_{stable}, (v(t) \mapsto x(t)), t)$
$s : s = stabilizing$
$x'_p : x'_p = bind(\mathbb{R}^+, \mathbb{R}, \mathbb{R}, v', x')$
THEN
act1 : $v, x :|$
$x' \in \mathbb{R}^+ \rightarrow \mathbb{R} \wedge v' \in \mathbb{R}^+ \rightarrow \mathbb{R} \wedge$
$bind(\mathbb{R}^+, \mathbb{R}, \mathbb{R}, v', x')$
solutionOf $ode(f_{stable}, (v(t) \mapsto x(t)), t)$
END

The previous models are extracts of a generic development that can be downloaded from yamine.perso.enseeiht.fr/HS_eventb_models.pdf.

5.3 About KeYmaera and Rodin: Assessments

The case study presented in Sect. 2 has been developed in both differential dynamic logic with KeYmaera and Event-B with Rodin. In both cases, the model of the system has been designed and the requirements have been proved. However, we have observed several differences in the use of these two modelling techniques.

KeYmaera supports the definition and verification of hybrid systems models expressed using $\mathsf{d}\mathcal{L}$ and hybrid programs. The logic is built-in and fixed once and for all and the proof rules are hard-coded into the KeYmaera prover. The advantage of such an approach is the availability of very powerful proof rules associated to the manipulation of differential equations (see Fig. 1), and in particular the differential invariant rule that defines an inductive proof schema (*loop* rule in Fig. 1) for differential equations.

To model hybrid systems in Event-B, we define the operational behaviour using the events. Invariants and other properties are defined at the same time. The model is seen as a set of events that perform either sensing or actuations. Proof by induction is obtained by proving invariant preservation by each event while KeYmaera advocates proof of invariant preservation on the whole hybrid program without a possibility of decomposing this hybrid program (as Event-B does for events). The specific proof rules for ODEs (like Cauchy-Lipschitz theorem) need to be added in the defined theories while they are built-in in KeYmaera. This task is cumbersome but should only be done once.

Our experiments with Event-B have been conducted in two manners. First, we have encoded directly the case study as an Event-B model in the spirit of $\mathsf{d}\mathcal{L}$ and KeYmaera. Contrary to KeYmaera, this process requires the definition of all the material related to the manipulation of continuous functions and ODEs.

Compared to KeYmaera, this process may be time-consuming due to the important proof effort just to set up the different functions and ODEs.

Secondly, we have developed an abstract model that generalises the theory of hybrid controllers. This model is designed and proved once and for all. It may be instantiated, using Event-B refinement, for specific cases by providing witnesses and proof of existence. Indeed, these feasibility proof obligations convey a technical point of differential equation theory; that is: the existence of solutions to a given problem. However, *KeYmaera* proofs seem to rely on the *ad hoc* existence of those solutions.

The definition of the generic model makes explicit definitions of all the concepts manipulated by the model. These definitions can be customised for specific kinds of controllers and ODEs (for example, we can add more constraints on ODEs to admit only linear ones). The Rodin theory plug-in helps to define the proof rules associated to the use of these definitions.

Finally, some issues still need to be solved. For example, the difficulty to define inductive Cartesian products ($S \times \cdots \times S$ or S^n for an arbitrary n) to define vectors of state variables. We have to use an inductive structure for this purpose and thus rewrite the *bind* generic operator. Secondly, the definition of a condition to assert the non-zeno property of the system described by ODEs must be addressed. This can be done by adding a condition on the existence of a piecewise decomposition of an ODE in a finite set of arbitrary intervals. KeYmaera makes this assumption but does not make it explicit.

Other Proof-Based Approaches. Here we briefly review three main contributions in formal modelling and proof based verification of hybrid systems.

As described above, differential dynamic logic using KeYmaera and KeYmaeraX tool suite have been used to model several cases of hybrid systems.

Additionally to the approach presented in this paper, other Event-B contributions can be mentioned. [12,18] use Event-B and the Rodin Platform [14] to model hybrid systems in a closed-loop model. Time is explicitly modelled using a specific state variable. The authors consider continuous functions and they define discrete and continuous transitions preserving invariants which characterise the correct behaviour of the described hybrid system.

[9] proposes the Hybrid Event-B extension of Event-B with built-in concepts for continuous behaviours, differential equations discrete and continuous (pliant) events. Several hybrid systems models (e.g. [8]) have been developed. A similar approach has been proposed to define continuous abstract state machines in [10]. For both approaches, there is no available supporting tool.

Last, we mention the work of [11] using a proof-based approach with COQ for the analysis of C hybrid systems programs. The approach uses formal annotations on discrete and continuous program elements as input to a set of provers.

Finally, we recall that several other approaches based on model checking of hybrid systems modelled as hybrid automata [5]. Due to space limitations these approaches are not discussed in this paper.

6 Conclusion and Future Work

The formal development of hybrid systems requires modelling of both discrete and continuous behaviours. In this paper, we have presented two formal developments of a case study related to a stop-sign controller of a car. The first one is based on differential dynamic logic with KeYmaera and the second one uses Event-B and Rodin. Both approaches proved useful to model such a system and refer to interactive proofs involving proof rules on differential equations. Besides, using the theory plug-in to extend Rodin's capabilities, Event-B proved to be well adapted for generalisation.

This work opens several research paths. First the generalisation we have presented in Sect. 5.2 can be improved in order to formally handle more features like invariants, guards or different kinds of ODEs. Offering such a possibility will allow us to produce generic templates of hybrid models that correspond to different types of hybrid systems (for example a non-linear system, or a polyhedral invariant). The ultimate objective is to hide the development details through the definition of development/proof patterns. Second, following our preliminary results in [6,7], the synthesis of a discrete controller from hybrid models similar to those presented in this paper is also a key issue. Last, we will address the simulation aspect related to the modelling of hybrid systems.

Finally, we advocate that Event-B together with the plug-in associated to powerful provers will allow us to achieve these goals.

Acknowledgment. We would like to thank Dr Thai Son Hoang for his help regarding Rodin's Theory plug-in.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. Int. J. Softw. Tools Technol. Transfer **12**(6), 447–466 (2010)
3. Abrial, J.R., Butler, M., Hallerstede, S., Leuschel, M., Schmalz, M., Voisin, L.: Proposals for mathematical extensions for Event-B. Technical report (2009). http://deploy-eprints.ecs.soton.ac.uk/216/
4. Alur, R.: Formal verification of hybrid systems. In: Proceedings of the Ninth ACM International Conference on Embedded Software, EMSOFT 2011, pp. 273–278. ACM, New York (2011)
5. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.-H.: Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In: Grossman, R.L., Nerode, A., Ravn, A.P., Rischel, H. (eds.) HS 1991-1992. LNCS, vol. 736, pp. 209–229. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-57318-6_30
6. Babin, G., Aït-Ameur, Y., Singh, N.K., Pantel, M.: Handling continuous functions in hybrid systems reconfigurations: a formal Event-B development. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) ABZ 2016. LNCS, vol. 9675, pp. 290–296. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33600-8_23

7. Babin, G., Aït-Ameur, Y., Singh, N.K., Pantel, M.: A system substitution mechanism for hybrid systems in Event-B. In: Ogata, K., Lawford, M., Liu, S. (eds.) ICFEM 2016. LNCS, vol. 10009, pp. 106–121. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47846-3_8
8. Banach, R.: Formal refinement and partitioning of a fuel pump system for small aircraft in hybrid Event-B. In: 2016 10th International Symposium on Theoretical Aspects of Software Engineering (TASE), pp. 65–72, July 2016
9. Banach, R., Butler, M., Qin, S., Verma, N., Zhu, H.: Core hybrid Event-B I: single hybrid Event-B machines. Sci. Comput. Program. **105**, 92–123 (2015)
10. Banach, R., Zhu, H., Su, W., Wu, X.: ASM, controller synthesis, and complete refinement. Sci. Comput. Program. **94**, 109–129 (2014)
11. Boldo, S., Clément, F., Filliâtre, J.C., Mayero, M., Melquiond, G., Weis, P.: Trusting computations: a mechanized proof from partial differential equations to actual program. Comput. Math. Appl. **68**(3), 325–352 (2014)
12. Butler, M., Abrial, J.R., Banach, R.: Modelling and refining hybrid systems in Event-B and Rodin. In: Petre, L., Sekerinski, E. (eds.) From Action Systems to Distributed Systems: The Refinement Approach. Computer and Information Science Series, Chap. 3, pp. 29–42. Chapman and Hall/CRC (2016)
13. Butler, M., Maamria, I.: Practical theory extension in Event-B. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) Theories of Programming and Formal Methods. LNCS, vol. 8051, pp. 67–81. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39698-4_5
14. Jastram, M.: Rodin User's Handbook, October 2013. http://handbook.event-b.org
15. Platzer, A.: Differential dynamic logic for hybrid systems. J. Autom. Reas. **41**(2), 143–189 (2008)
16. Platzer, A., Quesel, J.-D.: KeYmaera: a hybrid theorem prover for hybrid systems (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 171–178. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71070-7_15
17. Quesel, J.D., Mitsch, S., Loos, S., Aréchiga, N., Platzer, A.: How to model and prove hybrid systems with KeYmaera: a tutorial on safety. Int. J. Softw. Tools Technol. Transfer **18**(1), 67–91 (2016)
18. Su, W., Abrial, J.R., Zhu, H.: Formalizing hybrid systems with Event-B and the rodin platform. Sci. Comput. Program. **94**(Part 2), 164–202 (2014). Abstract State Machines, Alloy, B, VDM, and Z

Issues in Automated Urban Train Control: 'Tackling' the Rugby Club Problem

Richard Banach(✉)

School of Computer Science, University of Manchester,
Oxford Road, Manchester M13 9PL, UK
richard.banach@manchester.ac.uk

Abstract. Normally, the passengers on urban rail systems remain fairly stationary, allowing for a relatively straightforward approach to controlling the dynamics of the system, based on the total rest mass of the train and passengers. However when a mischievous rugby club board an empty train and then run and jump-stop during the braking process, they can disrupt the automatic mechanisms for aligning train and platform doors. This is the rugby club problem for automated urban train control. A simple scenario of this kind is modelled in Hybrid Event-B, and sufficient conditions are derived for the prevention of the overshoot caused by the jump-stop. The challenges of making the model more realistic are discussed, and a strategy for dealing with the rugby club problem, when it cannot be prevented, is proposed.

1 Introduction

With profuse apologies to Clement Clarke Moore: *'Twas early in the morning, when all thro' the house, Not a creature was stirring, not even a mouse...* aside, that is, from the stout adherents of a rugby club, who were bent on making their way to the *Métro* station, to board the otherwise empty first service of the day on the fully automated, unmanned line.[1]

As the train pulls out of the station, the dynamical variables are measured by the train system,[2] in order to gauge the weight of the passengers that have got on board—this, in order to be able to accurately predict the braking force that will be needed when the train pulls into the next station. The train becomes cognisant of the weight of the rugby club, at this point standing at the back of the train.

As the train starts to approach the next station, the rugby club start a run up the empty train towards the front. The velocity feedback control law governing the train's travel detects a shortfall in velocity and commands additional acceleration to bring the train up to speed, thereby adding to the momentum of

[1] Such as the Paris *Météor* Line 14, engineered using the B-Method.
[2] Acceleration, time taken to reach cruising speed, etc.

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 171–186, 2018.
https://doi.org/10.1007/978-3-319-91271-4_12

the whole train. The train starts to brake as it enters the next station. As the train is coming to a stop, the rugby club complete their run with a jump-stop, impulsively imparting their momentum to the train body. The train has calculated its braking force on the basis of the earlier, stationary rugby club, and has not taken into account the additional momentum. As a result of the jump-stop, the train's braking force is inadequate, and the train overshoots its intended stopping point ... by a sufficient distance for the misalignment with the platform side doors to exceed the permitted safety margin. The only option for such excess misalignment (taking into account the demands of rush-hour throughput) is that the train does not stop but continues to the next station. Having given a cheer, the rugby club make their way to the back of the train, which still works on the basis of the original weight estimate. As the next station is approached, they start a run ... you can guess the rest. On a circular line,[3] the rugby club can amuse themselves this way all day long, with the train never stopping until the end of service. This is the Rugby Club Problem for automated urban railways.[4]

The problem of a moderate, but nevertheless unacceptable overshoot of the door position by an automated urban train is easily solved if the train doors are equidistantly spaced. Then, it is enough to have an additional door or two at the front end of the platform. The train then aims for its normal position, and if an overshoot happens, the train can carefully, but quite quickly, inch along to the next spare door position, the equidistant spacing guaranteeing that all train doors will thereby be correctly aligned.[5] But the equidistant design is not widely adopted. To have enough doors per carriage to cope with a busy rush hour in an urban environment that is populated enough to justify an urban rail solution in the first place, puts considerable demands on the structural integrity of the carriages, leading to additional costs.[6]

Putting aside levity, the aim of this paper is to demonstrate that Hybrid Event-B [8,9] can deal fluently with the problem of modeling the kind of impulsive physics exhibited by the Rugby Club Problem. By now there are quite a number of existing case studies using Hybrid Event-B [2–7,11], but none of the ones published hitherto has focused on impulsive physics.

The remaining sections of the paper are as follows. In Sect. 2, we outline Hybrid Event-B, emphasising the elements that are useful in modelling impulsive physics. In Sect. 3, for lack of space, we present a very simple model of the Rugby Club Problem, displaying its essential characteristics, including how the impulsive elements are handled. In Sect. 4, we consider various alternatives and enhancements, which we discuss more briefly. Section 5 discusses how the Rugby Club Problem might be solved in the context of the modelling of this paper. Section 6 concludes.

[3] O. K. The *Météor* line is not circular, but you get the idea.

[4] I am indebted to Thiérry Lecomte of ClearSy [14] for this delightful story [19].

[5] Such a design is visible on the Shanghai Metro's circular line 4, as well as on some other, older Shanghai Metro lines, built when train door alignment control was less precise than today.

[6] The robustness of the carriages on the Shanghai line 4 would put much heavy rail to shame.

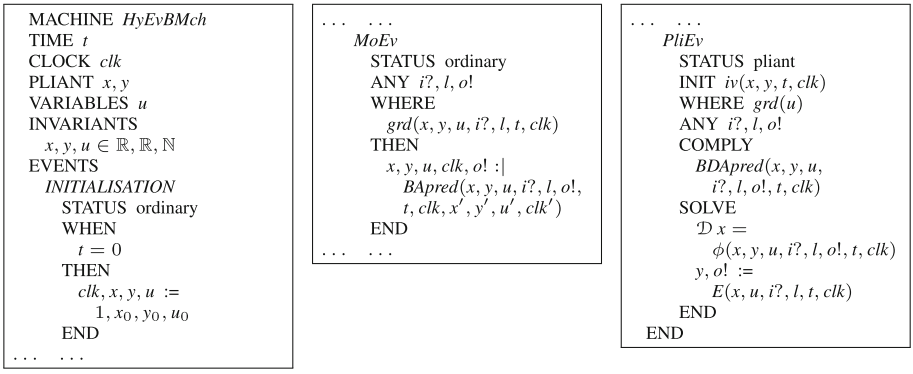


Fig. 1. A schematic Hybrid Event-B machine.

2 Hybrid Event-B, and Modelling Impulsive Physics

In this section, we outline Event-B and Hybrid Event-B for a single machine. Because it is more complex, we describe Hybrid Event-B first via Fig. 1, and show how it reduces to Event-B (which of course came earlier) by erasing the more recently added elements.

Figure 1 shows a schematic Hybrid Event-B machine. It starts with declarations of time and of a clock. Time is a first class citizen in that all variables are functions of time (which is read-only), explicitly or implicitly. Clocks are assumed to increase like time, but may be set during mode events. Variables are of two kinds. There are mode variables (like u) which take their values in discrete sets and change their values via discontinuous assignment in mode events. There are also pliant variables (such as x, y), declared in the PLIANT clause, which typically take their values in topologically dense sets (normally $\mathbb{R}$) and which are allowed to change continuously, such change being specified via pliant events.

Next are the invariants. These resemble invariants in discrete Event-B, in that the types of the variables are asserted to be the sets from which the variables' values *at any given moment of time* are drawn. More complex invariants are similarly predicates that are required to hold *at all moments of time* during a run.

Then, the events. The *INITIALISATION* has a guard that synchronises time with the start of any run, while all other variables are assigned their initial values as usual.

Mode events are analogues of events in discrete Event-B. They can assign all machine variables (except time). The schematic *MoEv* of Fig. 1, has parameters $i?$, l, $o!$, (input, local, and an output), and a guard *grd*. It also has the after-value assignment specified by the before-after predicate *BApred*, which can specify the after-values of all variables (except time, inputs and locals).

Pliant events are new. They specify the continuous evolution of the pliant variables over an interval of time. Figure 1 has a schematic pliant event *PliEv*.

There are two guards: *iv*, for specifying enabling conditions on the pliant variables, clocks, and time; and *grd*, for specifying enabling conditions on the mode variables (in [8] there is a detailed discussion justifying such a design).

The body of a pliant event contains three parameters $i?, l, o!$, (input, local, and output, again) which are functions of time, defined over the duration of the pliant event. The behaviour of the event is defined by the COMPLY and SOLVE clauses. The SOLVE clause contains direct assignments, e.g. of y and output $o!$ (to time dependent functions); and differential equations, e.g. specifying x via an ODE (with $\mathcal{D}$ as the time derivative).

The COMPLY clause is used to express any additional constraints that are required to hold during the pliant event via the before-during-and-after predicate *BDApred*. Typically, constraints on the permitted ranges of the pliant variables, can be placed here. The COMPLY clause can also specify at an abstract level, e.g. stating safety properties for the event without going into detail.

Briefly, the semantics of a Hybrid Event-B machine consists of a set of *system traces*, each of which is a collection of functions of time, expressing the value of each machine variable over the duration of a system run.

Time is modelled as an interval $\mathcal{T}$ of the reals. A run starts at some initial moment of time, t_0 say, and lasts either for a finite time, or indefinitely. The duration of the run $\mathcal{T}$, breaks up into a succession of left-closed right-open subintervals: $\mathcal{T} = [t_0 \ldots t_1), [t_1 \ldots t_2), [t_2 \ldots t_3), \ldots$. Mode events (with their discontinuous updates) take place at the isolated times corresponding to the common endpoints of these subintervals t_i, and in between, the mode variables are constant, and the pliant events stipulate continuous change in the pliant variables.

We insist that on every subinterval $[t_i \ldots t_{i+1})$ the behaviour is governed by a well posed initial value problem $\mathcal{D}\, xs = \phi(xs \ldots)$ (where xs is a relevant tuple of pliant variables). Within this interval, we seek the earliest time t_{i+1} at which a mode event becomes enabled, and this time becomes the preemption point beyond which the solution to the ODE system is abandoned, and the next solution is sought after the completion of the mode event.

In this manner, assuming that the *INITIALISATION* event has achieved a suitable initial assignment to variables, a system run is *well formed*, and thus belongs to the semantics of the machine, provided that at runtime:[7]

- Every enabled mode event is feasible, i.e.has an after-state, and on its completion enables a pliant event (but does not enable any mode event). (1)
- Every enabled pliant event is feasible, i.e. has a time-indexed family of after-states, and EITHER: (2)

[7] If a mode event has an input, the semantics assumes that its value only arrives at a time *strictly later* than the previous mode event, ensuring part of (1) automatically. By this means we can ensure a mode event executes asynchronously—and if the only purpose of having an input would be to ensure this asynchronous scheduling, we can introduce the 'async' status and omit the input altogether, as in Fig. 2.

(i) During the run of the pliant event a mode event becomes enabled. It preempts the pliant event, defining its end. ORELSE
(ii) During the run of the pliant event it becomes infeasible: finite termination. ORELSE
(iii) The pliant event continues indefinitely: nontermination.

Thus in a well formed run mode events alternate with pliant events. The last event (if there is one) is a pliant event (whose duration may be finite or infinite). In reality, there are several semantic issues that we have glossed over in the framework just sketched. We refer to [8] for a more detailed presentation (and to [9] for the extension to multiple machines). The presentation just given is quite close to the modern formulation of hybrid systems. See e.g. [22,23], or [13] for a perspective stretching further back.

The mode events of Hybrid Event-B, which permit the discontinuous state changes of the computational world to be represented, also allow impulsive physics to be conveniently modelled. For example, a billiard cue strikes a ball, changing its velocity discontinuously, or a capacitor discharges, instantaneously reducing the electrical potential across its plates to zero. However, unlike the computational world in which the programmer is at liberty to decide what discontinuous state changes take place, the physical world is governed by immutable laws, which must be adhered to yield a useful model. Thus, in the billiard ball example, it is the conservation of momentum that determines the relationship between the physical states before and after the strike. We might say that *'Hybrid Event-B cannot do your physics for you; but it can faithfully represent the physics that you know from elsewhere.'*

Connected with the preceding is the fact that discontinuous state changes in the physical world are stimulated by forces which are 'pure impulses'. And whereas discontinuous change can be represented quite directly in Hybrid Event-B, these pure impulses cannot. Physicists and engineers speak of such impulses as 'delta functions'—'zero everywhere except at a single point, were they are infinite, but with a finite integral'. Mathematically, that last phrase is meaningless; delta functions are not functions, but so-called distributions [18,24,25]. The nearest we get in Hybrid Event-B (or any other similar formalism) to the representation of a pure impulse is the (syntactic) description of the mode event that encapsulates the discontinuous change that results from the impulse. The occurrence of the mode event (at runtime) corresponds to the occurrence of the physical impulse causing the discontinuous change of state.

3 A Simple Rugby Club Problem Scenario

In this section, we present a very simple model of the rugby club scenario, formalised in Hybrid Event-B and, in particular, utilising the insights about impulsive physics just discussed. The model itself is in Fig. 2.

MACHINE *RugbyClub_1*
CLOCK *clk_A*
VARIABLES $mode, m_{in}, m_{pcv}, v_{rcr}, brTime$
PLIANT $m_{\rm eff}, v_T, brDist$
INVARIANTS
$mode \in \{STAT, ACCEL, CRUISE, DECEL\}$
$m_{in}, m_{pcv}, v_{rcr}, brTime \in \mathbb{R}^+, \mathbb{R}^+, \mathbb{R}^+, \mathbb{R}^+$
$m_{\rm eff}, v_T, brDist \in \mathbb{R}, \mathbb{R}, \mathbb{R}$
EVENTS
INITIALISATION
BEGIN
$clk\_A := BIGT$
$mode := STAT$
$m_{in}, m_{pcv} := M_T, M_T$
$v_{rcr} := 0$
$m_{\rm eff} := M_T$
$v_T := 0$
$brDist := 0$
$brTime := 0$
END
TrainStationary pliant
WHEN $mode = STAT$
THEN
COMPLY CONST$(m_{\rm eff}, v_T, brDist)$
END
RugbyClubBoards async
WHEN $mode = STAT$
THEN
$m_{in} := M_T + M_{rc}$
$m_{\rm eff} := M_T + M_{rc}$
END
TrainStarts async
BEGIN
$mode := ACCEL$
$clk\_A := 0$
END
TrainAccelerating pliant
WHEN $mode = ACCEL$
COMPLY CONST$(m_{\rm eff})$
THEN
$\mathcal{D}\, v_T = F_A / m_{in}$
$brDist := 0$
END
TrainAtSpeed
WHEN $mode = ACCEL \wedge v_T = V_{cr}$
THEN
$mode := CRUISE$
$m_{pcv} := (F_A\ clk\_A)/V_{cr}$ // . . . $= m_{in}$
$m_{\rm eff} := m_{in}$
END
TrainCruising pliant
WHEN $mode = CRUISE$
THEN
$\mathcal{D}\, v_T = -K(v_T - V_{cr})$
$brDist := 0$
$m_{\rm eff} := (M_T + M_{rc}) + \dfrac{M_{rc} v_{rcr}}{v_T}$
END
. . . . . .

. . . . . .
RugbyClubStartsRun async
WHEN $mode = CRUISE$
THEN
$v_{rcr} := V_{rcr}$
$v_T := V_{cr} - \dfrac{M_{rc}\, V_{rcr}}{M_T + M_{rc}}$
$m_{\rm eff} := \dfrac{(M_T + M_{rc})}{\left[1 - \dfrac{V_{rcr}}{V_{cr}} \left[1 + \dfrac{M_T}{M_{rc}}\right]^{-1}\right]}$
END
TrainBrakes async
WHEN $mode = CRUISE$
INIT $|\, v_T - V_{cr}\, | < \epsilon$
THEN
$mode := DECEL$
$brDist := -\dfrac{m_{pcv}}{F_D} \dfrac{V_{cr}^2}{2}$
$brTime := \dfrac{m_{pcv}\, V_{cr}}{F_D}$
END
TrainDecelerating pliant
WHEN $mode = DECEL$
THEN
$\mathcal{D}\, v_T = -\dfrac{F_D}{M_T + M_{rc}}$
$\mathcal{D}\, brDist = v_T$
$m_{\rm eff} := (M_T + M_{rc}) + \dfrac{M_{rc} v_{rcr}}{v_T}$
END
RugbyClubJumpStop async
WHEN $mode = DECEL \wedge v_T > 0$
THEN
$v_{rcr} := 0$
$v_T := \dfrac{m_{\rm eff}}{m_{in}} v_T$
$m_{\rm eff} := m_{pcv}$
END
TrainStopSucceed
WHEN $mode = DECEL \wedge v_T = 0 \wedge$
$|\, brDist\, | \leq BRTOL$
THEN
$clk\_A := BIGT$
$mode := STAT$
$m_{in}, m_{pcv} := M_T, M_T$
$brDist := 0$
$brTime := 0$
END
TrainStopFail
WHEN $mode = DECEL \wedge v_T = 0 \wedge$
$|\, brDist\, | > BRTOL$
THEN
$clk\_A := 0$
$mode := ACCEL$
$m_{in}, m_{pcv} := M_T, M_T$
$brDist := 0$
$brTime := 0$
END
END

Fig. 2. A simple Hybrid Event-B model of the urban rail Rugby Club Problem.

The model depends on a number of constants (which would be declared in a CONTEXT, which we do not show). There are: the phases of the model stored in the *mode* variable: *STAT*ionary; *ACCEL*erating; *CRUISE*ing; *DECEL*erating. There is also: $BIGT$, an initial value for a clock that is bigger than any value that could trigger the enabledness of any mode event; M_T, the mass of the train; M_{rc}, the mass of the rugby club; V_{cr} the cruising speed of the train; V_{rcr} the rugby club's running speed relative to the train's speed (when the members of the rugby club are, in fact, running).

A number of variables contain the state of the model. There is *mode*, already mentioned. There are a number of variables representing mass: m_{in}, the inertial mass of the system at any time; m_{pcv}, the mass perceived by the train at any time (based on the dynamical properties that it measures and the moments in time that it does so); v_{rcr}, the rugby club's running speed relative to the train at any time (regardless of whether the rugby club are, in fact, running or not at that time); *brTime*, the train's concept of the needed duration for the braking period, at the start of the braking period. These variables are mode variables, because they only need to get updated via mode events.

There are also pliant variables: m_{eff}, the effective mass of the system, i.e. the point mass which, when traveling at the train's velocity, would possess the same momentum as the whole train plus rugby club system, thus offering the same resistance to change in momentum as the whole system—it changes continuously when the rugby club is running during acceleration or braking, due to the continuously changing relative proportions that the train and the rugby club contribute to the overall momentum during the accelerating or braking episodes; v_T, the speed of the train at any time; *brDist*, the current remaining distance during the braking period until the train comes to a standstill, as computed by the train according to the dynamical properties that it measures.

In reality, not all of these variables are strictly necessary. Many can be dispensed with as they can be re-expressed in terms of constants and other variables. The variables in this category are: m_{in}, m_{pcv}, v_{rcr}, m_{eff} and *brTime*. We nevertheless retain them in order to make the ensuing explanation of the model easier to follow.

The invariants are trivial typing invariants in this simple model: *mode* is as described earlier, and the others are all either reals or non-negative reals. We discuss some possibilities for more complex invariants later.

We turn to what the model actually does. In order to save space in Fig. 2, we have economised on some notational matters. Thus: We have decanted events' STATUS declarations to a decoration at the end of the line containing the event name (where the STATUS is not 'ordinary'). We have used the 'async' STATUS to ensure a mode event does not execute eagerly.[8] We have slightly generalised the CONST declaration of [2] to cover a list of (pliant) variables that are to stay constant during the execution of a pliant event.

[8] Thus, 'STATUS async' is an abbreviation for the semantic device of giving the mode event an external input which is not used in the event's body. See footnote 7.

INITIALISATION starts the model with the train stationary in a station with no one on board. A clock *clk_A*, is set to an innocuous value $BIGT$; the *mode* is $STAT$; all the masses are set to be the train's inertial mass M_T; the train's velocity and the rugby club's relative velocity are set to zero; and all other variables are of no interest and are also set to zero.

The ensuing pliant event *TrainStationary* just perpetuates this state of affairs—all mode variables cannot change, and the pliant variables are held constant via the CONST declaration.

At some point during this phase the async event *RugbyClubBoards* is executed. Although boarding clearly does not take place instantaneously, only the overall change in mass makes any difference, and so there is no harm in modelling the process as an impulsive change to the mass during this event. The inertial mass m_{in} becomes $M_T + M_{rc}$, as does the effective mass m_{eff} (since the train system behaves as a single mass at this stage). Everything else stays the same. In particular, the train's perceived mass m_{pcv} remains unchanged since the train is, as yet, unaware of the rugby club. After this the *TrainStationary* event resumes, all variables maintaining their values, whether old values or newly acquired values.[9]

At some point after this the dynamics starts, and for this, we assume an conventionally idealised setup. Thus we assume the track is straight and level, the movement of the train is frictionless and suffers no other impediment (such as air resistance), and the train can be treated as one (or several) point mass(es) for the purpose of dynamical calculations.

In complex situations, dynamics is best treated via the d'Alembert-Lagrange approach, or an equivalent. See e.g. [15,17]. The foundations are not in fact as uncomplicated as the ancient pedigree of this subject might suggest; [12] gives a good discussion, not to mention the gargantuan [21]. For us, it will suffice to stick to a fairly low-level approach, *provided* we remember that Newton's Second Law equates *force* to rate of change of *momentum*, and not to mass times acceleration, as is usually stated, and to which the more accurate form usually reduces.

So, async event *TrainStarts* executes. It changes the mode to *ACCEL*erating and starts the clock. It thus enables the *TrainAccelerating* pliant event which states how the pliant variables change. Since the rugby club are stationary, the effective mass m_{eff} remains CONSTant at its value at the start of *TrainAccelerating*. The braking distance variable *brDist* is not needed yet, so is kept at zero.[10] The nontrivial element of the *TrainAccelerating* event is the

[9] In fact, *RugbyClubBoards* remains enabled during the resumed *TrainStationary* event, so could execute again. But *RugbyClubBoards* is async and idempotent, so no harm would be done.

[10] This could also have been handled via a CONST declaration. In fact, that would have been more convenient, since assignment to a (time dependent, in general) expression generates a verification condition to check that the initial value of the expression agrees with the value on entry to the pliant event, in order to ensure right continuity of the variable's history at the entry point to the pliant event, as required by the semantics [8]. Not mentioning *brDist* at all would entail the default behaviour for pliant variables during pliant events, namely of constraining them to simply obey any relevant invariants. This would be inappropriate here.

ODE that equates the rate of change of the train's momentum $\mathcal{D}\,(m_{in}\; v_T)$ to the force applied by the train. The latter is assumed to be a constant accelerating force F_A. Since there is no relative motion between the train and rugby club, and all the train and rugby club mass is treated as concentrated at the centre of gravity, we can take the mass element to be the intertial mass m_{in}, and we derive the statement found in Fig. 2.

Acceleration continues until the train achieves cruising velocity, detected by the guard $v_T = V_{cr}$ of the mode event *TrainAtSpeed*. This turns off the accelerating force and changes the mode to *CRUISE*ing. This also enables the train to calculate its overall perceived mass m_{pcv} from the information it has, namely clock value *clk_A*, applied force F_A and cruise velocity V_{cr}. Of course, since the motion has been so simple thus far, a straightforward application of Newtonian mechanics (namely, that change of momentum $m_{pcv} \times V_{cr}$ equals duration of applied force $F_A \times clk\_A$) shows that the answer m_{pcv}, is m_{in}, as noted in the accompanying comment, but the train can only use the information available to it, so we show the assignment to m_{pcv} expressed using those quantities.

TrainAtSpeed enables the *TrainCruising* pliant event. *brDist* is still not needed, so is assigned as previously. The train velocity v_T is controlled by a linear constant coefficients ODE, impelling v_T towards the stable equilibrium value V_{cr}. Since $v_T = V_{cr}$ immediately after *TrainAtSpeed*, there is no change in velocity at this time. Similarly, the effective mass m_{eff} remains as before, which is easy to see in the direct assignment to m_{eff} when we notice that $v_{rcr} = 0$ during this period.

During *TrainCruising*, async mode event *RugbyClubStartsRun* is enabled, and at some point is executed. Now the dynamics gets more interesting. Again we idealise the change of state as an impulsive change, since only the overall change in momentum matters, and the dynamics is completely lossless. The rugby club's relative velocity with respect to the train v_{rcr}, becomes V_{rcr}. Since momentum is conserved, using primes for after-values, we can write:

$$(M_T + M_{rc})v_T = (M_T + M_{rc})v'_T + M_{rc}\,V_{rcr} \tag{3}$$

from which, noting that $v_T = V_{cr}$, we derive the assignment to v_T that we see in *RugbyClubStartsRun*. The train effective mass m_{eff} becomes the mass that is needed to generate the momentum on either side of (3) when the velocity is the new train velocity. A slightly longer calculation, equating (3) to $m'_{\text{eff}}v'_T$, is needed to derive the expression for m_{eff} (given in terms of the cruise velocity V_{cr}) that appears in *RugbyClubStartsRun*.

After *RugbyClubStartsRun*, *TrainCruising* is still enabled. Since the train velocity is no longer V_{cr}, the feedback control law in *TrainCruising* now has work to do. Implicitly, an accelerating force is applied to impel the train velocity v_T towards V_{cr}, and it does work that adds to the total momentum of the system. Note that as v_{rcr} is non-zero, having become V_{rcr}, and given that v_T changes, so does m_{eff}, as can be derived from (3), reflecting the changing proportion of the overall momentum that the rugby club's relative run velocity contributes.

When v_T has returned close enough to V_{cr}, the async mode event *TrainBrakes* becomes enabled—we are assuming that the train velocity has recovered before the train starts to brake. We assume the train knows where it is relative to the next stopping position, and initiates braking at a point where, according to the train's perception about its dynamics, applying its fixed braking force F_D for an appropriate time will bring it to a halt just where needed. We assume that the train still imagines its overall mass is the originally calculated m_{pcv}, and taking the velocity to be V_{cr}, a simple Newtonian mechanics calculation of the (quadratic) displacement generated by a constant force yields the *brDist* value assigned in *TrainBrakes*, assuming further that the next stopping position is the origin of distance measurements, and that positive distances are oriented beyond the stopping position. The time taken to come to a halt is recorded in *brTime*—it is just the time needed to consume all of the assumed momentum $m_{pcv}\,V_{cr}$ by applying a force of magnitude F_D.

TrainBrakes changes the mode to *DECEL*erating, and thus enables pliant behaviour *TrainDecelerating*. During this period, it is the laws of physics, and not the train's perceptions, that determine what happens. Thus, the rate of change of velocity is governed by the momentum form of Newton's Law:

$$\mathcal{D}\,((M_T + M_{rc})v_T + M_{rc}\,V_{rcr}) = -F_D \tag{4}$$

In (4), only v_T can vary, the other symbols being constants. Thus we derive the ODE for v_T in *TrainDecelerating*. And v_T gives the time derivative of *brDist*. The effective mass m_{eff} is given by the same formula as in *TrainCruising*, for exactly the same reasons.

At some point during *TrainDecelerating*, but while the velocity is still positive, the rugby club come to the end of their run. The momentum that they 'stole' from the train when they initiated their run, and which was unknowingly made up by the feedback control law during *TrainCruising*, is now suddenly dumped back into the train when they do their jump-stop.

The physical consequences of this process are described in the async mode event *RugbyClubJumpStop*. The rugby club relative run velocity v_{rcr} changes from V_{rcr} to zero. Since the train system now behaves once more like a point mass, the effective mass must likewise become m_{in}. The process is governed by conservation of momentum, which, using primes for after-values as usual, yields the following:

$$m_{\text{eff}}\,v_T = m_{in}\,v'_T = m_{pcv}\,v'_T = m'_{\text{eff}}\,v'_T \tag{5}$$

This explains the assignments to the variables in *RugbyClubJumpStop*.

Once *RugbyClubJumpStop* has executed, *TrainDecelerating* is enabled once more.[11] But now, the train velocity which the decelerating phase has to deal with is suddenly greater than before. So the braking phase is extended compared with its previously anticipated duration.

[11] While *TrainDecelerating* executes, *RugbyClubJumpStop* is (more or less) always enabled, but as in other cases, it is an async event and its effect is idempotent, so executing it again would have no discernible effect.

It is intuitively clear that if the rugby club consists of extreme lightweights, and that if they run extremely slowly, the effect on the braking episode will be small due to the small amount of momentum at issue. Equally, if the rugby club are all very heavy, and they run very fast, then the effect on the braking episode will be more appreciable.

Mode events *TrainStopSucceed* and *TrainStopFail* handle these two possibilities. One or other is triggered when v_T hits zero (and the mode is still *DECEL*erating). The ideal stopping position is at $brDist = 0$. So if the discrepancy between the ideal and actual stopping positions when v_T hits zero does not exceed *BR*aking*TOL*erance, then *TrainStopSucceed* executes, and the train stops at the station, as it should. The state returns to its initial configuration and the rugby club can alight (presumably disappointed). The whole scenario can then repeat.

However, if the discrepancy between the ideal and actual stopping positions when v_T hits zero exceeds *BR*aking*TOL*erance, then the train returns to *ACCEL*erating mode, and the train moves towards the next station, with the (presumably elated) rugby club on board. In this case we have the classic rugby club problem which can then also repeat.

3.1 Analysis of the Jump-Stop Phenomenon

In this section we analyse more precisely the distinction between the *TrainStopSucceed* and *TrainStopFail* cases.

During an execution of *TrainDecelerating*, if the intial velocity of the train is v_{IN} and the pliant event executes for a time t_{EX}, then after this period, the velocity and distance travelled become:

$$v_{IN} - \frac{F_D\, t_{EX}}{(M_T + M_{rc})} \qquad \text{and} \qquad v_{IN}\, t_{EX} - \frac{F_D\, t_{EX}^2}{2(M_T + M_{rc})} \tag{6}$$

respectively. To work out the implications of the jump-stop, we need to consider two such episodes, separated by a *RugbyClubJumpStop*.

The braking period starts with the train moving forward with velocity V_{cr}. Suppose the rugby club do their jump-stop t_{JS} later than the start of braking. Then, substituting into (6), after the first braking episode, the velocity and distance travelled become:

$$v_{JS} = V_{cr} - \frac{F_D\, t_{JS}}{(M_T + M_{rc})} \qquad \text{and} \qquad d_{JS} = V_{cr}\, t_{JS} - \frac{F_D\, t_{JS}^2}{2(M_T + M_{rc})} \tag{7}$$

Then comes the jump-stop. According to *RugbyClubJumpStop*, the velocity needs to be rescaled by m_{eff}/m_{in}, which increases the velocity expression in (7) to:

$$v_{JS}\left[m_{in}+\frac{M_{rc}\,V_{rcr}}{v_{JS}}\right]\Big/\, m_{in} \;=\; V_{cr}-\frac{F_D\,t_{JS}}{(M_T+M_{rc})}+V_{rcr}\left[1+\frac{M_T}{M_{rc}}\right]^{-1} \;=\; v'_{JS} \tag{8}$$

Braking is then completed by another *TrainDecelerating* episode. This time the initial velocity is v'_{JS}. Using (6) with this intial value, the pliant behaviour executes until the velocity drops to zero. Naming this duration t_{HALT}, it is given by:

$$v'_{JS}-\frac{F_D\,t_{HALT}}{(M_T+M_{rc})} \;=\; 0 \qquad \text{thus} \qquad t_{HALT} \;=\; \frac{(M_T+M_{rc})\,v'_{JS}}{F_D} \tag{9}$$

and therefore, the distance covered in the second *TrainDecelerating* episode is, by (6):

$$d_{HALT} \;=\; v'_{JS}\,t_{HALT}-\frac{F_D\,t^2_{HALT}}{2(M_T+M_{rc})} \tag{10}$$

The total distance travelled during braking is therefore $d_{TOT}=d_{JS}+d_{HALT}$, subject to the constraint that $v_{JS}>0$. If we call $brDist_{TOT}$, the value of *brDist* assigned by *TrainBrakes*, it is the discrepancy between d_{TOT} and $brDist_{TOT}$ that must be compared to *BRTOL* to determine whether *TrainStopSucceed* or *TrainStopFail* will be enabled. We find:

$$\begin{aligned}
d_{TOT} \;&=\; V_{cr}\,t_{JS}-\frac{F_D\,t^2_{JS}}{2(M_T+M_{rc})}+v'_{JS}\,t_{HALT}-\frac{F_D\,t^2_{HALT}}{2(M_T+M_{rc})}\\
&=\; V_{cr}\,t_{JS}-\frac{F_D\,t^2_{JS}}{2(M_T+M_{rc})}+\frac{(M_T+M_{rc})\,v'^2_{JS}}{F_D}-\frac{F_D\,(\frac{(M_T+M_{rc})\,v'_{JS}}{F_D})^2}{2(M_T+M_{rc})}\\
&=\; V_{cr}\,t_{JS}-\frac{F_D\,t^2_{JS}}{2(M_T+M_{rc})}+\frac{(M_T+M_{rc})\,v'^2_{JS}}{2\,F_D}\\
&=\; V_{cr}\,t_{JS}-\frac{F_D\,t^2_{JS}}{2(M_T+M_{rc})}\\
&\quad+\frac{(M_T+M_{rc})}{2\,F_D}\left(V_{cr}-\frac{F_D\,t_{JS}}{(M_T+M_{rc})}+V_{rcr}\left[1+\frac{M_T}{M_{rc}}\right]^{-1}\right)^2
\end{aligned} \tag{11}$$

After a bit more working we get:

$$\begin{aligned}
d_{TOT} \;=\; \frac{(M_T+M_{rc})}{2\,F_D}\Bigg[&V^2_{cr}+V^2_{rcr}\left[1+\frac{M_T}{M_{rc}}\right]^{-2}+2\,V_{rcr}\,V_{cr}\left[1+\frac{M_T}{M_{rc}}\right]^{-1}\\
&-2\,V_{rcr}\frac{F_D\,t_{JS}}{(M_T+M_{rc})}\left[1+\frac{M_T}{M_{rc}}\right]^{-1}\Bigg]
\end{aligned} \tag{12}$$

So

$$d_{TOT} \;<\; \frac{(M_T+M_{rc})}{2\,F_D}\left[V^2_{cr}+V^2_{rcr}\left[1+\frac{M_T}{M_{rc}}\right]^{-2}+2\,V_{rcr}\,V_{cr}\left[1+\frac{M_T}{M_{rc}}\right]^{-1}\right] \tag{13}$$

The last step follows, because in the last two terms of (12), the negative one cannot exceed the positive one in magnitude because of the constraint on t_{JS} coming from $v_{JS} > 0$, which implies that t_{JS} reaches its maximum when $v_{JS} = 0$, at which point these last two terms cancel.

Therefore, if we can arrange it that $\mid \ brDist_{TOT} + d_{TOT} \mid < BRTOL$, we will always execute *TrainStopSucceed* at the end of the braking process, and will never execute *TrainStopFail*. We will have surmounted the rugby club problem.

We can express this insight as an additional, nontrivial invariant, where *HYP* denotes the relationships between the various constants of the model that have to be true in order that $\mid \ brDist_{TOT} + d_{TOT} \mid < BRTOL$ holds:

$$HYP \vdash \ mode = DECEL \wedge v_T = 0 \ \Rightarrow \ \mid \ brDist \ \mid \leq BRTOL \tag{14}$$

Thus, the enabledness of *TrainStopSucceed* at the crucial moment becomes provable.

Regarding $brDist_{TOT} + d_{TOT}$, which equals the last two terms of (13), we note that the V_{rcr}^2 term will be negligible in magnitude compared with the $V_{rcr}\, V_{cr}$ term. This enables us to derive a simple criterion that will be adequate for most engineering purposes:

$$brDist_{TOT} + d_{TOT} \ < \ \frac{V_{rcr}\ V_{cr}\ M_{rc}}{F_D} \tag{15}$$

4 The Rugby Club Problem—Further Discussion

The details of the control strategy actually used for urban rail control are commercially confidential, for obvious reasons. Nevertheless, it seems clear that the fact that there is a rugby club problem at all, signals a likely cause of it as being the discrepancy between a control strategy based on pure kinematics and one based on the complete dynamics. In this section, we briefly some discuss issues for more realistic modelling.

Several factors would need to be taken into account in a more realistic model: the track will not be straight and level; it will not sustain frictionless train travel; the train's wheels will not always make perfect rolling contact with the track (there will be some skidding at times); the control laws will not be as simple as we have chosen them to be in our models; in the confines of an underground tunnel, air resistance will cause significant drag on the train. And so on.

All these things will soak up some of the momentum of the train as it travels, requiring work from the engine to maintain speed. Simple realistic models of these phenomena will not be available. The best one might hope for, would be phenomenological models that predicted the relevant losses, based on tabulated data taken over many journeys under a variety of conditions. These data would have to be specific to each section of the route, and dealing with these aspects could seriously complicate the design of the critical code controlling the train's motion.

5 'Tackling' the Rugby Club Problem

Above, we suggested that if appropriate relationships could be made to hold between the various constants that characterised our model, then the rugby club problem might be overcome. In this section, we discuss how the rugby club problem may be addressed when such choices of constants are not available for whatever reason, while remaining within the simple modelling framework of Fig. 2.

In our model, the principal cause of the loss of coherence between the train's view of the dynamics and the physical reality could be attributed to the fact that the control law for the cruise phase was based exclusively on the train's velocity, whereas the true physics of the situation requires the accounting of momentum.

The obvious suggestion then, would be to change the control laws for the various phases of the dynamics to account for momentum more accurately. In our extremely idealised models this would not be hard to do, because in such simple models, the relationships between velocity and momentum are straightforward, and the cruise phase could easily detect how much momentum it had given away as it brought the train back up to speed. The train could then approach the stopping point more cautiously, knowing that the momentum it had given away would have to be given back soon.

However, when we consider doing the same thing in the context of the more realistic models contemplated in Sect. 4, this is easier said than done. The rugby club steals momentum from the train, but so do all the other sources of non-ideal motion that we mentioned. Distinguishing between 'natural losses' and 'unnatural losses' becomes nontrivial. Nevertheless, if natural and unnatural could be distinguished clearly enough, an optional 'more cautious stopping strategy' offers a potential way forward.

6 Summary and Conclusions

In the preceding sections we outlined the essentials of Hybrid Event-B, with a special focus on how impulsive physics can be handled. Then we constructed a Hybrid Event-B model of the rather engaging rugby club problem scenario described in the Introduction. For the purposes of arriving at a reasonably clear exploration of the rugby club problem which nevertheless fitted in a fairly short paper, our model had to simplify and idealise the situation rather severely. It was thus suffused with point mass and lossless dynamics in the familiar style of classical mechanics. The precision of the model allowed us to derive conditions that distinguished between the non-disruptive and disruptive case of the rugby club dynamics, and we discussed some options for adding more complex invariants to the model, based on these. We then discussed possibilities for reducing the degree of idealisation in the model, and thus the prospects for making it more realistic, thereby bringing it closer to applicability in practice.

It is worthwhile, at this point, making an observation about how the stated provability of the additional invariants that were mentioned came about. Most of

the analysis of this paper was performed in a fairly *ad hoc* manner. When dealing with a situation described by physical theories, this is, more or less, unavoidable. It follows in turn because physical theories are almost always expressed using a family of equalities. As such, any of the participating variables may (in the given situation) carry input values, with the other variables acquiring their values from the demanded equalities, as outputs. So the derivation process is not structured in a manner that is fixed at the outset, in the way that formal development processes tend to be. However, once the *ad hoc* reasoning has yielded its fruits, we can take a step back, and restructure what has been discovered in a manner that better fits a formal development process. It is in this manner that the provability that is claimed of the additional invariants emerges.

In the last section, we addressed how this modelling exercise could be used to overcome the rugby club problem, in cases where it could not be prevented by choosing appropriate constants. The crux of the matter would be to centre the control system for the train more firmly on the momentum dynamics of the physical system, than on purely kinematic aspects. Confidence in this assertion is supported by the fact that although a rugby club may be able to outwit a train control system whose design is insufficiently suspicious, they cannot cheat the laws of physics.

It is instructive to note the very major role played by physics knowledge in the exercise undertaken in this paper. Although computer scientists often find it convenient to downplay or neglect the influences of non-computing disciplines in the design of cyberphysical systems [13,16] (see, for example, the balance of content in references such as [1,20]), the importance of such influences cannot be denied, and the present exercise shows this eloquently. Cyberphysical systems are truly multidisciplinary and it is unwise to neglect any of the disciplines that contribute to a given system while emphasising just one (e.g. just the computing viewpoint). See [10] for a review of some of the less obvious elements that impact cyberphysical systems, discussed from a mathematical viewpoint.

References

1. Alur, R.: Principles of Cyberphysical Systems. MIT Press, Cambridge (2015)
2. Banach, R.: Pliant modalities in hybrid event-B. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) Theories of Programming and Formal Methods. LNCS, vol. 8051, pp. 37–53. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39698-4_3
3. Banach, R.: The landing gear system in multi-machine hybrid Event-B. Int. J. Soft. Tools for Tech. Trans. **19**, 205–228 (2017)
4. Banach, R.: Formal refinement and partitioning of a fuel pump system for small aircraft in hybrid Event-B. In: Bonsangue, M., Deng, Y. (eds.) Proceedings of IEEE TASE-16, pp. 65–72. IEEE (2016)
5. Banach, R.: Hemodialysis machine in hybrid event-B. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) ABZ 2016. LNCS, vol. 9675, pp. 376–393. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33600-8_32
6. Banach, R., Butler, M.: A hybrid Event-B study of lane centering. In: Aiguier, M., Boulanger, F., Krob, D., Marchal, C. (eds.) Complex Systems Design & Management, pp. 97–111. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-02812-5_8

7. Banach, R., Butler, M.: Cruise control in hybrid Event-B. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) ICTAC 2013. LNCS, vol. 8049, pp. 76–93. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39718-9_5
8. Banach, R., Butler, M., Qin, S., Verma, N., Zhu, H.: Core hybrid Event-B I: single hybrid Event-B machines. Sci. Comput. Program. **105**, 92–123 (2015)
9. Banach, R., Butler, M., Qin, S., Zhu, H.: Core hybrid Event-B II: multiple cooperating hybrid Event-B machines. Sci. Comput. Program. **139**, 1–35 (2017)
10. Banach, R., Su, W.: Cyberphysical systems: a behind-the-scenes foundational view. In: Mashkoor, A., Thalheim, B., Wang, Q. (eds.) Proceedings of Klaus-Dieter Schewe Festschrift 2018. College Publications (2018, to appear)
11. Banach, R., Van Schaik, P., Verhulst, E.: Simulation and formal modelling of yaw control in a drive-by-wire application. In: Proceedings of FedCSIS IWCPS-15, pp. 731–742 (2015)
12. Bloch, A., Krishnaprasad, P., Murray, R., Baillieul, J., Crouch, P., Marsden, J., Zenkov, D.: Nonholonomic Mechanics and Control. Springer, New York (2015). https://doi.org/10.1007/b97376
13. Carloni, L., Passerone, R., Pinto, A., Sangiovanni-Vincentelli, A.: Languages and tools for hybrid systems design. Foundations and Trends in Electronic Design Automation, vol. 1, pp. 1–193 (2006)
14. ClearSy. http://www.clearsy.com/
15. Fasano, A., Marmi, S.: Analytical Mechanics. Oxford University Press, Oxford (2013)
16. Geisberger, E., Broy, M. (eds.): Living in a networked world. In: Integrated Research Agenda Cyber-Physical Systems (agendaCPS) (2015). http://www.acatech.de/fileadmin/user_upload/Baumstruktur_nach_Website/Acatech/root/de/Publikationen/Projektberichte/acaetch_STUDIE_agendaCPS_eng_WEB.pdf
17. Goldstein, H., Poole, C., Safko, J.: Classical Mechanics. Addison Wesley, Boston (2001)
18. Horvarth, J.: Topological Vector Spaces and Distributions. Dover, Mineola (2012)
19. Lecomte, T.: Atelier B has Turned 20. In: Proceedings of ABZ-16, LNCS, vol. 9675, p. XVI. Springer (2016)
20. Lee, E., Shesha, S.: Introduction to Embedded Systems: A Cyberphysical Systems Approach. 2nd edn (2015). LeeShesha.org
21. Papastavridis, J.: Analytical Mechanics: A Comprehensive Treatise on the Dynamics of Constrained Systems, 2nd edn. World Scientific, Singapore (2014)
22. Platzer, A.: Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14509-4
23. Tabuada, P.: Verification and Control of Hybrid Systems: A Symbolic Approach. Springer, Heidelberg (2009). https://doi.org/10.1007/978-1-4419-0224-5
24. Treves, F.: Topological Vector Spaces, Distributions and Kernels. Dover, Mineola (2007)
25. Zemanian, A.: Distribution Theory and Transform Analysis: An Introduction to Generalized Functions, with Applications. Dover, Mineola (2003)

Refinement

Clarification of Ambiguity for the Simple Authentication and Security Layer

Farah Al-Shareefi(✉), Alexei Lisitsa, and Clare Dixon

Department of Computer Science, University of Liverpool,
Liverpool L69 3BX, UK
{F.M.A.Al-Shareefi,lisitsa,cldixon}@liverpool.ac.uk

Abstract. The Simple Authentication and Security Layer (SASL) is a framework for enabling application protocols to support authentication, integrity and confidentiality services. The SASL was originally specified in RFC 2222, and later updated in RFC 4422, using natural language. However, due to the richness of natural language this involves ambiguities and imprecision. Whilst there is an Oracle implementation of SASL, its documentation also contains informal descriptions and under-defined specifications of the RFCs. This paper provides clarification of ambiguity in SASL using Abstract State Machines (ASMs). This clarification is based on two ASM essential notions: a ground model to capture the intended SASL behavior in an understandable way, and a refinement notion to accurately explicate the ambiguous parts of the behavior. We also show some differences between RFCs and the description of the Oracle implementation. We believe our work can serve as a basis for further implementation and for formal analysis.

Keywords: Ambiguity · Simple Authentication and Security Layer
Abstract State Machines

1 Introduction

The Simple Authentication and Security Layer (SASL) is a framework that can be used by application protocols to perform authentication, and to optionally supplement it with what is called security layer services, including integrity and confidentiality. SASL was firstly described in Requests for Comments (RFC) 2222 [14], and then in RFC 4422 update [13], using natural language. Unfortunately, the RFCs, being stated in natural language that intrinsically has associated informality and imprecision, are sometimes ambiguous. Despite that, there is an Oracle implementation of SASL [15], its documentation also includes textual explanations and some unclear specification of the RFCs. In addition, this implementation involves hidden details about the functions which are called to achieve specific tasks.

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 189–203, 2018.
https://doi.org/10.1007/978-3-319-91271-4_13

To overcome the imprecision and ambiguity problems, formal methods can be used as they are based on mathematical foundations [8]. Among these methods, we choose the Abstract State Machine (ASM) method [11], since it can be used to specify systems in a rigorous mathematical, understandable, and scalable way [6].

In this paper, the ambiguities of the SASL textual explanations in the RFCs and Oracle implementation documents, are analyzed formally using the ASM method. This is achieved by implementing two strategies of the ASM method. First, the ground model directly captures the informal SASL behavior in an understandable and concise but precise enough manner. Second, the refinement strategy allows us to precisely explicate and re-elaborate the under-defined notions in the ground model. The refined specification is written in the executable ASMETA Language (AsmetaL) [10], since it is close to the ASM mathematical concepts, and it directly permits us to test specification errors.

The main contributions of this paper are:

- clarifying the ambiguities of the SASL informal descriptions in RFCs and Oracle implementation documents clearly in terms of ASMs;
- presenting a methodology for clarifying ambiguity that starts with the RFC document to capture its informal description, via the ASM ground model, then it explicates the potential description ambiguities depending on other document sources by using ASM refinement;
- highlighting the main differences between RFCs and the Oracle documents.

The rest of this paper is organized as follows. Section 2 presents background knowledge about the SASL framework and the ASM method. Section 3 describes the ASM formal specification, and highlights how this specification elucidates the main ambiguities of SASL. Section 4 discusses our results. Section 5 presents some related work. Finally, Sect. 6 concludes the paper.

2 Background

In this section, we describe both the Simple Authentication and Security Layer framework and the Abstract State Machine Method.

2.1 Simple Authentication and Security Layer

The Simple Authentication and Security Layer (SASL) was initially introduced in RFC 2222 [14], and later updated in RFC 4422 [13], as a framework for providing authentication support with an optional security layer service, such as integrity or confidentiality, to connection-oriented protocols, via substitutable mechanisms. Providing these services is achieved through using a shared abstraction layer which has a structured interface between intended protocols and mechanisms. With this layer, any SASL supported protocol, such as IMAP [18], SMTP [19], etc., can exploit any SASL supported mechanism, such as PLAIN [20], DIGEST-MD5 [12], etc.

Based on RFC 2222/4422 [13,14], the client and server of the SASL protocol application launch a negotiation about the selection of a suitable mechanism, then they negotiate the authentication. Basically, the client requests to connect with the server using SASL. Then, the server replies with a list of supported authentication mechanisms. Next, the client selects the best mechanism. After that, the authentication is started by the client via sending an authentication command, which involves the selected mechanism and optionally authentication data, to the server. The authentication exchange continues until the authentication succeeds, fails, or is aborted by the client or the server. During the authentication exchange, when the selected mechanism supports the security layer, the client and server negotiate the use of a security layer. If they both agree about using it, then both sides must negotiate the maximum size for the cipher text buffer, that each side is able to receive. The RFCs specification, however, leaves open a number of questions, in particular: how the server advertises its mechanisms' list, how the client selects the best mechanism, when the client and server agree about using the security layer and how it can be used, and how they negotiate the maximum cipher text buffer size. Some of these questions relate to the ambiguity and missing details of the informal description for the API routines in the Oracle implementation documentation [15].

According to the Oracle implementation [15], the application communicates with the structured interface by calling a suitable API routine, which in turn calls a mechanism plug-in interface. One of these routines is: the `sasl_client_start()` which is called by the client to select the best mechanism depending on the security properties. The main properties that restrict mechanism selection are: the **security policies**, such as NOPLAINTEXT, NOACTIVE, NOANONYMOUS, etc., and the maximum **Security Strength Factor (SSF)** [15] for the client, server, and mechanism. The SSF is an integer that denotes the security layer strength. When it is zero, it indicates only authentication, if it is one, it means both authentication and integrity, while if it is greater than one, it denotes authentication, integrity, confidentiality, and at the same time the key length for encryption. Also the server calls the `sasl_listmech()` routine to obtain the mechanisms' list that satisfies the security policies.

2.2 Abstract State Machines

Abstract State Machines (ASMs) [11] were first introduced by Gurevich as a versatile machine to model any algorithm at an appropriate level of abstraction. ASMs have been developed to a practical and mathematically well-founded method for high-level system design and analysis [6]. The ASM method is constructed from three essential notions: *ASMs, a ground model, and stepwise refinement* [6].

ASMs are transition systems which are based on abstract states, to model the system's structure, and on transition rules, to model the system's dynamic behavior. The ASM states are multi-sorted first-order structures, i.e, domains of objects coming with functions and relations defined on them. The functions in ASM states can be *static*, which are never updated, *controlled*, which are updated

by the machine itself, or *monitored*, which are updated by the machine's environment. ASM transition rules describe the modification of function interpretations from one state to the subsequent one. The basic transition rule is a *function update*: $f(t_1, \ldots, t_n) := t$. f is an arbitrary n-ary function and $t_1, \ldots, t_n, t$ are first-order terms, which are simultaneously updated to yield a new ASM state. There are some rule constructors, such as: **if then** (conditional rule), **par** (parallel execution of the grouped rules), **choose** (non deterministic selection), and **switch case** (extension of the conditional rule). ASMs can capture the formalization of a procedural *single-agent* and distributed *multiple agents* interacting in a synchronous and asynchronous way.

There is a specific class of ASMs called *control state ASMs* [6]. They can be employed for describing various system modes. Figure 1 shows a graphical representation of control states and the form of their transition rules.

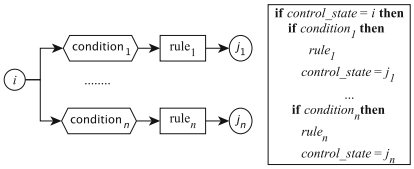


Fig. 1. Control state ASMs

Ground model ASMs are conceptual models for capturing informal requirements of a system in a precise, concise, flexible, and understandable way. The ground model can be represented graphically using control state ASMs. From a concise ground model, by **stepwise refinement**, a more detailed model can be obtained, through changing the states definition, or the flow of operations, or both of them.

Several projects have been developed around ASMs to make them executable, such as CoreASM [1], the ASMETA[1] [9] framework, etc. In this paper, we have chosen the ASMETA framework, that includes integrated tools, in particular the ASMETA Language (AsmetaL) and the ASMETA Simulator (AsmetaS) for writing and executing ASM models [10], respectively. The AsmetaL supports encoding of ASM models which is close to the ASM mathematical concepts.

3 The Formal SASL Specification

In this section, we show how the ASM method has been used to provide formal specifications for SASL. The main aim is to clarify precisely: how the server advertises the available mechanisms, how the client selects the best mechanism, how the client determines the maximum size for the cipher buffer, and how and when the security layer is negotiated. As the SASL framework has detailed and complex behavior, we separate the SASL into three phases: the mechanism negotiation phase, the authentication negotiation phase, and the security layer negotiation phase. In each phase, we will present (if necessary) the ground model for both client and server sides, that is depicted via the control state ASM, then we will focus only on refining the rules to clarify ambiguities in the RFCs

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

and Oracle implementation documentation[2]. Each refined rule is expressed in AsmetaL. The mapping from the graphical notation of the control state ASM to the AsmetaL notation is done according to the mapping shown in Fig. 1.

3.1 The Mechanism Negotiation Phase

Figure 2 shows the ground model at the client side for this phase. This figure is a direct interpretation of RFC 2222/4422. The client starts this phase by sending a request to ask the server to send its mechanisms' list. Whenever this list is received, the guard *At least one mechanism in the list is supported* checks if the client allows any mechanism in the list. If so, the client selects an acceptable mechanism from the server list and reaches the final state for this phase *Sending authentication request*. Otherwise, the client will send an abort response to the server, and waits for an abort reply from it. When the abort reply is received, the client aborts this exchange, by entering the *Abort* state.

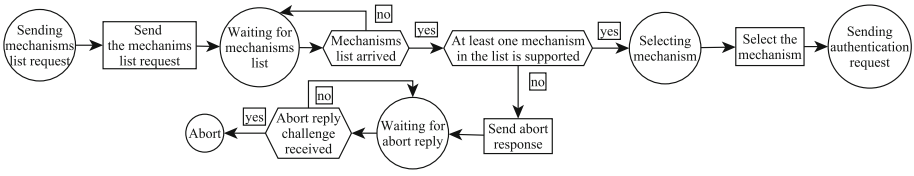


Fig. 2. Client side for mechanism selection phase - ground model

The server side for this phase is also based on RFC 2222/4422, see Fig. 3.

Fig. 3. Server side for mechanism selection phase - ground model

The ground model depicted in Fig. 3, schedules the main steps taken by the server for this phase. Initially, the server keeps waiting at the *Waiting for mechanisms list request* state until it receives a mechanism list request from the client. When this request arrives, the server obtains the available mechanisms' list to send it to the client. After sending this list, the server goes to the final state for this phase, which is *Waiting for authentication request*.

[2] All the rules for the refined model that is based on RFC 2222/4422 are available online at https://doi.org/10.5281/zenodo.1204257, while for the refined model which is based on the description of Oracle implementation documentation are available at https://doi.org/10.5281/zenodo.1204242.

It is not clear from Fig. 2, how the client chooses the desired mechanism. As stated by RFC 4422 [13], determining the best mechanism is the client's choice. This is specified in the `r_selectmech` rule shown in Code 1(a). In this Code, the mechanism selection is performed in an interactive manner with the client via the monitored function `insertMechanism`. The selected mechanism should be any mechanism in the arrived mechanisms' set, which is represented by the `arrivedMechList`. An arbitrarily chosen mechanism is stored in `selMech`.

```
rule r_selectmech=
  if contains(arrivedMechList, insertMechanism) then
       selMech:=insertMechanism
  endif
```

(a) The r_selectmech rule according to RFC 4422

```
function mHasGreatestSSF($m in Mechanisms, $c in Client)=
 forall $x in arrivedMechList with
   (($x!=$m) and allin(policies($x), policies($c)) implies
                                                ssf($m)>=ssf($x))
rule r_selectmech=
  choose $m in arrivedMechList with
     allin(policies($m), policies(self)) and
     mHasGreatestSSF($m, self)=true do
         selMech:=$m
```

(b) The refined r_selectmech rule according to Oracle implementation document

Code 1: The r_selectmech rule

On the other hand, the explanation of the Oracle implementation documentation [15] states that the client selects the best mechanism, depending on the maximum mechanism SSF and client's security policy. This can be re-elaborated by refining the rule in Code 1(a) into the one shown in Code 1(b). In the refined rule, we added further modelling vocabulary. Precisely, let the `ssf($m)` function be the SSF value for each mechanism `$m` in the `Mechanisms` domain, and `policies($m)` be the security policies set for each mechanism `$m`, while `policies(self)` is the security policies' set for client. The 0-ary function `self` is interpreted by the client agent as itself. Each `policies` set can be one or more elements from the domain `Policies={NOPLAINTEXT, NOANONYMOUS, NOACTIVE, MUTUALAUTH, NODICTIONARY}`. The `mHasGreatestSSF` function returns true if the selected mechanism has the greatest SSF value. The refined rule picks the best mechanism from the `arrivedMechList`, such that the security policies set of the selected mechanism includes all the elements in the client's set, and this mechanism has SSF value, which is greater than all the SSF values of the mechanisms that their sets include the client's policies set.

On the server side in Fig. 3, getting the available mechanisms' list needs elucidation. As indicated by RFC 4422 [13], the server just advertises the available mechanisms' list. This is specified in the `r_getmechs` rule shown in Code 2(a). In this code, let the `mList($c, self)` be a set of the advertised mechanisms

which will be sent to the `$c` client. The `saslmechs(self)` set contains one or more SASL mechanisms for server use. The server, in the `r_getmechs` rule, will simply make a copy of all the elements in the `saslmechs(self)` set and pass it to the `mList($c, self)`, which is initially empty set, to represent the advertised mechanisms' list.

```
rule r_getmechs($c in Client)=
  mList($c,self):=saslmechs(self)
```

(a) The r_getmechs rule according to RFC 4422

```
rule r_getmechs($c in Client)=
  let ($i=0) in
    while $i<size(saslmechs(self)) do
      seq
       let($m=at(asSequence(saslmechs(self)), iton($i))) in
         if (exist $p in policies(self) with
             contains(policies($m), $p)=true) then
               mList($c,self):=including(mList($c,self),$m)
         endif
       endlet
       $i:=$i+1
      endseq
  endlet
```

(b) The refined r_getmechs rule according to Oracle implementation document

Code 2: The r_getmechs rule

Obtaining the available mechanisms' list is described in the Oracle implementation documentation [15], as "The server can call sasl_listmech() to get a list of the available SASL mechanisms that satisfy the security policy". In this quoted statement, it is not obvious whether there is a specific policy for the SASL mechanisms and what is meant to satisfy this policy. In the Java security guide provided by Oracle [16], it says that there is a particular policy set for each SASL mechanism, such as the {NONANONYMOUS}, {NOPLAINTEXT, NOACTIVE, NODICTIONARY}, and {NONANONYMOUS, NOPLAINTEXT} for the PLAIN, EXTERNAL, and DIGEST-MD5 mechanisms, respectively. As an attempt to understand the exact meaning of 'satisfy the security policy', we analyse the server's reply (sending the available mechanisms' list to the client) in some SASL mechanism examples. For instance, in the DIGEST-MD5 mechanism example [12], the server sends the {PLAIN, DIGEST-MD5} list. We can see that these two mechanisms share the NONANONYMOUS policy. This means that the server adopts the NONANONYMOUS policy and it sends the mechanisms which satisfy this policy. Similarly, in the EXTERNAL mechanism example [13], the server sends the {DIGEST-MD5, EXTERNAL}. Again, these mechanisms in the list share the NOPLAINTEXT policy, which is supported by the server. Accordingly, the `r_getmechs` rule in Code 2(a) can be refined into the rule in Code 2(b).

In the refined `r_getmechs` rule, the server gets a mechanism from the `saslmechs` for the server use, which satisfies the following condition: the policy

set for this mechanism contains a policy of the server's policies set. In other words, the policy set for every mechanism in the advertised mechanisms' list supports at least one server's policy.

3.2 The Authentication Negotiation Phase

This phase is the longest phase in SASL. As a result, we divide the ground model for both client and server into two parts: one for achieving an initial step in this phase, and one for performing the later step(s). The number of the later steps is determined by the selected mechanism. Due to space restrictions we do not provide all of these constructed models[3]. The ground model for the client agent of the initial and later step(s) includes (if necessary) the `Get response` rule to get the required authentication data to the server. While, the ground model for the server agent of the initial and later step(s) includes (if necessary) the `Get challenge` rule to get the required authentication data to the client.

One underspecified aspect of this phase, is the negotiation about using the security layer and the maximum cipher buffer size, which are involved in the `Get response` and `Get challenge` rules on the client and server sides, respectively. In RFC 4422 [13], it was stated that when the selected mechanism supports a security layer, then a negotiation about using this layer must be carried out, but how this negotiation takes place is not defined. However RFC 2222 [14] defines this by stating that the negotiation includes exchanging a **bit-mask** (1: no security layer, 2: integrity, and 4: privacy), which corresponds to a security layer level. This bit-mask defines the unstated privacy service and ignores the confidentiality service.

On the other hand, in the explanation of the Oracle implementation documentation [15], the SSF value (0: authentication, 1: authentication and integrity, and >1: authentication, integrity, confidentiality and the key length), is used instead of a bit-mask. However, it is not clear how the client and server agree about using a security layer.

The Java security guide provided by Oracle [16] states that the selected mechanism, when its SSF value is greater than or equal to 1, tells the server to send its supported **Quality of Protection** (QOP) list, which includes one or more items from the following: **auth** (authentication), **auth-int** (authentication and integrity), and **auth-conf** (authentication, integrity, and confidentiality). Later, the client selects a protection value from this list according to its SSF value, and sends it to the server. The server verifies that the client's protection value is within its list, to save the session SSF value which is equivalent to the client's protection value. The saved SSF value represents the agreed security layer service. However, in this guide, there is insufficient detail about how the client and server determine the maximum buffer size when they agree about using the confidentiality service.

[3] The full ground models are available online at https://doi.org/10.5281/zenodo.1200216.

In the DIGEST-MD5 SASL mechanism example [12], it was stated that when the server sends its supported maximum buffer size (if desired), the client will check the availability of buffer size value in the received challenge. If it exists, the client will determine that the buffer size for this session is equal to subtracting 16 bytes from the minimum size of the received one and the client's supported one. If it is not available, the client will determine that the buffer size is equal to the default value 65536. Following this description, we present in Code 3 the specification of how the client determines the maximum buffer size.

```
if contains(receivCh(self), "maxbuf")=true then
  choose $max in Maxbuf with
  eq(at(receivCh(self), iton(indexOf(receivCh(self), "maxbuf")+1)),
  toString($max)) do
      if $max<maxBuf(self) then
          maxBufDetermined:=$max-16
      else
          maxBufDetermined:=maxBuf(self)-16
      endif
else
    maxBufDetermined:=65520
endif
```

Code 3: Specifying the maximum buffer size in the client side

In Code 3, the `receivCh(self)` is a sequence of String that represents the received challenge from the server, the integer domain `Maxbuf` contains the following possible values for the buffer's size {65535, 131071, 262143, 16777215}, and the `maxBuf(self)` is the client's maximum buffer size. First of all, the client checks if the `receivCh(self)` contains the server's maximum buffer size, to calculate the buffer size, or to set it to the default value. At the calculation, the client chooses an integer value from the `Maxbuf` domain, since the `receivCh(self)` is a sequence of String, which is equal to the string value contained in the `receivCh(self)`. Then the chosen value is compared with the client's buffer size to determine the buffer size, which is stored in `maxBufDetermined`.

3.3 The Security Layer Negotiation Phase

This phase is an optional phase. Performing this phase depends on the negotiation in the previous phase. This negotiation includes exchanging a bit-mask according to RFC 2222 [14], while it includes exchanging SSF value according to the Oracle documentation [15]. As we stated previously that the bit-mask does not define the confidentiality service, we specify this phase relying on the Oracle implementation documentation [15], as well as the RFC 4422 [13]. Furthermore, the specification for integrity and confidentiality protected messages are based on the RFC 2831 for the DIGEST-MD5 SASL mechanism [12], because the RFC 2222/4422 and the Oracle implementation documentation do not illustrate this specification. We annotate the main information for specifying this phase in Fig. 4.

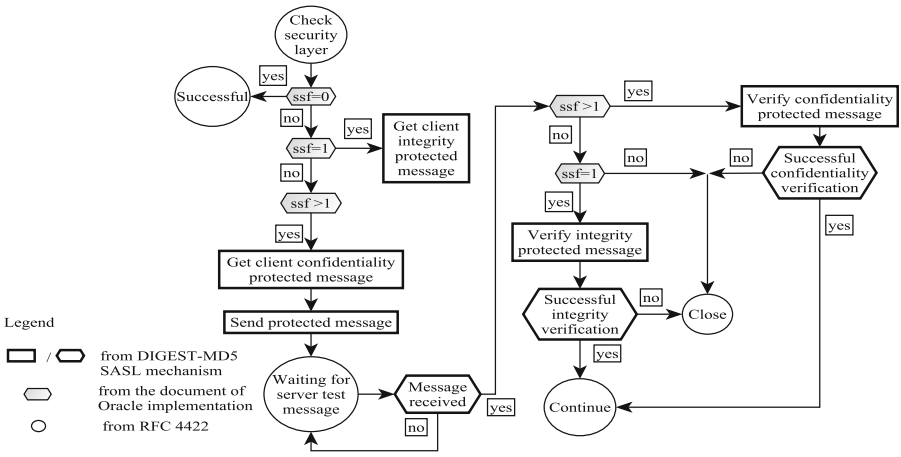


Fig. 4. Client side for security layer negotiation phase - ground model

Figure 4 illustrates the ASM ground model for negotiating the security layer service on the client side. The client starts this phase by checking the SSF value, that was agreed by both client and server in the authentication phase. If this value is zero, then the client will reach the final state *Successful*. This state indicates that the client has been authenticated successfully, and there is no security layer. If the SSF value is one, this means that the subsequent protocol messages must be integrity protected. Therefore, the flow goes to execute the *Get client integrity protected message*, to obtain a test message that appends with the computed Message Authentication Code (MAC) for the message sequence number and the message itself [12]. While if the SSF value is greater than one, then the following protocol messages must be confidentiality protected (encrypted). As a result, the client executes the *Get client confidentiality protected message*, to encrypt a test message together with its computed MAC [12]. The encryption is done according to the selected cipher, which is one of the following: rc4-40 (40 bit key), rc4-56 (56 bit key), rc4 (128 bit key), and aes-ctr (128 bit key). Later, the client sends the protected message to the server, and changes its state to *Waiting for server test message*. Whenever the client receives a protected message from the server, it will check the agreed SSF value. When this value is greater than one, the client will perform confidentiality verification (decrypt the message, compute the MAC and compare it with the received one). While, if the SSF value is one, the client will perform integrity verification (compute the MAC and compare it with the received one). In case that the verification succeeds, the client reaches the final state *Continue*, which means the client can continue the interactions after SASL. Whereas, the client terminates the connection with the server and changes its state to *Close*, when the verification fails.

As the ground model in Fig. 4, clearly shows how the client uses a security layer service, and how the SSF value guides the client to determine whether a security layer has been negotiated, we do not show the refinement for this model.

Furthermore, we do not present the server ground model for this phase, since it is similar to the client one, except that the server keeps waiting for a protected message from the client before sending its message.

As in this phase we need to encrypt and decrypt the message with regards to a suitable cipher, we specify the encryption and decryption actions in an abstract manner based on the ASM features, see Code 4.

```
dynamic abstract domain CipherText
controlled key: CipherText -> String
controlled plainMsg: CipherText -> Seq(String)
controlled plainText: Seq(String)
controlled cipher: CipherText
controlled method: CipherText -> String
rule r_encrypt($msg in Seq(String), $key in String, $method in String)=
  choose $e in CipherText with plainMsg($e)=$msg do
    cipher:=$e
  ifnone
      extend CipherText with $ciph do
       par
         plainMsg($ciph):=$msg
         key($ciph):=$key
         method($ciph):=$method
         cipher:=$ciph
       endpar
rule r_decrypt($cipher in CipherText, $key in String, $method in
                                                              String)=
  choose $ciphtext in CipherText with ($ciphtext=$cipher) and
     (key($ciphtext)=$key) and (method($ciphtext)=$method) do
       plainText:=plainMsg($ciphtext)
  ifnone
       plainText:=""
```

Code 4: Abstract specification for encryption and decryption

In Code 4, first, we introduce the specification signature. The `CipherText` is an infinite domain for the cyphertext. The unary function `key` represents the key for a given `CipherText` element. The nullary function name `plainMsg` is a sequence of Strings for the plain text. The `cipher` is an element of `CipherText` domain. The `method` is a cipher method that has been used to encrypt the plain message.

After the signature we specify two rules: the `r_encrypt` rule for converting the presented plain message into an encrypted one using the determined key and method, and the `r_decrypt` rule which transforms the encrypted message into a plain text one using a specific key and method. The `r_encrypt` rule, firstly, chooses an element in the `CipherText` domain, such that the plain text for this element is equal to the given message. This element represents the cyphertext for the message. When choosing an element returns nothing (the presented message has not been encrypted previously), this rule will generate a new cyphertext, given its plain text, key, and method, by extending the `CipherText` domain. While `r_decrypt` rule choose a cyphertext item from the `CipherText` domain, in such a way that this item equals to the given cyphertext, to return the plain text of this item. If there is no such item, this rule will return the empty string.

4 Results and Discussion

The main aim of this paper is to provide clarification of ambiguities in SASL using ASMs. Our methodology starts with reflecting the textual description in RFCs, using ground model notion, then it re-elaborates this description using other document sources by exploiting the refinement notion. Table 1 outlines the main ambiguities that have been investigated, and the source documents for both the ambiguity itself and its formal clarified specification.

From Table 1, we can see the following:

(1) selection of the best mechanism is ambiguous in RFC 4422 [13], as it just states that the client selects the best one. We try to elucidate this using the description of the Oracle implementation [15], which states that the client selects the best mechanism with the maximum SSF, and according to its security policy;
(2) advertising the available mechanisms' list is not clear in both RFC 4422, which only states that the server advertises the list, and the Oracle implementation, which states that the server advertises the mechanisms that satisfy the security policy. We convert the informal description of Oracle into a formal one, based on analysing the server reply in the document sources shown in Table 1. We conclude that satisfying the security policy means at least one server's policy must be supported by every mechanism in the advertised list;
(3) determining the maximum buffer size is under-defined in RFC 4422 [13] and the Oracle implementation. For explicating that, we use the explanation that is provided by the DIGEST-MD5 SASL mechanism [12];
(4) using the security layer in RFC 2222 needs more explication, as it states that using this layer relates to the agreed bit-mask, which does not consider the confidentiality service. Therefore, we rely on the Oracle implementation, that uses the SSF instead of a bit-mask, to show when this layer is used. Also, we rely on the DIGEST-MD5 SASL mechanism [12], to show how the client and server negotiate this layer.

This paper shows how the ASM formalism is valuable in clarifying the ambiguity, especially with its ground model and the refinement notions. The ground model can first capture the informal specification in understandable way and at the desired level of details. Then, it can be evolved via stepwise refinement into a precise and enhanced mathematical specification.

As we construct a formal specification and provide links between it and informal or underdefined resources, we could prove properties of the development specification using the ASMETA framework in a similar way to [2].

We present an executable AsmetaL specification for private key encryption and decryption, in an abstract style.

In our ASM specification, the timing aspects for SASL are not considered, since neither of RFC 2222/4422 and Oracle implementation documents give specification for that.

Table 1. The source document for each ambiguity and its formal clarified specification

No.	The ambiguity	The document source for ambiguity	The clarified specification	The document source for clarification
1	The client selects the best mechanism	RFC 4422 [13]	Code 1 (b)	Oracle implementation document [15]
2	The server advertises the available mechanisms' list	RFC 4422 [13], and Oracle implementation document [15]	Code 2 (b)	DIGEST-MD5 SASL mechanism [12], Oracle implementation document [15], and its Java security guide [16]
3	Determining the maximum cipher text buffer size	RFC 4422 [13], and Oracle implementation document [15]	Code 3	DIGEST-MD5 SASL mechanism [12]
4	How and when the security layer is negotiated	RFC 2222 [14]	Ground model in Fig. 4	Oracle implementation document [15], DIGEST-MD5 SASL mechanism [12], and RFC 4422 [13]

5 Related Work

Our work elucidates ambiguities in the informal description for SASL, based on the ASM method. Therefore, we will now discuss other work related to either elucidating ambiguity or to the ASM method.

In [4], the ASM formalism is used to get a formal model of the Kerberos Authentication System which is based on the Needham and Schroeder authentication protocol. The formal model is used as a basis to locate the minimum assumptions to guarantee the correctness of the system and to analyse its security weaknesses.

In [7], the ASM ground models of a content adaptation system employed for the interactions between different client devices and the Cloud, is presented. This work is extended in [3], by refining the initial model into a more detailed one, through focusing on the interactions between the client and the middleware server to retrieve information relating to the client's device. Furthermore, the modelling process has been supported by validation and verification activities which are integrated within the ASMETA framework.

In [17], abstract encryption and decryption is specified using the language AsmL. This specification is based on the object-oriented features and constructs, and thus it diverts from the theoretical model of ASMs.

The researchers in [5] use Higher-order logic (HOL4) to develop a rigorous post-hoc specification for TCP, UDP, and the Sockets API, that reflects

the behavior of different implementations, include: FreeBSD 4.6, Linux 2.4.20-8, and Windows XP SP1. They validate their specification against several thousand traces captured from these implementations, to test whether they meet this specification. This paper is notable in the context of our work as its authors are motivated by increasing clarity and precision over ambiguous informal specifications of the RFC, that may result in inconsistent implementations. In this paper, we do not consider validating that the implementation meets the specification. We focus on clarifying ambiguities in the RFC description, and on elucidating uncertainty in the textual explanation of the implementation. Furthermore, our specification is expressed using the ASM method, because it is accessible, as it requires a minimum of notational coding, unlike HOL4, which requires extensively annotating the mathematical definitions side-by-side with informal specification [5].

6 Conclusion and Future Work

We have provided the ASM specifications that elucidate ambiguities in the SASL framework. We have focused on the ambiguous parts in RFC 2222/4422 and Oracle implementation documents, including mechanism selection, providing mechanisms' list, defining when and how the security layer can be used, and determining the maximum cipher buffer size.

We have showed how the comprehensible specification has been achieved based on two ASM notions: a ground model and stepwise refinement. The ground model enabled us to reflect the desired behavior, which is explained in RFCs, in an understandable way. While the stepwise refinement helped us to explicate the ambiguous part of the desired behavior in an accurate way, using other document sources to inform us.

We convert the informal specification into formal one by expressing it in the ASM formalism, which is mathematically well-defined, precise, and easily understood.

To further our research we are planning to consider the security of the SASL, to show whether the SASL specification is secure. We intend to use a suitable security analysis technique to elicit security requirements for the SASL and to verify them at the verification level.

Acknowledgments. The third author was partially supported by the EPSRC funded RAI Hub FAIR-SPACE (EP/R026092/1).

References

1. The CoreASM Project. http://www.coreasm.org/
2. Al-Shareefi, F., Lisitsa, A., Dixon, C.: Abstract state machines and system theoretic process analysis for safety-critical systems. In: Cavalheiro, S., Fiadeiro, J. (eds.) SBMF 2017. LNCS, vol. 10623, pp. 15–32. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70848-5_3

3. Arcaini, P., Holom, R.M., Riccobene, E.: ASM-based formal design of an adaptivity component for a Cloud system. Formal Aspects Comput. **28**(4), 567–595 (2016)
4. Bella, G., Riccobene, E.: Formal analysis of the Kerberos authentication system. J. Univers. Comput. Sci. **3**(12), 1337–1381 (1997)
5. Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M., Wansbrough, K.: Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations, pp. 55–66. ACM Press (2006)
6. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-642-18216-7
7. Chelemen, R.M.: Modeling a web application for cloud content adaptation with ASMs. In: International Conference on Cloud Computing and Big Data (CloudCom-Asia), pp. 44–55. IEEE (2013)
8. Froome, P., Monahan, B.: The role of mathematically formal methods in the development and assessment of safety-critical systems. Microprocess. Microsyst. **12**(10), 539–546 (1988)
9. Gargantini, A., Riccobene, E., Scandurra, P.: Model-driven language engineering: the ASMETA case study. In: The Third International Conference on Software Engineering Advances, ICSEA, pp. 373–378. IEEE (2008)
10. Gargantini, A.M., Riccobene, E., Scandurra, P.: A metamodel-based language and a simulation engine for abstract state machines. J. Univ. Comput. Sci. **14**(12), 1949–1983 (2008)
11. Gurevich, Y.: Evolving algebras 1993: Lipari guide. In: Specification and Validation Methods, pp. 9–36. Oxford University Press (1995)
12. Leach, P., Newman, C.: Using Digest Authentication as a SASL Mechanism. RFC 2831 (2000)
13. Melnikov, A., Zeilenga, K.: Simple Authentication and Security Layer (SASL). RFC 4422 (2006)
14. Myers, J.: Simple Authentication and Security Layer (SASL). RFC 2222 (1997)
15. Oracle: Writing applications that use SASL. In: Developer's Guide to Oracle Solaris® 11 Security, Chap. 7, pp. 126–148. Oracle (2012)
16. Oracle: Java SASL API Programming and Deployment Guide. In: Java Platform, Standard Edition Security Developers Guide, Chap. 10, pp. 21–28. Oracle (2016)
17. Rosenzweig, D., Runje, D., Slani, N.: Privacy, abstract encryption and protocols: an ASM model - part I. In: Börger, E., Gargantini, A., Riccobene, E. (eds.) ASM 2003. LNCS, vol. 2589, pp. 372–390. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36498-6_22
18. Siemborski, R., Gulbrandsen, A.: IMAP Extension for Simple Authentication and Security Layer (SASL) Initial Client Response. RFC 4959 (2007)
19. Siemborski, R., Melnikov, A.: SMTP Service Extension for Authentication Initial Client Response. RFC 4954 (2007)
20. Zeilenga, K.: The PLAIN Simple Authentication and Security Layer (SASL) Mechanism. RFC 4616 (2006)

Systematic Refinement of Abstract State Machines with Higher-Order Logic

Flavio Ferrarotti[1], Senén González[1(✉)], Klaus-Dieter Schewe[2], and José María Turull-Torres[3]

[1] Software Competence Center Hagenberg, Hagenberg, Austria
{flavio.ferrarotti,senen.gonzalez}@scch.at
[2] Christian-Doppler Laboratory for Client-Centric Cloud Computing, Linz, Austria
kdschewe@acm.org
[3] Universidad Nacional de La Matanza, Buenos Aires, Argentina
jmturull1952@gmail.com

Abstract. Graph algorithms that involve complex conditions on subgraphs can be specified much easier, if the specification allows expressions in higher-order logic to be used. In this paper an extension of Abstract State Machines by such expressions is introduced and its usefulness is demonstrated by examples of computations on graphs, such as graph factoring and checking self-similarity. In a naïve way these high-level specifications can be refined using submachines for the evaluation of the higher-order expressions. We show that refinements can be obtained in an automatic way for well-defined fragments of higher-order logic that collapse to second-order, by means of which the naïve refinement is only necessary for second-order logic expressions.

1 Introduction

There are many examples of graph computation problems that involve complex conditions such as graph colouring [3], topological subgraph discovery [14], recognition of hypercube graphs [13], and many others (see also [6,11]).

Such graph algorithms are difficult to specify in common rigorous methods such as Abstract State Machines (ASMs) [7], B [1], Event-B [2] or TLA$^+$ [19], because the algorithms require the definition of characterising conditions for particular subgraphs that lead to expressions beyond first-order logic, which are supported in TLA$^+$ as well as in provers such as Coq and Isabelle, but not in B, Event-B or ASMs, where the logical expressions supported by the methods are intrinsically first-order (for good reasons). Therefore, for the sake of easily comprehensible high-level specifications it is advisable to extend rigorous methods to support also higher-order logic (HOL) and to investigate strategies for refinement to first-order. In this paper we propose such an extension for ASMs,

The research reported in this paper was partially supported by the **Austrian Science Fund (FWF) [I2420-N31]** for the project: *Higher-Order Logics and Structures*.

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 204–218, 2018.
https://doi.org/10.1007/978-3-319-91271-4_14

which we present in Sect. 2. In Sect. 3 we further contribute a demonstration of the usefulness of such an extension by graph computation examples.

An analogy is the integration of weak monadic second-order logic and sophisticated tree background structures [5] into DB-ASMs defining XML machines [24]. Due to the behavioural theory of database transformations [23] this extension does not increase expressiveness. That is, the problems of XML computations can also be solved without the extension, so there must exist refinements to lower-level DB-ASM specifications that do not use second-order logic.

The same applies to the HOL-extended ASMs. A naïve refinement strategy would be to simply define ASMs that evaluate HOL sentences. However, based on results in descriptive complexity theory quite often HOL sentences collapse at least to second-order. Examples are the characterisation of hypercube graphs, which can be easily expressed in third-order logic, but as the problem is in the complexity class NP, an equivalent formula in existential second-order logic exists [9]. Another example is the formula-value query [4]. A general treatment of query computation, i.e. evaluation of sentences, was presented in [16], the capture of fixed-point queries in HOL was handled in [22].

Another contribution in this paper are conditions, under which HOL sentences collapse naturally to second-order following our recent research in [12]. We show that in these cases we can obtain an automatic refinement of the HOL-extended ASM to a SOL-extended ASM, to which the naïve refinement strategy consisting on non-deterministically guessing the quantified relation variables can be applied. We describe this refinement strategy in Sect. 4. Note that for well known cases of SOL sentences the evaluation can be done quite efficiently, but this is not within the scope of this paper. We conclude the paper with a brief summary and outlook in Sect. 5.

2 Abstract State Machines with Higher-Order Logic

We assume familiarity with the essential concepts of Abstract State Machines (ASMs) as defined in [7]. At its core an ASM consists of a signature Σ and a rule r. A *signature* is a finite set of function symbols f, each associated with an arity $ar(f)$. For a fixed *universe* $\mathcal{U}$ (aka as base set) we consider structures S, i.e. a function symbol $f \in \Sigma$ of arity n is interpreted by a function $f_S : \mathcal{U}^n \to \mathcal{U}$. A *state* is such a structure.

Terms over Σ are built in the usual way, i.e. variables x drawn from a set V of variables are terms, and for terms $t_1, \ldots, t_n$ and a function symbol $f \in \Sigma$ of arity n also $f(t_1, \ldots, t_n)$ is a term. For convenience we allow all values $v \in \mathcal{U}$ to be used as constants, i.e. function symbols of arity 0, with $v_S() = v$ for all states S. For simplicity we always write simply v instad of $v()$. Terms t are interpreted in a state S by a value $val_S(t)$ subject to a variable assignment $\sigma : V \to \mathcal{U}$ in the usual way, i.e. $val_S(x) = \sigma(x)$ and $val_S(f(t_1, \ldots, t_n)) = f_S(val_S(t_1), \ldots, val_S(t_n))$. A *location* ℓ is composed of a function symbol f and a tuple $(v_1, \ldots, v_n)$ of values in $\mathcal{U}$, where n is the arity of f. The evaluation of ℓ in state S is $val_S(\ell) = f_S(v_1, \ldots, v_n) \in \mathcal{U}$. ASM rules over Σ are composed in the usual way using

assignments. $f(t_1, \ldots, t_{ar_f}) := t_0$ (with terms t_i over Σ),
branching. if φ **then** r_+ **else** r_- **endif**,
parallel composition. forall x **with** $\varphi(x)$ **do** $r(x)$ **enddo**,
bounded parallel composition. $r_1 \ldots r_n$, and
choice. choose x **with** $\varphi(x)$ **do** $r(x)$ **enddo**.

with variables x and formulae φ and $\varphi(s)$, respectively, i.e. terms that evaluate to true or false. An *update* is a pair (ℓ, v) composed of a location ℓ and a value $v \in \mathcal{U}$. As defined in detail in [7] for each state S a rule r (without free variables) yields an update set $\Delta(S)$, i.e. a (finite) set of updates[1]. If this update set is consistent, i.e. there are no two updates $(\ell, v_1), (\ell, v_2) \in \Delta(S)$ with $v_1 \neq v_2$, then in consequence it determines the succesor state $S' = S + \Delta(S)$ with $val_{S'}(\ell) = v$ for $(\ell, v) \in \Delta(S)$ and $val_{S'}(\ell') = val_S(\ell')$ for all other locations ℓ'.

This standard definition of ASMs tacitly exploits a *background structure* [5], i.e. a set of domains that are either ground domains $\{D_i\}_{i \in I}$ or constructed from these by domain constructors, plus function symbols that denote operators on these domains. All values in these domains are used as constants and all function symbols in the background are used in the definition of terms with the difference that their interpretation is fixed and does not depend on the state. In this way the universe, the set of states, the set of terms and the ASM rules are extended by a fixed background structure, but updates still only affect locations with a function symbol in Σ.

The minimum requirement for ASMs that permit unbounded parallelism is that the background structure captures truth values, operations on truth values, constructors for records and multisets and the corresponding operators (for details see [10]). As in [24], where tree structures, a hedge algebra and weak monadic second-order logic was defined as part of the background structure, we define next an appropriate extension of this minimum background structure to capture HOL constructs in ASMs. For this we need to build finite relations over $\mathcal{U}$ in a hereditarily finite way, and extend the set of formulae to capture HOL.

The set $\mathcal{R}_n(\mathcal{U})$ of n-ary relations over $\mathcal{U}$ is defined as set of all finite subsets of $\mathcal{U}^n$. Then $\mathcal{R}(\mathcal{U}) = \bigcup_{n \in \mathbb{N}} \mathcal{R}_n(\mathcal{U})$ defines the set of relations of order 1, $\mathcal{R}^k(\mathcal{U}) = \mathcal{R}(\mathcal{R}^{k-1}(\mathcal{U}))$ defines the set of relations of order k, and $\bigcup_{k \in \mathbb{N}} \mathcal{R}^k(\mathcal{U})$ defines all higher-order relations over $\mathcal{U}$. A more general definition of higher-order relations can be found in [11,20] among others. Such a definition does not alter the expressive power of the higher-order logics [16] and would unnecessarily complicate our presentation.

Then assume that the set of variables is partitioned as $V = \{V_i \mid i \in \mathbb{N}\}$, where V_i is the set of variables of order i. So far, in the definitions above we only exploited V_0. Accordingly, terms of order i can be variables $x \in V_{i-1}$, or they can have the form $f(t_1, \ldots, t_n)$ or $X(t_1, \ldots, t_n)$ with terms t_j of order $i-1$ and either a function symbol f of arity n or a variable $X \in V_i$. The formulae of HOL_i are then defined in the standard way:

[1] In the case of non-determinsm using the choice rule we actually obtain a set of update sets.

(i) Every well-formed formula in HOL_{i-1} is a well-formed formula of HOL_i;
(ii) $X(t_1, \ldots, t_n)$ with terms t_j of order $i-1$ and a variable $X \in V_i$ is a well-formed formula of HOL_i;
(iii) If φ and ψ are well-formed formulae of HOL_i, then also $\neg\varphi$, $\varphi \wedge \psi$ and $\varphi \vee \psi$ are well-formed formula of HOL_i;
(iv) If φ is a well-formed formula of HOL_i and $X \in V_i$, then $\exists X(\varphi)$ and $\forall X(\varphi)$ are well-formed formula of HOL_i.

We omit the standard interpretation of HOL formulae. In a HOL-extended ASM the conditions φ and $\varphi(x)$, respectively, in branching, parallel composition and choice rules can be HOL formulae, and the variables used in such rules can be higher-order variables.

3 Specification of Graph Algorithms with HOL-Extended ASMs

In this section we show the usefulness of HOL-extended ASMs for the high level formal specification of graph algorithms. We focus first in writing an HOL-extended ASM that decides whether a graph is self-similar. Afterwards, we consider the intricate problem of graph factoring. In this second case, we provide a detailed third-order sentence which decides this problem over finite relational structures. Due to space restrictions, we leave as an easy exercise for the reader the construction of an equivalent HOL-extended ASM.

In the examples, we make use of some additional notation. Tuples of the form $\langle r_1, \ldots, r_s \rangle$, where $r_i \geq 0$ for $i = 1, \ldots, s$, denote types of third-order variables. Upper case calligraphic letters such as $\mathcal{V}$ denote the actual third-order variables. A third-order variable $\mathcal{V}^\tau$ of type $\tau = \langle r_1, \ldots, r_s \rangle$ is a third-order variable which range over sets of s-tuples, where each element in position $1 \leq i \leq s$ of a given s-tuple is either a relation of arity r_i if $r_i > 0$ or an atom if $r_i = 0$.

3.1 Self Similarity of Graphs

Graphs are a powerful tool to study properties of theoretical and real life complex networks. A prime example is that of self-similarity of complex networks [25] (aka scale invariance) which has practical applications in diverse areas such as the world-wide web [8], social networks [15] and biological networks [21].

Given a network represented as a finite graph G, it is relevant to determine whether G can be built starting from some graph pattern G_b by recursively replacing nodes in the pattern by new, "smaller scale", copies of G_b. If this holds, then we say that G is self-similar. Examples of self-similar graphs are shown in Fig. 1. Note that they are constructed using a triangle as graph pattern.

Generalizing our example, we say that a graph $G = (V, E)$ is *self-similar* w.r.t. a graph pattern $G_p = (V_p, E_p)$ of size k, if there is a sequence of graphs $G_0, G_1, \ldots, G_n$ such that G_0 is isomorphic to G_p, G_n is isomorphic to G and, for every pair (G_i, G_{i+1}) of consecutive graphs in the sequence, there is a partition $\{P_1, \ldots, P_k\}$ of the set of nodes of G_{i+1} which satisfies the following conditions:

1. For every $j = 1, \ldots, k$, the sub-graph induced by P_j in G_{i+1} is isomorphic to G_i.
2. There is a graph G_t isomorphic to G_p with set of nodes $V_t = \{a_1, \ldots, a_k\}$ for some $a_1 \in P_1, \ldots, a_k \in P_k$ and set of edges

$$E_t = \{(a_i, a_j) \mid \text{there is an edge } (x, y) \text{ of } G_{i+1} \text{ such that } P_i(x) \text{ and } P_j(y).\}.$$

3. For very $1 \leq i < j \leq k$, the closed neighborhoods $N_{G_{i+1}}[P_i]$ and $N_{G_{i+1}}[P_j]$ of P_i and P_j in G_{i+1}, respectively, are isomorphic.

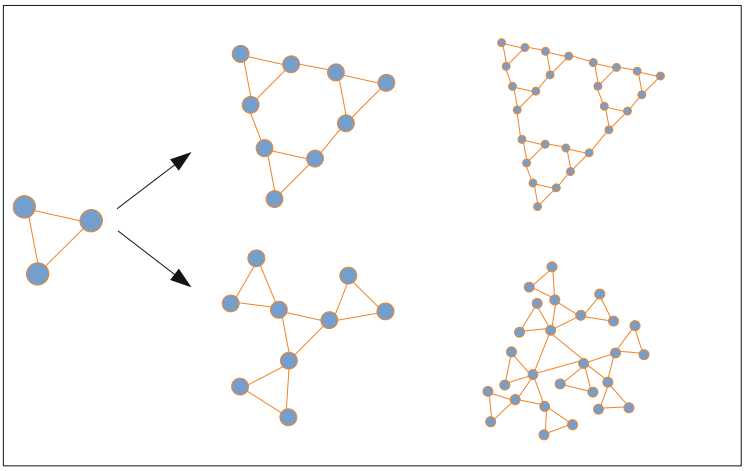


Fig. 1. Self-similar graphs.

We build a HOL-extended ASM M of signature Σ=$\{V, E, V_p, E_p, Accepts, k\}$ which, given a graph pattern of a fixed size, decides the self-similarity problem. V, V_b are unary and E, E_p binary static relation symbols. k is a constant symbol and *accept* a nullary function symbol. In every initial state S of M, we have that (V, E) and (V_p, E_p) are interpreted as the node and edge relations of the simple graphs G and G_p, respectively. Further, the set of nodes interpreting V_p has cardinality k. The main rule of M is as follows:

choose $\mathcal{V}$ $\mathcal{E}$ **with** $\alpha_{linear}(\mathcal{V}, \mathcal{E})$ **do**
  if $\alpha_{first}(\mathcal{V}, \mathcal{E}, V_p, E_p) \wedge \alpha_{last}(\mathcal{V}, \mathcal{E}, V, E) \wedge \varphi(\mathcal{V}, \mathcal{E})$ **then**
    $Accept := True$
  else
    $Accept := False$
  endif
enddo

where:

– $\mathcal{V}$ and $\mathcal{E}$ are third-order variables of types $\langle 1, 2\rangle$ and $\langle 1, 2, 1, 2\rangle$, respectively,

- $\alpha_{linear}(\mathcal{V},\mathcal{E})$ is a third-order logic formula which states that $\mathcal{V}$ is a set of simple graphs and that $\mathcal{E}$ is a third-order relation which defines a linear order of $\mathcal{V}$ (i.e., $(\mathcal{V},\mathcal{E})$ is third-order linear graph whose nodes are simple graphs),
- $\alpha_{first}(\mathcal{V},\mathcal{E},V_p,E_p)$ states that the first graph in $(\mathcal{V},\mathcal{E})$ is isomorphic to G_p,
- $\alpha_{last}(\mathcal{V},\mathcal{E},V,E)$ states that the last graph in $(\mathcal{V},\mathcal{E})$ is isomorphic to G, and
- $\varphi(\mathcal{V},\mathcal{E})$ states that for every pair (G_i,G_{i+1}) of consecutive graphs in $(\mathcal{V},\mathcal{E})$, there is a partition $\{P_1,\ldots,P_k\}$ of the set of nodes of G_{i+1} which satisfies conditions 1–3 above.

The formulae α_{linear}, α_{first}, α_{last} and φ are described below. To simplify the presentation, we frequently abuse the notation writing G_i instead of (V_i,E_i), where V_i is a unary relation variable and E_i is a binary relation variable. For instance, we write $\forall G_i(\mathcal{V}(G_i)\rightarrow\forall xy(E_i(x,y)\rightarrow V_i(x)\wedge V_i(y)))$ instead of the wff $\forall V_i\forall E_i(\mathcal{V}(V_i,E_i)\rightarrow\forall xy(E_i(x,y)\rightarrow V_i(x)\wedge V_i(y)))$. Sometimes we further abuse the notation writing for instance $\forall G_i(\mathcal{V}(G_i)\rightarrow E_i\subseteq V_i\times V_i)$ instead of the previous formula, $G=G'$ instead of $\forall x\forall y(V(x)\leftrightarrow V'(x)\wedge E(x,y)\leftrightarrow E'(x,y))$, and $G\cong G'$ instead of the formula stating that (V,E) and (V',E') are isomorphic graphs. We omit this last formula since it is already well-known (see for instance [13]).

- $\alpha_{linear}(\mathcal{V},\mathcal{E})\equiv\exists\mathcal{O}\Big(\mathcal{O}\subseteq\mathcal{V}\times\mathcal{V}\wedge\forall G\big(\neg\mathcal{O}(G,G)\big)\wedge$
  $\forall G_iG_j\Big(\mathcal{V}(G_i)\wedge\mathcal{V}(G_j)\rightarrow\big(\mathcal{O}(G_i,G_j)\vee\mathcal{O}(G_j,G_i)\vee G_i=G_j\big)\Big)\wedge$
  $\forall G_iG_j\Big(\neg\big(\mathcal{O}(G_i,G_j)\wedge\mathcal{O}(G_j,G_i)\big)\Big)\wedge$
  $\forall G_iG_jG_z\Big(\mathcal{O}(G_i,G_j)\wedge\mathcal{O}(G_j,G_z)\rightarrow\mathcal{O}(G_i,G_z)\Big)\wedge$
  $\forall G_iG_j\Big(\mathcal{E}(G_i,G_j)\leftrightarrow\big(\mathcal{O}(G_i,G_j)\wedge\forall G\big(\neg(\mathcal{O}(G_i,G)\wedge\mathcal{O}(G,G_j))\big)\big)\Big)\Big)$
- $\alpha_{first}(\mathcal{V},\mathcal{E},V_p,E_p)\equiv\exists G'\Big(\mathcal{V}(G')\wedge\forall G''\big(\mathcal{V}(G'')\rightarrow\neg\mathcal{E}(G'',G')\big)\wedge G'\cong G_p\Big)$
- $\alpha_{last}(\mathcal{V},\mathcal{E},V,E)\equiv\exists G'\Big(\mathcal{V}(G')\wedge\forall G''\big(\mathcal{V}(G'')\rightarrow\neg\mathcal{E}(G',G'')\big)\wedge G'\cong G\Big)$
- $\varphi(\mathcal{V},\mathcal{E})\equiv\forall G_iG_{i+1}\Big(\mathcal{V}(G_i)\wedge\mathcal{V}(G_{i+1})\wedge\mathcal{E}(G_i,G_{i+1})\rightarrow$
  $\exists P_1\ldots P_k\Big(V_{i+1}=\bigcup_{l=1}^{k}P_l\wedge\bigwedge_{1\leq l<m\leq k}P_l\cap P_m=\emptyset\wedge$
  $\psi_1(P_1,\ldots,P_k,G_i,G_{i+1})\wedge\psi_2(P_1,\ldots,P_k,G_{i+1})\wedge\psi_3(P_1,\ldots,P_k,G_{i+1})\Big)\Big)$
  where the formulae ψ_1, ψ_2 and ψ_3 express conditions 1–3 above, respectively.

- $\psi_1(P_1,\ldots,P_k,G_i,G_{i+1})\equiv\exists G_{P_1}\cdots\exists G_{P_k}\Big(\bigwedge_{l=1}^{k}P_l=V_{P_l}\wedge$
  $\forall xy\big(E_{P_l}(x,y)\leftrightarrow(E_{i+1}(x,y)\wedge V_{P_l}(x)\wedge V_{P_l}(y))\big)\wedge G_{P_l}\cong G_i\Big)$

- $\psi_2(P_1,\ldots,P_k,G_{i+1}) \equiv \exists G_t \exists x_1 \ldots x_k \Big(P_1(x_1) \wedge \cdots \wedge P_k(x_k) \wedge$
  $V_t = \{x_1,\ldots,x_k\} \wedge \forall xy \Big(E_t(x,y) \leftrightarrow \exists x'y' \big(E_{i+1}(x',y') \wedge \bigwedge_{l=1}^{k} P_l(x) \leftrightarrow P_l(x') \wedge$
  $\bigwedge_{l=1}^{k} P_l(y) \leftrightarrow P_l(y') \big) \Big) \wedge G_t \cong G_p \Big)$
- $\psi_3(P_1,\ldots,P_k,G_{i+1}) \equiv \forall G_a \forall G_b \Big(\Big(\big(\bigvee_{l=1}^{l} V_a = P_l \cup \{y | \exists x P_l(x) \wedge E_{i+1}(x,y)\} \big)$
  $\wedge \big(\bigvee_{l=1}^{l} V_b = P_l \cup \{y | \exists x P_l(x) \wedge E_{i+1}(x,y)\} \big) \wedge \forall xy \big(E_a(x,y) \leftrightarrow (E_{i+1}(x,y) \wedge V_a(x) \wedge$
  $V_a(y)) \big) \wedge \forall xy \big(E_b(x,y) \leftrightarrow (E_{i+1}(x,y) \wedge V_b(x) \wedge V_b(y)) \big) \Big) \rightarrow G_a \cong G_b \Big)$

Note that r is an accepting run of our HOL-extended machine M on an initial state S with input graph G of size n, only if r has length $\log_k n$.

We next refine M into a standard ASM M' which does not make use of higher-order expressions. The signature of M' extends the signature Σ of M with the nullary function *Mode* and the disjoint set of dynamic relation symbols $\Sigma_{aux} = \{P_1,\ldots,P_k,V_{P_1},\ldots,V_{P_k},E_{P_1},\ldots,E_{P_k},\ V_t,E_t,V_i,E_i,V_{i+1},E_{i+1}\}$. M' roughly works by unfolding the "third-order" choose of M into several steps. Instead of guessing a third-order linear digraph, M' starts with the pattern graph G_p and tries to reach G by non-deterministically guessing a path.

if *Mode* $= 0$ **then**
  Accept := *False* *Mode* := 1
  forall $x \in V_p$ **do** $V_i(x)$:= *True* **enddo**
  forall $(x,y) \in E_p$ **do** $E_i(x,y)$:= *True* **enddo endif**
if *Mode* $= 1$ **then** GuessRel *Mode* := 2 **endif**
if *Mode* $= 2 \wedge \psi_1' \wedge \psi_2' \wedge \psi_3'$ **then**
  if $\forall xy(V_{i+1}(x) \leftrightarrow V(x) \wedge E_{i+1}(x,y) \leftrightarrow E(x,y))$ **then** *Accept* := *True*
  else *Mode* := 1
    forall $x \in V_{i+1}$ **do** $V_i(x) := V_{i+1}(x)$ **enddo**
    forall $(x,y) \in E_{i+1}$ **do** $E_i(x,y) := E_{i+1}(x,y)$ **enddo endif endif**

where ψ_1' and ψ_2' are obtained by simply deleting the second-order quantification from ψ and ψ_2, respectively, ψ_3' is the formula equivalent to ψ_3 obtained by replacing the second-order universal quantification by a conjunction with k first-order expressions (which is possible since these quantifiers just range over the set of closed neighborhoods of $P_1,\ldots,P_k$ in G_{i+1}), and GuessRel non-deterministically assign a relation to each symbol in Σ_{aux} (in exactly the same way as GuessRelationS does for $S_1,\ldots,S_l$ in Definition 4.3).

3.2 Graph Factoring

Given two connected and loop-less undirected graphs, $G_1 = (V_1,E_1)$ of m nodes and $G_2 = (V_2,E_2)$ of n nodes, their (Cartesian) product is a graph $G_3 = (V_3,E_3)$, of $m \cdot n$ nodes. G_3 consists of m copies of G_2. Every sub-graph induced in G_3

by the set of m copies of a node of G_2 is isomorphic to G_1. As this operation is commutative, we can also define it as n copies of G_1.

An alternative definition that is easier to express in second-order logic is the following: V_3 is the set of nodes (u, v) s.t. $u \in V_1$, and $v \in V_2$, and E_3 is the set of edges $((u_1, v_1), (u_2, v_2))$ s.t. either $(u_1, u_2) \in E_1$ and $v_1 = v_2$, or $(v_1, v_2) \in E_2$ and $u_1 = u_2$.

We consider an input structure $\mathcal{A}$ of signature $\sigma_F = \langle V_I, E_I, \mathcal{F}_I \rangle$ consisting of a finite domain $D^{\mathcal{A}}$, a connected and loop-less undirected graph $(V_I^{\mathcal{A}}, E_I^{\mathcal{A}})$, and a third-order relation $\mathcal{F}_I^{\mathcal{A}}$ of type $\langle 1, 2, 1, 2 \rangle$, which in turn consists of a set of pairs of graphs $(V_{\mathcal{F}_I}^{\mathcal{A}}, E_{\mathcal{F}_I}^{\mathcal{A}})$, and $(V_{K\mathcal{F}_I}^{\mathcal{A}}, E_{K\mathcal{F}_I}^{\mathcal{A}})$. The first graph of each pair is a connected and loop-less undirected graph, and the second graph is a clique. We define *graph factoring* as a decision problem. A σ_F-structure $\mathcal{A}$ is in the class GraphFactoring iff the third-order relation $\mathcal{F}_I^{\mathcal{A}}$ is *a factoring*[2] of the graph $(V_I^{\mathcal{A}}, E_I^{\mathcal{A}})$, where the first graph of each pair in $\mathcal{F}_I^{\mathcal{A}}$ is a factor of the graph $(V_I^{\mathcal{A}}, E_I^{\mathcal{A}})$, and the size of the corresponding clique is the exponent.

Our formula roughly says that there is a third-order circuit $\mathcal{C} = (\mathcal{V}_\mathcal{C}, \mathcal{E}_\mathcal{C})$ whose gates compute the graph product of the two input graphs, whose roots are the factor graphs in the third-order relation $\mathcal{F}_I^{\mathcal{A}}$, where the fan-out of each factor graph in the circuit is the size of its corresponding clique $(V_{K\mathcal{F}_I}^{\mathcal{A}}, E_{K\mathcal{F}_I}^{\mathcal{A}})$ in the input structure $\mathcal{A}$, and whose output graph is the graph $(V_I^{\mathcal{A}}, E_I^{\mathcal{A}})$ in $\mathcal{A}$. A straightforward consequence of the definition of graph product is that the size of any factoring circuit $\mathcal{C}$ for a structure $\mathcal{A}$ is at most $2 \cdot \lceil \log(|V_I^{\mathcal{A}}|) \rceil$, and the size of the third-order relation ${\mathcal{F}_I}^{\mathcal{A}}$ on any given $\mathcal{A} \in$ GraphFactoring is at most $\lceil \log(|V_I^{\mathcal{A}}|) \rceil$. At the first abstraction level, the following pseudo third-order formula φ_F expresses graph factoring:

$$\exists \mathcal{V}_\mathcal{C} \mathcal{E}_\mathcal{C} \big(\text{FactoringCircuitForG}_\text{I}(\mathcal{V}_\mathcal{C}, \mathcal{E}_\mathcal{C}) \wedge \text{NodesConnLooplessUgraphs}(\mathcal{V}_\mathcal{C}, \mathcal{E}_\mathcal{C}) \\ \wedge \text{RootsPrimeGraphs}_\mathcal{C} \wedge \text{RootsIn}\mathcal{F}_{\text{I}\mathcal{C}} \wedge \text{SingleOutputG}_{\text{I}\mathcal{C}} \big) \qquad \text{where}$$

- $(\mathcal{V}_\mathcal{C}, \mathcal{E}_\mathcal{C})$, is a third-order graph, whose nodes are graphs, and whose edges are pairs of graphs, that is, $\mathcal{V}_\mathcal{C}$ is a third-order relation of type $\langle 1, 2 \rangle$, and $\mathcal{E}_\mathcal{C}$ is a third-order relation of type $\langle 1, 2, 1, 2 \rangle$, and
- NodesConnLooplessUgraphs says that each node in the third-order graph $(\mathcal{V}_\mathcal{C}, \mathcal{E}_\mathcal{C})$ is an undirected (i.e., symmetric) graph, which is connected and loop-less.

We present next the third-order pseudo-formulae for the main sub formulae in φ_F:

$$\text{FactoringCircuitForG}_\text{I}(\mathcal{V}_\mathcal{C}, \mathcal{E}_\mathcal{C}) \equiv \text{Digraph}(\mathcal{V}_\mathcal{C}, \mathcal{E}_\mathcal{C}) \wedge \text{Acyclic}(\mathcal{V}_\mathcal{C}, \mathcal{E}_\mathcal{C}) \wedge \\ \text{Connected}(\mathcal{V}_\mathcal{C}, \mathcal{E}_\mathcal{C}) \wedge \text{InDegree2}_\mathcal{C} \wedge \text{ProductOfParents}_\mathcal{C} \wedge \\ \text{LinearNonRoots}_\mathcal{C} \wedge \text{NonIsomorphicRoots}_\mathcal{C}$$

[2] Note that even when the factoring of a graph is unique up to isomorphism, the third-order relation $\mathcal{F}_I^{\mathcal{A}}$ provides one of the possible set of graphs which are factors of $(V_I^{\mathcal{A}}, E_I^{\mathcal{A}})$.

In turn:

$$\text{InDegree2}_{\mathcal{C}} \equiv \forall V_1 E_1 V_2 E_2 V_3 E_3 V_4 E_4 \Big([\mathcal{E}_{\mathcal{C}}(V_1, E_1, V_4, E_4) \wedge \mathcal{E}_{\mathcal{C}}(V_2, E_2, V_4, E_4) \wedge \mathcal{E}_{\mathcal{C}}(V_3, E_3, V_4, E_4)] \rightarrow [\text{EqualGraphs}(V_1, E_1, V_2, E_2) \vee \text{EqualGraphs}(V_1, E_1, V_3, E_3) \vee \text{EqualGraphs}(V_2, E_2, V_3, E_3)] \Big)$$

$$\text{ProductOfParents}_{\mathcal{C}} \equiv \forall V_3 E_3 \Big[\Big(\exists V_1 E_1 [\mathcal{E}_{\mathcal{C}}(V_1, E_1, V_3, E_3) \wedge \forall V_2 E_2 (\mathcal{E}_{\mathcal{C}}(V_2, E_2, V_3, E_3) \rightarrow \text{EqualGraphs}(V_1, E_1, V_2, E_2))] \rightarrow \text{Product}(V_1, E_1, V_1, E_1, V_3, E_3) \Big) \wedge \Big(\exists V_1 E_1 V_2 E_2 [\text{NotEqualGraphs}(V_1, E_1, V_2, E_2) \wedge \mathcal{E}_{\mathcal{C}}(V_1, E_1, V_3, E_3) \wedge \mathcal{E}_{\mathcal{C}}(V_2, E_2, V_3, E_3)] \rightarrow \text{Product}(V_1, E_1, V_2, E_2, V_3, E_3) \Big) \Big]$$

$$\text{Product}(V_1, E_1, V_2, E_2, V_3, E_3) \equiv \exists V_{\times} E_{\times} \Big(\big(\forall v_1 w_1 v_2 w_2 \big[(V_{\times}(v_1, w_1) \leftrightarrow (V_1(v_1) \wedge V_2(w_1))) \wedge (E_{\times}(v_1, w_1, v_2, w_2) \leftrightarrow [(v_1 = v_2 \wedge E_2(w_1, w_2)) \vee (w_1 = w_2 \wedge E_1(v_1, v_2))]) \big] \big) \wedge \text{Isomorphic}(V_{\times}, E_{\times}, V_3, E_3) \Big)$$

$$\text{LinearNonRoots}_{\mathcal{C}} \equiv \exists \mathcal{V}_{\mathcal{C}l} \mathcal{E}_{\mathcal{C}l} \Big(\text{EqualMonadicTO}\big(\mathcal{V}_{\mathcal{C}l}, \{\text{non root nodes in } \mathcal{C}\}\big) \wedge \text{EqualBinaryTO}\big(\mathcal{E}_{\mathcal{C}l}, \mathcal{E}_{\mathcal{C}} \restriction \{\text{non root nodes in } \mathcal{C}\}\big) \wedge \text{LinearDigraph}(\mathcal{V}_{\mathcal{C}l}, \mathcal{E}_{\mathcal{C}l}) \Big)$$

where $\mathcal{E}_{\mathcal{C}} \restriction \{\text{non root nodes in } \mathcal{C}\}$ is the restriction of the third-order binary relation $\mathcal{E}_{\mathcal{C}}$ to the subset $\{\text{non root nodes in } \mathcal{C}\}$ of the set $\mathcal{V}_{\mathcal{C}}$[3].

$$\text{NonIsomorphicRoots}_{\mathcal{C}} \equiv \forall V_1 E_1 V_2 E_2 \Big([\text{Root}_{\mathcal{C}}(V_1, E_1) \wedge \text{Root}_{\mathcal{C}}(V_2, E_2)] \rightarrow [\text{EqualGraphs}(V_1, E_1, V_2, E_2) \vee \neg \text{Isomorphic}(V_1, E_1, V_2, E_2)] \Big)$$

$$\text{RootsPrimeGraphs}_{\mathcal{C}} \equiv \forall V_1 E_1 \Big([\mathcal{V}_{\mathcal{C}}(V_1, E_1) \wedge \neg \exists V_2 E_2 (\mathcal{E}_{\mathcal{C}}(V_2, E_2, V_1, E_1))] \rightarrow \text{PrimeGraph}(V_1, E_1) \Big)$$

where $\text{PrimeGraph}(V_1, E_1) \equiv \neg \exists V_2 E_2 V_3 E_3 \Big(\text{Product}(V_2, E_2, V_3, E_3, V_1, E_1) \Big)$.

$$\text{SingleOutputG}_{\text{IC}} \equiv \mathcal{V}_{\mathcal{C}}(V_I, E_I) \wedge \neg \exists V_1 E_1 (\mathcal{E}_{\mathcal{C}}(V_I, E_I, V_1, E_1)) \wedge \forall V_2 E_2 \Big([\mathcal{V}_{\mathcal{C}}(V_2, E_2) \wedge \neg \exists V_3 E_3 (\mathcal{E}_{\mathcal{C}}(V_2, E_2, V_3, E_3))] \rightarrow \text{EqualGraphs}(V_2, E_2, V_I, E_I) \Big)$$

[3] To make this subformula more understandable we chose to use a standard algebraic notation mixed with the syntax of third-order.

$$\text{RootsIn}\mathcal{F}_{1\mathcal{C}} \equiv \forall V_0 E_0 \Big(\text{Root}_{\mathcal{C}}(V_0, E_0) \leftrightarrow \exists V_{K0} E_{K0}(\mathcal{F}_I(V_0, E_0, V_{K0}, E_{K0}))\Big) \wedge$$
$$\forall V_0 E_0 V_{K0} E_{K0} \Big(\mathcal{F}_I(V_0, E_0, V_{K0}, E_{K0}) \rightarrow [\text{Clique}(V_{K0}, E_{K0}) \wedge$$
$$\text{NumbOfProducts}_{\mathcal{C}}(V_0, E_0, V_{K0})]\Big)$$
$$\text{NumbOfProducts}_{\mathcal{C}}(V_0, E_0, V_{K0}) \equiv \exists \mathcal{H} \forall x_1 V_1 E_1 V_2 E_2 V_3 E_3 \Big(\Big[V_{K0}(x_1) \rightarrow$$
$$\exists V_4 E_4(\mathcal{H}(x_1, V_4, E_4))\Big] \wedge \Big[(\mathcal{H}(x_1, V_1, E_1) \wedge \mathcal{H}(x_1, V_2, E_2))$$
$$\rightarrow \text{EqualGraphs}(V_1, E_1, V_2, E_2)\Big] \wedge \Big[\mathcal{E}_{\mathcal{C}}(V_0, E_0, V_3, E_3) \rightarrow \exists x_1(\mathcal{H}(x_1, V_3, E_3))\Big] \wedge$$
$$\Big[\mathcal{E}_{\mathcal{C}}(V_0, E_0, V_3, E_3) \rightarrow \Big([(\text{InDegree1}_{\mathcal{C}}(V_3, E_3) \wedge \exists x_4 x_5 \big(V_{K0}(x_4) \wedge V_{K0}(x_5)$$
$$\wedge x_4 \neq x_5 \wedge \mathcal{H}(x_4, V_3, E_3) \wedge \mathcal{H}(x_5, V_3, E_3) \wedge \forall x_6[(V_{K0}(x_6) \wedge \mathcal{H}(x_6, V_3, E_3))$$
$$\rightarrow (x_6 = x_4 \vee x_6 = x_5)])] \vee \big[\text{InDegree2}_{\mathcal{C}}(V_3, E_3) \wedge \exists x_4 \big(V_{K0}(x_4) \wedge$$
$$\mathcal{H}(x_4, V_3, E_3) \wedge \forall x_5[(V_{K0}(x_5) \wedge \mathcal{H}(x_5, V_3, E_3)) \rightarrow (x_5 = x_4)]\big)\big]\Big)\Big)$$

The "quasi" injectivity of the function $\mathcal{H} : V_{K0} \mapsto \text{Children}_{\mathcal{C}}(V_0, E_0)$ in the formula above (expressed in the last five lines of the formula) is due to the fact that we avoid allowing multiple edges between two given nodes in the circuit $\mathcal{C}$, to make the formula simpler. Note that the only possible case where one single edge means that a (factor) graph is actually being used twice in the same product is at the (unique) node at depth one in the circuit. An example for this situation is the factoring circuit for an hypercube of order n, where the same factor graph (K_2) is used n times.

As to the sub formulae for Acyclic, Connected and LinearDigraph, they are rather standard and can be found among other sources in [13]. The other sub formulae not included in the present article are trivial.

4 Fragments of HOL that Collapse to Second-Order

We define a *general schema* of existential third-order logic (∃TO) formulae that describes a sequence of structures representing a computation by explicitly stating, which operations can be involved in the construction of a given structure in the sequence, when applied to the previous one. We then show that formulae under this general schema collapse to SOL plus transitive closure, SO(TC), and can be systematically refined into standard ASMs which do not use HOL formulae. If the length of the sequence of structures is further bounded by a polynomial in the size of the input, then this class of formulae collapses to plain SOL as per the results in [12].

Definition 4.1. Let Σ be a relational vocabulary, which may include constant symbols. We define $\mathfrak{T}[\Sigma]$ as the class of ∃TO formulae of the form:

$$\exists \mathcal{C} \mathcal{O}\big(\text{TotalOrder}(\mathcal{C}, \mathcal{O}) \wedge \forall \bar{X}\big(\text{First}(\bar{X}) \rightarrow \alpha_{\text{First}}(\bar{X}) \wedge \text{Last}(\bar{X}) \rightarrow \alpha_{\text{Last}}(\bar{X})\big) \wedge$$
$$\forall \bar{X} \bar{Y}\big(\mathcal{C}(\bar{X}) \wedge \mathcal{C}(\bar{Y}) \wedge \text{Succ}(\bar{X}, \bar{Y}) \rightarrow \varphi(\bar{X}, \bar{Y})\big)\big),$$

where

- $\mathcal{C}$ is a third-order variable of type $\bar{s} = (s_1, \ldots, s_l, s_{l+1}, \ldots, s_{l+k})$ with $l > 0$, $k \geq 0$, $s_i > 0$ for $i = 1, \ldots, l$ and $s_i = 0$ for $i = l+1, \ldots, l+k$,
- $\mathcal{O}$ is a third-order variable of type $\bar{s}\bar{s}$,
- $\bar{X} = (X_1, \ldots, X_l, x_1, \ldots, x_k)$ with X_i of ariy s_i for $i = 1, \ldots, l$,
- $\bar{Y} = (Y_1, \ldots, Y_l, y_1, \ldots, y_k)$ with Y_i of ariy s_i for $i = 1, \ldots, l$,
- TotalOrder$(\mathcal{C}, \mathcal{O})$, First$(\bar{X})$, Last$(\bar{X})$ and Succ$(\bar{X}, \bar{Y})$ denote fixed second-order formulae with third-order free variables which express that $\mathcal{O}$ is a total order over $\mathcal{C}$, that $\bar{X}$ is the first tuple (relational structure) in $\mathcal{O}$, that $\bar{X}$ is the last tuple in $\mathcal{O}$, and that $\bar{Y}$ is the immediate successor of $\bar{X}$ in $\mathcal{O}$, respectively,
- $\alpha_{\text{First}}(\bar{X})$ and $\alpha_{\text{Last}}(\bar{X})$ are arbitrary second-order formulae of vocabulary $\Sigma \cup \{\bar{X}\}$ which express the properties that the first and last tuples (relational structures) in the order $\mathcal{O}$ should satisfy,
- $\varphi(\bar{X}, \bar{Y})$ is an arbitrary second-order formula of vocabulary $\Sigma \cup \{\bar{X}\} \cup \{\bar{Y}\}$ which expresses the transition from $\bar{X}$ to $\bar{Y}$, i.e., the operations that can be used to obtain $\bar{Y}$ from $\bar{X}$.

It is easy to see that using this schema we can express many interesting queries in a clear and natural way, e.g. the examples in Sect. 3 and in [12,13].

The class of Boolean queries expressible by ∃TO formulae in $\mathfrak{T}$ corresponds exactly to that definable in SO(TC). The following definition of SO(TC) is from [18].

Definition 4.2. Let $\varphi(\bar{X}, \bar{x}, \bar{Y}, \bar{y})$ be a formula of second-order logic of some vocabulary Σ with free second-order (relation) variables $\bar{X} = (X_1, \ldots, X_l)$, $\bar{Y} = (Y_1, \ldots, Y_l)$ and free first-order variables $\bar{x} = (x_1, \ldots, x_k)$, $\bar{y} = (y_1, \ldots, y_k)$. Let $\text{arity}(X_i) = \text{arity}(Y_i) = r_i$ for $1 \leq i \leq l$. Given a structure $\mathbf{A}$ of vocabulary Σ, interpret φ as the third-order relation

$$\varphi^{\mathbf{A}} = \{(\bar{R}, \bar{a}, \bar{S}, \bar{b}) \mid R_i, S_i \subseteq A^{r_i};\ \bar{a}, \bar{b} \in A^k \text{ and } \mathbf{A} \models \varphi(\bar{R}, \bar{a}, \bar{S}, \bar{b})\}.$$

We write $[\text{TC}_{\bar{X},\bar{x},\bar{Y},\bar{y}}\varphi]$ to denote the reflexive, transitive closure of the binary third-order relation defined by $\varphi(\bar{X}, \bar{x}, \bar{Y}, \bar{y})$. We define SO(TC) as the closure of second-order logic with arbitrary occurrences of TC.

Theorem 4.1. *The class of Boolean queries definable in* $\mathfrak{T}$ *coincides with the class of Boolean queries definable in* SO(TC).

Proof (Sketch). We first show that SO(TC) $\subseteq \mathfrak{T}$. Let $\psi \in$ SO(TC). We assume that ψ is in the following normal form:

$$\psi \equiv \exists \bar{R}\bar{S}\bar{v}\bar{w}\big([\text{TC}_{X_1,\ldots,X_l,x_1,\ldots,x_k,Y_1,\ldots,Y_l,y_1,\ldots,y_k}\beta](\bar{R}, \bar{v}, \bar{S}, \bar{w}) \wedge \forall \bar{z}(\neg R_1(\bar{z}) \wedge \cdots \wedge \neg R_l(\bar{z}) \wedge S_1(\bar{z}) \wedge \cdots \wedge S_l(\bar{z}))\big) \quad (1)$$

where β is a quantifier-free SO formula, and the second-order variables X_i and Y_i all have the same arity r. Every SO(TC) formula can be written in this normal form (see [18, Exercise 10.23, p. 166]). We can translate ψ into an equivalent

formula in $\mathfrak{T}$ of the form described in Definition 4.1 by defining the sub-formulae $\alpha_{First} \equiv \forall \bar{z}(\neg X_1(\bar{z}) \wedge \cdots \wedge \neg X_l(\bar{z}))$, $\alpha_{Last} \equiv \forall \bar{z}(X_1(\bar{z}) \wedge \cdots \wedge X_l(\bar{z}))$ and $\varphi \equiv \beta$.

Finally, we show that $\mathfrak{T} \subseteq \mathrm{SO(TC)}$. Each $\varphi \in \mathfrak{T}$ has the form of the formula in Definition 4.1 and it is clearly equivalent to the following SO(TC)-formula: $\exists \bar{X}\bar{Y}\bar{x}\bar{y}(\alpha_{First}(\bar{X},\bar{x}) \wedge \alpha_{Last}(\bar{Y},\bar{y}) \wedge \mathrm{TC}_{\bar{R}\bar{v}\bar{S}\bar{w}}\varphi'](\bar{X},\bar{x},\bar{Y},\bar{y}))$, where φ' is obtained by substituting in φ the variables $\bar{X},\bar{x},\bar{Y},\bar{y}$ by $\bar{R},\bar{v},\bar{S},\bar{w}$, respectively. □

As PSPACE = SO(TC) (see [17,18]), $\mathfrak{T}$ also provides a descriptive characterization of PSPACE. Its expressive power equals that of first-order logic extended with a partial fixed-point operator and a linear order (see [18,26]).

Corollary 4.1. $\mathrm{PSPACE} = \mathrm{FO(PFP}, \leq) = \mathfrak{T}$.

Note that any partial fixed point formula of the form $[\mathrm{PFP}_{X,\bar{x}}\varphi(X,\bar{x})](\bar{t})$ can be rewritten as an equivalent $\mathfrak{T}$-formula where $\alpha_{First} \equiv \forall \bar{z}(\neg X(\bar{z}))$, i.e. initially X is empty, $\varphi \equiv \forall \bar{z}(Y(\bar{z}) \leftrightarrow \varphi(X,\bar{z}))$, where Y is obtained by applying the operator defined by φ to X, and $\alpha_{Last} \equiv \forall \bar{z}(X(\bar{z}) \leftrightarrow \varphi(X,\bar{z})) \wedge X(\bar{t})$, i.e. fixed point have been reached and $\bar{t}$ belongs to it.

Furthermore, in Definition 4.3 we describe how any formula in $\mathfrak{T}$ can be systematically refined into an equivalent standard ASM that does not use higher-order expressions.

Definition 4.3. Let φ be a SO(TC) sentence in the normal form described in (1). The *corresponding ASM* M_φ is an ASM with signature $\Sigma' = \Sigma \cup \{\bar{R}\} \cup \{\bar{S}\} \cup \{\bar{v}\} \cup \{\bar{w}\} \cup \{\bar{w}'\} \cup \{D, Mode, Accept\}$, where $D, Mode, Accept \notin \Sigma$ and $Mode = 0$ in every initial state of M_φ. Let the following be the main rule of M_φ:

if $Mode = 0$ **then** $Accept := False$ $\quad Mode := 1$
  forall $\bar{x} \in D^r$ **do** $R_1(\bar{x}) := False$ **enddo**
  ⋮
  forall $\bar{x} \in D^r$ **do** $R_l(\bar{x}) := False$ **enddo**
  choose $\bar{x} \in D^k$ **do** $\bar{v} := \bar{x}$ **enddo**
  choose $\bar{x} \in D^k$ **do** $\bar{w} := \bar{x}$ **enddo endif**
if $Mode = 1$ **then** GuessRelationsS $\quad Mode := 2$
  choose $\bar{x} \in D^k$ **do** $\bar{w}' := \bar{x}$ **enddo endif**
if $Mode = 2$ **then**
  if $\beta(\bar{X}/\bar{R}, \bar{x}/\bar{v}, \bar{Y}/\bar{S}, \bar{y}/\bar{w}')$ **then**
    if $\forall \bar{x}(S_1(\bar{x}) \wedge \cdots \wedge S_l(\bar{x})) \wedge \bar{w}' = \bar{w}$ **then** $Accept := True$
    else $Mode := 1$ $\quad \bar{v} := \bar{w}'$
      forall $\bar{x} \in D^r$ **do** $R_1(\bar{x}) := S_1(\bar{x})$ **enddo**
      ⋮
      forall $\bar{x} \in D^r$ **do** $R_l(\bar{x}) := S_l(\bar{x})$ **enddo endif endif endif**

GuessRelationsS =
  forall $\bar{x} \in D^r$ **do**
    choose y **with** $y = True \vee y = False$ **do** $S_1(\bar{x}) := y$ **enddo**
    ⋮
    choose y **with** $y = True \vee y = False$ **do** $S_l(\bar{x}) := y$ **enddo enddo**

Since we know from the proof of Theorem 4.1 that every $\mathfrak{T}$ sentence can be translated to an equivalent SO(TC) sentence and that every SO(TC) sentence can in turn be translated to the normal form described in (1), we get the following result.

Proposition 4.1. *Let φ be a third-order $\mathfrak{T}$ sentence (recall Definition 4.1). Then there is an* equivalent *ASM machine M_φ which has the form of the ASM in Definition 4.3 and can be built systematically starting from φ.*

Finally, we recall a result from [12] by which we can do better when the higher-order variables range over higher-order relations that are polynomially bounded in size, as in the case of the two examples in Sect. 3 and those mentioned in Sect. 1 (see also further examples in [12]). Let $\mathrm{HO}^{i,P}$ be the restriction of the HOL of order i to quantifiers that can only range over higher-order relations of total size bounded by a polynomial on the size of the input, then we have the following:

Theorem 4.2 ([12]). *For every order $i \geq 3$, every $\mathrm{HO}^{i,P}$ formula α can be algorithmically translated into an equivalent second-order formula α'.*

5 Conclusion

We presented an extension of ASMs by HOL and demonstrated its usefulness by a glimpse on graph algorithms that exploit complex conditions for subgraphs such as self-similarity and graph factoring. The extension permits easier high-level specifications of such algorithms. We then characterised fragments of HOL that can be transformed to second-order by automatic ASM refinement. Thus, the refinement strategy for HOL-extended ASMs is to first exploit such an automatic refinement followed by the definition of ASMs for the evaluation of second-order sentences.

A similar extension targeting computations over tree structures in connection with XML was handled previously. We believe that it will be helpful for the specification of comprehensible ground models, if more domain-specific extensions of ASM were investigated and tailored refinements for these extensions could be discovered. An open problem in this context are proofs, where it would be preferable not to have to deal with HOL. As naïve refinements in a standard way are always possible, we believe that proofs could be conducted on such automatic, though not optimal refinements. This has to be studied more carefully in follow-on research.

References

1. Abrial, J.R.: The B-Book - Assigning Programs to Meanings. Cambridge University Press, Cambridge (2005)
2. Abrial, J.R.: Modeling in Event-B - System and Software Engineering. Cambridge University Press, Cambridge (2010)

3. Abu-Khzam, F.N., Langston, M.A.: Graph coloring and the immersion order. In: Warnow, T., Zhu, B. (eds.) COCOON 2003. LNCS, vol. 2697, pp. 394–403. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-45071-8_40
4. Beaudry, M., McKenzie, P.: Circuits, matrices, and nonassociative computation. In: Proceedings of the Seventh Annual Structure in Complexity Theory Conference, pp. 94–106 (1992)
5. Blass, A., Gurevich, Y.: Background of computation. Bull. EATCS **92**, 82–114 (2007)
6. Bollobás, B.: Modern Graph Theory. Graduate Texts in Mathematics, vol. 184. Springer, Heidelberg (2002). https://doi.org/10.1007/978-1-4612-0619-4
7. Börger, E., Stärk, R.F.: Abstract State Machines. A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-642-18216-7
8. Dill, S., Kumar, R., Mccurley, K.S., Rajagopalan, S., Sivakumar, D., Tomkins, A.: Self-similarity in the web. ACM Trans. Internet Technol. **2**(3), 205–223 (2002)
9. Fagin, R.: Generalized first-order spectra and polynomial-time recognizable sets. In: Karp, R. (ed.) Complexity of Computations. SIAM-AMS Proceedings, vol. 7, pp. 27–41. American Mathematical Society (1974)
10. Ferrarotti, F., Schewe, K.D., Tec, L., Wang, Q.: A new thesis concerning synchronised parallel computing - simplified parallel ASM thesis. Theor. Comput. Sci. **649**, 25–53 (2016)
11. Ferrarotti, F.: Expressibility of higher-order logics on relational databases: proper hierarchies. Ph.D. thesis, Massey University, Wellington, New Zealand (2008). http://hdl.handle.net/10179/799
12. Ferrarotti, F., González, S., Turull-Torres, J.M.: On fragments of higher order logics that on finite structures collapse to second order. In: Kennedy, J., de Queiroz, R.J.G.B. (eds.) WoLLIC 2017. LNCS, vol. 10388, pp. 125–139. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-55386-2_9
13. Ferrarotti, F., Ren, W., Turull Torres, J.M.: Expressing properties in second- and third-order logic: hypercube graphs and SATQBF. Logic J. IGPL **22**(2), 355–386 (2014)
14. Grohe, M., Kawarabayashi, K., Marx, D., Wollan, P.: Finding topological subgraphs is fixed-parameter tractable. In: Proceedings of the 43rd Annual ACM Symposium on Theory of Computing (STOC 2011), pp. 479–488. ACM (2011)
15. Guimerà, R., Danon, L., Díaz-Guilera, A., Giralt, F., Arenas, A.: Self-similar community structure in a network of human interactions. Phys. Rev. E **68**, 065103 (2003)
16. Hella, L., Turull Torres, J.M.: Computing queries with higher-order logics. Theor. Comput. Sci. **355**(2), 197–214 (2006)
17. Immerman, N.: Languages which capture complexity classes (preliminary report). In: Johnson, D.S., et al. (eds.) Proceedings of the 15th Annual ACM Symposium on Theory of Computing (STOC 1983), pp. 347–354. ACM (1983)
18. Immerman, N.: Descriptive Complexity. Graduate texts in computer science. Springer, Heidelberg (1999). https://doi.org/10.1007/978-1-4612-0539-5
19. Lamport, L.: Specifying Systems, The TLA$^+$ Language and Tools for Hardware and Software Engineers. Addison-Wesley, Boston (2002)
20. Leivant, D.: Higher order logic. In: Gabbay, D.M., Hogger, C.J., Robinson, J.A., Siekmann, J.H. (eds.) Handbook of Logic in Artificial Intelligence and Logic Programming. Deduction Methodologies, vol. 2, pp. 229–322. Oxford University Press (1994)

21. Réka, A.: Scale-free networks in cell biology. J. Cell Sci. **118**(21), 4947–4957 (2005)
22. Schewe, K.D., Torres, J.M.T.: Fixed-point quantifiers in higher order logics. In: Kiyoki, Y., et al. (eds.) Information Modelling and Knowledge Bases XVII. Frontiers in Artificial Intelligence and Applications, vol. 136, pp. 237–244. IOS Press (2006)
23. Schewe, K.D., Wang, Q.: A customised ASM thesis for database transformations. Acta Cybernetica **19**(4), 765–805 (2010)
24. Schewe, K.D., Wang, Q.: XML database transformations. J. Univers. Comput. Sci. **16**(20), 3043–3072 (2010)
25. Song, C., Havlin, S., Makse, H.A.: Self-similarity of complex networks. Nature **433**, 392–395 (2005)
26. Vardi, M.Y.: The complexity of relational query languages (extended abstract). In: Lewis, H.R., et al. (eds.) Proceedings of the 14th Annual ACM Symposium on Theory of Computing (STOC 2014), pp. 137–146. ACM (1982)

Refinement of Timing Constraints for Concurrent Tasks with Scheduling

Chenyang Zhu(✉), Michael Butler, and Corina Cirstea

School of Electronics and Computer Science, University of Southampton, Southampton, UK
{cz4g16,mjb,cc2}@ecs.soton.ac.uk

Abstract. Event-B is a refinement-based formal method that is used for system-level modeling and analysis of concurrent and distributed systems. Work has been done to extend Event-B with discrete time constraints. However the previous work does not capture the communication and competition between concurrent processes. In this paper, we distinguish task-based timing properties with scheduler-based timing properties from the perspective of different system design phases. To refine task-based timing properties with scheduler-based timing properties based on existing trigger-response patterns, we introduce a non-deterministic queue based scheduling framework to schedule processes under concurrent circumstances, which addresses the problems of refining deadline constraint under concurrent situations. Additional gluing invariants are provided to this refinement. To demonstrate the usability of the framework, we provide approaches to refine this framework with FIFO scheduling policy as well as deferrable priority based scheduling policy with aging technique. We demonstrate our framework and refinement with a timed mutual exclusion case study. The model is proved using the Rodin tool.

Keywords: Event-B · Refinement · Timing · Concurrency Scheduling

1 Introduction

Cyber Physical Systems (CPS) have received much attention in recent years due to their capabilities with advanced processors, sensors and wireless communication. Timing and concurrency are two key features of CPS [4]. The physical world evolves with time and this needs to be taken into account of within the computing devices in CPS. Real-time constraints need to be introduced to computing devices to ensure that the devices are interacting with the physical world correctly. What's more, with the advanced processors of CPS, multiple threads of computation are executing concurrently to achieve the goal of computation as a whole. For this reason understanding models and design principles for timing and concurrency is critical for CPS. In addition, it is difficult to model a

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 219–233, 2018.
https://doi.org/10.1007/978-3-319-91271-4_15

complicated CPS with all the detailed features in one step. An abstraction and refinement approach can be adapted to manage complexity by modeling the system from abstract level to more concrete levels with reasoning to verify the consistency between refinement levels [7].

Event-B is a formal method for system-level modeling and analysis that is based on predicate logic and set theory [2]. Apart from its ability to model systems with precise mathematical abstractions, it also provides notions of abstraction and refinement. However, an explicit notion of real-time is not supported in Event-B, while real-time performance is critical for CPS. Existing work that extends Event-B models with timing properties uses a trigger-response pattern to model discrete time [16]. The pattern sets timestamps for trigger and response events and uses a tick event to prevent the global clock proceeding to a point where time constraints between trigger and response events would be violated. This pattern, however, does not distinguish timing properties for different system design phases. We define task-based timing properties as high level timing properties from system requirement specification phase, which place discrete time constraint on individual processes or tasks. These task-based timing properties can not describe the concurrent behaviour of tasks precisely. In real time systems, there are always several tasks running concurrently. High level time constraints for each task cannot guarantee the timing behaviour of the whole system. To model the behaviour of these concurrent processes, we define scheduler-based timing properties as concrete timing properties for the system design phase, which place discrete time constrains on the scheduler which schedules the concurrent tasks. Fairness can be imposed to restrict the nondeterministic behaviour of concurrent tasks. In many real-time applications, the weak guarantee of eventual occurrence of some event with weak/strong fairness assumption may be insufficient [11]. Alur and Henzinger proposed the definition of finitary fairness to impose a bound on the relative frequency in scheduling a set of events [3]. This definition of weak fairness requires that there is an unknown bound k for every computation of a system such that no enabled transition is postponed more than k consecutive steps.

We propose a nondeterministic queue based scheduling framework based on the idea of finitary fairness. Processes are placed in a nondeterministic position in the queue and once a process enters the queue, it cannot be postponed for more than k consecutive times. Additional gluing invariants are provided to use the framework to refine task-based deadline constraints with scheduler-based deadline constraints. Our approach is demonstrated by a timed mutual exclusion case study. Two alternative refinements from the nondeterministic queue to a FIFO scheduling policy as well as a deferrable priority based scheduling policy with aging technique are used to demonstrate the usability of the framework. In addition, we only need to prove the timing is satisfied by the nondeterministic queue since any refinement of the nondeterministic queue will also satisfy the same timing deadlines.

Section 2 introduces some related work on modeling discrete time in Event-B with trigger-response patterns. We also discuss some additional fairness assumptions on the tick event introduced in the trigger-response pattern. In Sect. 3 we introduce task-based deadlines used in the requirement specification phase. We use a timed mutual exclusion case study to illustrate the usage of task-based timing property. Section 4 refines the task-based deadlines to scheduler-based deadlines with a nondeterministic queue based scheduling policy as well as some additional gluing invariants. Section 5 gives two different implementations of the nondeterministic queue based scheduling policy. Section 6 gives the conclusion and some future work.

2 Related Work

2.1 Event-B

Event-B [2] is a formal modeling method based on set theory and first-order logic, which is usually used for system-level modeling and analysis with abstraction, refinement and reasoning on the model. Formal modeling is used to address the problem of lack of precision of specifications. However, formality on its own does not handle the problem of complex requirements and specifications [7]. Refinement helps to simplify the process of modeling with a stepwise approach. Gluing invariants which refer to variables of abstract and concrete machines are used to relate the states of concrete and abstract machines during refinement steps [14].

2.2 Time Modeling

Timing issues are critical in cyber physical systems. Timing analysis should be carried out together with the development of the system to improve the real-time performance as well as guarantee the safety of the whole system. Timed automata [5] that are supported by the UPPAAL [15] model checker have been used in industrial modeling of real time systems. It is challenging to model a complex system with the timed automata formalism and UPPAAL as it does not support refinement of the model. Some approaches such as counter example guided abstraction refinement have been brought up to add abstraction and refinement when modeling a complex system [12]. This approach uses a model checker to get the counter examples from the abstract model and uses these counter examples as guides to refine the system. However it is difficult to find the missing part from the model just from counter examples.

Event-B is a general purpose modeling language that lacks explicit support for expressing and verifying timing constraints [19]. Work has been done to add time constraints to Event-B. Butler and Falampin proposed an approach to model and refine timing properties in classical B [1], which adds a clock variable representing the current time and an operation which advances the clock [9]. The approach ensures the timing properties are satisfied by preventing

behaviours in which the clock advances to a point where deadlines would be violated. More work concerning time constraints such as delay, expiry, deadline and interval are presented recently [10,16,19]. These approaches define timing properties between different events, while Graf and Prinz introduce time to state transition systems [13].

2.3 Trigger-Response Pattern

To formally model the timing properties for the trigger-response pattern in control systems, Sarshogh and Butler proposed an approach that categorizes timing constraints in three groups: delay, expiry and deadline [16], which are denoted in (1a), (1b), (1c) below. All these three timing constraints follows a trigger-response pattern where trigger and response events are modeled as events in Event-B. (1a) shows that the *Response* event can only happen if the *delay* period has passed following the occurrence of the *Trigger* event. (1b) shows that if the *expiry* period has passed then the *Response* event can never happen. (1c) denotes that if the *Trigger* event occurs, then one of the events $Response_1$ to $Response_n$ must occur before *deadline* passes. To model these three timing properties in Event-B, a global clock as well as tick events are added to model the discrete time.

$$Delay(Trigger, Response, delay) \tag{1a}$$

$$Expiry(Trigger, Response, expiry) \tag{1b}$$

$$Deadline(Trigger, Response_1 \vee .. \vee Response_n, deadline) \tag{1c}$$

Figure 1 shows the trigger response pattern as an Event-B machine for the delay and deadline constraints, where *trigger* and *response* are modelled as events. The response event *response* must occur within time *ddl* of trigger event *trigger* occurring and can only occur if the delay period has passed. We use tT to refer to the time that trigger event happens, and we use tR to refer to the time that response event happens. Invariant @$inv1$ and @$inv2$ specify the delay and deadline timing property between *trigger* and *response* respectively. Guard @$grd3$ of the *response* event guarantees that the response is disabled when the global clock has not passed the delay period thus preserving @$inv1$. Guard @$grd1$ of the $Tick$ event constrains the global clock not to tick when the response event is missing its deadline thus preserving @$inv2$. @$inv3$ is needed to prove invariant @$inv2$. When modeling the expiry time constraint, @$grd3$ of *response* event should be $clk < tT + expiry$ to guarantee that the response is disabled when the global clock has passed the expiry period.

There are several patterns developed by Sarshogh to refine deadlines, delay and expiry. For example, to refine an abstract deadline D to sequential sub-deadlines $D_1..D_n$, there should be invariants to ensure the order of sequential sub-deadlines and the sum of the duration of sub-deadlines should be less than the abstract deadline duration [16].

```
invariants
    @inv1 tT<tR⇒tR−tT≥ dly
    @inv2 tR≥ tT⇒tR−tT≤ ddl
    @inv3 t=TRUE∧r=FALSE⇒clk−
          tT≤ ddl
  event trigger
    where
      @grd1 t=FALSE
      Ga(c,v)
    then
      @act1 t:= TRUE
      @act2 tT:= clk
      Act_a
  end
```

```
event response
  where
    @grd1 t=TRUE
    @grd2 r=FALSE
    @grd3 clk≥ tT+dly
    Gb(c,v)
  then
    @act1 r:= TRUE
    @act2 tR:= clk
    Act_b
end

event Tick
  where
    @grd1 t=TRUE∧r=FALSE⇒clk
          +1−tT≤ ddl
  then
    t:= FALSE
    r:= FALSE
end
```

Fig. 1. Model timing properties of trigger-response events with delay and deadline

Sarshogh's approach only handles the system with trigger and response pattern without specifying some possible interrupt events from the environment. To model a more complex system that supports interrupt events to interrupt current time intervals, Sulskus et al. brought up the notion of time interval constraints in Event-B [19].

The trigger-response pattern only models discrete time constraints, while the real-world events do not always happen at integer-value times. Continuous time can be modelled approximated by choosing the granularity of the global clock, which model the timed system with an approximate sense. Banach et al. presents the Hybrid Event-B extension which accommodates continuous behaviours in between discrete transitions [6]. Based on this extension, Butler et al. outlines an approach to modeling and reasoning about hybrid systems which uses continuous functions over real intervals to model the evolution of continuous values over time [8].

The trigger-response pattern also introduces a *Tick* event to proceed the global clock without any fairness assumption. Without fairness, a valid event trace may repeat trigger and response events without executing any *Tick* event, which makes the global clock never proceed. Guard @*grd*1 of *Tick* event from Fig. 1 shows that the only event that disables *Tick* event is itself. With weak fairness assumption on the *Tick* event, the *Tick* event is guaranteed to proceed the global clock in the system if the *Tick* event is enabled.

3 Task-Based Deadline Constraint

During the system design level, requirement specification are used to specify some high level specifications. We define task-based timing properties as high level timing properties to specify time constraints of individual tasks or processes. To better explain the definition, we use a simple timed mutual exclusion case study to demonstrate the usage of the framework. The timed mutual exclusion case study has two minimum requirements:

- No more than one process can be in its critical section at any time.
- If a process wishes to enter its critical section, it will enter the critical section within a certain deadline.

In the most abstract level, a mutual exclusion model is proposed which guarantees no two processes can be in the critical section at the same time. However a process can enter the critical section multiple times without allowing other process to proceed. Figure 2 gives the abstract mutual exclusion model. Here we use quantified variable p to represent one process. The event $wish(p)$ models the point at which process p wishes to enter the critical section. Event $enter(p)$ models the process entering the critical section while event $leave(p)$ models the process leaving the critical section. We add a task-based deadline constraint for each process in the first refinement, which ensures that if a process wishes to enter its critical section, it will enter the critical section within a certain deadline. The specification of the task-based deadline is presented in (2), which states that any process that wishes to enter the critical section, will enter the critical section within ddl. Figure 3 shows the refinement with task-based deadline for

```
invariants
  @inv1 wait ⊆PROCESS
  @inv2 process ⊆PROCESS
  @inv3 finite(process)
  @inv4 card(process)≤ 1
  @inv5 wait ∩process = ∅
event wish
  any p
  where
    @grd1 pro∈ PROCESS\wait
    @grd2 pro∈ PROCESS\process
  then
    @act1 wait:= wait∪{p}
end
```

```
event enter
  any p
  where
    @grd1 pro∈ wait
    @grd2 process = ∅
  then
    @act1 wait:= wait\{p}
    @act2 process:= process ∪{p}
end

event leave
  any p
  where
    @grd1 pro∈ process
  then
    @act1 process:= process\{p}
end
```

Fig. 2. Abstract model with timed mutual exclusion problem

each process. $t1(p)$ models the timestamp at which process p wishes to enter the critical section. $r1(p)$ models the timestamp process entering the critical section. @$inv4$ captures the task-based deadline constraint, and @$inv5$ is needed to prove that @$inv4$ is preserved. @$grd1$ of $Tick$ event ensures @$inv5$ is preserved.

$$\forall p \cdot p \in wait \;\Rightarrow\; Deadline(wish(p), enter(p), ddl) \tag{2}$$

```
1  invariants
2    @inv1 clk ∈ ℕ
3    @inv2 t1 ∈ wait →0..clk
4    @inv3 r1 ∈ process →0..clk
5    @inv4 ∀p·r1(p)>t1(p) ⇒r1(p)−t1(p
         )≤ ddl1
6    @inv5 ∀p·p∈ wait ⇒clk−t1(p)≤
         ddl1
7
8  event enter extends enter
9    then
10     @act3 r1(p):= clk
11 end
```

```
1  event wish refines wish
2    any p
3    where
4      @grd1 pro∈ PROCESS\wait
5      @grd2 pro∈ PROCESS\process
6    then
7      @act1 wait:= wait∪{p}
8      @act2 t1(p):= clk
9  end
10
11 event tick
12   where
13     @grd1 ∀p·p∈ wait ⇒clk+1−t1(p)
           ≤ ddl1
14   then
15     @act1 clk:= clk+1
16 end
```

Fig. 3. First refinement with task-based deadline constraint

4 Scheduler-Based Deadline Constraint with Nondeterministic Queue Based Scheduling

In concurrent computing, concurrent processes are executed by interleaving the execution steps of each process, which models processes in the outside world that happen concurrently. In real time systems, scheduling is used to make sure that all processes meet their deadlines [4]. A scheduler is used to allocate the resource to a process for some time.

In this case study, we specify two scheduler-based deadlines: (3a) and (3b). (3a) requires that when the system is idle, one of the requesting processes will enter the critical section within $ddl3$. Specifically, there are two cases that trigger the scheduling of the enter event: (1) a process wishes to enter and both the queue and the critical section are empty, and (2) some process leaves the critical section and there is some other process waiting in the queue. Observe here that events can act as timing triggers only under certain conditions, e.g., the wish event is only a timing trigger when the queue and critical section are empty.

To deal which such conditional triggers, we split the event into separate refinements representing separate cases. We refine the *wish* event into a *wish_empty* event, enabled when condition 1 is true, and a *wish_nonempty* event, enabled in all other cases. Similarly, we split the *leave* event into a *leave_nonempty* event, enabled when condition 2 is true, and a *leave_idle* event, enabled in the other cases (see Fig. 5). The events *wish_empty* and *leave_nonempty* are therefore used as trigger events in (3a), whereas the response event *enter* is the event modeling entering the critical section.

(3b) requires that, once a process enters the critical section, it will leave the critical section within $ddl2$. Therefore the trigger event is the *enter* event, whereas the response events should correspond to leaving the critical section. As the latter is now captured by *two* events, there are two response events in (3b).

$$Deadline(\{wish\_empty, leave\_nonempty\}, enter, ddl3) \tag{3a}$$

$$Deadline(enter, \{leave\_empty, leave\_nonempty\}, ddl2) \tag{3b}$$

To refine the task-based deadline constraint with scheduler-based deadline constraints, we propose a nondeterministic queue based scheduling framework to address the schedule order of the sequential execution of a set of events. In this framework, a queue is used to manage the ready processes. Each process is assigned a position in the queue, formally: $queue \in wait \rightarrowtail (0..N-1)$. When one process is ready, it is nondeterministically assigned a natural number that is not in the range of the queue. Only the process in the front of the queue ($min(ran(queue))$) can get the resource to run. The dequeue operation will decrease the indexes of all the other processes in the queue by the index of the front process plus one ($min(ran(queue)) + 1$) to guarantee that once a process is added to the queue, it will eventually get the chance to run. In the second refinement, we use this nondeterministic queue based framework to impose an order on the execution of the concurrent tasks. This refinement prevents a process from entering the critical section forever without allowing other processes to enter the critical section. The second refinement is shown in Fig. 4.

```
1  invariants
2    @inv2 queue ∈ wait ⤖ 0..N−1
3  event wish extends wish
4    any i
5    where
6      @grd3 i ∈ 0...N−1
7      @grd4 i∉ ran(queue)
8    then
9      @act3 queue(p):= i
10 end
```

```
1  event enter extends enter
2    any j
3    where
4      @grd4 p=queue∼(j)
5      @grd5 j=min(ran(queue))
6      @grd6 j∈ ran(queue)
7      @grd7 queue≠∅
8    then
9      @act4 queue:= (λ q·q∈ dom({p}⩤
           queue)| queue(q)−j−1)
10 end
```

Fig. 4. Second refinement with queue based scheduling framework

In order to prove that the scheduler deadlines refine the task-based deadlines, invariants capturing the relation between task-based deadlines and scheduler-based deadlines are needed. Assume that in the abstract machine the trigger event of one process p occurs at timestamp $t1$, and the deadline is ddl. In the refined machine the trigger event ($wish\_empty/wish\_nonempty$) starts at timestamp $t3$ and its deadline is $ddl3$. The trigger event ($enter$) starts at timestamp $t2$ and its deadline is $ddl2$. We assume that all processes have the same maximum possible deadline to enter and leave the critical section $ddl23 = ddl2 + ddl3$. The process p has to wait for all the processes ahead of it in the queue to enter and leave the critical section. The total waiting time is proportional to its index in the queue, which is $queue(p) * ddl23$. If the critical section is empty and the time that last process leaves the critical section is $t3$, p should enter the critical section within $t3 + queue(p) * ddl23 + ddl3$. If the critical section is not empty and the time that last process enters the critical section is $t2$, p should enter the critical section within $t2 + queue(p) * ddl23 + ddl23$. Based on different conditions, the sum of the refined sequential deadline should be less than abstract deadline $t1 + ddl1$, which is shown in @$inv9$ and $inv10$ in Fig. 6. @$inv9$ and $inv10$ present these two conditions as required gluing invariants. Assume there are N processes, the worst case is $N - 1$ processes in the waiting list. As @$axm8$ presents, in the worst case the refined deadline is less than the abstract deadline. Figure 6 shows the required axioms and invariants to refine the task-based deadlines to scheduler-based deadlines. @$inv5$ and @$inv8$ present the invariant for scheduler-based deadlines (3b), @$inv6$ and @$inv7$ present the invariants for scheduler-based deadlines (3a).

5 Two Implementation of Nondeterministic Queue-Based Framework

The nondeterministic queue based scheduling framework is a general framework that assign indexes to processes nondeterministically. By applying additional rules on the assignment of these indexes, the queue based scheduling framework can be refined to some scheduling policies such as First In First Out (FIFO) and deferrable priority based scheduling policy with aging technique.

First In First Out. FIFO is one of the scheduling policies that guarantee that the resources are assigned to each process with the order that they require the resource. The FIFO scheduling policy handles all processes without priorities. The queue based scheduling framework assigns each process with a corresponding natural number $k \in \mathbb{N}$, and FIFO scheduling policy limits this natural number to the current size of the queue. And when the critical section is empty, the process that is in the front of queue leaves the queue and enters the critical section. The indexes of all the other processes in the queue are reduced by one.

The refinement from the scheduler-based model is shown in Fig. 7. Initially the queue is empty and $qsize$ is zero. Whenever some process is added to the queue, it is assigned with the number of queue size and the queue size increases

```
event wish_empty extends wish
where
  @grd5 wait=∅∧process=∅
then
  @act5 t3 := clk
end

event wish_nonempty extends wish
where
  @grd5 wait≠∅∨process≠∅
end

event enter extends enter
then
  @act6 t2 := clk
  @act7 r3 := clk
end
```

```
event leave_nonempty extends leave
where
  @grd2 queue≠∅
then
  @act3 r2 := clk
  @act4 t3 := clk
end

event leave_idle extends leave
where
  @grd2 queue=∅
then
  @act2 r2 := clk
end

event tick refines tick
where
  @grd2 process=∅ ∧wait≠∅ ⇒clk
        +1−t3≤ ddl3
  @grd4 process≠∅ ⇒clk+1−t2≤ ddl2
then
  @act1 clk:= clk+1
end
```

Fig. 5. Third refinement to scheduler-based deadline constraint

```
axioms
  @axm7 ddl23 = ddl2+ddl3
  @axm8 ((N−1) ∗ ddl23)+ddl3 ≤ ddl1
invariants
  @inv5 r2>t2 ⇒r2−t2≤ ddl2 // deadline2( enter, leave_idle|leave_nonempty,
        ddl2)
  @inv6 r3>t3 ⇒r3−t3≤ ddl3 // deadline3( wish_empty|leave_nonempty, enter,
        ddl3 )
  @inv7 queue≠∅ ∧process=∅ ⇒clk−t3≤ ddl3 // required for enter to preserve
        inv6
  @inv8 process≠∅ ⇒clk−t2≤ ddl2 // required for leave to preserve inv5
  @inv9 ∀p·process=∅ ∧p∈ wait ⇒t3+(queue(p)∗ddl23)+ddl3 ≤ t1(p)+ddl1
  @inv10 ∀p·process≠∅ ∧p∈ wait ⇒t2+(queue(p)∗ddl23)+ddl23 ≤ t1(p)+ddl1
```

Fig. 6. Axioms and invariants needed to refine task-based deadline to scheduler-based deadline

by one. When the critical section is empty, the process in the front of queue *queue*(0) is removed from the queue, the indexes of all the other processes in the queue are reduced by one. The queue size also decreases by one. *wish_nonempty* uses the same refinement strategy as *wish_empty*.

```
event enter refines enter
  any p j
    where
    @grd1 pro∈ wait
    @grd2 process = ∅
    @grd6 j∈ ran(queue)
    @grd4 p=queue∼(j)
    @grd5 j=0
    @grd7 queue≠∅
  then
    @act1 wait:= wait\{p}
    @act2 process:= process ∪{p}
    @act3 r1(p):= clk
    @act4 queue:= (λ q·q∈ dom({p}⩤
          queue) | queue(q)−j−1)
    @act6 t2 := clk
    @act7 r3 := clk
    @act8 qsize:= qsize−1
end
```

```
invariants
  @inv1 qsize ∈ 0..N−1
  @inv2 ∀i·i≥ qsize⇒i∉ ran(queue)

event wish_empty refines wish_empty
  any p i
  where
    @grd1 pro∈ PROCESS\wait
    @grd2 pro∈ PROCESS\process
    @grd3 i =qsize
    @grd5 wait=∅∧process=∅
    @grd6 qsize<N
  then
    @act1 wait:= wait∪{p}
    @act2 t1(p):= clk
    @act3 queue(p):= i
    @act5 t3 := clk
    @act6 qsize:= qsize+1
end
```

Fig. 7. Refinement with FIFO queue

Deferrable Priority Based Scheduling with Aging Technique. Fixed priority scheduling policies assign the tasks with fixed priorities: $priorities \in event \rightarrow \mathbb{N}$. The scheduler will select the tasks with higher priorities to access the system resources before those with lower priorities. However there is a disadvantage of these scheduling policies that tasks with lower priorities may be starved when the tasks with higher priorities keep coming and jumping the queue. An aging technique is used to ensure that tasks with lower priorities eventually complete their execution. The general way to implement aging technique is to increase the priorities of the tasks with lower priorities while they are waiting in the ready queue. However with the increasing priorities of some processes, it will occupy the positions of some other processes. Deferrable priority based scheduling allows that when the position of one process is occupied by some other processes, this process is deferred with a random position after its assigned position. Using the timed mutual exclusion problem as an example, our approach to refine the scheduling framework to priority based scheduling with aging technique is to define a $pindex \in PROCESS \rightarrowtail 0..N-1$, where $pindex$ is a bijection function from $PROCESS$ to natural numbers. Equation (4) shows that the higher priority of a process is, the lower its index is.

$$\begin{aligned} &\forall a, b \cdot a \in PROCESS \wedge b \in PROCESS \wedge priority(a) < priority(b) \\ &\Rightarrow pindex(a) > pindex(b) \end{aligned} \tag{4}$$

To avoid the starving problem of processes with lower priorities, we add a rule to the priority based scheduling that when the position of some process is occupied

by some other process with lower priority, which means that the lower priority one has waited some time in the queue, the high priority one is deferred by some higher random index. Specifically, the indexes of the processes are decreasing by $min(ran(queue)) + 1$ when the process at the front queue, whose index is $min(ran(queue))$, leave the queue and enter the critical section. The *enqueue* operation will assign the process with its corresponding index in the queue. However, this would cause a conflict as this operation will make some processes occupy the spaces of some other processes. For example, process a's level is 3 and process b's level is 2. One process c is at the front of queue. When c leaves the queue, the index of a is reduced to 2. When b wishes to enter the queue, its position is taken by a. Here we choose the next available space available in the queue $i = min(k|k \in ran(pindex) \wedge k \notin ran(queue) \wedge k > pindex(p))$. When the position is not taken by other processes, the process takes its assigned position $pindex(p)$. The dequeue operation is the same as the basic queue based scheduling framework. Figure 8 shows the refinement from scheduler-based model with a deferrable priority based scheduling policy with aging technique.

```
1  event wish_empty refines wish_empty
2    any p i
3    where
4      @grd1 pro∈ PROCESS\wait
5      @grd2 pro∈ PROCESS\process
6      @grd3 {k|k∈ ran(pindex)∧k∉ ran(queue)∧k≥ pindex(p)}≠∅
7      @grd4 i=min({k|k∈ ran(pindex)∧k∉ ran(queue)∧k≥ pindex(p)})
8      @grd5 wait=∅∧process=∅
9    then
10     @act1 wait:= wait∪{p}
11     @act2 t1(p):= clk
12     @act3 queue(p):= i
13     @act5 t3 := clk
14 end
```

Fig. 8. Refinement with deferrable priority based scheduling with aging technique

Proof Statics. Table 1 shows the proof statics of the model. Here $m0$ is the abstract machine with a simple mutual exclusion problem. $m1$ refines $m0$ with the task-based deadline for each process. $m2$ refines $m1$ with a nondeterministic queue based scheduling policy. $m3$ refines $m2$ to the scheduler-based deadline with additional gluing invariants. $m4\_fifo$ and $m4\_priority$ are two different implementations of the nondeterministic queue. FIFO queue has more proof obligations than priority queue because additional invariants @$inv1$ and @$inv2$ of Fig. 7 are needed for the FIFO queue refinement.

Table 1. Proof statistics

Machine	Number of generate PO	Automatically proved	Automatically proved %
m0	7	7	100
m1	18	18	100
m2	9	7	77.8
m3	49	43	87.8
m4_fifo	17	12	70.6
m4_priority	9	7	77.8

6 Conclusions and Future Work

Based on a trigger-response approach to modeling deadlines in Event-B, we distinguish the concept of task-based timing properties and scheduler-based timing properties from the perspective of different system design phases. We define timing properties that place discrete time constraints on individual processes or tasks as task-based timing properties, which describe high level timing properties from system requirement specification phase. These task-based timing properties can not describe the concurrent behaviour of tasks precisely. In real time systems, schedulers are used to schedule concurrent tasks. To model the behaviour of these concurrent processes, we define scheduler-based timing properties as concrete timing properties for the system design phase, which place discrete time constrains on the scheduler which schedules the concurrent tasks. To refine task-based timing properties to scheduler-based timing properties, we introduce a nondeterministic queue based scheduling policy with some additional gluing invariants. The queue based scheduling policy can also be implemented as a FIFO queue scheduling policy or a deferrable priority based scheduling policy with aging technique. We prove that the two refinements of the nondeterministic queue satisfy the same deadlines with the mutual timed exclusion case study.

This paper does not address the possible time deadlock caused by the trigger-response pattern. For example, if the delay is larger than the deadline between the same trigger-response pair, there would be a point where the global clock cannot proceed as it is constrained by the deadline constraint not to proceed but also constrained by the delay constraint to proceed, a deadlock will occur in the model. Additional conditions to avoid these deadlocks and formal specifications of the enabledness and weak fairness assumptions on response events with proofs are left for future work. In addition, the mutual exclusion case study assumes no intermediate events between trigger and response events, while intermediate events are common in real systems such as CPS. More exploration is needed for the enabledness of intermediate and response events under different situations. Fairness and convergence assumptions on intermediate events and response events will help with the scaling of the proposed approach.

In the cases that the system does not require explicit mention of time, the notion of bounded fairness and finitary fairness allows one to express eventual occurrence of a set of events. Some work has been done to model fairness in Event-B [17,18]. Bounded fairness modeling as well as finitary fairness modeling can be researched further with some addition prove rules and refinement frameworks.

In order to explicitly represent timing properties in a cyber physical system, there are three typical time constraints to look into: period, deadline, worst-case execution time. More work can be done to apply some scheduling policies such as Rate-Monotonic (RM) and priority inheritance protocol based on the queue based scheduling framework to analyze real-time performance of CPS together with the mentioned time constraints in Event-B.

Acknowledgement. This work is supported in part by the scholarship from China Scholarship Council (CSC) under the Grant CSC NO. 201708060147.

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (2005)
2. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
3. Alur, R., Henzinger, T.A.: Finitary fairness. In: Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science, pp. 52–61, July 1994
4. Alur, R.: Principles of Cyber-Physical Systems. The MIT Press, Cambridge (2015)
5. Alur, R., Dill, D.L.: A theory of timed automata. Theoret. Comput. Sci. **126**(2), 183–235 (1994)
6. Banach, R., Butler, M., Qin, S., Verma, N., Zhu, H.: Core hybrid Event-B I: single hybrid Event-B machines. Sci. Comput. Program. **105**, 92–123 (2015)
7. Butler, M.: Mastering System Analysis and Design through Abstraction and Refinement. IOS Press, Amsterdam (2013). http://eprints.soton.ac.uk/349769/
8. Butler, M., Abrial, J.R., Banach, R.: Modelling and refining hybrid systems in Event-B and Rodin. In: Petre, L., Sekerinski, E. (eds.) From Action System to Distributed Systems: The Refinement Approach. Taylor & Francis, April 2016. https://eprints.soton.ac.uk/376053/
9. Butler, M., Falampin, J.: An approach to modelling and refining timing properties in B, January 2002. https://eprints.soton.ac.uk/256235/
10. Cansell, D., Méry, D., Rehm, J.: Time constraint patterns for Event B development. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 140–154. Springer, Heidelberg (2006). https://doi.org/10.1007/11955757_13
11. Dershowitz, N., Jayasimha, D.N., Park, S.: Bounded fairness. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 304–317. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39910-0_14
12. Dierks, H., Kupferschmid, S., Larsen, K.G.: Automatic abstraction refinement for timed automata. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 114–129. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75454-1_10
13. Graf, S., Prinz, A.: Time in state machines. Fundam. Inform. **77**, 143–174 (2007)

14. Jastram, M., Butler, P.: Rodin User's Handbook: Covers Rodin V.2.8. 2.8covers Rodin. Createspace Independent Pub, North Charleston (2014). https://books.google.co.uk/books?id=ws2WoAEACAAJ
15. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. Int. J. Softw. Tools Technol. Transf. **1**(1–2), 134–152 (1997)
16. Sarshogh, M.R., Butler, M.: Specification and refinement of discrete timing properties in Event-B. In: AVoCS 2011 (2011). https://eprints.soton.ac.uk/272480/
17. Sekerinski, E., Zhang, T.: Finitary fairness in Event-B. In: Dagstuhl Seminar on Refinement Based Methods for the Construction of Dependable Systems. Dagstuhl, Germany (2009)
18. Sekerinski, E., Zhang, T.: Finitary fairness in action systems. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) ICTAC 2013. LNCS, vol. 8049, pp. 319–336. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39718-9_19
19. Sulskus, G., Poppleton, M., Rezazadeh, A.: An interval-based approach to modelling time in Event-B. In: Dastani, M., Sirjani, M. (eds.) FSEN 2015. LNCS, vol. 9392, pp. 292–307. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24644-4_20. http://eprints.soton.ac.uk/377201/

Verifiable Code Generation from Scheduled Event-B Models

Mohammadsadegh Dalvandi(✉), Michael Butler, Abdolbaghi Rezazadeh, and Asieh Salehi Fathabadi

University of Southampton, Southampton, UK
{md5g11,mjb,ra3,asf08r}@ecs.soton.ac.uk

Abstract. Scheduled Event-B (SEB) augments Event-B with a scheduling language to make the control flow in an Event-B model explicit and facilitate derivation of algorithmic structure in Event-B refinement. A concrete SEB model has a concrete algorithmic structure associated with it. Although this structure reduces the difficulty of code generation, there is still some gap between the model and executable code. This work formulates the translation of SEB models to a programming language called Dafny and proposes an approach in which a number of assertions are generated from the model that allows the verification of the generated code in a static program verifier.

1 Introduction

Event-B is a general purpose formal method which is designed to target a set of different domains including distributed systems, sequential programs, and embedded systems. This generality is achieved by not fixing the behavioural semantics of Event-B models [11]. Although this approach provides a great degree of freedom in using the method, the process of using Event-B in some domains (e.g. sequential program development) remains underdeveloped and not always easy to follow. In our previous work [6] we introduced Scheduled Event-B (SEB). SEB augments Event-B with a scheduling language and provides a number of refinement rules to facilitate derivation of algorithmic structure in Event-B refinement. It allows the modeller to introduce the algorithmic structure using the scheduling language from the very abstract level. The model, together with its algorithmic structure, is then refined towards a concrete level. The final refinement step results in a concrete Event-B model (i.e. a model with concrete data structures and no non-determinacy) and a concrete algorithmic structure (i.e. a deterministic algorithmic structure). It is assumed that this final refinement level is the closest possible model to the final implementation, i.e. a one to one mapping between the model constructs and the target language constructs exists. The most basic building block of a SEB model is an event. An event may have multiple actions (i.e. assignments) which model state changes and are executed simultaneously. If we consider assignments to be the most basic executable building blocks of an executable program, then each event should be

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 234–248, 2018.
https://doi.org/10.1007/978-3-319-91271-4_16

broken down to a number of assignments in the target language. Since event actions are considered to be executed at the same time, the syntactic ordering between them is not important. However, in a programming language, the order in which the assignments are sequentially executed may change the final state of the program. Due to this fact, when an event is sequentialised (i.e. its actions are translated to sequentially composed assignments), then the imposed ordering on assignments should be verified to prove that the sequential execution of the assignments will change the state in the same way that the execution of the atomic event changes it. This verification task can be carried out at the Event-B level. However, the problem with doing this in Event-B is a huge overhead caused by the introduction of new auxiliary variables, program counters, new invariants and events required for modelling and verification of the sequential execution of the actions of a single event. To avoid the aforementioned overhead, we can delegate this verification task to a program verifier which is much more sequential composition friendly than Event-B. This can be achieved by placing assertions in the program generated from an Event-B model in a way that proving the assertions implies that the sequentialised assignments change the state in the same way that the original atomic event does.

The above proposed approach takes advantage of abstraction and refinement offered by Event-B in developing an algorithm correctly and also benefits from modern and powerful program verifiers for verification of low level properties of the final implementation in order to prove that the final generated program implements the Event-B model correctly. In this paper we use the Dafny programming language and its verifier as our target language for implementing Event-B models.

This paper has two main contributions. First, it provides a set of rules for transforming a SEB model to an executable code in Dafny. Second, it introduces an approach for sequentialisation of atomic events and verifying its correctness using Dafny verifier. The rest of this paper is organised as follows: Sect. 2 provides background information required for understanding this work including an introduction to Event-B, Scheduled Event-B, and Dafny. Section 3 provides a set of transformation rules for transforming a SEB model to Dafny code. Section 4 discusses the verification of sequentialised model using Dafny verifier. Finally Sect. 5 discusses the future work and concludes the paper.

2 Background

2.1 Event-B

Event-B is a formal modelling language based on set theory and predicate logic for modelling and reasoning about systems, introduced by Abrial [2]. Event-B is greatly inspired by Action Systems [4] and the B-Method [1]. Modelling in Event-B is facilitated by an extensible platform called Rodin [3]. A model in Event-B usually has two main parts: a *context* and a *machine*. A context is the static part (types and constants) of a model which is specified using carrier sets, constants and axioms. A machine is the dynamic part (variables and events)

of a model which is specified by means of variables, invariants and events. An *event* models the state change in the system. Each event may have a number of assignments called actions which are executed simultaneously. Each event may also have a number of guards. Guards are predicates that describe the necessary conditions which should be true before an event can occur. An event can be parametrised by means of event parameters. A general Event-B event has the following form:

$$\boxed{\text{Evt} \triangleq \textbf{any } t \textbf{ when } P(t,v) \textbf{ then } S(t,v) \textbf{ end}}$$

where Evt is the name of the event, t is a set of parameters, v is the set of model variables, $P(t, v)$ is a set of guards and $S(t, v)$ is a set of actions. Modelling a complex system in Event-B can largely benefit from abstraction and refinement. Refinement is a stepwise process which starts from an abstract level and continues towards a more concrete level by a series of successive steps in which new details of functionality are added to the model in each step [5]. The abstract level models the general purpose of the system by specifying *what* the system is supposed to achieve. Each refinement level adds more details on *how* the goal of the system can be achieved. It is essential that the correctness of each refinement is proved, i.e. proving that each refinement "displays the same behaviour" as the abstract one [15].

Refinement of an Event-B model may consist of refining existing events and/or adding new events, variables and invariants. The new events must not diverge. This means that they should not be enabled for ever. Each refinement may involve introducing new variables to the model. This usually results in extending abstract events or adding new events to the model. It is also possible to replace abstract variables by newly defined concrete variables (*data refinement*). Concrete variables are related to abstract variables through *gluing invariants*. A gluing invariant associates the state of the concrete machine with that of its abstraction. All invariants of a concrete model including gluing invariants should be preserved by all events. All abstract events may be refined by one or more concrete event.

2.2 Scheduled Event-B

In Event-B the control flow between events are implicitly encoded using event guards. Whenever the guards of an event are true, the event is considered to be enabled and can be executed. The lack of explicit control flow in Event-B can make algorithm and sequential program development difficult. To deal with the problem of control flow, in our previous work we introduced Scheduled Event-B (SEB) [6]. SEB augments Event-B with an explicit control flow construct called a *schedule*. Each refinement level has an associated schedule. A schedule provides the modeller with a set of abstract and concrete programming-like control constructs and allows the introduction of the control flow to a model from the very abstract level. SEB also provides a number of rules for schedule refinement. The rules allow the modeller to refine the abstract schedule along with the abstract

model to a more concrete level. The final level of refinement will result in a concrete algorithm with only deterministic control constructs left in it. Figure 1 shows the abstract and concrete control structures provided by SEB.

⟨*Schedule*⟩ ::= *Event*
| ⟨*Schedule*⟩ ; ⟨*Schedule*⟩
| ⟨*Schedule*⟩ □ ⟨*Schedule*⟩
| ⟨*Schedule*⟩*
| **if**(⟨*Cond*⟩){⟨*Schedule*⟩}, {**elseif**(⟨*Cond*⟩){⟨*Schedule*⟩}},[**else**{⟨*Schedule*⟩}]
| **while**(⟨*Cond*⟩){⟨*Schedule*⟩}

⟨*Cond*⟩ ::= *Predicate*

Fig. 1. The scheduling language. The language is presented in EBNF [18].

The simplest form of a schedule is a single event. *Event* denotes an event in the schedule. A schedule may contain one or more Event-B events. A sequential order can be imposed by the sequential composition operator (;). Nondeterministic choice ($S_1 \square S_2$) and iteration (S^*) are the abstract control structures. Iteration is required to be finite. This is enforced by proving convergence of events. The aforementioned control structures allow us to retain the event structure (guards and actions together) so that data refinement reasoning is localised to pairs of corresponding abstract and refining events using the standard definition of the Event-B refinement. The concrete control structures include deterministic `if..else` branches and `while` loops with explicit conditions (*Cond*). The branch and loop conditions should be valid Event-B predicates as defined in [2]. Non-deterministic choices and iterations can be refined to deterministic branches and loops, respectively. Schedule refinement rules are defined in [6]. Figure 2 depicts how a schedule is refined alongside with the Event-B model. To illustrate scheduled Event-B, we use the binary search algorithm presented in [6] here. We only provide the most concrete Event-B model of the search algorithm:

Machine $m3$ **refines** $m2$ **Sees** $c0$
Variables r, k, i, j
Initialisation $r := 0, k := (n-1)/2, i := 0, j = n-1$

Event *search_inc*
refines *search*
where
grd1: $f(k) < v$
then
act1: $k := (k+j+1)/2$
act2: $i := k+1$
End

Event *search_dec*
refines *search*
where
grd1: $f(k) > v$
then
act1: $k := (i+k-1)/2$
act2: $j := k-1$
End

Event $found$
refines $found$
where
grd1: $f(k) = v$
then
act1: $r := k$
End

In the above model, f represents an array modelled as a total function in Event-B ($f \in 0..n-1 \rightarrow \mathbb{Z}$). The concrete schedule associated with the above model is as follows:

$$\begin{array}{c} initialisation; \\ \textbf{while}(f(k) \neq v)\{\textbf{if}(f(k)<v)\{search\_inc\}\ \textbf{else}\{search\_dec\}\}; \\ found \end{array}$$

The above schedule defines the control flow of the Event-B model. The schedule contains a `while` loop and an `if..else` branch with explicit conditions. The explicit conditions in the schedule allow the guards in the events to be eliminated when generating the code (see Sect. 3). Variables r, k, i, and j are of type integer. f is a sorted array defined in the context $c0$. The above model does not include the context $c0$ or any of the abstract machines.

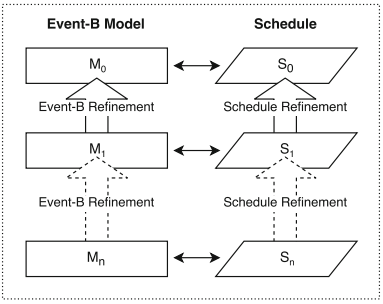


Fig. 2. Event-B and schedule refinement

2.3 Dafny

Dafny is an imperative, class-based language [13], which allows both strong and weak typed variables. Dafny implements the verification method of Hoare logic where a program can be specified with pre- and post-conditions. In the Dafny language, pre- and post-conditions are influenced by the Eiffel language [16] and the concept of *design-by-contract* [17]. Dafny is an object-oriented programming language with generic classes and allows creation of objects which gives rise to pointers [14]. Despite the fact that Dafny is a class-based language, it does not support subclasses and inheritance. However, there is a built-in **object** type that is a super-type of all class types. Dafny supports inductive datatypes and has its own specification constructs. Standard pre- and post-conditions, framing constructs and termination metrics are included in the specifications. In this paper we call these specification constructs, *code contracts*. The language also offers recursive functions, sets, sequences and some other features to support specification. Dafny allows the definition of **ghost** variables. A ghost variable is a variable that is used by the Dafny verifier and ignored at run time. A ghost

variable is used for specification purposes only and does not appear in any part of the implementation. Specifications and ghost variables are omitted by the compiler and are used just during the verification process.

The Dafny verifier attempts to verify different parts of a program locally (modular verification) and infer the correctness of the whole system from those locally verified parts. The Dafny verifier translates a Dafny program to an immediate verification language known as Boogie 2 [12]. This is done in a way that the correctness of the generated Boogie program implies the correctness of the Dafny program. First-order verification conditions then are generated by the Boogie tool and passed to the Z3 SMT solver [8].

3 Translating Concrete SEB Models to Dafny

A scheduled Event-B model, in its final refinement level, has a concrete schedule (i.e. the schedule has only events, `;`, `while` and/or `if..else`) associated with it. It is assumed that all constructs in the model are refined to a concrete level and all non-deterministic assignments are replaced with deterministic ones. The concrete schedule is assumed to be a correct refinement of the abstract one with respect to the refinement rules introduced in [6]. This section explains how a SEB model is translated to Dafny implementation. We will use the model of the binary search algorithm presented in Sect. 2.2 as an example to illustrate the translation. To formulate the translation of SEB models to Dafny, we define a function called $SEB2DFY$. The function accepts an Event-B model (M) (consisting of a machine and the context it sees) and a schedule (S) and returns generated code and contracts:

$$SEB2DFY(M, S) \triangleq \ SEB2DFY_{\text{class}}(M, S) \tag{1}$$

Function $SEB2DFY_{\text{class}}$ defines a class including a method implementing the algorithm. They are discussed in the following sections. The input model M and schedule S are expected to be refined to a concrete level as explained earlier.

3.1 Dafny Method Generation

The focus of SEB is on development and verification of sequential algorithms. SEB does not yet cover concepts like method calls or recursions. With this in mind, for the purpose of code generation, it would be an appropriate decision to map a SEB model to a class with a method implementing the algorithm based on the provided schedule S. Based on this decision, function $SEB2DFY_{\text{class}}$ will return a class with a single method with the same name as the model which was passed to it:

$$\begin{aligned} SEB2DFY_{\text{class}}(M, S) \triangleq \ & \texttt{class}\ mchn\{ \\ & SEB2DFY_{\text{mtd}}(M, S) \\ & \} \end{aligned} \tag{2}$$

Function $SEB2DFY_{\text{mtd}}(M, S)$ defines the way that the method should be generated:

$$
\begin{aligned}
SEB2DFY_{\text{mtd}}(M,S) \triangleq\ & \texttt{method}\ mchn(SEB2DFY_{\text{args}}(M)) \\
& SEB2DFY_{\text{pre}}(M) \\
& \{ \\
& \quad SEB2DFY_{\text{var}}(v_1, inv_{v1}) \\
& \quad SEB2DFY_{\text{var}}(v_2, inv_{v2}) \\
& \quad \vdots \\
& \quad SEB2DFY_{\text{var}}(v_n, inv_{vn}) \\
& \quad SEB2DFY_{\text{alg}}(M,S) \\
& \}
\end{aligned}
\tag{3}
$$

In the above class and method *mchn* is a placeholder for the name of the machine being translated. If there is a value that the algorithm needs to receive in order to perform a specific task on it such as an unsorted array to be sorted, it is usually declared and specified in the model context using constants and axioms. In this case the constant is mapped to an input argument which is passed to the method and the axioms specifying it are transformed to method pre-conditions. Functions $SEB2DFY_{\text{args}}(M)$ and $SEB2DFY_{\text{pre}}(M)$ are used to generate the method's input arguments and its necessary pre-conditions. Assume that we have a model containing machine *mchn* and a context with constants $a_1, \ldots, a_{\text{k}}$ where each constant is of type $T_1, \ldots, T_{\text{k}}$, respectively. Function $SEB2DFY_{\text{args}}(M)$ has the following definition:

$$
SEB2DFY_{\text{args}}(M) \triangleq\ a_1 : T_1, \ldots, a_{\text{k}} : T_{\text{k}} \tag{4}
$$

If the context of model M has n axioms specifying input arguments then function $SEB2DFY_{\text{pre}}(M)$ has the following definition:

$$
\begin{aligned}
SEB2DFY_{\text{pre}}(M) \triangleq\ & \texttt{requires}\ SEB2DFY_{\text{pred}}(axm_1) \\
& \vdots \\
& \texttt{requires}\ SEB2DFY_{\text{pred}}(axm_{\text{n}})
\end{aligned}
\tag{5}
$$

The `requires` keyword is used in Dafny to declare method pre-conditions. The function $SEB2DFY_{\text{pred}}$ transforms an Event-B predicate to its Dafny equivalent. Function $SEB2DFY_{\text{var}}$ gives rise to generation of variable declarations including typing invariants. Finally, $SEB2DFY_{\text{alg}}(M, S)$ generates the implementation and necessary contracts. This function will be discussed in detail in the rest of this paper.

3.2 Algorithm Generation

An important step in transforming a SEB model to Dafny code is the generation of the code implementing the algorithm. Function $SEB2DFY_{\text{alg}}(M, S)$

formulates this step. Schedule S contains key information about the algorithmic structure of model M. A schedule is usually comprised of a number of sub-schedules (which are either a control structure or a single event) ordered using sequential composition operator:

$$S \triangleq S_1 ; \ldots ; S_{\mathrm{n}}$$

If a schedule is comprised of a number of sub-schedules like the above, then function $SEB2DFY_{\mathrm{alg}}(M, S)$ is defined as follows:

$$\begin{aligned} SEB2DFY_{\mathrm{alg}}(M, S) \triangleq \quad & SEB2DFY_{\mathrm{alg}}(M, S_1) \\ & \vdots \\ & SEB2DFY_{\mathrm{alg}}(M, S_{\mathrm{n}}) \end{aligned} \tag{6}$$

As mentioned before, a sub-schedule may be a control structure (branch or loop) or an event. The general form of a branch sub-schedule is as follows:

$$S_{\mathrm{i}} \triangleq \texttt{if(}c_1\texttt{)\{}\, s_1 \,\texttt{\}elseif(}c_2\texttt{)\{}\, s_2 \,\texttt{\}}\ldots\ \texttt{else\{}\, s_n \,\texttt{\}}$$

where $c_1, \ldots, c_{\mathrm{n}-1}$ are branch conditions (in the form of Event-B predicates) and $s_1, \ldots, s_{\mathrm{n}}$ are schedules. In this case the definition of $SEB2DFY_{\mathrm{alg}}(M, S_{\mathrm{i}})$ is as follows:

$$\begin{aligned} SEB2DFY_{\mathrm{alg}}(M, S_{\mathrm{i}}) \triangleq \quad & \mathrm{if}(SEB2DFY_{\mathrm{pred}}(c_1))\{ \\ & \quad SEB2DFY_{\mathrm{alg}}(M, s_1) \\ & \} \\ & \mathrm{elseif}(SEB2DFY_{\mathrm{pred}}(c_2))\{ \\ & \quad SEB2DFY_{\mathrm{alg}}(M, s_2) \\ & \} \\ & \vdots \\ & \mathrm{else}\{ \\ & \quad SEB2DFY_{\mathrm{alg}}(M, s_{\mathrm{n}}) \\ & \} \end{aligned} \tag{7}$$

If sub-schedule S_{j} is a loop then it has the following general form:

$$S_{\mathrm{j}} \triangleq \texttt{while(}c\texttt{)\{}\, s \,\texttt{\}}$$

where c is the loop condition and s is a schedule representing the body of the loop. The definition of $SEB2DFY_{\mathrm{alg}}(M, S_{\mathrm{j}})$ is as follows:

$$\begin{aligned} SEB2DFY_{\mathrm{alg}}(M, S_{\mathrm{j}}) \triangleq \quad & \mathrm{while}(SEB2DFY_{\mathrm{pred}}(c))\{ \\ & \quad SEB2DFY_{\mathrm{alg}}(M, s) \\ & \} \end{aligned} \tag{8}$$

Now that we defined $SEB2DFY_{\text{alg}}$ for branches and loops, we need one more definition for the case that a (sub-)schedule is a single event. This case will be discussed in the next section in detail.

3.3 Events to Sequential Statements

The most basic component of a schedule is an event. Event-B events usually have a number of guards and actions. In [6] we showed that a correct schedule allows us to eliminate event guards because guards should follow explicit schedule guards. Elimination of event guards is facilitated through a number of guard propagation and elimination rules. These rules allow us to propagate explicit schedule guards (loop or branch conditions) to events and eliminate event original guards safely. As an example consider the following schedule:

$$\textbf{while}(a)\{\textbf{if}(b)\{evt\}\}$$

where a and b are predicates and evt is an event. If the control reaches event evt then the schedule guarantees that the following condition holds right before the execution of evt:

$$a \ \wedge b$$

Guards of evt can be eliminated safely in the program if the following condition holds:

$$a \ \wedge b \ \Rightarrow \ grd(evt)$$

where $grd(evt)$ denotes evt guards. Guard propagation and elimination rules are discussed in detail in [6].

If we eliminate event guards then we are left with event actions. Since event actions are assumed to be executed simultaneously in Event-B, no ordering is assumed between them. Translation of an event to code involves sequentialisation of event actions and imposing a suitable sequencing on execution of them using sequential composition.

As explained before, since Event-B events are executed atomically, the syntactic ordering between the actions are not important. However when actions of an event are translated to a series of assignments in a programming language, the order in which they appear in the program can change the outcome. For instance, consider the following event from the model of binary search algorithm introduced earlier:

Event *search_inc*
refines *search*
where
grd1: $f(k) < v$
then
act1: $k := (k+j+1)/2$
act2: $i := k + 1$
End

If the actions of the event are translated to sequentially composed assignments in Dafny with the same order that they have in the event, the resulting program will change the state in a different way than the event. This is due to the fact that the right-hand side of action *act2* is dependent on variable k whose value is being updated by action *act1*. In this case the problem disappears if we re-order the actions since act1 is independent of variable i. However this is not a general solution since action may be mutually dependent. We can use auxiliary variables to make the right-hand side of actions independent from the left-hand side of the other actions. To do this, one auxiliary variable should be introduced for each variable that is being modified and used by the event. The auxiliary variable needs to be initialised with value of its associated variable. All the occurrences of the left-hand side variables in the right-hand side expressions of the actions should then be replaced by the auxiliary variables. For instance, the actions of the above event should be translated to the following code:

```
var aux_k := k;
k := (aux_k + j + 1) / 2;
i := aux_k + 1;
```

As can be seen in the above code, the ordering between third and fourth assignments, with the help of auxiliary variables, does not matter any more. To formulate this, assume that we have the following general event:

Event *evt*
where
  $G(v)$
then
  act1: $v_1 := E_1(v)$
  $\vdots$
  actn: $v_n := E_n(v)$
End

The definition of $SEB2DFY_{\text{alg}}(M, S)$ when S is a single event evt ($S \triangleq evt$) is as follows:

$$\begin{aligned} SEB2DFY_{\text{alg}}(M, evt) \triangleq \quad & SEB2DFY_{\text{ghost}}(M, evt) \\ & SEB2DFY_{\text{aux}}(v_1) \\ & \vdots \\ & SEB2DFY_{\text{aux}}(v_n) \\ & SEB2DFY_{\text{act}}(v_1 := E_1[v \backslash aux\_v]) \\ & \vdots \\ & SEB2DFY_{\text{act}}(v_n := E_n[v \backslash aux\_v]) \\ & SEB2DFY_{\text{post}}(M, evt) \end{aligned} \tag{9}$$

where v and aux_v are sets of model variables and auxiliary variables, respectively. $E[v \backslash aux\_v]$ is the result of substituting aux_v for all occurrences of v in

E. Functions $SEB2DFY_{\text{ghost}}$ and $SEB2DFY_{\text{post}}$ which appeared on the first and last lines of the above definition, are used for contract generation purposes which will be discussed in the next section. Function $SEB2DFY_{\text{aux}}$ receives a variable and generates an auxiliary variable declaration and initialisation:

$$SEB2DFY_{\text{aux}}(v) \triangleq \quad var\ aux\_v := v; \tag{10}$$

Function $SEB2DFY_{\text{act}}$ receives an action (a) in the form of $v := E(v)$ and has the following definition:

$$SEB2DFY_{\text{act}}(a) \triangleq \quad v := SEB2DFY_{\text{exp}}(E); \tag{11}$$

$SEB2DFY_{\text{exp}}$ transforms an Event-B expression to Dafny based on a set of translations rules for translating Event-B expressions to Dafny. Due to space limitation, we omit expression and predicate translation rules here.

4 Verification of Event Sequentialisation

In the previous section we discussed the sequentialisation of an event in detail. This section discusses how we can prove its correctness. In order to be able to verify the sequentialisation, along with the translation of the actions of each event, we generate assertions representing the expected behaviour of the program based on before-after predicate of those actions.

Before we continue to explain our approach for verifying the correctness of event sequentialisation, we justify why this step is done at Dafny level. Although it is possible to sequentialise an event in Event-B and to impose a sequential order on the execution of its actions and prove its correctness, it involves the overhead of adding a number of new events, guards, and program counters and also extending the scheduling language and refinement rules proposed in [6] to accommodate sequentialisation. Due to this, performing event sequentialisation in a programming language designed for development of sequential programs seems to be a more appropriate choice than trying to sequentialise actions in Event-B level which involves the aforementioned overhead.

Previously, we discussed how an event is translated to a number of sequential statements in Dafny. A program (or part of a program) in Dafny may be specified (annotated) using code contracts (method's pre- and post-conditions and assertions). The Dafny verifier checks an annotated program text against its specification in order to prove that the program behaves as intended. In order to prove that an event is correctly transformed to Dafny code (sequentialised correctly), a number of code contracts should be generated from the event in the form of assertions.

The way that the state is changed by an Event-B event can be expressed by a before-after predicate. By transforming an event's before-after predicate to Dafny assertions, we will be able to verify the correctness of event sequentialisation. Functions $SEB2DFY_{\text{ghost}}$ and $SEB2DFY_{\text{post}}$ will generate the necessary ghost variables and assertions for verification of sequentialisation.

To illustrate the generation of required assertions, recall event *search_inc* from the model of the binary search algorithm introduced in the previous sections:

Event *search_inc*
refines *search*
where
grd1: $f(k) < v$
then
act1: $k := (k+j+1)/2$
act2: $i := k + 1$
End

After the execution of the above event, the value of variables k and i are changed in the following way:

$$k' = (k + j + 1)/2 \ \wedge \ i' = k + 1 \tag{12}$$

where k' and i' are the value of variables k and k after the execution of the event and k and i are the value of variable k and i before the execution of *search_inc*. A block of code implements event *search_inc* correctly, if it has the same behaviour as the event, i.e. it changes the state in the same way. If we want to verify that a block of code sequentialises the event actions correctly, then we need to generate assertions like (12) in Dafny based on event before-after predicates.

The challenge here is how to refer to the before and after values (unprimed and primed variables) in an assertion in Dafny. A variable in a Dafny assertion always refers to the current value of the variable. So to be able to transform a before-after predicate like (12) to an assertion, we need to have access to the value of variables before execution of the block of code implementing the event. We transform the event *search_inc* and its before-after predicate to Dafny code and assertions using ghost variables for storing the before values (unprimed variable) of variables k and i:

```
1 ghost var old_k = k;
2 ghost var old_i = i;
3 var aux_k := k;
4 var aux_i := i;
5 k := (aux_k + j + 1) / 2;
6 i := aux_k + 1;
7 assert k == (old_k + j + 1) / 2;
8 assert i == old_i + 1;
```

Variables *old_k* and *old_i* are ghost variables and are used to keep the before value of variables k and i, respectively. A ghost variable is a variable that is used by the Dafny verifier and ignored at run time. Function $SEB2DFY_{\text{ghost}}$ facilitates the generation of the required ghost variables. It receives the model and a specific event, and works directly on the event before-after predicate and generates one ghost variable for every unprimed variable that appeared in the event before-after predicate and initialises it with the value of the unprimed

variables. For practical reasons, we decided to generate one `assert` statement per each action. The assertions are yielded by replacing unprimed variables in the before-after predicate with their ghost counterparts and primed variables with the original variables. Function $SEB2DFY_{\text{post}}$ generates assertions required for verification.

In the above code, lines 1–2 and 7–8 are required for verification only. Lines 3–6 are the code implementing the event. The guard of the event is not used here. This is because the guard is available to the Dafny verifier since it would be a condition in the `if` statement generated based on the schedule given in Sect. 2.2. Any code that can satisfy the assertions (lines 7–8) can replace lines 3–6.

Apart from the verification of sequentialisation, having assertions in the generated code is useful for another purpose as well. The embedded assertions make it possible to verify further amendments to the implementation at the code level to prove that the new code complies with its abstract specification (event). For instance in the above code, any implementation that satisfies the assertions (lines 7–8) can replace the code implementing the abstract event (lines 3–6).

5 Conclusion and Future Work

In this paper, we proposed an approach for generation of verifiable implementation from Scheduled Event-B [6] models. The paper outlined the necessary rules for translating the algorithmic structure of a model together with rules for sequentialisation of Event-B atomic events in Dafny. We also introduced a way for generating Dafny assertions that allows us to verify the correctness of the sequentialisation phase. Overall, our approach benefits from combining the verification power of Dafny together with abstraction and refinement offered by Event-B. We have applied this approach to a number of examples, including the model of Schorr-Waite algorithm introduced in [6] and generated code and contracts required for verification.

In our previous work [7], we introduced another approach for generating Dafny code contracts from Event-B models. The proposed approach generates Dafny method pre- and post-conditions from a group of atomic Event-B events in a way that any implementation that satisfies the generated pre- and post-conditions is considered to be a correct implementation of the Event-B abstract model. There are two main differences between the approach presented in this paper and the one presented in our previous work. First, the previous approach only focuses on contract (method's pre- and post-conditions) generation and no implementation is generated while in this work, both implementation and code contracts (i.e. assertions) are generated. The second difference is the granularity of generated contracts. In the previous work we generated contracts at method level in a way that the overall behaviour of the method was annotated, while in this work, the contracts are generated in a lower granularity where the behaviour of small blocks of code (sequentially composed assignments) inside a method are annotated.

In [10], we extended Event-B code generation tool [9] and applied it to an Event-B model of a learning-based RTM (Runtime Management system) in embedded system design to generate C implementation from the model. The code generation tool supports portability of the platform-independent model from which platform-specific implementations are automatically generated. However, our experience shows that there are a number of limitations with the current Event-B code generation tool. The first limitation is that the algorithmic structure of the program can only be introduced at the final level of refinement and the modeller cannot benefit from refinement in derivation of algorithmic structure. The other limitation is that the current structuring language is too restrictive and does not allow the modeller to define nested programs. Another limitation is that the verification is only done at the Event-B level and no verification is performed on the generated code.

In future, we want to mechanise the process of generation of the code and contracts from scheduled Event-B models. We also envisage to apply the approach presented in this paper to other case studies including the Event-B model of RTM introduced in [10] to further validate our approach.

Acknowledgments. This work was funded in part by the EPSRC PRiME Project (EP/K034448/1), www.prime-project.org.

References

1. Abrial, J.-R., Lee, M.K.O., Neilson, D.S., Scharbach, P.N., Sørensen, I.H.: The B-method. In: Prehn, S., Toetenel, H. (eds.) VDM 1991. LNCS, vol. 552, pp. 398–405. Springer, Heidelberg (1991). https://doi.org/10.1007/BFb0020001
2. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
3. Abrial, J.-R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. Int. J. Softw. Tools Technol. Transf. **12**(6), 447–466 (2010)
4. Back, R.J.R., Kurki-Suonio, F.: Distributed cooperation with action systems. ACM Trans. Program. Lang. Syst. **10**(4), 513–554 (1988)
5. Butler, M.: Mastering system analysis and design through abstraction and refinement (2013)
6. Dalvandi, M., Butler, M., Rezazadeh, A.: Derivation of algorithmic control structures in Event-B refinement. Sci. Comput. Program. **148**(Suppl. C), 49–65 (2017). Special Issue on Automated Verification of Critical Systems (AVoCS 2015)
7. Dalvandi, M., Butler, M.J., Rezazadeh, A.: Transforming Event-B models to Dafny contracts. In: ECEASST, vol. 72 (2015)
8. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
9. Edmunds, A., Butler, M.: Tasking Event-B: an extension to Event-B for generating concurrent code. Event Dates: 2nd April 2011, February 2011
10. Fathabadi, A.S., Butler, M.J., Yang, S., Maeda-Nunez, L.A., Bantock, J., Al-Hashimi, B.M., Merrett, G.V.: A model-based framework for software portability and verification in embedded power management systems. J. Syst. Architect. **82**, 12–23 (2018)

11. Hallerstede, S.: On the purpose of Event-B proof obligations. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 125–138. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87603-8_11
12. Leino, K.R.M.: This is Boogie 2. Manuscript KRML 178:131 (2008)
13. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
14. Leino, K.R.M., Monahan, R.: Dafny meets the verification benchmarks challenge. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 112–126. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15057-9_8
15. Malcolm, G., Goguen, J.A.: Proving correctness of refinement and implementation. Computing Laboratory, Programming Research Group, Oxford University (1994)
16. Meyer, B.: Eiffel: The Language. Prentice-Hall, Inc., Upper Saddle River (1992)
17. Meyer, B.: Design by Contract. Prentice Hall, Inc., Upper Saddle River (2002)
18. Wirth, N.: Extended Backus-Naur Form (EBNF). ISO/IEC 14977:2996 (1996)

Hybrid ERTMS Case Study

The Hybrid ERTMS/ETCS Level 3 Case Study

Thai Son Hoang[1], Michael Butler[1(✉)], and Klaus Reichl[2]

[1] ECS, University of Southampton, Southampton, UK
{t.s.hoang,mjb}@ecs.soton.ac.uk
[2] Thales Austria GmbH, Vienna, Austria
klaus.reichl@thalesgroup.com

Abstract. This document presents a description of the European Rail Traffic Management System (ERTMS) case study. ERTMS is a system of standards for management and interoperation of signalling for railways by the European Union (EU). The case study focuses on the *ERTMS Level 3 Hybrid* principle, which accommodates different types of trains including ERTMS trains equiped with the Train Integrity Monitoring System (TIMS), ERTMS trains without TIMS, and non-ERTMS trains.

Keywords: ERTMS · ETCS · Level 3 Hybrid

1 Introduction

The case study concerns the European Rail Traffic Management System (ERTMS)[1], the system of standards for management and interoperation of signalling for railways by the European Union (EU)[2]. The aim of ERTMS is to replace the different national train control and command systems in Europe with a seamless European railway system. The advantages of ERTMS include increased capacity, higher reliability rates, improved safety, and open supply market.

There are three signaling levels for ERTMS[3].

Level 1. Communication between trains and trackside equipment by means of transponders called Euro-balises. Trackside equipment is needed for detecting train location and train integrity[4] and lineside signals are required.

Level 2. Communication between trains and trackside equipment is provided by the Global System for Mobile Communications - Railway (GSM-R). Trackside equipment is needed for determining train location and integrity while lineside signals are optional.

[1] http://ertms.net.
[2] http://en.wikipedia.org/wiki/European_Rail_Traffic_Management_System.
[3] http://ec.europa.eu/transport/modes/rail/ertms/what-is-ertms/levels_and_modes_en.
[4] Train integrity means the train is complete and has not been accidentally split.

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 251–261, 2018.
https://doi.org/10.1007/978-3-319-91271-4_17

Level 3. The train determines its location using fixed positional transponders and supervises its integrity using the on-board Train Integrity Monitoring System (TIMS). This means that trackside detection equipment is not required.

There are different options depending on levels of maturity in terms of definition and development, leading to several ERTMS *Level 3* types. Our case study focuses on *Level 3 Hybrid* which is the most mature and is developed using existing technology solution augmented for optimisation [3].

Abbreviations. Figure 1 shows the list of abbreviations used in this document. A more complete glossary of terms and abbreviations referenced here can be found in [2].

EoA	End of Authority
ERTMS	European Rail Traffic Management System
EU	European Union
GSM-R	Global System for Mobile Communications - Railway
MA	Movement Autority
TIMS	Train Integrity Monitoring System
TTD	Trackside Train Detection
VSS	Virtual Sub-Section

Fig. 1. List of abbreviations

Requirements Taxonomy. In this document, we use ASM to indicate an *assumption* and REQ to indicate a *requirement* of the system. The list of requirements in this document is intended to provide a high level view of the system and does not cover all system details. We refer the reader to [1] for the detailed principles of the system under consideration.

Structure. The rest of this document is as follows. Section 2 gives an overview of the system. Section 3 presents a more detailed description of various aspects of the system under consideration. We briefly review the state machine for the Virtual Sub-Section (VSS), the key idea for the ERTMS Level 3 Hybrid principle, in Sect. 4. Section 5 gives a short conclusion on our expectation for the case study.

2 System Overview

It is expensive and challenging to fit trains with ERTMS and the Train Integrity Monitoring System (TIMS) so *Level 3 Hybrid* copes with different train configurations (TIMS-equipped, ERTMS without TIMS, and non-ERTMS). *Level 3 Hybrid* uses a limited amount of trackside detection. In the case of TIMS-equipped trains, the capacity of the line can be increased using *fixed virtual blocks*. In order to achieve this purpose, each Trackside Train Detection (TTD)

is divided into several VSSes. The scope of the case study is the management of the VSSes (more detailed specification is in [1]). We will not consider the interlocking system, e.g., how train routes are set and unset. More specifically, we can consider that the trains travel on a straight line and in the same direction.

ASM 1	The trains travel along a *straight line* track and in the *same direction*.

ASM 2	The train track is partitioned into several fixed TTD sections.

ASM 3	Each TTD is partitioned into one or more fixed VSS.

The overview of the relevant part of the system can be seen in Fig. 2. The trackside has a sub-system for managing the VSS, which communicates the VSS status information to the Movement Authority (MA) authorisation sub-system. The MA authorisation sub-system sends information related to the MAs to the trains and also informs the VSS management sub-system about the issued MAs. In order to decide the VSS status, the VSS management sub-system receives the TTD status from the interlocking system and the position reports from the trains (depending on the trains' type).

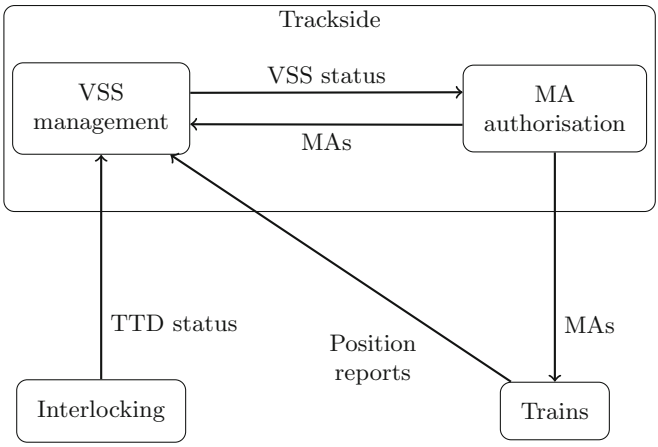


Fig. 2. System overview

We describe in more detail the various aspects of the system in the next section.

3 Level 3 Hybrid with Fixed Virtual Blocks

3.1 TTD Sections and VSSes

We consider the TTD information as reliable and safe. In particular, a TTD section is reported as free only if there are no trains or no part of a train located on the TTD. Subsequently, the VSS on a free TTD can be regarded as "free".

ASM 4	A TTD can be reported as "free" or "occupied"

ASM 5	A TTD is reported as *free* if and only if there are no trains or a part of a train located on the TTD.

Due to the discrepancy of the timing and spatial information of the trackside detection, two additional (internal) statuses of VSS are specified: "ambiguous" and "unknown". Status "ambiguous" indicates that a train is present but its status is not known, whereas status "unknown" indicates that the occupancy sub-section is not proven.

REQ 6	A VSS can have one of the following statuses: "free", "occupied", "ambiguous", or "unknown"

REQ 7	A VSS is *free* when there are no trains or no part of a train located on the VSS.

REQ 8	A VSS is *occupied* if there is exactly one train or a part of a train located on the VSS.

REQ 9	A VSS is *ambiguous* if there is a train occupying the VSS but its status is not known.

REQ 10	A VSS is *unknown* if the occupancy of the VSS is not proven.

3.2 Types of Trains

Depending on the train's equipment, the status of a VSS is computed differently based on the train position information and the TTD information:

- A TIMS-equipped ERTMS train (an *integer* train) precisely occupies the relevant VSS in which it is located.
- An ERTMS train not fitted with TIMS also occupies the sections in the rear (until the end of the trackside detection section).
- A non-ERTMS train occupies the whole TTD section.

As a result, a non-TIMS train can follow an integer train on VSS sections, but other trains can only follow it on a separate trackside detection section. Capacity gain for *Level 3 Hybrid* can be achieved only for ERTMS trains and full gain is achieved only for TIMS-fitted trains.

REQ 11	The system should accomodate three types of trains: TIMS-equipped ERTMS, ERTMS not fitted with TIMS, and non-ERTMS.

REQ 12	A TIMS-fitted ERTMS train occupies the relevant VSSes that it is located on.

REQ 13	An ERTMS train without TIMS occupies the relevant VSSes that it is located on, and also all the VSSes in the rear until the end of the TTD section.

REQ 14	A non-ERTMS train occupies the whole TTD section that it is located on.

The status of a VSS is computed based on the TTD status and the train position reports.

3.3 Movement Authority

We will not need to consider *how* the MAs of the trains are computed or how they are related to routes. (A route is a contiguous sequence of connected sections.) The MA of a train defines (beside other information) a position on the track, called the End of Authority (EoA), which must not be passed by the train. Depending on the type of a train and its location within the track, the EoA can be defined in terms of a VSS or of the trackside sections. However, since VSS status depends on a train's MA, we will need to consider what has been set as the train MA with the assumption that the trains will be safe from collision if they respect the provided MAs. For the purpose of issuing MAs, only "free" state of VSSes is required to be distinguished from the other states, i.e., "occupied", "ambiguous", or "unknown" (which will be treated as "occupied").

ASM 15	For non-ERTMS trains, their EoAs are defined in terms of TTD sections.

ASM 16	For ERTMS trains, their EoAs are defined in terms of the VSSes.

ASM 17	The MAs are disjoint, i.e., trains will be safe from collision if they respect the provided MAs.

3.4 Timers

A timer can have one or more *start events* and zero or more *stop events*. Any start/stop event of a timer will start/stop the corresponding timer. A timer without a stop event once started will run until it is expired. Once expired, this timer will stay in the same state until it is reset when the start condition is met again.

REQ 18	A timer has one or more start events.

REQ 19	A timer has zero or more stop events.

REQ 20	A timer without a stop event once started will run until expired and stay in the "expired" state until reset when the start condition is met again.

There are two main types of timers implemented in the trackside, namely, *waiting* timers and *propagation* timers. The waiting timers are to avoid unnecessary changes of VSS status due to the delay in communication of train position, train integrity information, etc. The propagation timers are to avoid unnecessary propagation of the "unknown" state to the VSS sections with no immediate risk of having a train or a part of a train located on them. We describe some of the important timers here. The complete list of the timers is in [1, Sect. 3.4].

Mute timers. A waiting timer called "mute timer" is assigned to each train. Each *mute timer* runs continually and whenever some information is received from the train, the timer is reset. This timer is used to decide if communication between the trackside and the train is lost.

REQ 21	A *mute timer* is assigned to each train.

REQ 22	Each mute timer runs continually.

REQ 23	A mute timer is reset whenever some information is received from the train.

Wait integrity timers. A waiting timer called a "wait integrity timer" is assigned to each train. Each *wait integrity timer* runs continually and whenever integrity confirmation is received from the train and no change of train length has been reported since the previous position report, the timer is reset. This timer is used to decide if the train has lost integrity.

REQ 24	A *wait integrity timer* is assigned to each train.

REQ 25	Each wait integrity timer runs continually.

REQ 26	A wait integrity timer is reset whenever integrity confirmation is received from the train and no change of train length has been reported since the previous position report.

Disconnected propagation timers. A "disconnected propagation timer" is assigned to each VSS. The start event for a "disconnected propagation timer" is that the "mute timer" of a train located on the VSS expired. The stop event for this timer is when the connection of the train is reestablished. This timer is used to propagate the "unknown" status of VSS due to train disconnection.

REQ 27	A *disconnected propagation timer* is assigned to each VSS.

REQ 28	The start event of a disconnected propagation timer is when the mute timer of a train located on the VSS expires.

REQ 29	The stop event of a disconnected propagation timer is when connection of the train is restored.

Ghost train propagation timers. A "ghost train propagation timer" is assigned to each TTD. The start event for a "ghost train propagation timer" is either (1) the TTD become "occupied" without any train on it or (2) the TTD become "occupied" without any MA associated with it. There is no stop event for this timer. This timer is used to propagate the "unknown" status of VSS due to ghost trains (see Sect. 3.5).

REQ 30	A *ghost train propagation timer* is assigned to each TTD.

REQ 31	The start event of a ghost train propagation timer is when the TTD becomes "occupied" without any train or MA associated with it.

REQ 32	There is no stop event for a ghost train propagation timer.

3.5 Ghost Trains and Shadow Trains

In some situation, objects might be detected by the TTD but are unknown to the trackside system (this could due to some physical objects occupied the track or some virtual objects due to trackside failure). They are called ghost trains. For example, when a train is split, the rear part will become a ghost train. When a ghost train is following a normally operated Level 3 train (i.e., an integer train), it is called a *shadow train*.

REQ 33	Ghost trains are objects detected by the TTD but are unknown to the trackside.

REQ 34	A ghost train following an integer train is called a shadow train.

To protect the system against ghost trains, the VSS status "unknown" is used and propagated according to the "ghost train propagation timer" (see [1, Sect. 4.2.2]). To protect the system against a shadow train hazard, the VSS status "ambiguous" is used (more information is in [1, Sect. 4.5]).

3.6 Train Connectivity

The communication between the trackside and a train is considered to be lost when the mute timer for the train expires. When the train is disconnected from the trackside, the VSS sections within the train's MA up to either the limit of the first free TTD or the first VSS of the MA are set to"unknown" (they are propagated according to the "disconnected propagation timer"). A disconnected train can reconnect, i.e., the trackside receives a position report from the train after its mute timer has expired. In this case, the status of different VSSes are updated depending on whether they are occupied by the train or in the front of the train or in the rear of the train. Also, the updated VSS status will depend on whether or not the train confirms its integrity with no change in its length. In any situation, the unknown VSSes in rear of the train would become "free" if the TTD section is released. More information is in [1, Sects. 3.8 and 4.2.1]

REQ 35	The communication between the trackside and a train is considered to be lost when the mute timer for the train expires.

REQ 36	When the trackside receives a position report from a disconnected train, the communication between the trackside and the train is reestablished

4 The State Machine for VSS

For a VSS, its state machine can be seen in Fig. 3. Depending on the situation, the status of a VSS can be changed between any two of the four states, i.e., "free", "unknown", "ambiguous", "occupied". Extensive details of the transitions can be found in [1, Sect. 5] and are not repeated here. In particular, for each transition, there are several situations where the VSS status is changed according to the transition.

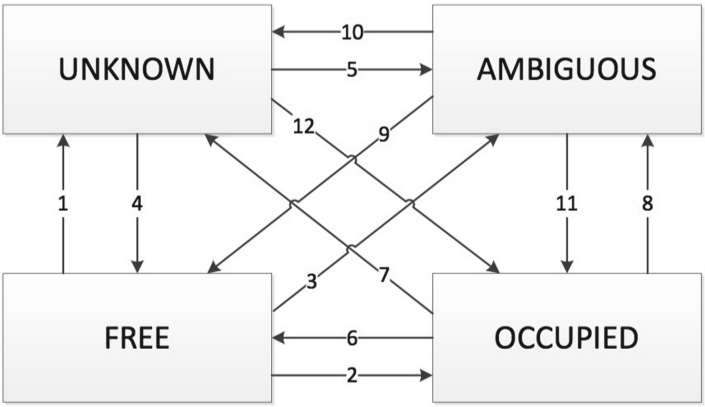


Fig. 3. The state machine of a VSS [1]

5 Conclusion

We have given an overview of the ERTMS Level 3 Hybrid principles. We are looking for solutions that address the various challenges of the case study, and also provide insights into the case study and/or the formal methods used. For the case study, we expect the solutions will illustrate what can be guaranteed by the system (e.g., in terms of collision-free), and/or explanation about various hazard-mitigating mechanisms of the system. Regarding formal methods, we expect to see a justification of the "need" and the "value" of the methods and/or tools in addressing a complex industrial challenge.

Acknowledgements. The organisers would like to thank the EEIG ERTMS Users Group (EUG) for the Principles on "Hybrid ERTMS/ETCS Level 3" document [1] released on 14/07/2017.

References

1. EEIG ERTMS Users Group, Brussels, Belgium. Hybrid ERTMS/ETCS Level 3: Principles, July 2017. Ref. 16E042 Version 1A
2. ERA, UNISIG, EEIG ERTMS Users Group. Glossary of Terms and Abbreviations: ERTMS/ETCS, 3.3.0 edition, May 2016. http://www.era.europa.eu/Document-Register/Documents/SUBSET-023%20v330.pdf
3. Furness, N., van Houten, H., Arenas, L., Bartholomeus, M.: ERTMS level 3: the game-changer. IRSE News View 232, April 2017

Modeling the Hybrid ERTMS/ETCS Level 3 Standard Using a Formal Requirements Engineering Approach

Steve Jeffrey Tueno Fotso[1,2(✉)], Marc Frappier[1], Régine Laleau[2], and Amel Mammar[3]

[1] Université de Sherbrooke, GRIL, Sherbrooke, Québec, Canada
{Steve.Jeffrey.Tueno.Fotso,Marc.Frappier}@USherbrooke.ca
[2] Université Paris-Est Créteil, LACL, Créteil, France
laleau@u-pec.fr
[3] Télécom SudParis, SAMOVAR-CNRS, Evry, France
amel.mammar@telecom-sudparis.eu

Abstract. This paper presents a specification of the hybrid ERTMS/ETCS level 3 standard in the framework of the case study proposed for the 6th edition of the ABZ conference. The specification is based on the method and tools, developed in the *ANR FORMOSE* project, for the modeling and formal verification of critical and complex system requirements. The requirements are specified with *SysML/KAOS* goal diagrams and are automatically translated into *B System* specifications, in order to obtain the architecture of the formal specification. Domain properties are specified by ontologies with the SysML/KAOS domain modeling language, based on *OWL* and *PLIB*. Their automatic translation completes the structural part of the formal specification. The only part of the specification, which must be manually completed, is the body of events. The construction is incremental, based on the refinement mechanisms existing within the involved methods. The formal specification of the case study is composed of seven refinement levels and all the proofs have been discharged with the Rodin prover.

Keywords: Requirements engineering · Goal diagrams
Domain modeling · Ontologies · *SysML/KAOS* · *B System*

1 Introduction

In this paper, we are interested in using the *FORMOSE* approach [2] on the case study proposed for the 6th edition of the ABZ conference [7]. This case study deals with the specification of the *hybrid ERTMS/ETCS level 3* standard [5,14]. The case study is described in two main documents. The first one, [7], describes the hybrid ERTMS/ETCS level 3 protocol in a general way and restricts the scope of the study. The second one, [5], offers a technical and detailed description of the protocol specification. It provides the safety requirements that the system

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 262–276, 2018.
https://doi.org/10.1007/978-3-319-91271-4_18

must guarantee. The FORMOSE method includes the *SysML/KAOS* requirements engineering language [6,11] for modeling requirements with goal diagrams. Domain properties are modeled with ontologies, using the SysML/KAOS domain modeling language [20,22]. Once done, translation rules [13,21,23], supported by tools [13,23], allow the automatic generation of the *B system* specification. The goal diagrams give the set of *B System* components, each goal gives an event. As the refinement links defined between these components have to represent the SysML/KAOS refinements, new proof obligations are generated. The domain model gives the structural part of the specification. It consists of variables, constrained by an invariant, and constants, constrained by properties. The *Rodin* tool [3] has been used to support the verification and the validation of the *B System* specification, especially to prove the safety invariants and the refinement logic. The complete specification can be found in [24]. Compared to direct specification approaches using only plain *Event-B* such as [10,12], the FORMOSE method provides a more structured and methodological process to the formal specification of the system. It allows a decoupling between formal specification handling difficulties and system modeling, a better reusability and readability of models, and a strong traceability between the system formal specification and SysML/KAOS models, which are abstractions of the system and domain descriptions.

The remainder of this paper is structured as follows: Sect. 2 briefly describes the *B System* formal method, the SysML/KAOS goal and domain modeling languages and the rules for obtaining the *B System* specifications. Follows a presentation, in Sect. 3, of the work done on the case study and in Sect. 4, of the discussion related to it. The discussion includes a short comparison with a companion paper on the same case study, but specified using only plain *Event-B*. Finally, Sect. 5 reports our conclusions.

2 Context

2.1 B System

Event-B is an industrial-strength formal method for *system modeling* [1]. It is used to incrementally construct a system specification, using refinement, and to prove properties. Proof obligations are defined to prove invariant preservation by events (invariant has to be true at any system state), event feasibility, convergence and machine refinement [1]. *B System* is an *Event-B* syntactic variant proposed by *ClearSy*, an industrial partner in the *FORMOSE* project [2], and supported by *Atelier B* [4]. A *B System* specification consists of components. Each component can be either a system or a refinement and it may define static or dynamic elements. Constants, abstract and enumerated sets, and their properties, constitute the static part. The dynamic part includes the representation of the system state using variables constrained through invariants and updated through events. Each event has a ***guard*** and an ***action***. The *guard* is a condition that must be satisfied for the event to be triggered and the *action* describes the update of state variables. Although it is advisable to always isolate the static

and dynamic parts of the *B System* formal model, it is possible to define the two parts within the same component. In the following sections, our *B System* models will be presented using this facility.

2.2 SysML/KAOS Goal Modeling

SysML/KAOS [6,11] is a requirements engineering method which extends the *SysML* UML profile with a set of concepts from KAOS [8] to represent functional and non-functional requirements. It combines the traceability features provided by *SysML* with goal expressiveness provided by *KAOS*. SysML/KAOS goal models allow the representation of requirements to be satisfied by the system and of expectations with regard to the environment through a goal hierarchy. The hierarchy is built through a succession of refinements using different operators: ***AND***, ***OR*** and ***MILESTONE***. An ***AND*** *refinement* decomposes a goal into subgoals, and all of them must be achieved to realise the parent goal. An ***OR*** *refinement* decomposes a goal into subgoals such that the achievement of only one of them is sufficient for the accomplishment of the parent goal. A ***MILESTONE* refinement** is a variant of the AND refinement which allows the definition of an achievement order between goals. A SysML/KAOS goal can be *functional* or *non-functional*. The scope of this document is limited to functional goals. A functional goal describes the *expected behaviour* of the system once a certain condition holds [11]: *[if* ***CurrentCondition*** *then] sooner-or-later* ***TargetCondition***. SysML/KAOS allows the definition of a functional goal without specifying a current condition. In this case, the expected behaviour can be observed from any system state.

2.3 Formalisation of SysML/KAOS Goal Models

The formalisation of SysML/KAOS goal models is the focus of the work done by [13]. The proposed rules allow the generation of a formal model whose structure reflects the hierarchy of the SysML/KAOS goal diagram: one component is associated with each hierarchy level; this component defines one event for each goal. The semantics of refinement links between goals is expressed in the formal specification with a set of proof obligations which complement the standard proof obligations for *invariant preservation* and for *event actions feasibility* [1]. It could also have been expressed using control variables. However, control variables make the formal specification less readable and maintainable. In addition, each action of a formal event that updates a control variable generates new proof obligations, which complicates the formal verification process. Regarding the new proof obligations, they depend on the goal refinement operator used. For an abstract goal G and two concrete goals G_1 and G_2:[1]

[1] For an event `G`, `G_Guard` represents the guards of G and `G_Post` represents the post condition of its actions.

- For the *AND* operator, the proof obligations are
  - $G_1\_Guard \Rightarrow G\_Guard$
  - $G_2\_Guard \Rightarrow G\_Guard$
  - $(G_1\_Post \wedge G_2\_Post) \Rightarrow G\_Post$
- For the *OR* operator, they are
  - $G_1\_Guard \Rightarrow G\_Guard$
  - $G_2\_Guard \Rightarrow G\_Guard$
  - $G_1\_Post \Rightarrow G\_Post$
  - $G_2\_Post \Rightarrow G\_Post$
  - $G_1\_Post \Rightarrow \neg G_2\_Guard$
  - $G_2\_Post \Rightarrow \neg G_1\_Guard$
- For the *MILESTONE* operator, they are
  - $G_1\_Guard \Rightarrow G\_Guard$
  - $G_2\_Post \Rightarrow G\_Post$
  - $\Box(G1\_Post \Rightarrow \Diamond G2\_Guard)$ (each system state, corresponding to the post condition of G_1, must be followed, at least once in the future, by a system state enabling G_2)

Nevertheless, the generated specification does not contain the system structure, composed of variables, constrained by an invariant, and constants, constrained by properties.

2.4 SysML/KAOS Domain Modeling

The SysML/KAOS domain modeling language [20,22] uses ontologies to represent domain models. It is based on *OWL* [18] and *PLIB* [16], two well-known ontology modeling languages. Each domain model corresponds to a refinement level in the SysML/KAOS goal model. The *parent* association represents the hierarchy of domain models. A domain model can define multiple elements. For this case study, a domain model can define concepts, attributes, datasets and predicates. A concept represents a collection of individuals with common properties. It can be declared variable ($isVariable = TRUE$) when the set of its individuals can be dynamically updated by adding or deleting individuals. Otherwise, it is constant ($isVariable = FALSE$). A data set represents a collection of data values. An attribute captures links between concepts and data sets. It can be variable or constant, functional or total. A predicate expresses constraints between domain model elements, using the first order logic. Each predicate has a body which represents its antecedent and a head which represents its consequent. The head can be omitted if it is always TRUE. Gluing invariants represent links between variables defined within a domain model and those appearing in more abstract domain models. They are extremely important because they capture relationships between abstract and concrete data during refinement and are used to discharge proof obligations.

2.5 From SysML/KAOS Domain Models to B System Specifications

The translation rules are fully described in [21]. They allow the extraction of the structural part of the system formal specification from domain models. Table 1 represents some rules that are relevant for our purposes. It should be noted that o_x designates the result of the translation of x and that *abstract* is used for "without parent".

Table 1. Summary of the translation rules

	Domain model		B System	
	Element	Constraint	Element	Constraint
Abstract domain model	DM	$DM \in$ DomainModel	o_DM	$o\_DM \in$ System
		DM is not associated to a parent domain model		
Domain model with parent	DM	$\{DM, PDM\} \subseteq$ DomainModel	o_DM	$o\_DM \in$ Refinement
	PDM	DM is associated to PDM through the *parent* association		o_DM refines o_PDM
		PDM has already been translated		
Abstract concept	CO	$CO \in$ Concept	o_CO	$o\_CO \in$ AbstractSet
		CO is not associated to a parent concept		
Attribute	AT CO	$CO \in$ Concept	o_AT	**IF** the *is Variable* property of AT is set to $FALSE$
	DS	$DS \in$ DataSet		
		$AT \in$ Attribute		**THEN** $o\_AT \in$ Constant
		CO is the *domain* of AT		**ELSE** $o\_AT \in$ Variable
		DS is the *range* of AT		**END**
		CO and DS have already been translated		$o\_AT \in o\_CO \leftrightarrow o\_DS$[1]
Data value	Dva DS	$Dva \in$ DataValue $DS \in$ DataSet	o_Dva	$o\_Dva \in$ Constant
		Dva is a value of DS		$o\_Dva \in o\_DS$
		DS has already been translated		

[1] Depending on attribute properties, this relation may become a partial or total function.

3 Specification of the Hybrid ERTMS/ETCS Level 3 Standard

3.1 Main Characteristics of the Standard

The Hybrid ERTMS/ETCS level 3 protocol (HEEL3) has been proposed to optimize the use and occupation of railways [5,7,14]. It thus proposes the division of the track into separate entities, each named Trackside Train Detection (TTD). In addition, each TTD is subdivided into sub-entities called Virtual Sub-Sections (VSS). A TTD has two possible states: *free* and *occupied* with a safety invariant stating that if a train is located on a TTD, then the state of the TTD must be set to *occupied*. In addition to these two states, a VSS may have the *unknown* or the *ambiguous* state. The *ambiguous* state is used when the information available to the system suggest that two trains are potentially present on the VSS. The *unknown* state is used when the system can guarantee neither the presence nor the absence of a train on the VSS. For an optimal safety, Movement Authorities (MA) are evaluated and assigned to each connected train. The MA of a train designates a portion of the track on which it is guaranteed to move safely. ERTMS (European Rail Traffic Management System) designates a protocol and a set of tools that allow a train to know and report its position. Similarly, TIMS (Train Integrity Monitoring System) designates the component that allows a train to know and report its integrity and its size. HEEL3 considers trains equipped with TIMS (TIMS trains), which can report themselves as *integer* or not; trains equipped with ERTMS (ERTMS trains), which can report their position (connected trains) or not (unconnected trains); and finally, trains that are equipped neither with a ERTMS nor with a TIMS (unconnected trains).

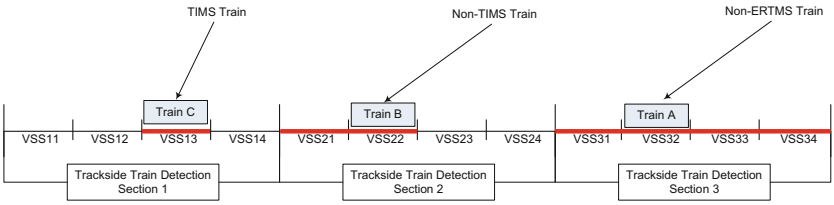


Fig. 1. Overview of the dependence between the capacity exploitation and the presence of ERTMS and TIMS [14]

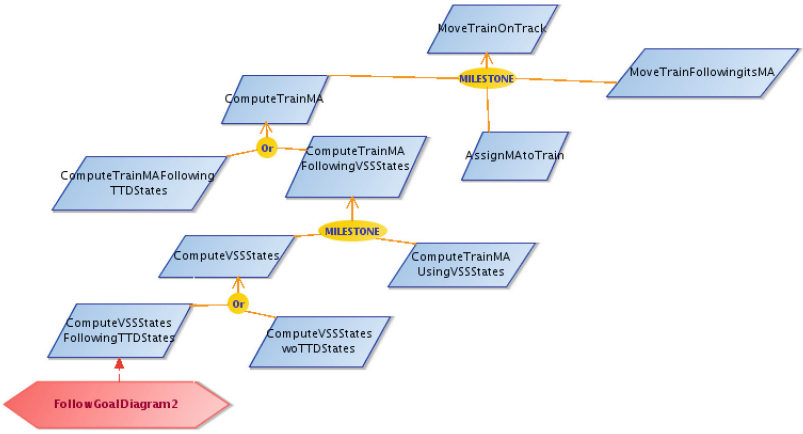


Fig. 2. The SysML/KAOS goal diagram

Figure 1 is an overview of the influence of the presence of ERTMS and TIMS on the track capacity exploitation [14], considering trains that behave normally. A TIMS train, is considered to occupy a whole VSS. A non-TIMS train (that is ERTMS) is considered to occupy all the VSSs from its front to the rear end of the TTD section where it is located. Finally, a non-ERTMS train (unconnected train) is considered to occupy the whole TTD section where the system guesses it is.

3.2 The Goal Diagram

The SysML/KAOS requirements engineering method allows the progressive construction of system requirements from refinements of stakeholder needs. Thus, even if the management of VSSs is the purpose of the case study, we need to put it into perspective with more abstract objectives that will explain what VSSs are useful for. Figure 2 is an excerpt from the SysML/KAOS functional goal diagram focused on the main system purpose: move trains on the track (`MoveTrainOnTrack`). To achieve it, the system must ensures that the train has a valid MA (`ComputeTrainMA`). If the MA has been recomputed, then the system must assign the new MA to the train (`AssignMAtoTrain`). Finally, the train

has to move following its assigned MA (`MoveTrainFollowingItsMA`). The second refinement level of the SysML/KAOS goal diagram focuses on the informations needed to determine the MA of a train: the MA computation can be based only on TTD states (`ComputeTrainMAFollowingTTDStates`) or following VSS states (`ComputeTrainMAFollowingVSSStates`) [5]. When the computation is only based on TTD states, it corresponds to the *ERTMS/ETCS Level 2* protocol. When VSS states are involved, it corresponds to the *ERTMS/ETCS Level 3* protocol. The MA computation based on VSS states requires the update of the states of VSSs (`ComputeVSSStates`) and the computation of the MA (`ComputeTrainMAUsingVSSStates`). Finally, depending on the type of the ERTMS/ETCS level 3 implementation, it is possible to use or not the TTD states when computing the VSS states (Table 1 of [14]). If TTD states are not required (*virtual (without train detection)* level 3 type), it corresponds to `ComputeVSSStateswoTTDStates`, with the disadvantage of only allowing the circulation of trains equipped with TIMS. If TTD states are used (*hybrid* level 3 type), it corresponds to `ComputeVSSStatesFollowingTTDStates`.

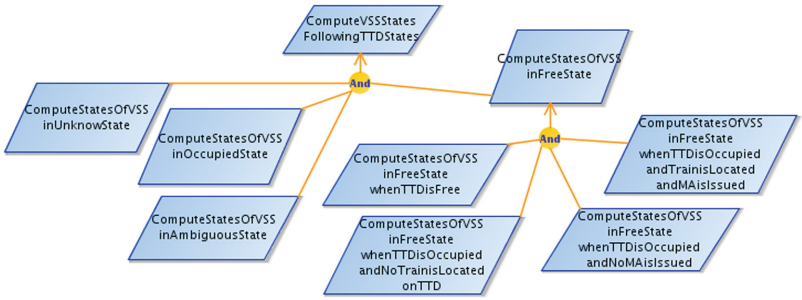


Fig. 3. SysML/KAOS goal diagram of the VSS state computation purposes

Figure 3 is an excerpt from the SysML/KAOS functional goal diagram focused on the purpose of VSS state computation with the use of TTD states (`ComputeVSSStatesFollowingTTDStates`). The computation of the current VSS states can be split into the determination of the current states of VSSs previously in the unknown state (`ComputeStatesOfVSSinUnknownState`), in the occupied state (`ComputeStatesOfVSSinOccupiedState`), in the ambiguous state (`ComputeStatesOfVSSinAmbiguousState`) and in the free state (`ComputeStatesOfVSSinFreeState`) (Fig. 7 of [5]). The last refinement level is focused on VSSs previously in the free state. Its goals come from the requirements of the transition *#1A* of Table 2 of [5]. When the TTD is free, then the VSSs remain free (`ComputeStatesOfVSSinFreeStateWhenTTDisFree`). When the TTD is occupied and no train is located on it or no MA is issued, then the VSSs move in the unknown state (`ComputeStatesOfVSSinFreeStateWhenTTDisOccupiedandNoTrainisLocatedonTTD`, `ComputeStatesOfVSSinFreeStateWhen`

`TTDisOccupiedandNoMAisIssued`). The other transitions are the purpose of `ComputeStatesOfVSSinFreeStateWhenTTDisOccupiedandTrainisLocatedandMAisIssued`.

The rest of this section consists of a presentation of the SysML/KAOS domain models associated with the most relevant refinement levels of the goal diagrams and of a description of the *B System* specifications obtained from goals and ontologies. From the goal model, we distinguish seven refinement levels which are translated into seven *B System* components. The formal specification has been verified using *Rodin* [3], an industrial-strength tool supporting the *Event-B* method [1]. We have in particular discharged all the proof obligations associated with the safety invariants that we have identified and with the SysML/KAOS refinement operators that appear in the goal diagram. For the sake of concision, we will present here only the first three refinement levels. The full specification can be found in [24].

3.3 The Root Level

Figure 4 represents the domain model associated with the top most goal `MoveTrainOnTrack` of the diagram of Fig. 2. It represents the entities needed for the specification of the movement of a train on the track and their characteristics. For instance, the concept `TRAIN` models the set of trains. The attribute `connectedTrain` models the subset of `TRAIN` that broadcast their location at least once and for each, the current connection status. The attribute `front` models the estimated position of the front of each connected train. For each connected train equipped with a TIMS, the attribute `rear` models the estimated

```
domain model ertms_etcs_case_study {
        concepts:
                concept TRAIN is variable: false
        attributes:
                attribute connectedTrain domain: Train range: BOOL {
                        is variable: true  is functional: true   is total: false
                }
                attribute front domain: dom(connectedTrain) range: TRACK {
                        is variable: true  is functional: true   is total: true
                }
                attribute rear domain: dom(connectedTrain) range: TRACK {
                        is variable: true  is functional: true   is total: false
                }
        data sets:
                custom data set TRACK
        data values:
                data value a type: NATURAL
                data value b type: NATURAL
        predicates:
                p0.1: a<b                              p0.2: TRACK=a..b
                p0.3: !tr. (tr : dom(rear) => rear(tr) < front(tr))
}
```

Fig. 4. SysML/KAOS domain modeling of the goal diagram root level

position of its rear[2]. Thus, $dom(front) \setminus dom(rear)$ represents the set of trains equipped with a ERTMS and not equipped with a TIMS. Predicates represent constraints on domain model elements. Each predicate is prefixed with an identifier `p<i>.<j>` where `<i>` designates the refinement level number and `<j>` designates the number of the predicate of this refinement level. For example, the predicate `p0.2` defines `TRACK` as the data range $a..b$.

SYSTEM ertms_etcs_case_study
SETS TRAIN
CONSTANTS a b TRACK
PROPERTIES
axm1: $a \in \mathbb{N}$ axm2: $b \in \mathbb{N}$ p0.1: $a < b$
p0.2: $TRACK = a \,..\, b$
VARIABLES connectedTrain front rear
INVARIANT
inv1: $connectedTrain \in TRAIN \nrightarrow BOOL$
inv2: $front \in dom(connectedTrain) \rightarrow TRACK$
inv3: $rear \in dom(connectedTrain) \nrightarrow TRACK$
p0.3: $\forall tr \cdot (tr \in dom(rear) \Rightarrow rear(tr) < front(tr))$
Event MoveTrainOnTrack $\widehat{=}$
any tr len
where
grd1: $tr \in connectedTrain^{-1}[\{TRUE\}]$
grd2: $len \in \mathbb{N}_1$
grd3: $front(tr) + len \in TRACK$
then
act1: $front(tr) := front(tr) + len$
act2: $rear := (\{TRUE \mapsto rear \lhd\!\!\!- \{tr \mapsto rear(tr) + len\}, FALSE \mapsto rear\})(bool(tr \in dom(rear)))$
END END

Fig. 5. *B System* specification of the root level of the goal diagram of Fig. 2

Figure 5 represents the *B System* model obtained from the translation of the root level of the goal diagram of Fig. 2 and of the associated domain model of Fig. 4. The domain model gives rise to sets, constants, properties, variables and invariants of the formal specification. Predicates involving variables give rise to invariants and the others to properties. The *isFunctional* and *isTotal* characteristics of attributes, are used to guess if an attribute should be translated into a partial or total function. The root goal is translated into an event for which the body has been manually specified: the movement of a connected train (`grd1`) results in the incrementation of the position of its front (`act1`) and its rear (`act2` in the case of an *INTEGER* train) of the value corresponding to the movement. Of course, the movement can only be done if the train stays on the track (`grd3`).

3.4 The First Refinement Level

Figure 6 represents the domain model associated with the first refinement level of the SysML/KAOS goal diagram of Fig. 2. It refines the one associated with the root level and introduces an attribute named `MA` representing the MA assigned to a connected train. The MA of a train is modeled as a contiguous part of the track (`p1.1`), containing the train (`p1.2` and `p1.3`). Finally, the predicate `p1.4` asserts that the MA assigned to two different trains must be disjoint. The predicates `p1.2` and `p1.3` are gluing invariants, linking the concrete variable `MA` with the abstract variables `front` and `rear`.

[2] The rear is deduced from the front and length of the train, since a train equipped with a TIMS broadcast its length and its integrity.

```
domain model ertms_etcs_case_study_ref_1 parent domain model ertms_etcs_case_study {
        attributes:
                attribute MA domain: dom(connectedTrain) range: POW(TRACK) {
                        is variable: true  is functional: true   is total: false
                }
        predicates:
                p1.1: !tr. (tr : dom(MA) => #p,q.(p..q<:TRACK & p<=q & MA(tr)=p..q)))
                p1.2: !tr. (tr : dom(MA) => (front(tr) : MA(tr)))
                p1.3: !tr. (tr : dom(rear) & tr : dom(MA) => rear(tr) : MA(tr))
                p1.4: !tr1,tr2. ((tr1 : dom(MA) & tr2 : dom(MA) & tr1 /=
                        tr2)=>MA(tr1) /\ MA(tr2)={})
}
```

Fig. 6. SysML/KAOS domain modeling of the goal diagram first refinement level

REFINEMENT ertms_etcs_case_study_ref_1
REFINES ertms_etcs_case_study
VARIABLES connectedTrain front rear MA MAtemp
INVARIANT
inv1: $MA \in dom(connectedTrain) \nrightarrow \mathbb{P}(TRACK)$
p1.1: $\forall tr \cdot (tr \in dom(MA) \Rightarrow (\exists p, q \cdot (p \mathinner{..} q \subseteq TRACK \land p \leq q \land MA(tr) = p \mathinner{..} q)))$
p1.2: $\forall tr \cdot (tr \in dom(MA) \Rightarrow front(tr) \in MA(tr))$
p1.3: $\forall tr \cdot (tr \in dom(rear) \cap dom(MA) \Rightarrow rear(tr) \in MA(tr))$
p1.4: $\forall tr1, tr2 \cdot (((\{tr1, tr2\} \subseteq dom(MA) \land tr1 \neq tr2) \Rightarrow MA(tr1) \cap MA(tr2) = \emptyset)$
inv6: $MAtemp \in dom(connectedTrain) \nrightarrow \mathbb{P}(TRACK)$
inv7: $\forall tr \cdot (tr \in dom(MAtemp) \Rightarrow (\exists p, q \cdot (p \mathinner{..} q \subseteq TRACK \land p \leq q \land MAtemp(tr) = p \mathinner{..} q)))$
theorem s1: $ComputeTrainMA\_Guard \Rightarrow MoveTrainOnTrack\_Guard$
theorem s2: $ComputeTrainMA\_Post \Rightarrow AssignMAtoTrain\_Guard$
theorem s3: $AssignMAtoTrain\_Post \Rightarrow MoveTrainFollowingItsMA\_Guard$
theorem s4: $MoveTrainFollowingItsMA\_Post \Rightarrow MoveTrainOnTrack\_Post$

Event
ComputeTrainMA $\widehat{=}$
any tr p q len
where
grd1: $tr \in connectedTrain^{-1}[\{TRUE\}]$
grd2: $p \mathinner{..} q \subseteq TRACK \land p \leq q$
grd3: $front(tr) \in p \mathinner{..} q$
grd4: $tr \in dom(rear) \Rightarrow rear(tr) \in p \mathinner{..} q$
grd5: $p \mathinner{..} q \cap union(ran(\{tr\} \lhd MA)) = \emptyset$
grd6: $len \in \mathbb{N}_1$
grd7: $front(tr) + len \in TRACK$
then
act1: $MAtemp(tr) := p \mathinner{..} q$
END

AssignMAtoTrain $\widehat{=}$
any tr len
where
grd1: $tr \in connectedTrain^{-1}[\{TRUE\}] \cap dom(MAtemp)$
• • •
grd6: $front(tr) + len \in MAtemp(tr)$
then
act1: $MA(tr) := MAtemp(tr)$
END

MoveTrainFollowingItsMA $\widehat{=}$
any tr len
where
grd1: $tr \in connectedTrain^{-1}[\{TRUE\}] \cap dom(MA)$
grd2: $len \in \mathbb{N}_1$
grd3: $front(tr) + len \in MA(tr)$
then
act1: $front(tr) := front(tr) + len$
act2: $rear := (\{TRUE \mapsto rear \lhd\!\!- \{tr \mapsto rear(tr) + len\}, FALSE \mapsto rear\})(bool(tr \in dom(rear)))$
END
END

Fig. 7. *B System* specification of the first refinement level of the diagram of Fig. 2

Figure 7 represents the *B System* model obtained from the translation of the first refinement level of the goal diagram of Fig. 2 and of the associated domain model of Fig. 6. Each refinement level goal is translated into an event for which the body has been manually specified: the current MA of the train is computed and stored into a variable named `MAtemp` (event `ComputeTrainMA`). Because the computation of the MA is out of the scope of the case study [7], the event simply nondeterministically choose an MA, with respect to the safety invariants. This MA is then assigned to the train by updating the variable `MA` (event `AssignMAtoTrain`) and taken into account for the train displacement

(event MoveTrainFollowingItsMA). Theorems s1, s2, s3 and s4 represent the proof obligations related to the usage of the *MILESTONE* operator between the root and the first refinement levels. Since each proof obligation has been modeled as an *Event-B* theorem, it has been proved based on system properties and invariants. To deal with the fact that *Event-B* does not currently support the temporal logic, we have used the proof obligation $G1\_Post \Rightarrow G2\_Guard$ for the invariants s2 and s3, instead of $\Box(G1\_Post \Rightarrow \Diamond G2\_Guard)$ (Sect. 2.3), since $(G1\_Post \Rightarrow G2\_Guard) \Rightarrow (\Box(G1\_Post \Rightarrow \Diamond G2\_Guard))$. By using this trick, we replace the proof obligation involving operators of the temporal logic with a more constraining proof obligation. The trick is only useful if it is possible and easier to discharge the newly introduced proof obligation. The full specification of s1 is given below:

theorem s1: $\forall tr, p, q, len \cdot (((tr \in connectedTrain^{-1}[\{TRUE\}]) \land (p..q \subseteq TRACK \land p \leq q) \land (front(tr) \in p..q) \land (tr \in dom(rear) \Rightarrow rear(tr) \in p..q) \land (p..q \cap union(ran(\{tr\} \lhd MA)) = \emptyset) \land (len \in \mathbb{N}_1) \land (front(tr) + len \in TRACK)) \Rightarrow ((tr \in connectedTrain^{-1}[\{TRUE\}]) \land (len \in \mathbb{N}_1) \land (front(tr) + len \in TRACK)))$

It expresses the fact that the activation of the guard of ComputeTrainMA for certain parameters is sufficient for the activation of the guard of MoveTrainOnTrack for this same group of parameters.

3.5 The Second Refinement Level

Figure 8 represents the domain model associated with the second refinement level of the diagram of Fig. 2. It refines the one associated with the first refinement level and introduces two concepts named TTD and VSS. The attributes stateTTD and stateVSS represent the states of the corresponding concepts. The predicates p2.1..p2.8 define each TTD as a contiguous part of the track and each VSS as a contiguous part of a TTD. The predicates p2.9 and p2.10 are used to state that if a train is located on a TTD, then its state must be occupied: a train $tr \in TRAIN$ is located on $ttd \in TTD$ if $front(tr) \in ttd$ (p2.9) or if tr is equipped with a TIMS ($tr \in dom(rear)$) and $(rear(tr)..front(tr)) \cap ttd \neq \emptyset$ (p2.10). Finally, the predicates p2.11..p2.13 states that two different trains must be in disjoint parts of the track: for two trains tr1 and tr2, if they are equipped with TIMS, then the track portions that they occupy should just be disjointed (p2.11); if they are on the same TTD and one of them, (tr2), is not equipped with a TIMS, then, the second, (tr1), must be equipped with a TIMS and tr2 must be in the rear of tr1 (p2.12); if none of them is an INTEGER train, then they must be in two distinct TTDs (p2.13). The predicates p2.9 and p2.10 are gluing invariants, linking the concrete variable stateTTD with the abstract variables front and rear. The *B System* specification raised from the translation of the second refinement level includes the result of the translation of the domain model of Fig. 8, two new events (ComputeTrainMAFollowingTTDStates, ComputeTrainMAFollowingVSSStates), an extension of the event MoveTrain FollowingItsMA taking into account the new safety invariants and the theorems representing the proof obligations related to the usage of the *OR* operator between the first and second refinement levels.

```
domain model ertms_etcs_case_study_ref_2 parent domain model ertms_etcs_case_study_ref_1 {
        concepts:
                concept TTD is variable: false      concept VSS is variable: false
        attributes:
                attribute stateTTD domain: TTD range: TTD_STATES {
                        is variable: true  is functional: true   is total: true
                }
                attribute stateVSS domain: VSS range: VSS_STATES {
                        is variable: true  is functional: true   is total: true
                }
        data sets:
                enumerated data set VSS_STATES { elements :
                        data value OCCUPIED   data value FREE
                        data value UNKNOWN   data value AMBIGUOUS
                }
                enumerated data set TTD_STATES { elements :
                        data value OCCUPIED    data value FREE
                }
        predicates:
                p2.1: TTD <: POW1(TRACK)  p2.2: union(TTD) = TRACK  p2.3: inter(TTD) = {}
                p2.4: !ttd. (ttd : TTD => #p,q.(p..q<:TRACK & p<q & ttd=p..q)))
                p2.5: VSS <: POW1(TRACK)  p2.6: union(VSS) = TRACK  p2.7: inter(VSS) = {}
                p2.8: !vss. (vss : VSS => #p,q,ttd.(ttd : TTD & p..q<:ttd & p<q & vss=p..q)))
                p2.9: !ttd,tr. ( tr : dom(front) \ dom(rear) & ttd : TTD & front(tr) : ttd)
                        => (( ttd |-> OCCUPIED ) : stateTTD)
                p2.10: !ttd,tr. (tr : dom(rear) & ttd : TTD & (rear(tr)..front(tr))/\ttd /= {})
                        => (( ttd |-> OCCUPIED ) : stateTTD)
                p2.11: !tr1,tr2. (tr1 : dom(rear) & tr2 : dom(rear) & tr1 /= tr2)
                        => ( (rear(tr1)..front(tr1))/\(rear(tr2)..front(tr2))={})
                p2.12: !tr1,tr2,ttd.(tr1 : dom(rear) & tr2 : dom(front)\dom(rear) & tr1 /= tr2
                        & ttd : TTD & front(tr2) : ttd & rear(tr1)..front(tr1))/\ttd /= {})
                        => ( front(tr2)<rear(tr1))
                p2.13: !tr1,tr2,ttd. ( tr1 : dom(front)\dom(rear) & tr2 : dom(front)\dom(rear)
                        &tr1 /= tr2 & ttd : TTD & front(tr1) : ttd) => ( front(tr2) /: ttd)
}
```

Fig. 8. SysML/KAOS domain modeling of the goal diagram second refinement level

4 Discussion

Benefits. This case study allowed us to benefit from the advantages of a high-level modeling approach within the framework of the formal specification of the hybrid ERTMS/ETCS level 3 requirements: decoupling between formal specification handling difficulties and system modeling; better reusability and readability of models; strong traceability between the system formal specification and the goal model, which is an abstraction of the case study description. Using the FORMOSE approach, we have quickly built the refinement hierarchy of the system and we have determined and formally expressed the safety invariants. The approach bridges the gap between the system textual description and its formal specification. Its use has made it possible to better present the specifications, excluding predicates, to *stakeholders*[3] and to better delineate the system boundaries. Using Rodin [3], we have formally verified and validated the safety invariants and the goal diagram refinement hierarchy. Through ProB [9], we have

[3] Stakeholders, here, include the co-authors of this paper and the members of the FORMOSE project involved in the study. We plan an assessment on more external entities.

animated the formal model. The full specification can be found in [24]. One conclusion of our work is that the description of the standard, as it exists in the documents [5,7,14], does not guarantee the absence of train collisions. Indeed, since the standard allows the movement of unconnected trains on the track, nothing is specified to guarantee that an unconnected train will not hit another train (connected or not). The animation of the specification allows the observation of these states. The only guarantee that the safety invariants expressed in [5,7,14] bring is that a connected train will never hit another train.

Comparison. We have also specified in a companion paper [10] the case study using plain Event-B, in the traditional style. Two distinct specifiers (first author of [10] and first author of this paper) wrote each specification without interacting with each other during specification construction. Critical reviewing by the team was then conducted after the specifications were built. The specification in [10] includes four refinement levels. The TTDs and trains are introduced in the root level and the VSSs are introduced in the second refinement level, as refinements of TTDs. The MAs and VSS states are introduced in the third refinement level (M3), for train movement supervision. A strategy is proposed to prove the determinism of the transitions of VSS states. The state variables of [10] are partitioned into environment variables and controller variables, and similarly for events. Environment events only modify environment variables. Controller events read environment variables and update controller variables. In this paper, we only model controller events; state variables represent the controller view of the environment. The execution ordering and the refinement strategy are enforced using proof obligations expressed as theorems, whereas in [10] there is no proof about these aspects. In [10], the safety properties are introduced in the last refinement level; here, we introduce them in the first (predicate `p1.4`) and second (predicates `p2.9`..`p2.13`) refinements. In [10], all trains equipped with ERTMS are equipped with TIMS, so they broadcast their front and rear; here, we consider ERTMS trains with or without TIMS, so a ERTMS train may or may not broadcast its rear. The FORMOSE approach makes it possible to trace the source and justify the need for each formal component and its contents, in relation with the SysML/KAOS goal and domain models. The FORMOSE approach therefore represents a more structured and methodological process to the formal specification of the system.

Difficulties. The expression of theorems representing proof obligations associated to SysML/KAOS refinement operators was difficult because there is no way in Rodin to designate the guard and the post condition of an event within predicates. Table 2 summarises the key characteristics related to the formal specification. The proof obligations have been discharged using the Rodin tool extended with *Atelier B provers* [17] and *SMT solvers* [19]. Customised auto-tactic/post-tactic profiles, including the added provers, with extended timeouts, have been defined. It seemed that the provers have a lot of trouble with data ranges such as $p..q$ and with conditional actions such as $rear := (\{TRUE \mapsto rear \lhd\!\!- \{tr \mapsto$

$rear(tr) + len\}, FALSE \mapsto rear\})(bool(tr \in dom(rear)))$ defined in the component `ertms_etcs_case_study` to simulate an *if-then-else* in order to avoid the definition of a second event.

Table 2. Key characteristics related to the formal specification

Refinement level	L0	L1	L2	L3	L4	L5	L6
Invariants	4	11	13	4	6	5	9
Proof obligations (PO)	20	40	50	13	5	5	14
Automatically discharged POs	17	30	30	11	0	0	4
Interactively discharged POs	3	5	20	2	5	5	10

5 Conclusion and Future Work

This paper focusses on the use of the FORMOSE approach for the high level modeling of system requirements, of domain properties and of safety invariants related to the hybrid ERTMS/ETCS level 3 standard [5,7,14]. Translation rules, supported by tools [13,23], have then been applied to obtain a formal specification containing the system structure and the skeleton of events. The Rodin tool [3] has been used to verify and validate the formal specification, especially to prove the safety invariants and the refinement logic, after the completion of the body of events. The full specification can be found in [24]. A comparison with a companion paper on the same case study, but specified using only plain *Event-B*, has been done.

Work in progress aims at improving the representation of domain predicates (to make them more user-friendly) and at evaluating the impact of updates on *B System* specifications within SysML/KAOS models. We are also working on integrating the approach within the open-source platform *Openflexo* [15] which federates the various contributions of *FORMOSE* project partners [2].

Acknowledgment. This work is carried out within the framework of the *FORMOSE* project [2] funded by the French National Research Agency (ANR). It is also partly supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

1. Abrial, J.: Modeling in Event-B - System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. ANR-14-CE28-0009: Formose ANR project (2017)
3. Butler, M., Jones, C.B., Romanovsky, A., Troubitsyna, E. (eds.): Rigorous Development of Complex Fault-Tolerant Systems. LNCS, vol. 4157. Springer, Heidelberg (2006). https://doi.org/10.1007/11916246
4. ClearSy: Atelier B: B System (2014). http://clearsy.com/
5. EEIG ERTMS Users Group: Hybrid ERTMS/ETCS Level 3: Principles. Ref. 16E042 Version 1A, July 2017

6. Gnaho, C., Semmak, F., Laleau, R.: Modeling the impact of non-functional requirements on functional requirements. In: Parsons, J., Chiu, D. (eds.) ER 2013. LNCS, vol. 8697, pp. 59–67. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-14139-8_8
7. Hoang, T.S., Butler, M., Reichl, K.: The hybrid ERTMS/ETCS level 3 case study. In: ABZ, pp. 1–3 (2018)
8. van Lamsweerde, A.: Requirements Engineering - From System Goals to UML Models to Software Specifications. Wiley, Hoboken (2009)
9. Leuschel, M., Butler, M.: ProB: a model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45236-2_46
10. Mammar, A., Frappier, M., Tueno, S., Laleau, R.: An Event-B Model of the ERTMS/ETCS Level 3 Standard (2018). info.usherbrooke.ca/mfrappier/abz2018-ERTMS-Case-Study
11. Mammar, A., Laleau, R.: On the use of domain and system knowledge modeling in goal-based event-B specifications. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 325–339. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47166-2_23
12. Mashkoor, A., Jacquot, J.: Utilizing Event-B for domain engineering: a critical analysis. Requir. Eng. **16**(3), 191–207 (2011). https://doi.org/10.1007/s00766-011-0120-5
13. Matoussi, A., Gervais, F., Laleau, R.: A goal-based approach to guide the design of an abstract Event-B specification. In: ICECCS 2011, pp. 139–148. IEEE Computer Society (2011)
14. Nicola, F., van Henri, H., Laura, A., Maarten, B.: ERTMS level 3: the game-changer. In: IRSE News View, p. 232, April 2017
15. Openflexo: Openflexo project (2015). http://www.openflexo.org
16. Pierra, G.: The PLIB ontology-based approach to data integration. In: Jacquart, R. (ed.) Building the Information Society. IIFIP, vol. 156, pp. 13–18. Springer, Boston, MA (2004). https://doi.org/10.1007/978-1-4020-8157-6_2
17. Deploy Project: Rodin Atelier B Provers Plug-in (2017). https://www3.hhu.de/stups/handbook/rodin/current/html/atelier_b_provers.html
18. Sengupta, K., Hitzler, P.: Web ontology language (OWL). In: Alhajj, R., Rokne, J. (eds.) Encyclopedia of Social Network Analysis and Mining, pp. 2374–2378. Springer, New York (2014). https://doi.org/10.1007/978-1-4614-6170-8_113
19. SYSTEREL: Rodin SMT Solvers Plug-in (2017). http://wiki.event-b.org/index.php/SMT_Solvers_Plug-in
20. Tueno, S., Laleau, R., Mammar, A., Frappier, M.: Towards using ontologies for domain modeling within the SysML/KAOS approach. In: 25th IEEE International Requirements Engineering Conference on IEEE Proceedings of MoDRE Workshop (2017)
21. Tueno, S., Laleau, R., Mammar, A., Frappier, M.: Formal Representation of SysML/KAOS Domain Models. ArXiv e-prints, cs.SE, 1712.07406, December 2017
22. Tueno, S., Laleau, R., Mammar, A., Frappier, M.: The SysML/KAOS Domain Modeling Approach. ArXiv e-prints, cs.SE, 1710.00903, September 2017
23. Tueno, S., Laleau, R., Mammar, A., Frappier, M.: The SysML/KAOS Domain Modeling Language (Tool and Case Studies) (2017). https://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser/tree/master
24. Tueno, S., Laleau, R., Mammar, A., Frappier, M.: SysML/KAOS Approach on the Hybrid ERTMS/ETCS Level 3 Case Study (2018). https://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser/tree/master/ABZ18_ERTMS

Modelling the Hybrid ERTMS/ETCS Level 3 Case Study in SPIN

Paolo Arcaini, Pavel Ježek, and Jan Kofroň(✉)

Charles University, Faculty of Mathematics and Physics,
Prague, Czech Republic
{arcaini,jezek,kofron}@d3s.mff.cuni.cz

Abstract. The SPIN model checker has been successfully applied to the modelling, validation, and verification of different safety-critical systems. In this paper, we model and validate the Hybrid ERTMS/ETCS Level 3 Case Study using SPIN; in particular, we show the assumptions we made to keep the state space limited, and present the problems and ambiguities that arose during the modelling. Although SPIN offers several advantages in terms of validation and verification facilities, its modelling language PROMELA is limited if compared to higher level notations of other formal methods. Therefore, we discuss the advantages and disadvantages of using the tool, and how it could be improved in terms of modelling facilities.

1 Introduction

In the context of the ABZ 2018 conference, the Hybrid ERTMS/ETCS Level 3 Case Study [6] has been proposed as benchmark for comparing the strengths (and weaknesses) of different state-based formal methods. The solutions provided for the case study should demonstrate the modelling, validation, and verification facilities of different methods.

The aim of the Hybrid ERTMS/ETCS Level 3 [6] is to increase the throughput of railway tracks, by integrating the physical information coming from the *trackside detection system* with information transmitted by the train itself regarding its position and integrity. In pre-Level 3 systems the railway track is divided in TTD (*trackside train detection*) sections and entering and exiting each TTD is physically detected; in such a situation, a whole TTD section is blocked when there is a train inside (i.e., no two trains can be in a single TTD[1]). In Hybrid ERTMS/ETCS Level 3 the train also periodically sends information about its position and integrity to the trackside system; in this situation, each TTD is further divided into several *virtual sub-sections* (VSSs) and the system

The research reported in this paper has been partially supported by the Czech Science Foundation project number 17-12465S.

[1] Actually, two trains can be in a TTD if they are operating in *on-sight mode* in which the drivers are fully responsible for the train movement; this setting, however, is an exceptional case that is not a part of normal operational mode.

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 277–291, 2018.
https://doi.org/10.1007/978-3-319-91271-4_19

should avoid presence of two trains on the same VSS. We remind to [6] for the complete description of the requirements.

As widely demonstrated in literature [14], there is no golden method for system analysis, and each particular method can provide a better support for some aspects (e.g., modelling), but be deficient on some others (e.g., scalability in verification). High-level notations as Abstract State Machines [4] and B [1] provide a wide support for modelling and refining the model and can be also used for documentation purposes when discussing with the stakeholders. In addition, they also provide different facilities for verification in terms of model checking [2, 15]; they usually translate their models into the notation of an existing model checker [2,7,16,17] and use this for the verification. The problem of this approach is that the mapping usually introduces a non-trivial overhead that limits their scalability.

On the other hand, implementing the problem directly in the notation provided by the model checker (e.g., PROMELA for SPIN [12] or the input notation of NuSMV [8]) usually allows one to obtain a more simple model that scales better. This is due to the fact that the notation provided by model checkers is rather limited and allows to get a better understanding of the consumed resources. However, such notations are usually less readable than notations as ASMs and B. Therefore, there is a trade-off between readability and scalability.

In this paper, we propose a solution of the aforementioned case study in SPIN. In developing the system, we tried to abstract as much as possible from all the unnecessary details, but still preserving the soundness of the verification. Since we have taken the approach of modelling the system directly in the input language of an explicit model checker, our solution can be taken as a baseline comparison for evaluating the performances of solutions developed in higher level notations. Such comparison could be used to assess the overhead introduced by mapping tools (and eventually bring to their improvement). On the other hand, we also aim at identifying those features that are missing in PROMELA (e.g., logging and visualization facilities) and that could be added to the language without compromising the performance.

Section 2 illustrates how we modelled the case study in PROMELA, and Sect. 3 describes the experiments we conducted. Section 4 discusses some problems we faced during the development of the model and some lessons learned. Finally, Sect. 5 reviews some related work and Sect. 6 concludes the paper.

2 Model

2.1 SPIN Modelling Platform

The PROMELA language is the input language for the SPIN model checker [12]. A PROMELA model consists of global variables and definitions of process types. Each process type can be instantiated resulting in a process (instance), which becomes the active entity of the model. A process consists of local variables and a sequence of statements, which are executed basically in the order in which they are written. The value of local variables and the process program counter

define the state of the process. The model is then defined as all allowed interleavings of particular processes' statements; in turn, the (global) model state is the composition of states of all processes and the values of global variables.

Spin allows for both simulation and verification of the models. While during verification, the entire state space, i.e., all states of the model are explored, the simulation can be seen as one particular execution, i.e., one particular interleaving of processes' statements, similarly to an execution of a multi-threaded program.

Spin provides several means for specifying, and also verifying specific model properties. They include asserts (as known from imperative programming languages, such as Java and C/C++), LTL formulae, checking for deadlocks and non-progress cycles, and so-called "never claims" [12]. While asserts can be used in a similar way as in common programs to check variable values at particular model places (i.e., expressing safety properties), LTL and non-progress cycles allow the developer for expressing and checking more complex properties, including both safety and liveness ones. Never-claims provide even more precise way for specification of the properties in an imperative way (complementary to declarative LTL).

2.2 Description of Model

In this section, we give a general overview of the model we developed.

The model (the `model.pml` file) consists of two main parts, each one represented by a Promela process—a `reality` process and a `trackside` process. On the other hand, each train is represented by a data structure (`Train`), thus being a passive entity in our model.

The `reality` process represents the real situation. It manipulates an array of `VSS` (`real[]`) representing the actual position of the trains and updates the fields of the `Train` structure accordingly.

The trains are moved either according to their movement authority in a random manner, or according to a defined scenario. The operation mode and the particular scenario to be executed is determined by the value of the `sce` variable; if none is defined, random mode takes place. The scenarios are specified in a separate file `scenarios.pml`.

The `trackside` process represents the behaviour of the trackside infrastructure. It receives the information reported by trains (by means of reading the fields of the `Train` structure) and changes the state of particular VSSs accordingly. The VSSs and their states are stored in an array (`vss[]`), similarly to the real state (`real[]`).

Code 1 shows an excerpt of the data structures used in the model. Each VSS (i.e., a `VSS`) can be in one of the four states `FREE`, `OCCUPIED`, `AMBIGUOUS`, and `UNKNOWN` as described in the case study document [6]. The VSSs of the `real` array, instead, can be set just either to `FREE` or to `OCCUPIED`; the `real` array is not accessed by the trackside detection system, but it is only used for debugging and verification purposes to compare the assumed state with the real one.

```
mtype = {REAL, TRACKSIDE};
mtype schedule = REAL;

mtype = {FREE, OCCUPIED, AMBIGUOUS, UNKNOWN};
typedef VSS {
    mtype state;
    byte ttd;
}

VSS vss[VSSCOUNT];
VSS real[VSSCOUNT];

mtype = {TTDFREE, TTDOCCUPIED};
typedef TTDSection {
    byte firstVSS;
    mtype state;
    bool ghost;
}
```

```
TTDSection ttd[TTDCOUNT];

typedef Train {
    byte front;
    byte rear;
    bool connected;
    bool integer;

    byte eom;
    byte eoma;
    bool hasreported;
    byte reportedposition;
    byte reportedintegrity;
    ...
}

Train trains[TRAINCOUNT];
```

Code 1. Data structures

Array `ttd` represents the real state of each TTD section; note that the TTD information is always considered safe by the trackside, and so there is only one copy of the array. In order to model the division of TTDs in VSSs, each `TTDSection` stores the index of the first VSS in the section. Similarly, each `VSS` stores an index of the `TTDSection` of which it is a part.

The `Train` structure involves fields both representing the real situation (e.g., position of the train) and those communicated to the trackside (e.g., reported position). The most important `Train` fields are:

- `front` and `rear` representing the real VSSs containing the front end and the rear end of the train (either the same one, or two consecutive ones)—see Sect. 2.3;
- `eoma` is the end of movement authority that the trackside grants to the train;
- `eom` is the destination of the train (i.e., "end of mission");
- `hasreported` tells whether the train has reported in its last step;
- `reportedposition`, `reportedintegrity`, ... represent the information reported by the train through the PTD; for example, `reportedposition` is the last reported position of the front end of the train.

Often, we use high values (254, 255) to model "unknown" or "invalid" values. For example, if a train has never reported, the last reported position is set 255.

The behaviour of the model is specified by the rules partially shown in Code 2. The model alternatively executes two processes: the `reality` process models the movement of the train, while the `trackside` process models the trackside system. The scheduling is determined by the value of the `schedule` variable. An alternative approach would be using the `d_step` or `atomic` blocks, but this solution brings several issues: (1) when using `d_step` blocks, the non-deterministic choices would not be explored, (2) a statement inside an `atomic` block can become blocked (by mistake in the model), so the other process would get executed (unexpectedly), and (3) when experimenting with `d_step` blocks, spin reported

```
#define rule2A (ttdstate == TTDOCCUPIED) && (trainidonvss != 255) && ...
#define rule7A trainidonvss != 255 && ((mutetimer[0] == 2 && trainidonvss == 0) || ...

...
inline updateVSS(i) {
    ...
    if
        :: vss[i].state == FREE ->
            if
                ...
                :: rule2A -> vss[i].state = OCCUPIED; log("transition #2A taken\n");
                ...
            fi
        :: vss[i].state == OCCUPIED ->
            if
                ...
                :: rule7A -> vss[i].state = UNKNOWN; log("transition #7A taken\n");
                ...
            fi
        :: vss[i].state == AMBIGUOUS -> ...
        :: vss[i].state == UNKNOWN -> ...
    fi
}

proctype trackside() {
    do
    :: schedule == TRACKSIDE ->
        ...//set timers
        atomic {
            for (i : 0 .. VSSCOUNT - 1) {
                updateVSS(i);
            }
        }
        ...//update end of movement authority
        schedule = REAL;
    :: timeout -> break;
    od;
}

...

proctype reality() {
    do
    ::schedule == REAL ->
        if
            :: (vss[0].state == FREE) && (alive < 2) -> spawntrain(alive); alive++;
            :: trains[0].alive -> move(0); // train 0 moves
            :: trains[1].alive -> move(1); // train 1 moves
            ...
        fi;
        trainreport(0); // train 0 can either report or not
        trainreport(1); // train 1 can either report or not
        schedule = TRACKSIDE;
    od;
}
```

Code 2. Rules

artificial deadlocks. We assume that the last issue is caused by too many statements inside a single `d_step` block.

When executed, the `reality` process non-deterministically performs one of the following actions:

– spawn of a new train (up to the fixed maximum train number of 2);
– progress of the spawned trains. Each train can perform one of the following actions:
  • if the train occupies only one VSS, it can either move only the front in the next VSS (so occupying two VSSs), or move entirely in the next VSS. The type of movement is chosen non-deterministically;
  • if the train spans over two VSSs, the rear of the train can be moved to the VSS containing the front;
  • the train disappears if it reaches `eom`, or the end of the modelled part[2];
  • the train can decide not to move from the current position. This models the situation in which the train moves while staying inside the same VSS(s);
  • the connected train can disconnect and vice versa;
  • the train can also split into two trains, if there is just one train in the system so far.

Moreover, at each step, the train can either report or not. The report always includes information about the position of the train, but may or may not involve integrity information.

At each step, the `trackside` process:

1. non-deterministically sets the starting and expiration of timers;
2. updates the states of the VSSs according to the rules of the case study document;
3. updates the `eoma` of the trains up to the first free VSS.

2.3 Abstractions

One of the main difficult aspects in modelling is to decide which details of the requirements can be abstracted away as not necessary for checking the correctness of the system. Leaving out details of the model has two advantages: the model is simpler to understand and maintain, and can be handled by verification tools (i.e., to tackle the state explosion problem). In the following, we report on the abstractions we applied to the case study requirements.

Train Length. We do not explicitly model the train length. We assume that a train can fit in a VSS and, therefore, during its journey, it can span at most over two VSSs. This decision is motivated by the specification document [6] (including the scenarios) that only considers these train lengths.

Number of Trains. The scenarios also report at most two trains. We assume this situation to be general enough to capture situations with a greater number of trains. As in the case of the train length, we were inspired by the specification document and the motivation to keep the model state space smaller.

[2] Note that the case study assignment [11] considers movement only in one direction, i.e., no backward moves.

Train Behaviour. We assume that the train can do at most one action at a time: move the front end, move the rear end, move entirely to the following VSS, disconnect, or reconnect. Therefore, for example, it is not possible that the reality process moves and disconnects a train in a single step. However, we can still model a given combination of train actions in several consecutive steps; for example, the simultaneous train movement and disconnection is captured by two steps, in which the train first moves and then disconnects. Our experiments with scenarios show that this approach includes also all the one-step VSS updates, so we consider it an over-approximation. In addition to that, the train can also disappear after reaching either its `eom` or the end of the modelled railway track.

Timers. The timers are modelled in quite a precise way. Each timer is started and stopped when the conditions for it [6] are met. Since Promela does not provide any real time support, the timers in our model non-deterministically expire after they are started. This can lead to unrealistic situations in the model, which would not appear in practice. For example, a train can move over several VSS without reporting its position and without expiring its mute timer. On the other hand, there is no precise relation between the timer expiration time, train speed and VSS lengths in the requirements document [6], which we consider its particular deficiency[3]. Our approach also allows us not to explicitly model the *wait integrity timers*, since they can be covered by the *integrity loss propagation timers*.

3 Experiments

We run all the experiments on a Linux blade server with Xeon X5687 CPU with 192 GB RAM. The model file together with the scenario definitions and output of scenario simulations are available at http://d3s.mff.cuni.cz/~kofron/abz18casestudy.html. Modelling the whole case study took about one month: two weeks for creating the model and other two weeks for debugging it.

In order to validate our approach, we simulated the nine scenarios reported in the requirements [6]; in order to automatize the approach, we had to specify in the model itself a mechanism for forcing some particular steps: more details are given in Sect. 4. In almost all the steps of all the scenarios, we were able to reproduce the exact VSSs configuration, using the same rules reported in the requirements to update the single VSSs. In some particular steps, instead, our simulation differs because of errors and/or ambiguities in the specification; we detail all of them in Sect. 4.

In addition to validation, we performed a more detailed analysis in terms of formal verification. We ran several verification runs with different settings (stack size limits, storage modes—exhaustive vs. bitstate hashing) to cover as large part of the state space as possible. We learned that the state space is branching a lot and so that it makes sense to run the verifier with both large

[3] Formulations in [6] such as "A value between 5–10 s would seem to be practical" and "... this timer could be set to a value of at least 27 s ..." are not of much use.

and small stack sizes. In sum, we ran the verification over more than a week, being able to explore over 6×10^{11} states. Of course, we are not aware of the total size of the state space, however, several bitstate hashing verifications were successfully accomplished. Even though being just approximative method, no error was found this way.

We attempted at proving a set of safety properties (assertions in the model) regarding the correct movement of trains:

- In order to avoid train collision, we check that a train does not move in a VSS occupied by another train. We actually found a violation of this property when the mute timer of the first train is started and does not expire while the train is moving over several VSSs; in this case, the chasing train can proceed and enter the VSS of the first train[4]. The violation is due to the fact that, as explained in Sect. 2.3, we do not put any constraint on the timer expiration (for example, a timer can start and never expire or it can take arbitrarily long). We think that there should be a relation between the train speed, timers duration, and the lengths of VSSs that, however, is not articulated in the requirements. The assertion violation was found in about 30 s, using the stack size of 4,000 states.
- We also check that a train does not move beyond its end of movement authority (EoMA) nor beyond its end of mission (EoM). This property can be violated (and we assume that in practice it is—c.f. step 2 of scenario 8) in the "on-sight mode", which we do not attempt to model. The reason for this is that the safety requirements of the system cannot be guaranteed in this mode.

In addition to functional correctness of the modelled system, we checked whether the requirements are consistent. The conditions specified in the state machine for the VSS (see Sect. 5 in [6]) should guarantee that, for a VSS, it is not possible that two rules bringing to different target states are applicable at the same time; when this is possible, the requirements explicitly specify the priority among the rules. Therefore, the update of VSSs should be deterministic. However, it could still be that the requirements document is not correct or that we wrongly implemented the rules. In order to check that the update of VSSs is deterministic, we performed an additional check. Before updating a VSS (by non-deterministically selecting one rule that is applicable), we count all the rules that are actually applicable and, if more than one applies, we raise an assertion violation. In this way, we found that in a particular scenario two rules are applicable: in step 5 of scenario 8, both rules #10A (the one reported in the requirements document) and #9A can be applied for VSS12. The VSS is the last one of TTD10, it is in state AMBIGUOUS, and a train has just left it and crossed the TTD border. Rule #10A is applicable when "VSS is left by all reporting trains", while #9A when "TTD is free"; both conditions are clearly satisfied in the current situation. The same problem appears in the update of VSS 12 in step 7 of scenario 9. We believe that the problem is due to

[4] The simulation output of the assertion violation can be found at http://d3s.mff.cuni.cz/~kofron/abz18casestudy.html.

an ambiguous description of rule #10A in the requirements document; indeed, the description of the rule refers to paragraphs 3.6 and 3.7 that regard non-integer trains; however, in scenarios 8 and 9 both trains are always integer, and, therefore, it is not clear why rule #10A should apply. Also, it is not clear to us what "all reporting trains" refers to—all reporting trains at the same TTD or all reporting trains in the system?

4 Discussion

In this section, we discuss the problems we faced during the model development. In particular, we focus on the problems that are caused by the deficiencies of the adopted modelling language (see Sect. 4.1), and on those that arise when reading the requirements (see Sect. 4.2).

4.1 Missing Facilities

Promela provides a limited support to debug/log the model by means of standard printing to the standard output. In order to visualize the train movement, we had to add suitable printing outputs into the model. Figure 1 shows an excerpt of the simulation of a scenario[5]. For each simulation step, we report events related to trains and signals, which VSSs have been updated and by which rule, and EoMAs of existing trains. Moreover, we also visually depict the real position of the trains in the first line (`A`, `a`, `B`, `b` for the first train connected, for the first train disconnected, for the second train connected, and for the second train disconnected, respectively), the VSSs statuses in the second line, and the TDDs statuses in the third line. The last line shows the TTD number.

Although the implemented solution worked pretty well for our purposes, some formal methods provide nicer ways to visualize the model evolution; for example, for the B method, ProB provides an animator [13] that allows to visualize specific pictures associated with model states. The advantages of this method are several: first of all, the visualization can be much nicer and understandable than that obtainable by standard text printing; moreover, since the visualization is defined in a separate function, there is a clear separation of concerns (specification of the behaviour and logging) in the model. As future work, we could consider to add some animation facilities to Promela/Spin.

Another feature that we missed during the development is a proper support for guided simulation. Spin allows to simulate the model by choosing, at each step, which state to take as next state (by selecting the values of variables that are non-deterministically updated); however, if the model is big, doing a manual simulation can be particularly cumbersome. Some formal methods allow to specify *scenarios* of the model execution by writing script in which non-deterministic

[5] Note that in Spin, we sometimes need to perform multiple steps in order to model a single step of a scenario reported in the requirements document; therefore, the step numbers (6 and 7) reported in the figure are different from the corresponding steps of the requirements document (steps 3 and 4).

```
Step 6 of scenario 9
Train 1 disconnects
Train 0 reported having left VSS3
Train 0 reported with integrity
Train 1 NOT reported
Timer ghosttimer expired
VSS2: transition #1F taken
VSS3: transition #1F taken
VSS4: transition #8B taken
Train A - eoma: 9
Train B - eoma: 3

b                               A
U       U       U       U       A       F       F       F       F       F
0       0       0       0       0       F       F       F       F       F
0       0       1       1       1       2       2       2       3       3
--------------------------------------------------------------------------------

Step 7 of scenario 9
Moving front of train 0 forward
Moving both front and rear of train 1 forward
Train 0 reported having left VSS3
Train 0 reported with integrity
Train 1 NOT reported
VSS5: transition #3A taken
Train A - eoma: 9
Train B - eoma: 3

        b                       A       A
U       U       U       U       A       A       F       F       F       F
0       0       0       0       0       0       0       0       F       F
0       0       1       1       1       2       2       2       3       3
--------------------------------------------------------------------------------
```

Fig. 1. Two steps (steps 3 and 4) of simulation of scenario 9

choices are fixed and the model is forced to perform a given number of steps (in a kind of test script). The main advantage of these tools is that scenarios can be executed as many times as necessary (usually, after the model update) in order to check the correctness of the model in that particular situation. In our model, we provided a basic support for scenarios. A scenario is described by means of an array of steps (typedef `Step` shown in Code 3) that specifies the non-deterministic choices to perform at each step. A `Step` is constituted by some variables as `train[]` and `mutetimer[]`; the `train[]` array, for example, encodes which actions should be performed by each train (moving, disconnecting, reporting, etc.). When running the model, we can specify whether it must be run randomly (if variable `sce` is not defined) or if it must read the choices specified in a given scenario `sce`. In order to drive the simulation according to the scenario commands, we had to modify the model such that, in all the points in which a non-deterministic choice is done, the choice specified in the scenario is chosen. Since a scenario variable can encode multiple commands, in the model we use proper masks to extract the commands.

Such approach allowed us to easily specify all the scenarios reported in the case study document. The main drawback of the approach is that the reading of the scenario had to be hard-coded in the model itself, decreasing the readability

```
typedef Step {
  byte train[2];
  byte mutetimer[2];
  byte disconnecttimer[VSSCOUNT];
  byte integritylosstimer[VSSCOUNT];
  byte shadowtimerA[TTDCOUNT];
  byte shadowtimerB[TTDCOUNT];
  byte ghosttimer;
  byte eoma[2];
  byte eom[2];
}

typedef Scenario {
  Step step[SIMULATIONSTEPS];
}
```

```
inline initScenarios() {
  ...
  //step 3
  scenarios[9].step[6].train[0] = 48;//train 0 reports
  scenarios[9].step[6].ghosttimer = 2;//ghost timer expires
  scenarios[9].step[6].train[1] = 5; //trains 1 disconnects
  //the end of movement authority of train 1 is extended to VSS3
  scenarios[9].step[6].eoma[1] = 3;
  //step 4
  //train 0 moves front and reports
  scenarios[9].step[7].train[0] = (2 | 48);
  //train 1 moves entirely without reporting
  scenarios[9].step[7].train[1] = (8);
  ...
}
```

Code 3. Excerpt of scenario 9 specification

and maintainability of the model. As future work, we plan to develop a higher support for scenario (e.g., a DSL for writing scenarios) as, for example, that provided for the ASM method [5].

4.2 Issues in Modelling the Requirements

One of the advantages of adopting formal methods is that they allow to highlight the inconsistencies and/or ambiguities contained in the requirements. Although the case study document [6] is already quite detailed, there are still some parts that we had problem in understanding. In the following, we review all these issues and describe how we handled them.

Delay in TTD Processing. According to the requirements [6], the "TTD information is considered as safe", i.e., it reports "free only if no train is present on the TTD section". Therefore, on the base of this requirement, we always consider the TTD information trustworthy; however, step 7 of scenario 5 reports a case in which "due to the delay time of the TTD detection system, the TTD is still considered occupied". We agree that the information provided in the requirements is not stating that the TTD is occupied only if the train is present; however, we think that the requirement is ambiguous and can bring to this misunderstanding. We are not sure about the length of the delay with respect to the train speed, report frequency, etc., to understand if it is important to consider this particular delay in the model. Therefore, since we are still not sure about which should be the correct behaviour, we decided to keep the model that we produced starting from the reading of the requirements and so we free the TTD section as soon as the train leaves it; therefore, we do not support step 7 of scenario 5.

Updating the End of Movement Authority. The requirements [6] do not exactly specify how and when the *end of movement authority* (EoMA) is modified and by whom. In the model, we assumed that the EoMA is modified by the track side authority after each update of the VSSs states: the EoMA is extended as

long as the VSSs are free or unknown or end of movement EoM is not reached. This is also motivated by the specification scenarios and it seems to be the most permissive choice that still preserves the safety of the system.

Order of Update. At first, we assumed that the update of VSSs depends on the previous state of the other VSSs; actually, this seems to be not true. In step 3 of scenario 9, VSS23 must become ambiguous if the previous VSS is unknown; however, the previous VSS becomes unknown in the same step. Therefore, we update the VSSs from left to right; however, we are not sure whether this assumption is correct.

Loosing the Integrity After Train Split. The requirements do not specify which is the integrity status of a train (actually the integrity status of the two parts of the train) when it splits. At first, we assumed that the train can be either integer (if it splits on purpose and it is aware of its integrity) or non-integer (if it splits accidentally). However, from scenario 5, it seems that the train always looses its integrity when it splits, and so we modelled this behaviour.

On Sight Mode. Requirements [6] mention the possibility for a train to operate in *on sight* (OS) mode "that gives the driver partial responsibility for the safe control of his train" [9]. We do not support the OS mode in the model, because handling it would not allow any kind of safety check regarding the correct operation of trackside detection system (as the driver could bring the train in an unsafe situation). However, we support the OS mode in scenarios, in which we can force the train to perform a given not allowed movement, as proceeding after its eoma; in this way, we have been able to reproduce step 2 of scenario 8.

Inconsistencies in the Scenarios. We identified some inconsistencies in some scenarios that report wrong rules for the reported state transitions of the VSS. In particular, in steps 6 and 7 of scenario 8, rule #2A is used to modify VSS22 and VSS23 from UNKNOWN to OCCUPIED; however, rule #2A goes from FREE to OCCUPIED. In other cases, instead, we think that the scenarios are not precise and they do not mention an additional transition that must be taken before the reported one:

- in step 8 of scenario 6, rule #6A is used to modify VSS23 from AMBIGUOUS to FREE (however, rule #6A goes from OCCUPIED to FREE); we think that the requirements imply that rule #11 must be taken before.
- in step 5 of scenario 8, rule #11A is used to modify VSS21 from UNKNOWN to OCCUPIED (however, rule #11A goes from AMBIGUOUS to OCCUPIED); we think that the requirements imply that rule #5 must be taken before. The same issue appears in step 7 of scenario 9 for VSS21.

5 Related Work

At the time of writing, we are not aware of any formalization and/or validation and verification of the Hybrid ERTMS/ETCS Level 3 system.

Regarding the Spin model checker, it has been applied to the modelling and verification of different safety-critical systems[6]; Havelund et al. [10], for example, applied it to the verification of the multithreaded plan execution module of an artificial intelligence-based spacecraft control system architecture part of the DEEP SPACE 1 mission.

A common approach in model checking models developed in high-level notations is to exploit existing model checkers as Spin, NuSMV, UPPAAL, etc. For example, in [3], the authors discuss the advantages of using high-level notations in hardware design. They observe that HW designers are used to high level notations as Bluespec and they have problems when dealing with the lower level notations; the authors claim that the notation of the verification tool should be transparent to the designer, who should specify the model and the properties in the same high level notation, without caring about the intricacies of the notation of the verification language. With this aim, a common approach is to automatically translate high level models in models of existing model checkers; this requires to define the mapping from the source notation to the target notation, and also a reverse translation of the counterexamples returned by the model checkers in concepts of the source notation.

However, such translation often introduces an overhead that affects the scalability of the verification; for example, this is reported for translation of ASMs to NuSMV [2], of UML models to Promela [7], and of Simulink models to NuSMV [16].

On the other hand, a more recent approach is to develop model checkers directly handling the high level notation, as ProB that directly model checks B models. In [14], the model checker ProB is compared with Spin. The author notices that, whenever the number of states of a B model and a Promela model are the same (models developed for the same problem), Spin outperforms ProB of several orders of magnitude, as Spin performs verification directly in C and it employs several optimizations (e.g., partial order reduction, bitstate hashing), while ProB uses an interpreter written in Prolog. On the other hand, the author also shows that, in some cases, the ProB model checker behaves better as it avoids the state explosion occurring in Spin (if the `atomic` construct is not used), it employs a mixed depth-first breadth-first strategy, and it exploits symmetries present in high-level models.

6 Conclusions

In this paper, we proposed the modelling, validation, and verification of the Hybrid ERTMS/ETCS Level 3 Case Study in Spin. The tool allowed us to model all the requirements of the case study, reproduce all the scenarios reported in the case study document, and verify the model. We have shown that, although very powerful in terms of verification, Spin misses some facilities (logging and scenario specification) that could help in debugging and validating the model. For this work, we devised an approach to encode scenarios that, however, requires to

[6] http://spinroot.com/spin/success.html.

modify the model itself; as future work, we plan to design a language for writing scenarios (as test cases) and implement a tool that, given a model and a scenario for it, drives the SPIN simulation over the model as specified in the scenario.

References

1. Abrial, J.: Modeling in Event-B - System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Arcaini, P., Gargantini, A., Riccobene, E.: AsmetaSMV: a way to link high-level ASM models to low-level NuSMV specifications. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 61–74. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11811-1_6
3. Arvind, D.N., Katelman, M.: Getting formal verification into design flow. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 12–32. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68237-0_2
4. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-642-18216-7
5. Carioni, A., Gargantini, A., Riccobene, E., Scandurra, P.: A scenario-based validation language for ASMs. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 71–84. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87603-8_7
6. Hybrid ERTMS/ETCS Level 3. Technical report, EEIG ERTMS Users Group, July 2017
7. Chen, J., Cui, H.: Translation from adapted UML to promela for CORBA-based applications. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 234–251. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24732-6_17
8. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: an opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_29
9. Glossary of terms and abbreviations. Technical report, ERA * UNISIG * EEIG ERTMS USERS GROUP, May 2016
10. Havelund, K., Lowry, M., Penix, J.: Formal analysis of a space-craft controller using SPIN. IEEE Trans. Softw. Eng. **27**(8), 749–765 (2001)
11. Hoang, T.S., Butler, M., Reichl, K.: The hybrid ERTMS/ETCS level 3 case study. Technical report (2018)
12. Holzmann, G.J.: The SPIN Model Checker - Primer and Reference Manual. Addison-Wesley, Boston (2004)
13. Ladenberger, L., Bendisposto, J., Leuschel, M.: Visualising event-B models with B-motion studio. In: Alpuente, M., Cook, B., Joubert, C. (eds.) FMICS 2009. LNCS, vol. 5825, pp. 202–204. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04570-7_17
14. Leuschel, M.: The high road to formal validation. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 4–23. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87603-8_2
15. Leuschel, M., Butler, M.: ProB: an automated analysis toolset for the B method. Int. J. Softw. Tools Technol. Transf. **10**(2), 185–203 (2008)

16. Meenakshi, B., Bhatnagar, A., Roy, S.: Tool for translating simulink models into input language of a model checker. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 606–620. Springer, Heidelberg (2006). https://doi.org/10.1007/11901433_33
17. Prigent, A., Cassez, F., Dhaussy, P., Roux, O.: Extending the translation from SDL to Promela. In: Bošnački, D., Leue, S. (eds.) SPIN 2002. LNCS, vol. 2318, pp. 79–94. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46017-9_8

Using a Formal B Model at Runtime in a Demonstration of the ETCS Hybrid Level 3 Concept with Real Trains

Dominik Hansen[1(✉)], Michael Leuschel[1], David Schneider[1], Sebastian Krings[1], Philipp Körner[1], Thomas Naulin[2], Nader Nayeri[2], and Frank Skowron[2]

[1] Institut für Informatik, Heinrich-Heine-Universität Düsseldorf, Düsseldorf, Germany
{hansen,leuschel,schneider,krings,korner}@cs.uni-duesseldorf.de
[2] Thales Deutschland GmbH, Berlin, Germany
{thomas.naulin,nader.nayeri,frank.skowron}@thalesgroup.com

Abstract. In this article, we present a concrete realisation of the ETCS Hybrid Level 3 concept, whose practical viability was evaluated in a field demonstration in 2017. Hybrid Level 3 (HL3) introduces Virtual Sub-Sections (VSS) as sub-divisions of classical track sections with Trackside Train Detection (TTD). Our approach introduces an add-on for the Radio Block Centre (RBC) of Thales, called Virtual Block Function (VBF), which computes the occupation states of the VSSs according to the HL3 concept using the train position reports, train integrity information, and the TTD occupation states. From the perspective of the RBC, the VBF behaves as an Interlocking (IXL) that transmits all signal aspects for virtual signals introduced for each VSS to the RBC. We report on the development of the VBF, implemented as a formal B model executed at runtime using ProB and successfully used in a field demonstration to control real trains.

Keywords: B-method · Animation · Model-based testing · ETCS

1 Introduction and Requirements

The specification "Hybrid ERTMS/ETCS Level 3" (HL3) [1] describes a novel train control concept, incorporating classical trackside train detection, radio-based position reports, and train integrity information. The main difference between the HL3 concept and a solution without any trackside train detection (pure Level 3) is that not all trains need to be equipped with an ETCS on-board unit and a TIMS (Train integrity monitoring system). In addition, the information from the underlying trackside train detection system can be used as fall back to, e.g., handle degraded situations and to improve the performance.

In June 2017 the Heinrich Heine University Düsseldorf (HHU) was asked by Thales Deutschland GmbH to contribute to a field demonstration of feasibility of the ETCS Hybrid Level 3 principles. The call for tender was initiated by ProRail

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 292–306, 2018.
https://doi.org/10.1007/978-3-319-91271-4_20

Netherland, with a demonstration planned on a test track at the ETCS National Integration Facility (ENIF), provided by Network Rail (UK) for December 2017.

This resulted in the present cooperation between Thales and HHU, with additional support provided by ClearSy. The goal was to develop an executable version of the HL3 specification, called Virtual Block Function (**VBF**), which is an add-on for the existing Thales Radio Block Centre (**RBC**) without adapting the RBC core functionalities. The main idea is that the VBF partitions each Trackside Train Detection section (**TTD**) into Virtual Sub-Sections (**VSS**). For the RBC, the track is thus decomposed into finer grained sections compared to the TTDs. The VBF computes the occupation status of each VSS by using the TTD occupation status and train position reports including train integrity information. For example, in Fig. 1 at the bottom you can see that we have two areas each with a trackside detection device (realised by axle counters or track circuits). The VBF knows that the left one is occupied and the right one is free. However, for the RBC it simulates the existence of six areas and six trackside detection devices. Based on the train position information, the VBF can already free part of the occupied left track for following trains, enabling higher throughput without having to install additional trackside equipment.

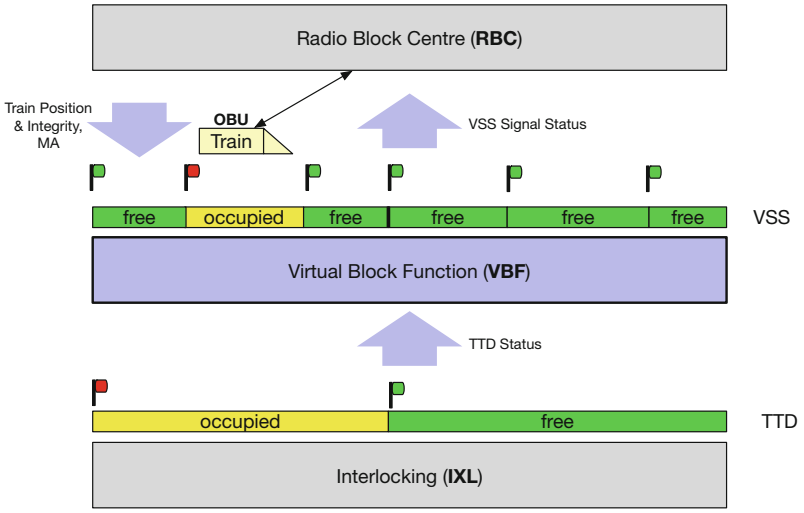


Fig. 1. The role of the VBF (Virtual Block Function)

In the following sections, we will report on our experience building a software product for the VBF based on a formal B model. In Sect. 2 we outline our tasks and early design decisions. Section 3 provides an overview of the formal B model and the modelling challenges, along with some ambiguities and inconsistencies we found in the HL3 specification. Section 4 describes the architecture of VBF

software which embeds the B model. Visualisation was important in our project and we discuss it Sect. 5. We conclude with discussion about practical results and insights gained in Sect. 6.

2 Project Constraints and Design Decisions

Due to the strict deadline and the very short time span for the project, it was decided to use off-the-shelf RBC and interlocking systems and use a **formal B model** [2] of the VBF as an **executable demonstrator**. More precisely:

- The Thales RBC core was to be used as is, without modifications for HL3. (Thales owns a product line for the RBC software to configure the generic software to the project specific requirements).
- The interlocking was used as is, without modifications for HL3.[1]
- The VBF had to be developed from scratch as an add-on for the RBC, which was to mimic an interlocking and transmits the signal aspects for the virtual signals to the RBC. The VBF contains a VSS state machine, with four possible states (free, occupied, unknown and ambiguous) for each VSS, exactly as required by the HL3 specification.

The following main tasks are the focus of this paper:

T1: Providing evidence that the HL3 principles are consistent and complete to handle possible hazards and to allow the desired operational behaviour.
T2: Implementation of the VBF as an independent software unit by supporting the given interfaces to the other components. The implementation should be conform to the HL3 principles.

To accomplish the first task, we decided to derive a formal B model from the HL3 specification. The decision was based on diverse work (e.g., [3–9]) which provided evidence that B is well suited for the railway domain. Moreover, first experiments were very promising: in a few days it was possible to model some simpler transitions of the HL3 specification.

For task T2, we intended to implement all interfaces (boundaries) to other components by hand and to use a classical testing approach to ensure their correct functioning. To reuse the formal model from task T1 for task T2, we had three options:

1. Using the model as a template to implement the VBF core by hand.
2. Generating code from the model and combine this code and the handwritten boundaries.
3. Executing the model at runtime by incorporating the execution engine and the handwritten boundaries.

[1] Except for the TTD occupation status which has to be send from the IXL to the VBF/RBC.

The first option would require us to maintain both the model and the code. This could be time-consuming if there were changes to the specification (due to feedback from ProRail, the specification was changed considerably). With the second approach, we would have to use an existing code generator (there was no time to develop our own) and thus have to refine our abstract B model down to implementation level B0—also time-consuming. Concerning the third option, we had already gained some experience of integrating ProB [10] as the execution engine in different software products [11,12]. Given our time constraints, the third option was the only feasible option, but it also posed the biggest research challenge: using a formal model at runtime interacting with various hardware and software components.

3 The Formal B Model

Below, we present some relevant aspects of our B model along with some source code snippets. Due to space limitations we cannot cover all interesting aspects, such as the modelling of timers and time.

3.1 Basic Datatypes

The modelling of the track was relatively straightforward, which is not surprising since B's relations can be used to represent graphs and B provides many convenient operators on relations and functions, which are just a special case of graphs (see, e.g., Chap. 14 of "Modeling in Event-B" [13]).

However, for pragmatic reasons, we did not use Event-B [13] but rather classical B [2] for modelling the VBF. For example, we have modelled the VSSs, TTDs and trains as classical B strings. For simulation and execution purposes, we had to read topology and configuration data from XML files. The conversion of the XML file into B data structures for the VBF model is also done in classical B using records and strings.[2] Finally, we have used other features, such as machine composition and operation calls (see Sect. 3.2), not readily available in Event-B.

Below, we try to give a flavour of our modelling by showing some derived data structures for the track topology.

```
PROPERTIES
  VSS : POW(STRING)
& TTD : POW(STRING)
& VSS /\ TTD = {}
& next_vss : VSS +-> VSS
& vss_ttd: VSS --> TTD // maps VSS to their TTD
& TTD_STATE = {free,occupied} // TTDs only have two states
& next_ttd : TTD +-> TTD
& last_vss: TTD --> VSS
```

[2] The conversion is not shown in this paper since the XML data format is proprietary.

```
& /*@label "the last vss is part of its TTD" */
  !t.(t:TTD => vss_ttd(last_vss(t)) = t)
& /*@label "a successor of a last vss is in another TTD" */
  !(t,n).(t:TTD & last_vss(t)|->n : next_vss => vss_ttd(n) /= t)
...
```

For example, the `next_vss` constant is a partial function which links VSS to their successor VSS. The direction of the track is thus constant for any given execution run.[3] However, the direction of the track can be toggled, since the conversion of the XML data is parameterised. Observe that we allow the IF-THEN-ELSE to be applied to expressions and use an external B function (see Sect. 6.3 in [11]) to read in the track data from an XML file.

```
PROPERTIES
   TRACK_DATA =  READ_XML("./resources/prj_ENIF_01@STR.xml")
...
 & C_VSSSequence = DeriveVSSSequence(TRACK_DATA)
...
 & next_vss = UNION(i, ii).(
    i : dom(C_VSSSequence) & ii : dom(C_VSSSequence) & ii = i + 1
    | {IF RUNNING_DIRECTION = "LEFT_TO_RIGHT"
       THEN C_VSSSequence(i)  |-> C_VSSSequence(ii)
       ELSE C_VSSSequence(ii) |-> C_VSSSequence(i) END
     } )
```

Train Status. Modelling the integrity state of trains revealed some ambiguities and inaccuracies within the HL3 specification. The concept "integer" (for a train) is used in different contexts within the specification. We try to explain the differences with the aid of our model:

```
SETS
   REPORTED_TRAIN_INTEGRITY = {lost_integrity, confirmed_integrity,
                               no_integrity_information}
 ; INTERNAL_TRAIN_INTEGRITY = {integer, not_integer}
PROPERTIES
 TRAIN_INTEGRITY_MAPPING = {
  "TRAIN_INTEGRITY_CONFIRMED_BY_INTEGRITY_MONITORING_DEVICE"
                                        |-> confirmed_integrity,
  "TRAIN_INTEGRITY_CONFIRMED_BY_DRIVER"  |-> confirmed_integrity,
  "NO_TRAIN_INTEGRITY_AVAILABLE"         |-> no_integrity_information,
  "TRAIN_INTEGRITY_LOST"                 |-> lost_integrity}
...

INVARIANT
```

[3] Every scenario in the HL3 specification only has a single linear track with trains running in one direction. Points are not considered by the current version of the HL3 specification and they were not required for the field tests at ENIF.

```
  registeredTrains : POW(STRING) &
& train_reportedTrainIntegrity
                   : registeredTrains --> REPORTED_TRAIN_INTEGRITY
& train_integrity  : registeredTrains --> INTERNAL_TRAIN_INTEGRITY
...
```

According to the ERTMS/ETCS specifications [14], a train can send four possible integrity status values within a train position report, which are represented by the domain of the constant `TRAIN_INTEGRITY_MAPPING`. Within the VBF, we only need to distinguish between three, which are represented by the enumerated set `REPORTED_TRAIN_INTEGRITY`. The surjective function `TRAIN_INTEGRITY_MAPPING` defines the respective mapping.

Moreover, the HL3 specification [1, Sect. 3.5] defines a further integrity state by using the terms "integer" and "not integer" which is represented by the enumerated set `INTERNAL_TRAIN_INTEGRITY`.[4] Yet, an unambiguous mapping from the reported train integrity to the internal train is missing in the HL3 specification [1]. Thus, we were forced to find a sensible interpretation; we defined the following two conditions as triggers for the transition from "integer" to "non-integer":

- *"train reports 'lost integrity"'*
- *"PTD [Positive Train Detection] with no integrity information is received outside of the integrity waiting period"*

Both conditions are part of the transitions #7B and #8A [1, Sect. 5.1.1.6]. The change of the train length (the remaining condition of #7B and #8A) does not affect the internal integrity status of a train but can have a consequence for VSS states as it triggers the "train integrity propagation timer" of the VSSs where the train is located.

The following operation manipulates the internal train integrity variable in our model:

```
Train_SetIntegrityStatus(train, integrityStatus) =
  PRE integrityStatus : REPORTED_TRAIN_INTEGRITY
  THEN
    train_reportedTrainIntegrity(train) := integrityStatus ||
    IF integrityStatus=lost_integrity
    THEN  train_integrity(train) := not_integer
    ELSIF integrityStatus = confirmed_integrity
    THEN  StartTimerDelta(train|->WAIT_INTEGRITY_TIMER)
          || train_integrity(train) := integer
    ELSIF // no information available
      train |-> WAIT_INTEGRITY_TIMER : expiredTimers
    THEN train_integrity(train) := not_integer
    END
  END
```

[4] The term "internal" refers to the internal state of the VBF.

However, the model checker ProB directly reported an invariant violation. This is because a train does not register itself by a train position report, thus the variable `train_reportedTrainIntegrity` is not a total function with the registered trains as its domain. As a consequence, we had to make a further decision by treating a train as `non_integer` before the VBF receives the first position report (interpretation to the safe side). We always tried to avoid partial functions as it would mostly introduce handling of special cases. Moreover, the description in the HL3 specification is imprecise regarding when to start the first "wait integrity timer": *"A 'wait integrity timer' runs continuously for every train [...]"* [1, Sect. 3.4.1.3.1]. We decided to start the timer with first train position reported but not with the registration.

We found a further inaccuracy with regard to the integrity status in the specification: *"For an integer train the confirmed rear end location of the train is derived from [...]"* [1, Sect. 3.3.3.1]. Here, the term "integer train" is used which corresponds to the internal train integrity of our model. However, in Sect. 3.3.3.4 it is stated that *"the confirmed rear end of the train location is never updated by position reports with integrity status 'Lost' or 'No information available"'* [1, Sect. 3.3.3.4]. Thus, Sect. 3.3.3.1 of the specification should rather start with *"For a train which reports confirmed integrity"* since a train can be integer while reporting "No integrity information available".

Train Location. Another essential concept in HL3 specification is the definition of the train location (in our case the image of the train location seen by the VBF) which is frequently referred within the state machine transitions of the HL3 specification. We mapped each registered train to a set of VSS within our model:

```
INVARIANTS
  ...
  & train_location : registeredTrains --> POW(VSS)
  & /*@label The train location must not have any gaps */
    !loc.(loc: ran(train_location)
      => #s.(s : iseq(loc)
            & !i,ii.(i : 1..size(s-1) => s(i) |-> s(i + 1) : next_vss)))
```

In most cases, we just want to know if a certain train is located on a certain VSS. For these cases, the data structure for `train_location` is very convenient. Alternatively, we could have used a relation but we prefer functions over relations except for the `next_vss` constant which is frequently inverted in our model. The order of the VSS is not incorporated into the location definition as this information is already contained in the `next_vss` constant. The condition that a train location must not have any gaps (which is not explicitly mentioned in the HL3 specification) can also easily be expressed with the aid of this constant.

While the modelling of the train location data structures was relatively straightforward, the updates to this variable are, in our opinion, the most underspecified part of the HL3 specification. Some issues referring to the location are:

- Minor: *"As long as the TTD where the max safe front end is reported is free, the train location is not extended onto the VSS which are part of this free TTD"* [1, Sect. 3.3.2.1.2]. This is imprecise as the condition should be: only if the max safe front is reported to be on the next free TTD but not the estimated front of the train.
- Fundamental: *"[. . .] the train location is derived from the estimated front end [...] of the last position report [. . .] as well as from TTD information [. . .]."* Is the train location only updated/changed by processing train position reports (in this case the TTD information will of course be considered)? Or does a single TTD change event without a train position report also update the train location? We had tried both alternatives and in the end we decided to use a train position report as the only trigger to update the train location. (The other alternative, forced us to adapt several transitions in order to be able to replay all scenarios of the HL3 specification.)

3.2 State Machine Transitions and Priorities

Below, we show the B translation of the state machine transition (#9A) of the HL3 specification.

```
DEFINITIONS
 Guard9A(vss) == vss:VSS & vss_state(vss) = ambiguous
  & /*@label "(TTD is free)" */
    ttd_state(vss_ttd(vss)) = free
...
OPERATIONS
VSS_Ambiguous_Free_9A(vss) =
  SELECT
    Guard9A(vss)
  THEN
    vss_state(vss) := free ||
    // state of the virtual signal which protects the vss
    vss_signalState(vss) := PROCEED ||
    ...
  END
```

The reason for separating out the guards into DEFINITIONS (in a separate file) is to encode the priorities of the HL3 specification. We have experimented with various ways of encoding the priorities, and have finally pursued a solution based on using a large IF-THEN-ELSE with the guards as conditions, calling respective operations of a subsidiary machine. The IF-THEN-ELSE ensures that the priorities of the transitions are respected, e.g., that transition `2A` has priority over `3`. A return variable `out` stores the exact VSS transition taken for debugging and analysis.

```
out <-- VSSUpdateStep(vss) = PRE vss : VSS
  THEN
    IF Guard1A(vss) THEN VSS_Free_To_Unknown_1A(vss) || out := "1A"
```

```
    ELSIF Guard1B(vss) THEN VSS_Free_To_Unknown_1B(vss) || out := "1B"
    ...
    ELSIF  #train.( train : registeredTrains & Guard11B(vss, train) )
      THEN
        ANY train WHERE train : registeredTrains & Guard11B(vss, train)
        THEN
          VSS_Ambiguous_Occupied_11B(vss, train) || out := "11B"
        END
    ELSE
      out := "NONE"
    END
  END
```

Execution of all VSS updates in a VBF cycle is done by a B WHILE loop calling `VSSUpdateStep`.

3.3 Animation of Scenarios

The HL3 document describes a number of scenarios in addition to the VSS state machine. We used these scenarios as test specifications, i.e., to check that these scenarios are feasible in our model (detection of inconsistencies).

To animate the scenarios with ProB, we developed an environment model and composed it with the VBF core model (software model) to obtain a system model. The environment model has knowledge of the "real" (physical) position of a train, which allows it to move the train and to send train positions reports which are inputs of the VBF. Figure 2 shows a system state where the "real" position differs from the train position within the VBF. In this case, the physical train has already moved to VSS21 and the VBF still sees the train in VSS12. Note that this is a very common situation as trains usually only send its position cyclically (e.g., each 6 s). Otherwise, this state can be seen as the situation where Train1 has already sent its position report but the VBF has not yet received it due to the delays of the communication interface.

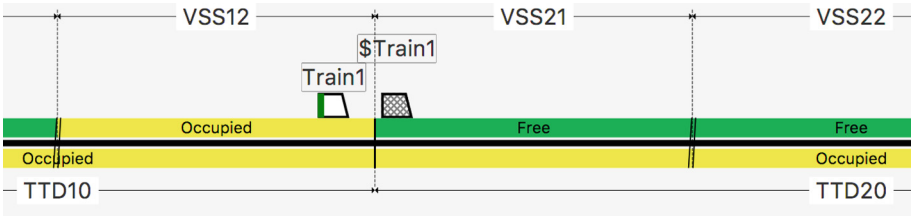


Fig. 2. Environment Model: "physical" train position ($Train1) vs. train position image in the VBF (Train1)

In summary, with the environment model it is possible to trigger all interfaces of the VBF by generating the following inputs:

- Train position reports including train integrity information
- Train registration message
- Train deregistration message
- Train data message (includes the train length)
- TTD occupation status
- Movement Authorities (MA) for trains

The environment model can make use of different tracks. For example, we used the track snippet from the HL3 specification to validate its scenarios and used the real track for onsite execution and to define a test plan for onsite execution.

While animating the scenarios of the HL3 specification, we detected more issues.[5] One issue, which is easy to understand but hard to find without tool support, is the following: in scenario 4 (Start of Mission/End of Mission) at step 8, it is stated that all VSS of TTD 20 go to "unknown" because the disconnect propagation timer of VSS 22 has expired. This is wrong because after the deregistration of the train in step 7, the train will be immediately treated as a ghost train and the corresponding transition #1A will apply. The result for the remaining VSSs of TTD20 is the same but at a different point in time; the VSSs go directly to "unknown" and not just after the disconnect propagation timer (of VSS22) has expired. As an aside, we think that transition #1A is erroneous, too: there should be an "and" instead of the "or" in *"(no FS MA is issued or no train is located on this TTD)"*. Otherwise, a connected train (with a FS MA) which physically enters a free TTD would always be treated as a ghost train because the TTD occupation usually arrives before a new train position report. In this case, the second condition *"no train is located on this TTD"* would be fulfilled which would allow applying transition #1A.

Besides the validation of the scenarios, the environment model permitted us to specify system level invariants. For example, the system state shown in Fig. 3 should never occur. Here, a physical train ($Train2), which is not connected, is located on a VSS which is seen as "free". The threat in this situation is that another train (not displayed in the figure) in rear of the non-connected train could receive a movement authority (FS MA) for VSS31 and VSS32. We were able produce a scenario which finally led to this state caused by an invalid stopping criterion for the ghost train propagation.[6]

Replaying Recorded Runs with ProB. Simulations runs (with On-Board-Unit simulators) as well as demonstration runs (with real trains) were logged by the VBF and could be replayed in the animator. This was vital, as it allowed us to analyse defects without inspecting (huge) RBC, IXL and Java log files. Log replay was also used to define timer values of the HL3 specification.

[5] Overall we detected more than 30 issues which we reported to authors of the HL3 specification.

[6] The scenario is too complex to be presented in this paper.

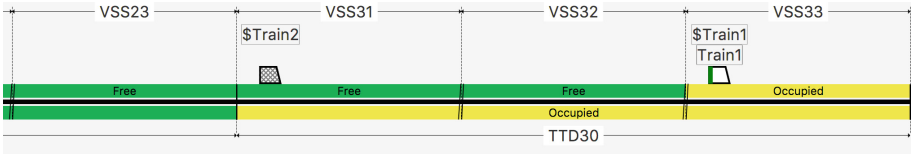


Fig. 3. Invalid system state: Non-connected train ($Train2) is located on a VSS with state "free".

4 Architecture

The VBF model described above is part of a larger application developed to conduct the field demonstration. The application embeds the VBF model using the ProB Java API [15] (often referred to as ProB2) and manages all the model's interactions with the outside world. The Java API exposes all of ProB's animation and model checking features to programs running on the Java Virtual Machine. This approach has been successfully used in several applications that use B models at runtime [12] and is the basis for a new ProB UI that is currently being developed.

The responsibilities of the application are: firstly, to interact with external input sources such as the RBC and others that provide information about the current state of the track, of physical and of simulated trains, etc. Secondly, to process these inputs and forward them to the model. And lastly, to act on the newly computed state of the model to update the visualisation and send updates to the RBC.

Figure 4 provides an overview of the application's architecture. The external inputs are provided via a variety of inputs, such as UDP packages, XML-RPC calls, plain files, etc. These inputs represent train information from the RBC as well as TTD information from real and simulated trains. These events are

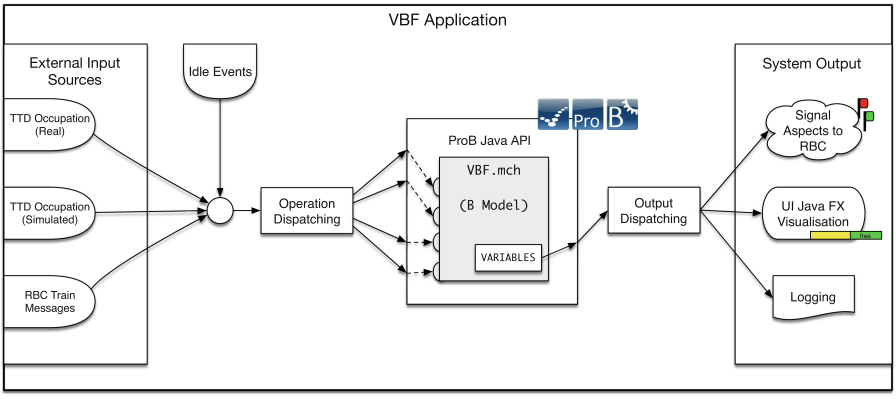


Fig. 4. Application architecture

received by the application, normalised and dispatched to the model. In case there are no external events, the application will, after a given delay, begin sending idle events to the model in specific intervals until it receives new external events. These events are used to update the timers in the model and compute an updated system state even in the absence of external events. Each type of input event is dispatched to a corresponding operation of the B machine by executing one guided animation step and computing a new state of the model. From each new model state computed by ProB, we derive an application-level state representation. This representation is based on the state variables of the model. These variables are exposed through the ProB Java API and extracted from the state, mapped to Java structures and used to compute the application's outputs. From this application-level state the signal aspect changes are extracted and sent to the RBC. The state is provided to the visualisation layer to update the track diagram and information tables. Lastly, the delta between two states is logged for debugging purposes.

5 Visualisation

One requirement for the actual onsite field demonstration was to provide a visualisation for checking the correct functioning of the VBF. Additionally, our experience has shown [16–18] that a visualisation combined with an interactive animator can be especially useful in early stages of the development such as the modelling and analysis stage.

Thus, our intention was to develop one visualisation that could be used in the early stages and during the field demonstration. As a consequence, the visualisation was developed as a separated software component with clearly defined interfaces for it to be integrated both into the ProB-Animator and the final VBF product. In both cases, the state information is extracted from the same (core) model. The only difference is that within the ProB-Animator the model is interactively controlled via an environment model by a user and in the final VBF software, the model is controlled via the real interfaces of the VBF.

Having the visualisation in the early stages of the project provided the following benefits:

- We quickly spotted mistakes in the specification and the model.
- We used the visualisation to communicate the model within our team and to the domain experts.
- We were able to replay the scenarios in the HL3 specification and detected inconsistencies between them and the state machine description.
- The visualisation enabled us to let a domain expert act as a tester by interactively inspecting the model.

For the project, we have also developed a new feature in ProB, namely to export an entire animation trace into an HTML file with one visualisation per state. This feature was useful to send entire animation scenarios to domain experts.

For the main application, we created a custom visualisation using the JavaFX UI framework. The visualisation is linked to the B model's state, and updates itself as soon as a new state is provided. As such, the same visualisation could be used as a plugin in the ProB-Animator during development.

Figure 5 shows a screenshot of the VBF visualisation running as a plugin in the ProB-Animator.

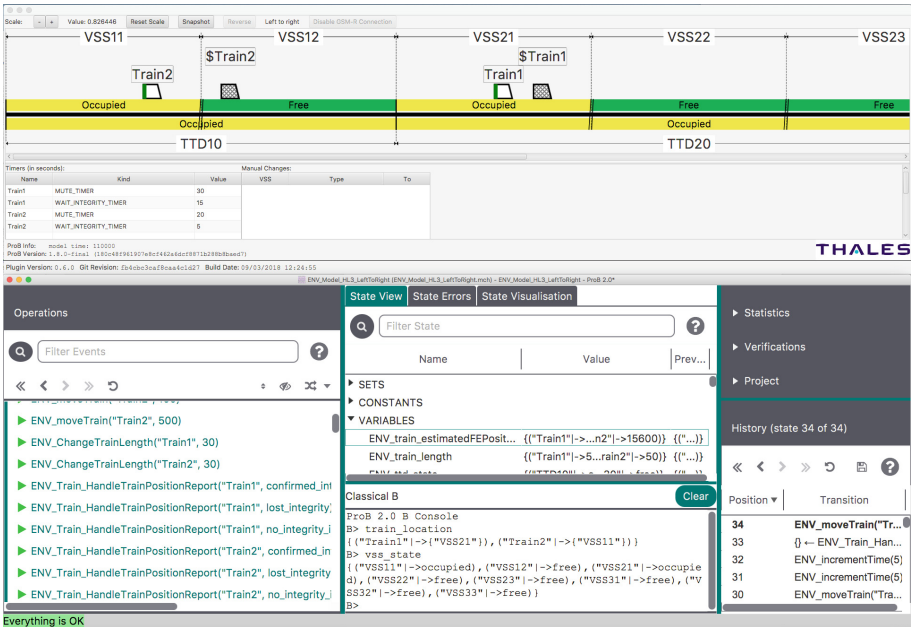


Fig. 5. Screenshot of the visualisation running as a ProB-Animator plugin

6 Practical Results, Discussion, Conclusion

Building upon the Thales domain knowledge, the formal B model was developed from July until the end of October (including the embedding application), with fine-tuning performed afterwards. A first integration with the Thales RBC was carried out in the beginning of November. The field demonstrations were carried out in November and December 2017. The VBF demonstrator was finished on time and on budget, and the demonstration of the HL3 principles using the Thales RBC was successful. The VBF model (without environment) consists of 13 B Machines, 14 definition files and has 45 constants and 28 variables. The required scenarios were demonstrated, with simulated and real trains. Five persons from HHU worked on the VBF demonstrator (two on the formal B model, three on the boundaries and the visualisation). Also, within the project, some ProB extensions were developed.

ProB had two different roles in our project. Its first role was, as described in Sect. 4, the execution engine for our B model. From the formal methods perspective, it is interesting to note that the B model can be used to control simulated and real trains in real time. Moreover, no problems with ProB occurred at runtime, performance and memory consumption were no issues.[7] In addition, the ProB Java API turned out to be a flexible way to link a formal model to external data sources or components.

In its second, more common role, ProB was the central tool in the validation process of the model and specification. Animation combined with visualisation were crucial for the success of the project, in particular to replay and validate the scenarios of the HL3 specification. We think this approach, of using animation and custom visualisations at every stage of development – especially the early ones – should be more widely used for safety critical (e.g., SIL 4) projects in industry. For example, the specification engineer can take over some work of the testing team as he is able to interactively derive test cases from the model[8], which are much more precise and consistent compared to the description of the scenarios contained in the HL3 specification.

From the project, we can conclude that formal models can be useful and cost-effective for demonstrators. Animation with forward/backward stepping and visualisation were extremely useful in the development process. We were able to develop a complete formalisation of the HL3 specification: the B formal model can now serve as an *executable reference specification*, for understanding the HL3 principles, for deriving test cases from it or possibly to generate code using Atelier-B.

Acknowledgements. Jens Bendisposto, David Geleßus, Christoph Hein-zen, Antonia Pütz, Yumiko Takahashi, Fabian Vu and Michelle Werth for all the work that went into the ProB Java API and the new ProB-Animator UI. We thank Mirko Aigner, Stefano Allrath, Burkhard Börner, Joachim Jost, Editha Nentzl, Sebastian Neuhau, Michael Schilling, Wilfried Seibt, Tom Seidel and Tino Wegner from Thales as well as the staff from ClearSy for their work and support on the demonstrator. Moreover, we are thankful to the authors of the HL3 specification and the reviewers of ABZ for their useful feedback.

References

1. Hybrid ERTMS/ETCS Level 3. Principles Ref: 16E042, Version: 1A, EEIG ERTMS Users Group, 123–133 Rue Froissart, 1040 Brussels, Belgium, 7 2017
2. Abrial, J.-R.: The B-Book. Cambridge University Press, Cambridge (1996)
3. Dollé, D., Essamé, D., Falampin, J.: B dans le transport ferroviaire. L'expérience de siemens transportation systems. Tech. Sci. Inform. **22**(1), 11–32 (2003)

[7] For example, in one 6-min run ProB's response time was—with one exception—between 0.03 and 0.14 s per event. One event required 0.31 s, possibly due to garbage collection being triggered.

[8] Note that we talk here about product and system level tests and not just unit tests.

4. Essamé, D., Dollé, D.: B in large-scale projects: the canarsie line CBTC experience. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 252–254. Springer, Heidelberg (2006). https://doi.org/10.1007/11955757_21
5. Leuschel, M., Falampin, J., Fritz, F., Plagge, D.: Automated property verification for large scale B models with ProB. Formal Asp. Comput. **23**(6), 683–709 (2011)
6. Lecomte, T., Burdy, L., Leuschel, M.: Formally Checking Large Data Sets in the Railways. CoRR, abs/1210.6815 (2012)
7. Sabatier, D., Burdy, L., Requet, A., Guéry, J.: Formal proofs for the NYCT Line 7 (flushing) modernization project. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) ABZ 2012. LNCS, vol. 7316, pp. 369–372. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30885-7_34
8. Sabatier, D.: Using formal proof and B method at system level for industrial projects. In: Lecomte, T., Pinger, R., Romanovsky, A. (eds.) RSSRail 2016. LNCS, vol. 9707, pp. 20–31. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33951-1_2
9. Comptier, M., Déharbe, D., Perez, J.M., Mussat, L., Pierre, T., Sabatier, D.: Safety analysis of a CBTC system: a rigorous approach with Event-B. In: Fantechi, A., Lecomte, T., Romanovsky, A. (eds.) RSSRail 2017. LNCS, vol. 10598, pp. 148–159. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68499-4_10
10. Leuschel, M., Butler, M.: ProB: a model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45236-2_46
11. Hansen, D., Schneider, D., Leuschel, M.: Using B and ProB for data validation projects. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) ABZ 2016. LNCS, vol. 9675, pp. 167–182. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33600-8_10
12. Schneider, D., Leuschel, M., Witt, T.: Model-based problem solving for university timetable validation and improvement. In: Bjørner, N., de Boer, F. (eds.) FM 2015. LNCS, vol. 9109, pp. 487–495. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19249-9_30
13. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
14. ERTMS/ETCS Baseline 3. System Requirements Specification Ref: SUBSET-026-3, Issue: 3.0.0, EEIG ERTMS Users Group, 123–133 Rue Froissart, 1040 Brussels, Belgium, December 2008
15. Bendisposto, J., Clark, J., Dobrikov, I., Körner, P., Krings, S., Ladenberger, L., Leuschel, M., Plagge, D.: ProB 2.0 Tutorial. In: Proceedings of the 4th Rodin User and Developer Workshop, TUCS Lecture Notes, Turku, June 2013. Turku Centre for Computer Science
16. Ladenberger, L., Bendisposto, J., Leuschel, M.: Visualising Event-B models with B-motion studio. In: Alpuente, M., Cook, B., Joubert, C. (eds.) FMICS 2009. LNCS, vol. 5825, pp. 202–204. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04570-7_17
17. Ladenberger, L.: Rapid creation of interactive formal prototypes for validating safety-critical systems. Ph.D. thesis, University of Düsseldorf, Germany (2017)
18. Hansen, D., Ladenberger, L., Wiegard, H., Bendisposto, J., Leuschel, M.: Validation of the ABZ landing gear system using ProB. In: Boniol, F., Wiels, V., Ait Ameur, Y., Schewe, K.-D. (eds.) ABZ 2014. CCIS, vol. 433, pp. 66–79. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07512-9_5

Validating the Hybrid ERTMS/ETCS Level 3 Concept with Electrum

Alcino Cunha and Nuno Macedo(✉)

INESC TEC & Universidade do Minho, Braga, Portugal
nfmmacedo@di.uminho.pt

Abstract. This paper reports on the development of a formal model for the Hybrid ERTMS/ETCS Level 3 concept in Electrum, a lightweight formal specification language that extends Alloy with mutable relations and temporal logic operators. We show how Electrum and its Analyzer can be used to perform scenario exploration to validate this model, namely to check that all the example operational scenarios described in the reference document are admissible, and to reason about expected safety properties, which can be easily specified and model checked for arbitrary track configurations. The Analyzer depicts scenarios (and counter-examples) in a graphical notation that is logic-agnostic, making them understandable for stakeholders without expertise in formal specification.

1 Introduction

The *European Rail Traffic Management System* (ERTMS) is a system of standards for management and interoperation of signalling for railways by the EU, that aims to replace the various national systems with a seamless European railway system[1]. The *European Train Control System* (ETCS), the ERTMS control command part, defines 3 levels of signalling that a system can operate on, depending on the trackside equipment used, how the on-board systems communicate with the trackside, and on which functions are processed on-board or by the trackside. In Level 3, positive train detection (PTD) information, including the train position and integrity information, is detected and reported by the on-board system directly to the trackside, which, based on logical rather than physical track block sections, decides whether it is safe to issue movement authorities (MA), reporting them back to the on-board system via radio. By removing the need for physical trackside detection, the implementation cost is reduced, while the use of virtual blocks allows for arbitrarily small sections, improving the performance and adaptability of the system.

For Level 3 to be feasible, PTD information must be reliable and the communication between the on-board and trackside systems guaranteed at all times. These pre-conditions are not easily met, which has led to the proposal of a Hybrid Level 3 concept [2], that combines PTD information with limited trackside detection. These trackside train detection sections (TTD) are then broken into smaller

[1] http://www.ertms.net/.

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 307–321, 2018.
https://doi.org/10.1007/978-3-319-91271-4_21

virtual subsections (VSS). Each of these VSSs, besides being identified as free or occupied, may also become ambiguous or unknown, whenever discrepancies in the information are detected. This allows for trains with non-ideal equipment or with communication problems to still use the line, albeit below full capacity.

This paper reports on the modelling and subsequent validation and verification of the Hybrid ERTMS/ETCS Level 3 (HL3) concept in Electrum [5], and was developed as an answer to the ABZ 2018 call for case study contributions. Electrum is a lightweight formal specification language that extends Alloy [4] with mutable relations and temporal logic operators. The result is a language as simple and flexible as Alloy, but with improved support for the specification of reactive systems and for the model checking of safety and liveness LTL properties. Its Analyzer [3] provides support for both bounded (through SAT likewise Alloy) and unbounded (through SMV) model checking, whose solutions (or counter-examples) are presented back to the user in a unified graphical interface.

The resulting HL3 model, as well as relevant design decisions, are presented in Sect. 2. Electrum concepts are presented as needed. Section 3 describes how the model was validated using the Analyzer, including the encoding of the operational scenarios, while Sect. 4 explores some desirable safety properties that can be automatically verified. Section 5 discusses the results and some identified challenges, and Sect. 6 points directions to future work. It should be noted that the authors had no *a priori* domain knowledge, and that this work was mainly based on the provided "*Principles*" document for the HL3 [2].

2 Modelling

The section presents an Electrum model for the HL3 concept, which is available online[2]. Proper abstraction is key to achieve a model that is representative of the system under study but that is still prone to being automatically analysed for relevant properties and easily understood by all interested parties. In the HL3, the main abstraction points arise from the mismatch between certain continuous aspects of the rail traffic management domain and the necessarily discrete nature of state-based modelling languages like Electrum. These include concerns with train length changes, as well as real-time issues related to communication delays and the use of timers to optimize the performance of the system. Relevant design decisions regarding such issues are explained as the model is presented. The Electrum language is also presented by example throughout the section. The formal presentation of its syntax and semantics are presented elsewhere [5].

2.1 Static Structural Components

In Electrum, likewise Alloy, structure is introduced through the declaration of *signatures*, that represent sets of uninterpreted atoms, and *fields*, that create relationships between multiple atoms. In Electrum such signatures may either

[2] http://haslab.github.io/Electrum/ertms.ele.

```
open util/ordering[VSS] as V
open util/ordering[TTD] as D

enum State
  { Unknown, Free, Ambiguous, Occupied }

sig VSS {
  var state   : one State,
  var jumping : lone Train
}

sig TTD {
  start : VSS,
  end   : VSS,
} { end.gte[start] }

fact trackSections {
  all ttd:TTD-D/last |
    ttd.end.V/next = (ttd.D/next).start
  first.start = V/first and last.end = V/last
}

fun VSSs[t:TTD] : set VSS {
  t.start.*V/next&t.end.*(~V/next)
}
fun parent[v:VSS] : one TTD {
  max[(v.*V/prev).~start]
}
```

```
sig Train {
  var pos_front : one VSS,
  var pos_rear  : one VSS,
  var MA        : one VSS
}
var sig connected in Train {}
var sig report_front,report_rear in Train {}

fun mute : set Train {
  Train-(report_rear+report_front)
}
fun disintegrated : set Train {
  report_front - report_rear
}

fun MAs[t:Train] : set VSS {
  knownRear[tr].*V/next & (tr.MA).*V/prev
}

fun knownFront[t:Train] : one VSS {
  { v:VSS | t not in report_front since
    (t in report_front and v = t.pos_front) }
}
...
fun occupied : set TTD {
  { d : TTD |
    some VSSs[d]&Train.(pos_rear+pos_front) }
}
```

Fig. 1. Excerpt of the structure of the HL3 model.

be static (by default) or variable (those marked as **var**), and can be restricted by simple multiplicity constraints. Static signatures represent the possible configurations on which a system can act, and although they can still be loosely defined and solved during the analysis process, they stay frozen throughout the evolution of the system. In the HL3 model, as shown in Fig. 1, these represent the available trains (signature `Train`) and the valid configurations of trackside train detection sections (signature `TTD`) and virtual subsections (signature `VSS`). Tracks are simply comprised by discrete sequences of VSS atoms, which in Electrum can be achieved by imposing a total order. Fields `start` and `end` simply register exactly **one** `VSS` in which a `TTD` starts and ends, respectively. This representation does not consider any particular dimension of the blocks or trains, which essentially affects reasoning about the minimum safe rear end position that occurs in transition #11A and the start event of the ghost propagation timer [2].

Relational expressions combine such signatures and fields (and other constants) using standard relational operators like union (`+`), intersection (`&`), difference (`-`), join (`.`) or the binary converse (~), or with transitive (`^`) or reflexive-transitive (`*`) closure operators. Primed expressions refer to their value in the succeeding state. Relational expressions can also be constructed by comprehension. Primitive relational formulas are either inclusion (**in**) or equality (`=`) tests, or basic multiplicity tests, which can be combined through common Boolean operators, first-order quantifications or future and past LTL operators. Such formulas can be imposed as axioms that always hold in a model through *facts*

such as `trackSections`, which universally quantifies over `TTD` elements to guarantee that the complete track is partitioned. *Functions* and *predicates* can be used for reusable expressions and formulas, respectively. For instance, function `VSSs` calculates all the subsections of a `TTD` through transitive closure operations, possible due to the imposed total order on `VSS`. This declarative definition allows for the analysis of properties over every valid track partition within a finite universe, a cumbersome task in languages without support for first-order logic.

2.2 Dynamic Structural Components

The dynamic structure of a model consists of the mutable elements of the system, those whose state evolves in time. In Electrum such signatures and fields are declared as **`var`**. In the HL3 model, as depicted in Fig. 1, these regard the physical state of the trains and the state of the trackside and the on-board systems.

Each train has an exact physical position (not necessarily known by the trackside) for its front and rear ends, represented by variable fields `pos_front` and `pos_rear`, that point to exactly **`one`** `VSS` at each time. Variable sub-signatures are used to represent the state of the PTD communication between each train and the trackside. Variable signature `connected` represents trains connected to the trackside at each instant, which will be modified by start (SoM) and end of mission (EoM) events, while `report_front` and `report_rear` denote which trains reported front and rear information at that instant, respectively. Dynamic auxiliary function `mute` identifies all trains not communicating in each instant, while function `disintegrated` identifies trains that failed to report their integrity (no rear report). Other auxiliary functions combine this information to retrieve the currently known information about a train. For instance, `knownFront` retrieves the last reported front position of a train using the past operator **`since`**.

Besides PTD information, an optimization is implemented in HL3 to detect the position of trains transitioning between TTDs, in order to avoid delays due to "jumping trains" [2, p. 12]. This information is assumed to exist by the VSS state machine (namely transition #2B). The field `jumping` on VSSs registers such occurrences and is also used when retrieving train position information; its content is fixed by a fact omitted from the excerpt.

HL3 proposes a state machine for VSSs, that combines the TTD and PTD information to determine the current state of a VSS, which is then used to issue MAs. This is encoded by field `state` that at each instant assigns to each VSS an element from the enumeration `State`. Unlike PTD information, the state of TTDs is considered safe: if there is a train located within it, it is reported as being occupied. Although this communication may have delays, we have opted to make it instantaneous and exact, as defined by function `occupied`. Implementing such delay would be straight-forward, by encoding an action that denotes whether TTD information has been received in each state, likewise the train PTD reporting predicates that will be presented in Sect. 2.3. However, this would increase the complexity of the model considerably and have little impact in the behaviour specified in [2] (it slightly affects Scenario 5 as there is no delay on detecting the free state of TTD20). Finally, the trackside system registers the

```
var sig disconnect_ptimer in VSS {}
var sig integrity_loss_ptimer in VSS {}

var sig shadow_timer_A in TTD {}
var sig shadow_timer_B in TTD {}
var sig ghost_ptimer in TTD {}

var sig mute_timer in Train {}
var sig integrity_timer in Train {}

pred set_mute_timer {
  mute_timer in mute
}
pred set_integrity_timer {
  integrity_timer in disintegrated
}
```

```
pred set_shadow_timer_A {
  shadow_timer_A in start_shadow_timer_A
}
fun start_shadow_timer_A : set TTD {
  { ttd : TTD | once {
      previous ttd in occupied
      ttd not in occupied
      previous ttd.end.state = Ambiguous } }
}
...
pred set_timers {
  set_mute_timer and set_disconnect_ptimer
  set_integrity_timer and set_integrity_loss_ptimer
  set_shadow_timer_A and set_shadow_timer_B
  set_ghost_ptimer
}
```

Fig. 2. Excerpt of the timers specification in the HL3 model.

end of the current `MA` for each train. All the VSSs that comprise a train's MA are calculated by the dynamic functions `MAs` using transitive closure operators.

To avoid performance deterioration due to communication fluctuations, HL3 implements a set of timers to avoid unnecessary state transitions. Each of these timers has start and end events, and is assigned to either a VSS, a TTD or a train. To model whether such timers have expired, for each of these elements variable sub-signatures were introduced, as depicted in Fig. 2. For instance, variable sub-signature `mute_timer` contains at each instant the trains for which that timer is expired. A predicate for each type of timer denotes whether the start conditions have been met. For those with dual start and stop events, this is straight-forward. For instance, a `mute_timer` may be triggered if a train is mute. Other timers, like the shown `shadow_timer_A`, must query over every previous state whether the start condition was met using the past operator **`once`**. Predicate `set_timers` aggregates all these predicates. The reference document states that, once expired, timers remain so until the start conditions are met again [2, p. 14]. Yet, this behaviour renders some of the scenarios inconsistent (see Sect. 3.3), so we have opted not to implement it. No particular time duration was imposed on timers, so only the possibility of expiration is modelled, and not its enforcement. Since each step does not represent any particular real-time interval, the free expiration allows for the designer to test different interleavings. Electrum has limited support for integers, which could allow for the eventual codification of real-time timers. However, during analysis these are translated into their bitwise representation so that they can be handled by the SAT solvers, which encumbers the process for complex integer expressions or larger integer values.

2.3 System Evolution

The system evolves as the trains move and report PTD information, and the trackside updates the states of the VSSs and the trains' MAs. Actions that model this behaviour can easily be represented in Electrum as declarative predicates that relate the current state of variable elements with the succeeding one using primed expressions. More advanced actions may freely use LTL operators.

```
pred move [t:Train] {
  t.pos_front' in t.pos_front.(iden+V/next)
  t.pos_rear' in t.pos_front'.(iden+V/prev)
  t.pos_rear' in t.pos_rear.(iden+V/next)
  { t in connected
    t in report_rear' => t in report_front'
  } or {
    t not in report_front'
    t not in report_rear' }
  t in connected iff t in connected'
}
```

```
pred som[t:Train] {
  t not in connected
  connected' = connected + t
  report_rear' = report_rear + t
  report_front' = report_front + t
  pos_front' = pos_front
  pos_rear' = pos_rear
}
```

Fig. 3. Excerpt of the train evolution of the HL3 model.

```
pred states[vss:VSS] {
  vss.state' = (
    n01[vss]   =>  Unknown else
    n02[vss]   =>  Occupied else
    n03[vss]   =>  Ambiguous else
    n04[vss]   =>  Free else
    n12[vss]   =>  Occupied else
    n05[vss]   =>  Ambiguous else
    n06[vss]   =>  Free else
    n07[vss]   =>  Unknown else
    n08[vss]   =>  Ambiguous else
    n09[vss]   =>  Free else
    n10[vss]   =>  Unknown else
    n11[vss]   =>  Occupied else
                   vss.state )
}
```

```
pred n09 [v:VSS] {
  v.state = Ambiguous
  after (n09A[v] or n09B[v])
}
pred n09A [v:VSS] {
  parent[v] not in occupied
}
pred n09B [v:VSS] {
  some t:Train {
    t not in disintegrated
    v not in knownLoc[t]
    previous v not in knownLoc[t]
    parent[v] not in shadow_timer_A
    parent[v] in start_shadow_timer_A }
}
```

Fig. 4. Excerpt of the VSS state machine specification in the HL3 model.

A train in the developed model can be updated by 4 events, some of which are presented in Fig. 3. SoM (`som`) and EoM (`eom`) actions simply connect or disconnect a train to the trackside. A `split` action models the breaking up of a train into two, affecting its integrity. A two-carriage train is modelled by two trains that have had exactly the same state up to that point; during break up, the front one will fail to report the rear position, resulting in lost integrity, and the rear one will be disconnected from the trackside. Finally, the `move` action updates the physical position of the train and may or not report PTD information to the trackside. To keep the evolution of the system manageable, the train is allowed to move forward at most one subsection in each step, and the rear is always kept at most one subsection away from the front, although these restrictions could easily be relaxed. A disconnected train never reports to the trackside, while connected ones may or not do so; reports lacking rear information will model integrity loss events. Although MA policies are beyond the HL3 concept, it is assumed that trains may move outside assigned MA for operational reasons [2, p. 6]. Our model assumes that a connected train moves within its MA, while disconnected ones may disregard it. Notice that most of these actions are encoded as declarative predicates that allow for a range of behaviours at each instant. For simplicity purposes, all trains are assumed to be in the track at all times (multiplicity **one** on positions), so trains may not enter or leave the track. Modelling such

```
pred MAs {
  all t:connected-mute_timer |
    t.MA' = t.MA or (knownFront[t].*next&t.MA'.*prev).state = Free or after OS[t]
  all t:(Train-connected)+mute_timer |
    t.MA' = t.MA or t.MA' = knownFront[t]
}
...
fact trace {
  always {
    set_timers and MAs and all v:VSS | states[v]
    (all t:Train | move[t]) or (some t1,t2:Train | split[t1,t2] or som[t1] or eom[t1]) }
}
```

Fig. 5. Excerpt of the MA assignment and trace specification in the HL3 model.

behaviour is easily done by creating additional "dummy" VSSs at the beginning or end of the track, as in Scenarios 8 and 9.

When processing PTD reports, [2, p. 11] assumes that the front and rear end reports are independent events, with the front one always being processed first in case of simultaneous reports. Forcing this behaviour at all times would however double the number of steps in the generated traces, which would possibly encumber the solving process. Thus, besides the independent processing of front information (represented by reports without rear information), our model also allows the simultaneous processing of front and rear reports. In fact, in the operational scenarios, PTD reports are collapsed into a single step, and the only scenario where this phenomenon is relevant is Scenario 9; in this case the reporting event was forced to be split into two steps in its encoding, the first missing rear information. Lastly, there are 3 events in [2] related to the integrity of the train that trigger the same VSS state transitions (#7B and #8A) and the integrity loss propagation timer (a train reports lost integrity, changed train length or its wait integrity timer expired); as these always occur in conjunction, they were abstracted into a single condition where the train fails to report the rear information, which simplified the model without affecting the overall behaviour.

Predicate `states` in Fig. 4 updates the state of the VSSs by encoding the state machine defined in [2, p. 6]. Depending on the current state of each VSS and on the available PTD and TTD information, each transition condition is tested in an order that preserves the imposed priorities. As an example, the condition for transition #9 between ambiguous and free states is depicted in Fig. 4. Due to the complexity (and occasional ambiguity) of these conditions, this construction process was iterative with the encoding of the operational scenarios (Sect. 3.2). Section 3.3 discusses some potential issues detected in [2] in this process.

The assignment of MAs is outside the scope of the HL3 concept [2], but the validation of the model requires that some reasonable, even if loose, policy is encoded. Its declarative definition (predicate `MAs` at Fig. 5) allows for alternative behaviours. For connected trains, either the MA remains unchanged or is updated to a VSS that is only separated from the front end of the train by free VSSs. To model the on-sight (OS) operational mode, that gives full privileges to the driver, the MA may also be set to the last VSS of the track (used in

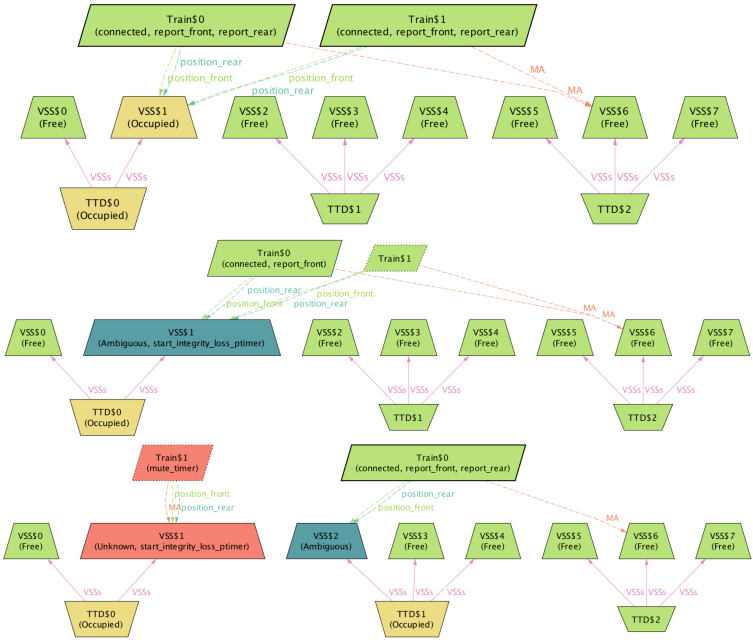


Fig. 6. 3 succeeding steps of the HL3 operational Scenario 2.

Scenarios 6 and 8). For disconnected trains, either the MA is preserved, or removed altogether (by assigning it the currently known position of the train).

All the actions are enforced through a fact `trace` (Fig. 5) that guarantees that all solution traces are created from the application of these actions. These usually encode interleaving semantics of actions, but since this would lead to an explosion of steps in each trace, we allow all trains to move in each step.

3 Validation

A conceptual model must be validated against the requirements and with other relevant stakeholders. The Electrum Analyzer provides support to generate solutions to the model that satisfy provided properties, allowing for the specification and exploration of scenarios, as well as providing a logic-agnostic graphical visualizer.

3.1 Scenario Visualisation

The Electrum Analyzer provides a graph visualiser for depicting the found instances, whose appearance can be customisable through themes. This is essentially an extension to the Alloy Analyzer to natively support infinite temporal traces through loopbacks. These logic-agnostic graphical instances are understandable for stakeholders without expertise in formal specification, and have

previously proven to be suitable for establishing a common interpretation of the requirements [7]. We focused on providing a visualisation theme that allowed both software designers and ERTMS/ETCS domain experts to communicate through a common scheme.

The Analyzer's theme editor provides basic customisation functionalities (e.g., changing the shape, colour and border of different signatures and fields). Additional customizations can be performed by defining functions that return sets of elements, whose result is calculated at static time by the visualizer. This enables, for instance, drawing a VSS according to its current state, by creating functions that for each state retrieves, by comprehension, VSSs elements for which that state holds, e.g., `fun occupied : set VSS {{vss:VSS | vss.state = Occupied}}`.

Given the theme customizations, the Alloy Analyzer applies a graph representation algorithm and distributes nodes among layers, a process that is oblivious of the underlying semantics of the nodes and edges. The only mechanism available to the user to change the shape of this graph is to reverse the direction of edges. In our HL3 model, this resulted in a graph that, although layered into TTDs, VSSs and trains, did not preserve the order on TTD and VSS blocks, hindering the readability of scenarios. To overcome this, we implemented a small modification of the Electrum Analyzer where information regarding totally ordered sets (`TTD` and `VSS` in HL3) is passed down to the visualizer and, when possible, used to order such elements in the same graphical layer.

Using the developed theme[3], the appearance of HL3 instances and counterexamples in the Electrum Analyzer is that of the snapshot in Fig. 6 for the operational Scenario 2. Both TTD sections and VSS subsection appear layered and ordered, with different colours depending on their current state (a textual label is also present). A train representation depicts (textually and graphically) its position, reporting status and MA. Expired timers are also depicted. Figure 6 in particular denotes a `split` event, where two trains with a shared state break up, leaving one disconnected (`Train$0`) and the other moving forward.

3.2 Modelling the Operational Scenarios

Electrum specifications can be animated through **`run`** commands that, given a desirable property and a finite scope for the declared signatures, automatically search for satisfying instances. Each signature scope denotes the maximum (or **`exactly`** the) number of elements that will be considered by the Analyzer. When performing bounded model checking, the maximum trace length that will be considered is imposed by a scope on **`Time`**. Once a solution is found, additional non-isomorphic solutions can be efficiently navigated through the Analyzer.

The HL3 concept [2] provides a set of operational scenarios that proved essential to validate the model during development. All 9 scenarios were encoded as predicates in Electrum in order to guarantee that our model was not overconstrained, and were used as regression tests for any succeeding modifications.

[3] http://haslab.github.io/Electrum/ertms.thm.

```
pred S2 {
  let v11 = V/first, v12 = v11.next, v21 = v12.next, ... {
  some disj t1,t2:Train {
    split[t1,t2]
    always t1.MA = v32
    t1 in report_front;t1 in report_front;...
    t1 in report_rear and after (t1 not in report_rear and after (...))
    t1.pos_front = v12;t1.pos_front = v12;...
    t1.pos_rear = v12 and after (t1.pos_rear = v12 and after (...))
    ... } }
}
```

Fig. 7. Excerpt of the Scenario 2 specification in the HL3 model.

These can be automatically generated by the bounded model checking procedures of the Analyzer through the **run** commands available in the provided Electrum model. Using the provided theme, these can visualized in a style similar to the one presented in Fig. 6 for Scenario 2. The outcome of all 9 scenarios can be consulted in Electrum's website[4]. Some inconsistencies between the VSS state machine and the operational scenarios were also detected during this process, which are discussed in Sect. 3.3.

Specifying concrete instances with several steps in Electrum is verbose, since LTL does not allow the reference to concrete time instants, requiring the creation of formulas with nested **after** operators. This was manifest when developing the HL3 model, where every scenario has at least 8 steps. This led us to explore potential language extensions to help specifying such scenarios, including the introduction of a new operator that acts as syntactic sugar during the specification of the traces: rather than `p and after (q and after r)` one can now simply write `p;q;r`. Figure 7 presents an excerpt of the predicate encoding Scenario 2, with rear information encoded with standard LTL operators and front information with the new operator. Running this predicate, which results in the trace depicted in Fig. 6, can be done through the following command:

```
run S2 for 8 Time, exactly 2 Train, exactly 3 TTD, exactly 8 VSS
```

All operational scenarios have 3 TTD sections and 8 VSS subsections and either 1 or 2 trains, so the scopes can be bound exactly in the commands. At the beginning of the development of HL3, scope **Time** denoted the maximum trace lengths that would be explored by the bounded analysis procedures, such that a scope n on **Time** would launch an iterative process where traces up to n are checked. This is important, since the absence of a counter-example for length n does not entail its absence for some $m < n$. However, in the HL3 model we are aware of the exact number of steps that comprises each scenario, and, since this number is not particularly small (at least 8 states), the incremental iterative process encumbers the solving process. Thus, the Analyzer was adapted to support ranges or exact bounds for trace lengths, allowing for the faster generation of scenarios. It should also be noted that according to bounded model checking semantics [1], evidences for **always** constraints (or counter-examples to **eventually** ones) require infinite traces, represented as a finite prefix that

[4] https://github.com/haslab/Electrum/wiki/ERTMS.

loops back into itself. Thus, scenarios must not deadlock at the last state, but somehow loop back into a previous state. This is not possible for every state (e.g., Scenario 9), meaning that the trace length scope may need to be increased.

Notice that the scenario predicates do not completely fix the states. Instead, they focus on establishing the movement of the train (as well as some timer and MA events) and leave the VSS state machine act freely, whose state will be solved by the Analyzer.

Electrum is also useful to explore scenarios with looser restrictions, when the user wants to reason about model instances that satisfy certain properties. For instance, to explore whether the existence of jumping trains is problematic (recall that field `jumping` registers the occurrence of jumping trains) one can simply run:

```
run {eventually some jumping} for 8 Time, 3 Train, 3 TTD, 8 VSS
```

Alternative solutions, with arbitrary track configurations within the scope, can then be quickly iterated, helping the user detect problematic instances.

3.3 Possible Issues with the HL3 Concept

Model validation allowed us to detect possible ambiguities or under-specifications in the HL3 concept. Note that this analysis is essentially based on [2] without any *a priori* domain knowledge. Two of these issues regard the VSS state machine triggering conditions, namely #1A and #5A, that when codified as described in the document result in a behaviour that does not match that of the operational scenarios. Condition #1A triggers the transition between a free VSS into unknown whenever the parent TTD is occupied without a train located *or* without MA assigned. Yet some scenarios do not reflect this behaviour, like Scenario 7, where VSS33 should transition to unknown since no train is located in the occupied TTD30. Removing the second disjunct (or converting the condition into a conjunction) results in the expected behaviour. Transition #5A between unknown and ambiguous should be triggered whenever a train is *located* in the VSS. For the remainder transitions, "located" was assumed to denote the last known position of the train. Yet, several scenarios break under this interpretation for #5A, like VSS22 at Scenario 4 that remains unknown even though the last reported position of the train was that VSS. Only considering trains reporting to be in that VSS in that instant matches the scenarios' behaviour.

Another issue regards the indefinite expiration of timers. Although [2, p. 14] states that expired timers remain expired until the start conditions are met again, this behaviour does not seem to be followed in the operational scenarios. For instance, in Scenario 9, if the ghost propagation timer remains expired, VSSs at TTD30 should transition from free to unknown according to #1F.

4 Verification

Proper validation increased our confidence that the model effectively abstracts the behaviour specified in the HL3 concept. The next logical step is to verify

whether such model behaves as expected. Similar to the **run** commands, **check** commands in Electrum instruct the Analyzer to search for instances that break a certain assertion within a fixed scope. However, there is no explicit notion of correctness defined in [2]. Moreover, this correctness is dependent on behaviour that is outside the scope of [2], namely the policy for extending and shortening MAs, as well as how the train acts upon those MAs. As a consequence, this exercise was mainly exploratory, although we hope that these preliminary results can foment the discussion among domain experts and lead to more formally defined safety requirements for implementations of the HL3 concept.

A reasonable correctness property is that, if PTD communication never fails and the integrity of the trains is never compromised, then no states other than free or occupied are assigned to the VSSs. In fact, it should be the case that every VSS with a train on it is set as occupied and the others as free. Recall that we had already imposed two (reasonable, in our perspective) assumptions regarding MAs in Sect. 2.3: (*i*) trains connected to the trackside always move within the assigned MAs, and (*ii*) to connected trains, the trackside will assign MAs between the currently known position and a succeeding free VSS or grant an OS MA. Proving these properties required the additional restriction (*iii*) that OS MAs are never assigned. This should be expected since this would allow trains to freely move ignoring trackside information. Electrum allows the definition of assertions as regular formulas, which given these pre-conditions can be encoded as:

```
assert trains_Occupied {
  (init and always
      (no mute and no disintegrated and no t:Train | after OS[t])) =>
    always Train.(pos_front+pos_rear).state = Occupied}
check trains_Occupied for 8 VSS, 3 TTD, 2 Train, exactly 12 Time
```

where state predicate `init` forces all trains to be reporting and the track to have a consistent state in the initial state.

More interesting safety properties allow failures in communication or non-integral trains, which necessarily involves reasoning about timers. Recall that our model, in order to be flexible, did not impose any particular duration on the timers, i.e., the number of steps that the starting condition must hold in order to the timer to expire. Since timer duration necessarily affects the correctness of the system, our safety assertions assumed a conservative approach where every timer expires instantaneously (predicate `auto_timer`). Guaranteeing these properties required however the additional pre-condition that (*iv*) disconnected trains do not move outside the assigned MA. This allowed us to show that, even with problematic trains, the state is correctly assigned to the VSSs, for instance, that the free state is never assigned to a VSS with a train on it:

```
assert timers_Free {
  (init and always (auto_timer and
     (all t:Train | t.pos_front in MAs[t] and not (after OS[t]))) =>
    always Train.pos_front.state != Free}
```

More complex assertions could test alternative timer durations and reason about possible interleaving issues among different kinds of timers.

5 Discussion

Being an extension to Alloy, it is important to compare the verbosity and readability of Electrum models with those developed in standard Alloy. Thus, a similar encoding of the HL3 concept was developed in Alloy as well[5], which, given the complexity of the case study, enabled us to clearly picture the cons and pros of the two languages. The static structural components of the system are identical in either Electrum or Alloy. Differences arise when modelling the dynamic components, as Alloy requires time, evolution and dynamic properties to be explicitly modelled. This would require the explicit declaration of the signature `Time` and the conversion of all variable signatures and fields to a state idiom [4], where, e.g., field `pos_train` would be declared with type `VSS one → Time`. Then, temporal formulas must explicitly quantify over time instants. For instance the `trains_Occupied` assertion in Alloy would take the shape:

```
assert trains_Occupied {
  (init[first] and all s:Time | no mute[s] and
      no disintegrated[s] and no t:Train | OS[s.next,t])) =>
    all s:Time | Train.((pos_front+pos_rear).s).(state.s) = Occupied}
```

The tradeoff is that Electrum does not allow quantification over time instants. In most cases there is an alternative encoding for such expressions using standard LTL. For instance, in Alloy one can retrieve the last state `s` in which a train reported, treat it as a first-level entity throughout relational formulas and expressions, and use it to query the state of the system at that state, as in a function that retrieves the VSSs currently known to be occupied by a train:

```
fun knownLoc[s:Time,t:Train] : set VSS {
  let s1 = max[s.*prev&t.report_front] |
  t.pos_front.s1 + t.pos_rear.s1}
```

Electrum does not allow explicit references to time instants, but the same behaviour was encoded using the **since** past operator (Fig. 1). Other expressions are necessarily more verbose in Electrum. For instance, evaluating a field `r` over every instant except `t` can be encoded in Alloy as `r.(Time-t) = a`, while in Electrum it would have to be expanded into a sequence of **after** expressions.

The Analyzer allowed for the automatic generation of scenarios and checking of assertions through bounded and unbounded model checking. All analyses were run in a quad-core Intel Core i5-4200U Haswell with 4 GB RAM, the bounded relying on MiniSAT 2.2.0 and the unbounded on nuXmv 1.1.1. All the 9 scenarios were generated by bounded model checking procedures, since their exact trace length is known, with performance times ranging from 20 s for Scenario 1 to 276 s for Scenario 9. The safety properties presented in Sect. 4 were verified

[5] http://haslab.github.io/Electrum/ertms.als.

through both bounded and unbounded model checking, since the latter provides additional correctness guarantees but has worse scalability. For instance, property `trains_Occupied`, for 5 `VSS`, 2 `TTD` and 2 `Train` elements is verified by the bounded procedure in 49 s and by the unbounded in 1273 s. Note that such analysis considers every possible track configuration with that number of sections, 8 for this scope. Previously we proposed an automatic decomposed solving strategy [6] that solves each of these configurations in parallel, which allowed the unbounded performance to be cut down to 73 s. A larger scope, with 8 `VSS` and 3 `TTD` elements can be verified in 20 m by the bounded procedure, analysing all 42 valid track configurations.

Bounded model checking can sometimes have unpredictable effects for those unaccustomed with its semantics. As already reported, the infinite traces imposed by global constraints forbid deadlocks at the last state, forcing the trace to loopback into a previous state. Since this not necessarily true in every trace, it may lead to unexpected unsatisfiable commands and longer traces. A related issue regards the use of past-time operators which, due to the finite nature of the trace and the alternative past state when reasoning at the loopback state, can also lead to unpredictable behaviour.

6 Conclusions

The complexity of the HL3 concept has tested Electrum and its Analyzer to their limits, allowing us to fully explore their potential and identify possible improvements and future lines of research. Some improvements (minor changes to the visualizer, a new temporal operator for formulas over traces, more control on the scope of trace lengths) were also implemented throughout the development of the HL3 model. The proposed model could still be further developed to allow reasoning about some HL3 aspects that were abstracted in the current version, including delays on TTD reports and forcing independent front and rear reports, although we expect them to have a considerable toll on performance.

The definition of the operational scenarios was the most cumbersome task, so we are currently exploring potential extensions to the language to ease that process, including variants of temporal logic with support for intervals that would allow the definition of properties over ranges of steps. It should be noted however that Electrum's (and Alloy for that matter) greatest strength is on the exploration of scenarios, and not the specification of fixed instances. Although we advocate that the current graphical feedback can be understood by stakeholders without background on formal specification, we also believe that there is room for improvement. We are currently working on techniques specifically tailored for the visualization and animation of traces.

We hope that this preliminary work can help clarify some ambiguities in the HL3 concept and motivate the ERTMS/ETCS community to explore the potential of formal specification and analysis methodologies.

Acknowledgements. The authors would like to thank David Chemouil for the support provided during the model checking of the model. This work is financed by the ERDF - *European Regional Development Fund* through the *Operational Programme for Competitiveness and Internationalisation* - COMPETE 2020 and by National Funds through the Portuguese funding agency, FCT - *Fundação para a Ciência e a Tecnologia* within project POCI-01-0145-FEDER-016826.

References

1. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Adv. Comput. **58**, 117–148 (2003)
2. EEIG ERTMS Users Group: Hybrid ERTMS/ETCS Level 3 - Principles (2017)
3. INESC TEC, ONERA: Electrum Analyzer, v1.0. Available Under the MIT License (2018). https://github.com/haslab/Electrum/releases/tag/v1.0
4. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2012). revised edn
5. Macedo, N., Brunel, J., Chemouil, D., Cunha, A., Kuperberg, D.: Lightweight specification and analysis of dynamic systems with rich configurations. In: SIGSOFT FSE. pp. 373–383. ACM (2016)
6. Macedo, N., Cunha, A., Pessoa, E.: Exploiting partial knowledge for efficient model analysis. In: D'Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 344–362. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_23
7. Moreira, J.M., Cunha, A., Macedo, N.: An ORCID based synchronization framework for a national CRIS ecosystem. F1000Research 4(181) (2015)

The ABZ-2018 Case Study with Event-B

Jean-Raymond Abrial(✉)

Marseille, France
jrabrial@neuf.fr

1 Introduction

Usually, case studies dealing with train systems concentrate on the safety of trains circulating on tracks equipped with points and crossings and protected by means of traffic lights and speed limit sign postings as in [1–3]. Train drivers are supposed to follow such indications. The goal of formal approaches used in such case studies is to prove that trains may circulate safely on such tracks provided drivers act correctly. This is done by constructing models of such complex systems and by using formal proof techniques.

The case study presented in this paper addresses different issues. Points and crossings are not incorporated in this case study and there are no traffic lights nor speed limit sign postings. In fact, drivers are still in charge of trains but some centralised structures called *tracksides* ensure train protections. There are various tracksides disposed along the track. Each of them is responsible for a certain portion of the track. All this is made possible through some radio connections between trains and tracksides and also by the presence of some on-board as well as trackside numerical systems.

So far, so good. But, unfortunately such radio connections and numerical systems might fail or be temporarily or permanently out of order. Moreover, it is quite possible that trains without any connections nor numerical devices (e.g. freight trains) be circulating on such tracks together with other well equipped trains. The challenge then is to ensure that trains can circulate safely in spite of these difficulties. This is the general purpose of this case study. A more precise scope for this case study is presented in Sect. 2.5.

The system under study is called *Hybrid ERTMS/ETCS Level 3* (European Real Traffic Management System/European Train Control System). The adjective *Hybrid* is due to the presence of additional physical equipment along the track in, so-called *TTD* sections (Trackside Train Detection sections). Such physical equipment are able to detect the presence of all trains on corresponding sections, in particular those which are not equipped with radio connections. Such additional equipment are in principle not necessary for handling well equipped connected trains. However, since they are there, tracksides use them as auxiliary aids.

The *reference document* on which this case study is based is [4]. This document is well written and apparently very complete. However, I found it rather difficult to master, even after numerous reading sessions. This is the reason why I propose a synthesis in the next section. This synthesis presents what I learned from such readings. It is then quite sure that this corresponds to some **simplifications** of the case study. It is also quite possible that my understanding is

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 322–337, 2018.
https://doi.org/10.1007/978-3-319-91271-4_22

not correct in some places. Anyway, what I will present in this paper is exactly what is written in the next section.

The rest of this paper is organised as follows: in Sect. 2, as I already said above, I give my own informal explanations about the requirements of this system, Sect. 3 contains the reference requirements, Sect. 4 gives an idea on how refinements could be structured, Sect. 5 contains the development of the formal model, finally Sect. 6 propose some highly subjective comments of mine on this system.

2 Requirement Document: Explanations

In this section, I propose an informal synthesis of the information needed to study this system. The reason for doing this synthesis is again that the reference document is very analytical. As a matter of fact, explanations for a given information are often spread in many different places of the 48 pages of this document. For example, details concerning the notion of *train integrity* can be found in more than 40 different places in the reference document.

Again, I do not pretend that descriptions contained in this section are correct according to the reference document. They just correspond to what I concluded after careful readings of this document. The formalisation described in this paper will correspond to what is presented in this section.

2.1 Main Components of the System

The system under study is made of three entities: the *trackside*, the *track* and the *trains*. See details in the coming subsections.

2.1.1 Tracksides

The *trackside* is the entity in charge of determining precisely the position of trains situated on the part of the track which the trackside is supposed to control. The trackside has a second role: that of sending some information to connected trains. Such information concern the ability of trains to proceed or not on the track. Note that this second role of the trackside is not part of this case study. As a matter of fact, I assume that safety is ensured by this information and that trains will follow them.

The best positions of trains, as recorded by the trackside, are called VSS (Virtual sub-sections). VSS are continuously ordered *virtual* sub-sections of continuously ordered *physical* TTD sections (Trackside Train Detection sections) of the track. Each VSS belongs to a unique TTD section. The various VSS of a TDD section are continuously numbered. Conversely, each TTD section has at least one VSS in it. See more information on VSS and TTD sections in Sects. 2.1.2 and 2.1.3 below.

TTD and VSS are made *free* or *occupied* by the trackside depending on the positions of trains on them. We shall see in Sect. 2.4 that VSS might have two more states.

In the best case, where a train is connected to the trackside, VSS which are occupied by a train are computed by the trackside from some information

received from trains under the form of a, so-called, *position*. In other cases, where trains are unable to send information to the trackside, such positions are nevertheless known to the trackside by means of physical equipments situated along the track within each physical TTD section. In this latter case, positions of the train on the trackside are thus not given by VSS anymore but rather by TTD sections.

2.1.2 Tracks

The *track*, under the control of the trackside, is made of various ordered physical TTD sections (Trackside Train Detection sections). As already explained in Sect. 2.1.1 above, each TTD section contains a special *physical equipment* able to check that a train resides on this section. Such special equipment are permanently connected to the trackside and are considered to be *safe*.

2.1.3 Trains

There are three kinds of *trains*: (1) those equipped with ERTMS (European Real Traffic Management System) and with TIMS (Train Integrity Monitoring System) and *connected* to the trackside, (2) those also equipped with ERTMS and TIMS but which are temporarily *disconnected* from the trackside, and finally (3) those which are not equipped with ERTMS (and thus not equipped with TIMS): such trains are *not connectable*, they are called *ghost trains*. Consequences of these definitions can be seen below in 1, 2, 3 and 4. Note that ERTM and TIMS are supposed to be *unsafe* equipment.

1. Trains which are equipped with ERTMS are able to communicate with the trackside.
2. Trains which are equipped with ERTMS and also with TIMS allow such trains to occupy precisely certain VSS within the trackside, in particular front end and rear end VSS (see more on this in Sect. 2.6). Note that VSS occupied by a given train are *contiguous* from the rear end to the front end of this train. Likewise, TTD sections occupied by a given train are also *contiguous*.
3. The VSS occupied by trains which are equipped with ERTMS and TIMS but which are not considered to be, so-called, *integer trains*, are computed differently by the trackside (see more on this in Sect. 2.6).
4. The trackside records a train which is not equipped with ERTMS by the TTD sections on which it resides. Note again that TTD sections occupied by a given train are *contiguous*.

I suppose that *two different trains have no common VSS*. Again, this fundamental property is not part of this case study. I thus assume that it is always respected. It is a slight **simplification** from the reference document where it is said somewhere that two different connected trains can share a common VSS.

2.2 More on Train Information

In this section, I define more precisely the kind of information which is sent regularly to the trackside by trains equipped with ERTMS and TIMS. Here are

these information: train *position*, train *length* and train *integrity*. Such information are detailed below in the following subsections. Note that more information are sent by the train to the trackside. Such data help the trackside to position the train with more precision. These data are the *max safe front end*, the *min safe rear end* and the *min safe front end*. More details on this can be seen in subsequent subsections.

2.2.1 Position

The exact nature of a train *position*, sent by trains to tracksides, is not made precise in the reference document. Note that the position in question is that of the front end of the train. The only thing which can be said about this position is that the trackside, when receiving it, is able to transform it into the unique VSS occupied by the front end of the train. As a **simplification**, I suppose that trains are never running backwards and that the resulting occupied TTD section or front end VSS numbers are not decreasing when trains move.

2.2.2 Length

By receiving the train *length*, the trackside is able to compute the VSS occupied by the rear end of the train. Again, it should be noted that all VSS situated in between the first (front end) VSS and the last (rear end) VSS of *integer trains* (see Sect. 2.2.3 below) are all occupied by such trains.

2.2.3 Integrity

When the train *integrity* is confirmed, this ensures that the train length received by the trackside is reliable. As a consequence, this length can be safely used in order to compute the VSS of the rear end of the train (see more in Sect. 2.3). When the train integrity is not confirmed, the VSS of the rear end of the train is computed differently by the trackside (see more in Sect. 2.3).

I understand (but I am not sure) from my readings of the reference document that train integrity and train length changes are independent notions. It would be then quite possible that a train integrity is lost whereas the train length remains unchanged. Conversely, it is quite possible that a train length change occurs while the train integrity is unchanged. This is what I deduce from my readings but **I am not certain of this**.

2.2.4 Max Safe Front End

The *max safe front end* is added by the trackside to the front end position received from the train. The new position, extended in this way, is then the one that is transformed by the trackside into a VSS.

2.2.5 Min Safe Rear End

The *min safe rear end* is subtracted by the trackside from the rear end position of the train as computed by the trackside using the front end position of the train and the train length. This usage of the min safe rear end is only made when the train integrity is confirmed by the train. In this case, the rear end position is

said to be *confirmed*. Note that explanations given here were not so explicitly stated in the reference document: what I express here is my own conclusion from what I have read in this document.

2.2.6 Min Safe Front End

The *min safe front end* is used by the trackside for computing the rear end position of the train in case integrity is not confirmed by the train. The trackside computes the rear end position of the train by subtracting the min safe front end plus the length of the train from the front end position of the train. In this case, the rear end position is only said to be *assumed*. Note again that explanations given here were not so explicitly stated in the reference document: what I express here is my own conclusion from what I read in this document.

2.3 Limits of the Trackside and of the Train Movements

The trackside I consider in this case study controls a certain maximum number of TTD sections, therefore a certain maximum number of VSS. The connected trains are supposed to have been given permission to circulate on some contiguous VSS of the ones controlled by the trackside: the set of such contiguous VSS for a given connected train is called its Movement Authority (*MA*). The last VSS of this MA is called the End of Authority (*EOA*) of the train. In this paper, I suppose to **simplify** that the EOA of connected trains are all the same and equal to the maximum number of VSS. Moreover, I suppose that this assignment of EOA is permanent.

2.4 State of a VSS in the Trackside

Here I almost quote directly some words seen in the reference document: "Besides the two states (*free*, *occupied*) which at least exist for a TTD ... two additional states are needed for a VSS to cover all operational situations. State *unknown* when there is no certainty if the VSS is *occupied* or not. State *ambiguous* when the VSS is known to be *occupied* by a train but when it is unsure whether another (not connected) train is also present in the same VSS".

When the trackside computes the new VSS of the front end of a train some VSS are modified from *free* to *occupied*. Note that all VSS contained in a *free* TTD are necessarily *free*. Also note that all VSS are in state *free* initially. In the reference document, the initial state of all VSS is *unknown*. I made a change here because I do not understand how such VSS could become *free*.

A connected trains can only move forward within its MA. Moreover a connected train can only move to a *free* VSS. These assumptions are not proved in this case study: they are thus assumed to be always true.

2.5 Precise Scope of the Case Study

With a now precise definition in Sect. 2.4 of the various states which can be associated with a VSS, we can make it clear what the main scope of this case study is, namely the *management of the VSS* [5]. In other words, I want to

formally study under which circumstances the status of a VSS in the trackside stays as previously established or is modified and thus how it is modified.

At this point it is important to make precise the various circumstances under which the state of a VSS could be modified. Here they are: move of connected trains (Sect. 2.6), connected trains disconnections (Sect. 2.7), disconnected trains reconnections (Sect. 2.8), detection of ghost trains (Sect. 2.9), and the move of ambiguous trains (Sect. 2.10). Notice that I have not retained in this paper potentiel disconnections and reconnections of ambiguous trains.

2.6 Moving Connected Trains

In this section, I synthesise the move of connected trains, as seen by the trackside. Once a message from a connected train is received by the trackside, the new VSS occupied by the train is computed. Let us suppose that the position of the train is a number and, of course, the various auxiliary information as well. I also suppose to have a function, *vss*, transforming a position into a VSS. Notice that this function is necessarily tabulated in the trackside as VSS have different sizes. Let *new_pos* be defined as follows:

$$new\_pos = position + max\_safe\_front\_end$$

The *front_end_vss* of the train is defined as follows:

$$front\_end\_vss = vss(new\_pos)$$

The *rear_end_vss* of the train is defined as follows in case the *integrity* of the train is confirmed:

$$rear\_end\_vss = vss(new\_pos - length - min\_safe\_rear\_end)$$

Finally, when the *integrity* of the train is not confirmed, the *rear_end_vss* of the train is defined as follows:

$$rear\_end\_vss = vss(new\_pos - length - min\_safe\_front\_end)$$

a result of these computations, we have the following transitions on the state of some VSS: from *free* to *occupied* and from *occupied* to *free*.

I think that what is said in this section is a **simplification** of the reference document.

2.7 Connected Train Disconnections

The trackside contains seven kinds of timers: (1) *mute timers*, (2) *wait integrity timers*, (3) *shadow timers A*, (4) *shadow timers B*, (5) *disconnect propagation timers*, (6) *ghost train propagation timers* and (7) *integrity lost propagation timers*.

In order to **simplify**, I will consider the first kind of timers only in this paper.

A *mute timer* is associated with each connected train by the trackside. This timer is reset each time a message from a connected train is received by the

trackside. When this timer expires without another message being received from the same connected train, then this train is considered to be *disconnected*.

As a result, all VSS occupied by the train are made *unknown*. We have thus a transition from *occupied* to *unknown* for such VSS. Likewise, all free VSS of the MA of this train in front of it until a free TTD or an occupied VSS by another train are encountered, are made *unknown* as well. We have thus a transition from *free* to *unknown* for such VSS.

What is said here is at least what I understood from my readings of the reference document, but I am not absolutely certain that this corresponds to the intent of the reference document. It is probably a **simplification** of mine.

2.8 Reconnection a Disconnected Trains

We have seen in Sect. 2.7 how the disconnection of a connected train can be detected by means of the expiration of a mute timer. After this mute timer has expired, if the trackside receives again a message from the disconnected train then it means that this train reconnects. In this case, we have to revert to the situation that was present before the disconnection occurred. As a **simplification**, I suppose that the information sent by the reconnecting train are simple: integrity is preserved, there is no change in the train length and the train still resides on the same TTD sections. The reconnection proceeds in two steps:

1. All VSS modifications occurring after the disconnection are reset as in their previous states.
2. Then the VSS of the train are modified according to the just received information.

After a train reconnection some VSS may enjoy the following transitions: from *unknown* to *occupied* or *unknown* to *free*.

2.9 Ghost Trains

A *ghost train* is also called in the reference document a *shadow train* when it follows a connected train. Such ghost trains are detected by the trackside when a free TTD section becomes unexpectedly occupied. This is indicated by the physical equipment in charge of this TTD section as explained in Sect. 2.1. This discovery has two consequences as indicated below in 1 and 2:

1. All VSS of the TTD section concerned with a ghost train are made *unknown*.
2. If the next TTD section is occupied by a connected train t whose rear end VSS is situated just after a sequence of free VSS from the beginning of this TTD, then the state of all free VSS of the TTD section until the rear end VSS of train t are made *unknown*. Moreover, the state of all VSS of the train t are made *ambiguous*. I call such a train t an *ambiguous train*.

   To summarise: the TTD section of the rear end VSS of such an ambiguous train is preceded by a TTD section with a ghost train. Moreover, the VSS situated behind the rear end VSS of such an ambiguous train are all *unknown*. These are permanent properties of ambiguous trains. All this is

what I deduced from my readings of the reference document although, to the best of my knowledge, I have not seen this written as such in the reference document.

All this corresponds to VSS transitions from *free* to *unknown* and from *occupied* to *ambiguous*. I clearly **simplified** things here. This is more precisely described in the reference document. I have taken this simple approach because I am not sure to have fully understood what is written in the reference document.

2.10 Moving Ambiguous Trains

An ambiguous train t can be moved like any other connected train (see Sect. 2.6). According to the properties of ambiguous trains (see Sect. 2.9), this move may have various consequences as indicated below in 1, 2, 3 and 4:

1. If the rear end VSS of the ambiguous train t is moved forward but remains in the same TTD section where it was previously, then the VSS that are normally made *free* when a connected train moves (Sect. 2.6), are made *unknown* instead.
2. If the rear end VSS is moved forward and leaves the TTD section where it was previously and if the left TTD section is not occupied after this move of train t, then t is not ambiguous anymore and all its VSS become *occupied* again. Moreover all VSS of its previous TTD section are made *free* again. We have here transitions from *unknown* or *ambiguous* to *free*.
3. If the rear end VSS is moved forward and leaves the TTD section where it was previously and if the left TTD section is still occupied, it means that the ghost train has moved. As a consequence, all VSS of this TTD section are made *unknown* and the train t remains ambiguous in the next TTD section.
4. If the front end of the ambiguous train t moves forward then some *free* VSS become *ambiguous*.

What is said in this section is not written as such in the reference document. It is something I concluded after my reading, thus **it can be wrong** according to the reference document.

2.11 Disconnecting and Reconnecting Ambiguous Trains

Disconnections and reconnections of ambiguous trains, which are quite possible, are not treated in this paper.

2.12 Synthesis of VSS State Transitions

The state of a VSS can be: free, occupied, unknown or ambiguous. Therefore we might have twelve transitions. Here is a synthesis of such transitions:

From	To	Section
free	occupied	2.6
occupied	free	2.6
occupied	unknown	2.7
unknown	free	2.8
unknown	occupied	2.8
free	unknown	2.9 2.7

From	To	Section
occupied	ambiguous	2.9
free	ambiguous	2.10
ambiguous	occupied	2.10
ambiguous	unknown	2.10
ambiguous	free	2.10
unknown	ambiguous	

Transitions shown in the above table are not the same as those mentioned in the reference document. This is because I **simplified** the case study in many places.

3 Requirement Document: Reference

There are two kinds of reference requirements: those dealing with the constant environment (labeled ENV) and those concerned by the functionalities of the system (labeled FUN).

The track is made of a sequence of physical train detection sections, called TTD sections	ENV-1
Each TTD section can be free or occupied	ENV-2
Each TTD section is made of a sequence of virtual sub-sections, called VSS	ENV-3
Each VSS can be free, occupied, unknown or ambiguous	ENV-4

A free TTD section can only contain free VSS	ENV-5
Trains can be connected, disconnected or ghost	ENV-6
Connected trains send messages to the trackside	FUN-1
Messages contain the following information: front end position, integrity, length, max safe front end, min safe rear end and min safe front end	FUN-2
From the received message, the trackside is able to compute the VSS occupied by a connected train. See section 2.6 for more details	FUN-3
A mute timer is associated with each connected train. This timer is reset when a message is received	FUN-4
After mute timer expiration, the corresponding train is considered to be disconnected	FUN-5
The VSS occupied by a disconnected train are made unknown. Some other VSS too. It is explained in section 2.7.	FUN_6
A disconnected train can be reconnected. The situation of the train is reverted as indicated in section 2.8	FUN-7
A ghost train can be detected on a normally free TTD section	FUN-8

The VSS of the TTD section of a ghost train are made unknown. Some other VSS are made ambiguous as indicated in section 2.9	FUN_9
VSS are subjected to some transitions as indicated in section 2.12	FUN-10

4 Refinement Strategy

The formal model presented in this paper is an abstraction only of the real system. In this abstraction, I suppose that connected trains are able to send directly their front end and rear end VSS to the trackside. This initial model takes account of reference requirements ENV-1 to ENV-6 and FUN-6 to FUN-10.

A further refinement would thus be necessary to formalise the way the trackside is able to compute such VSS. This corresponds to requirement FUN-3.

Another refinement would take account of the communication between trains and trackside. This corresponds to requirements FUN-1 and FUN-2.

A final refinement would introduce the mute timer. This corresponds to requirements FUN-4 and FUN-5.

I suppose that simplifications proposed in this paper could be overcome by further refinements.

5 Formal Model

The formal model constructed and proved with the Rodin Toolset is freely available in [6]. I warmly recommend readers to access this formal model as only part of it is shown in this paper. This formal model is entirely proven. It contains 327 proof obligations among which 230 were proven automatically (70%). The remaining proofs were done interactively. They were not difficult although sometimes a bit hairy.

5.1 Constants

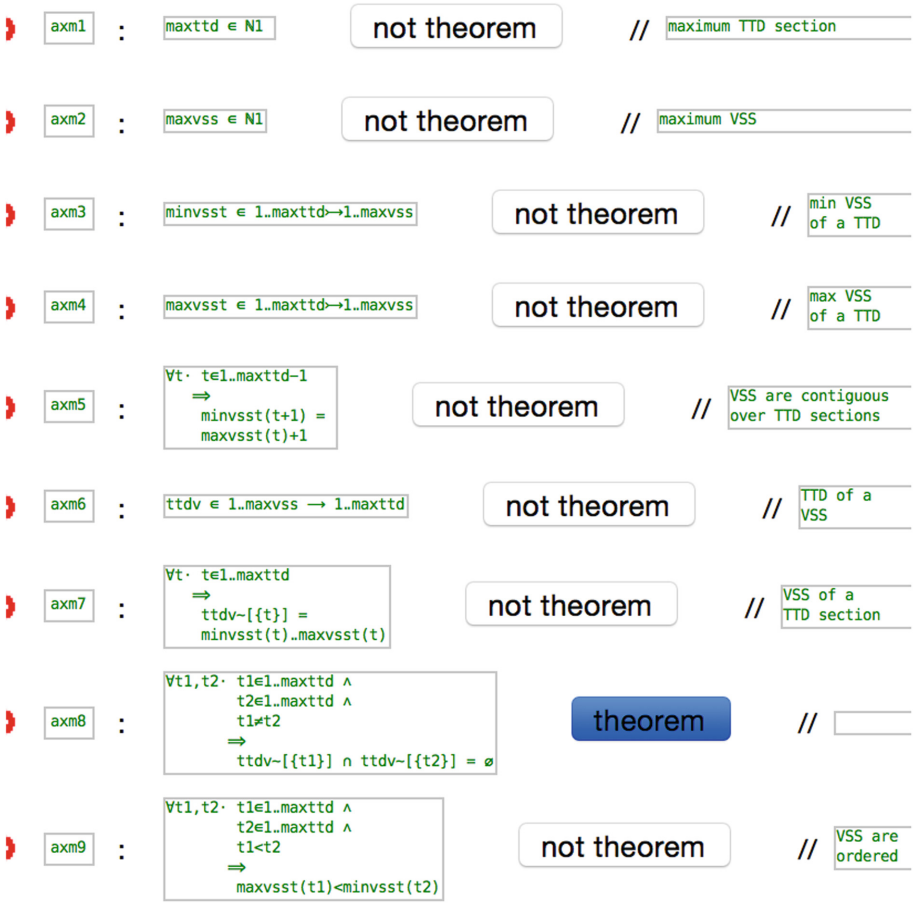
In this abstract model, the following constants are used:

1. *train*: this constant denotes the set of train.
2. *maxttd*: TTD section are named by means of a natural number interval from 1 to *maxttd*.

3. *maxvss*: VSS are named by means of a natural number interval from 1 to *maxvss*.
4. *minvsst*, *maxvsst*: these constants denote functions yielding the minimum and maximum VSS of each TTD section.
5. *ttdv*: this constant denotes a function yielding the TTD section of each VSS.

5.2 Axioms

Axioms are shown in the following screen copy:



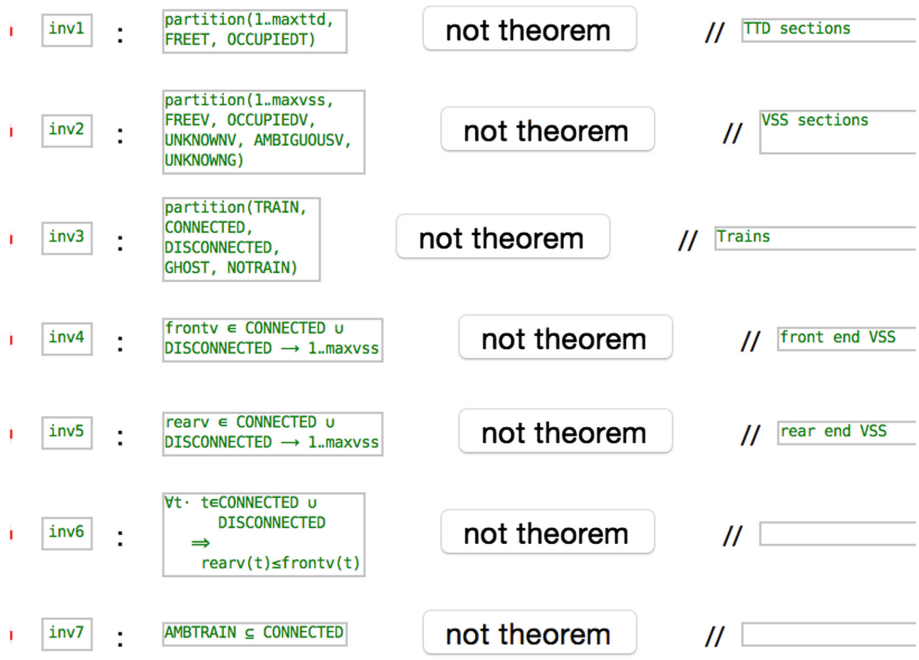
5.3 Variables

In this abstract model, the following variables are used:

1. $freet, occupiedt$: these subsets partition the set of TTD sections.
2. $freev, occupiedv, unknownv, ambiguousv, unknowng$: these subsets partition the set of VSS. Notice the presence of two unknown subsets. More precisely, $unknownv$ denotes the subset of unknown VSS resulting from a disconnection, whereas $unknowng$ denotes the subset of unknown VSS resulting from the presence of a ghost train.
3. $connected, disconnected, ghost, notrain$: these subsets partition the set $train$.
4. $frontv, rearv$ denote functions yielding the front end and rear end VSS of a connected or disconnected train.
5. $ambtrain$: this variable denotes the subset of connected trains that are followed by a ghost train.
6. $lastU$: this variable denotes the last VSS made unknown as a result of a disconnection.

5.4 Invariants

Invariants are shown in the following screen copies. Besides the typing of the variables, it is possible to see various properties and incompatibilities:




inv8 : ∀t· t∈CONNECTED\AMBTRAIN ⇒ rearv(t)‥frontv(t) ⊆ OCCUPIEDV
not theorer
inv9 : ∀t· t∈AMBTRAIN ⇒ rearv(t)‥frontv(t) ⊆ AMBIGUOUSV
not theorem
inv10 : ∀t· t∈DISCONNECTED ⇒ rearv(t)‥frontv(t) ⊆ UNKNOWNV
not theorem
inv11 : ttdv[OCCUPIEDV] ⊆ OCCUPIEDT
not theor
inv12 : ttdv~[FREET] ⊆ FREEV
not theorem
inv13 : ∀t1,t2· t1∈CONNECTED ∪ DISCONNECTED ∧ t2∈CONNECTED ∪ DISCONNECTED ∧ t1≠t2 ⇒ (rearv(t1)‥frontv(t1)) ∩ (rearv(t2)‥frontv(t2)) = ∅
no
inv14 : lastU ∈ DISCONNECTED ⟶ 1‥maxvss
not th
inv15 : ∀t· t∈DISCONNECTED ⇒ lastU(t)≥frontv(t)
not theorem
inv16 : ∀t· t∈DISCONNECTED ⇒ frontv(t)+1‥lastU(t)⊆UNKNOWNV
not th
inv17 : ∀t1,t2· t1∈DISCONNECTED ∧ t2∈DISCONNECTED ∧ t1≠t2 ⇒ rearv(t1)‥frontv(t1) ∩ frontv(t2)+1‥lastU(t2) = ∅
not t
inv18 : ∀t1,t2· t1∈DISCONNECTED ∧ t2∈DISCONNECTED ∧ t1≠t2 ⇒ frontv(t1)+1‥lastU(t1) ∩ frontv(t2)+1‥lastU(t2) = ∅
not t
inv19 : ∀tr,t· tr∈AMBTRAIN ∧ t = ttdv(rearv(tr)) ⇒ t>1 ∧ t−1∈OCCUPIEDT ∧ ttdv~[{t−1}]⊆UNKNOWNG ∧ minvsst(t)‥rearv(tr)−1⊆UNKNOWNG
nc

5.5 Events

The initial abstract machine contains the following events:

1. Moving connected non-ambiguous trains (2 events).
2. Disconnecting connected non-ambiguous trains (2 events).
3. Reconnecting disconnected trains (1 event).
4. Introducing ghosts and ambiguous trains (2 events).
5. Moving ambiguous trains (4 events).

Some abstract events are missing: disconnection and reconnection of ambiguous trains.

6 Some Comments

To the best of my knowledge (but I might be wrong), I did not find enough information in the reference document on the following issues: moving ghost trains, moving disconnected trains, the case of two or more ghost trains following each other, the case of a ghost train following a disconnected train which is reconnecting, the introduction of a new connected train within a trackside, the cooperation between two successive tracksides.

Concerning the structure of the reference document: I found it *too flat*, resulting in difficult readings.

Concerning this system in general, I found it a bit dangerous: I think it might be very difficult for tracksides to master simultaneously the safety of ghost trains, ambiguous trains, connected trains and disconnected trains. I cannot imagine how the trackside could take good safety decisions in the presence of unknown or ambiguous VSS. Another issue is that of traffic lights: they should be maintained on tracks because of the presence of non-connected trains. But drivers are said to be present in connected trains. It is well known fact that drivers are influenced by such traffic lights in spite of the automatic behaviour of the trackside.

I suggest that non-connected trains are not allowed to circulate on tracks where connected trains are under the controls of tracksides. Connected trains should not be connected anymore when they reach normal tracks. The issue of disconnected trains is, in my opinion, very rarely occurring. In this case, the driver (or the train itself automatically) should stop the train immediately until reconnection occur.

Acknowledgments. I would like to thank Asieh Salehi and Dominique Cansell for their help in preparing this paper.

References

1. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering, Chap. 17. Cambridge University Press, Cambridge (2010)
2. Butler, M.: A system-based approach to the formal development of embedded controllers for a railway. Des. Autom. Embed. Syst. **6**, 355–366 (2002)
3. Haxthausen, A.E., Peleska, J.: Formal development and verification of a distributed railway control system. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1709, pp. 1546–1563. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48118-4_32
4. EEIG ERTMS Users Group: Hybrid ERTMS/ETCS Level 3: Principles, July 2017. Ref. 16E042 Version 1A
5. Hoang, T.S., Butler, M., Reichl, K.: The Hybrid ERTMS/ETCS Level 3 Case Study. ECS University of Southampton
6. To some (but not all) MAC readers: after loading this file, you may have to compress it (zip) before importation in the Rodin toolset. To all readers: after importing, it might be the case that some proof obligations are not discharged. In this case, click on 'Proof Replay on Undischarged POs' under 'Proof Obligations'. http://users.ecs.soton.ac.uk/asf08r/ABZ_Case_2018

Diagram-Led Formal Modelling Using iUML-B for Hybrid ERTMS Level 3

Dana Dghaym(✉), Michael Poppleton, and Colin Snook

ECS, University of Southampton, Southampton, UK
{dd4g12,mrp,cfs}@ecs.soton.ac.uk

Abstract. We demonstrate diagrammatic Event-B formal modelling of a hybrid, 'fixed virtual block' approach to train movement control for the emerging *European Rail Traffic Management System* (ERTMS) level 3. We perform a refinement-based formal development and verification of the no-collision safety requirement. The development reveals limitations in the specification and identifies assumptions on the environment. We reflect on our team-based approach to finding useful modelling abstractions and demonstrate a systematic modelling method using the UML-like state and class diagrams of iUML-B. We suggest enhancements to the existing iUML-B method that would have benefitted this development.

1 Introduction

Railway control systems are safety-critical, and it is common for railway safety standards (e.g. CENELEC EN-50126, EN-50128/9) to recommend the use of formal modelling and verification to certify their correctness. We present our application of a diagrammatic formal modelling method to such a system.

The *European Rail Traffic Management System* (ERTMS)[1] [6] will comprise a single ATP (automatic train protection) system and a single GSM radio communication system train-to-trackside, to replace the variety of current national train control solutions. Hybrid ERTMS Level 3 [8] - a compromise between full ERTMS Levels 2 and 3 - aims to increase network capacity at reduced cost, using existing trackside train detection equipment together with radio communication.

This case study concerns a physical environment of trains, and communication by radio and trackside equipment. The case study concerns the control of trains moving on a linear track which is part of a wider network controlled by an interlocking system which is out of scope of this case study. A train movement controller called the *Radio Block Centre* (RBC) manages the *Movement Authority* (MA) granted to each train in mission. The focus of this work, called the *Virtual Block Detector* (VBD), conservatively estimates train locations to a finer granularity than physically detected track sections, and thus reports free

All data supporting this study are openly available from the University of Southampton repository at http://doi.org/10.5258/SOTON/D0403.

[1] http://ertms.net.

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 338–352, 2018.
https://doi.org/10.1007/978-3-319-91271-4_23

virtual track sub-sections available for train movement. Trains and trackside report location data to the VBD. In turn the VBD reports free track sections to RBC. The MA granted to each train consists of a set of sections that the train is permitted to move into. A *controlled* train is instructed that the MA sections are free; a *trusted* train is instructed that they might not be free. The key safety property which we verify is that controlled trains do not run into trains that are ahead of them.

Motivation and Contribution. The case study presents challenges - addressed by our contribution - for a formal development method, typical of challenges arising in safety-critical cyber-physical systems. First is the development of a useful model reflecting the component architecture of the target system (VBD) interacting with its physical environment and other system components. The model enables us to verify functional safety properties of the specification. Second, we need readable models so that domain experts are able to validate the model. While we do not focus on validation in this paper (other than for our own sanity checks of the model), future work will include running scenarios in a form of model acceptance test. Third, we describe how we tackle the diffcult process of turning a detailed and complex specification that contains ambiguity and relies on tacit domain knowledge, into a formally precise model containing useful abstractions. (We view this contribution as particularly useful for industrial partners to enable them to adopt formal modelling techniques). Fourth, we have the emergent critique of the specification document: assumptions on the environment, omissions, ambiguities, errors etc.

The refinement-based Event-B modelling method [1] is an appropriate choice since it allows us to verify key properties while leaving certain features, and interacting components, abstract and underspecified. The architecture can be layered through the refinement: each layer can focus on an abstract component interface, the environment, or a specific feature of the target system. Event-B has strong tool support [2] for verification and validation in the form of theorem provers and model-checkers. Diagrammatic modelling notations and tools are available which help in conceptual modelling: we use iUML-B class diagrams and state-machines [14,16,17]. One of our goals is to show that using iUML-B leads to a readable formal specification (or at least more readable than plain Event-B), which is easier for domain experts to validate.

Structure. The paper is structured as follows. We next recall Event-B and iUML-B basics in Sect. 2. Section 3 reviews our development process, and Sect. 4 gives our system analysis. The refinement strategy is then summarised (Sect. 5), followed by a detailed account of modelling in Sect. 6. Next is related work (Sect. 7), followed by the conclusion in Sect. 8.

2 Event-B and iUMLB

Event-B [1,9] is a refinement-based formal method for system development. An Event-B model contains two parts: *contexts* for static data, and *machines*

for dynamic behaviour specified by *variables* `v`, *invariant* predicates `I(v)` that constrain the variables, and *events*. An event comprises a guard denoting its enabling-condition and an action describing how the variables are modified when the event is executed. In general, an event `e` has the following form, where `t` are the event parameters, `G(t, v)` is the guard of the event, and `v := E(t, v)` is the action of the event.

```
e == any t where G(t,v) then v := E(t,v) end
```

Event-B is supported by the *Rodin platform* (Rodin) [2], an extensible toolkit which includes facilities for modelling, verifying the consistency of models using theorem proving and model checking techniques, and validating models with simulation-based approaches.

iUML-B [14,16,17] provides a diagrammatic modelling notation for Event-B in the form of state-machines and class-diagrams. The diagrammatic elements share the repository of an Event-B model, and contribute to that model. For example a state-machine will automatically generate the Event-B data elements (sets, constants, axioms, variables, and invariants) to implement the states, and contribute additional guards and actions to existing events. Class diagrams provide a way to visually model data relationships. Classes, attributes and associations are linked to Event-B data elements (carrier sets, constants, or variables) and generate constraints on those elements.

3 Process

Formal models are often presented as if they were developed in perfect inexorable steps when, in practice, they never are. We give an overview of our informal team-based process illustrating the iterations that involved many misunderstandings failures and re-work. Although we had some feedback from domain experts on terminology and detailed clarifications, this was not substantial as we wanted the case study to test our ability to use formal methods to understand and interpret the specification. The domain experts were not involved in the process described in this section. The team consisted of research and academic staff who had some experience of formal modelling of railway applications such as interlockings and crossings, but no previous experience of communications-based, virtual section train control.

Systems Analysis. While the *Hybrid ERTMS Level 3* (HLIII) specification is quite well presented in terms of explanatory scenarios, its focus makes it a detailed requirements specification for the VBD. It does not explain the overall system aims and principles so well. We therefore started by reverse engineering our understanding of the system in order to understand its purpose and the concepts that it is based on. This involved analysis of the information in the specification, discussions and sketching whiteboard diagrams such as components, entity relationship and state-machine diagrams. The diagram-based analysis naturally led into the iUML-B modelling. The systems analysis identifies the main components in the system and the information flow between them.

This is necessary for the model to reflect the appropriate responsibilities of the VBD verses assumptions it makes upon other components. As with most stages of the modelling process, the analysis was iterative. The modelling helped our understanding of the system and our new understanding helped us choose better abstractions for modelling. For example initially we assumed that only connected trains were 'in-mission'. However, when modelling we realised that when a connection is lost the system relies on the fact that the train will continue to respect its MA and this implies that the train is still in-mission. This new understanding of the system led us to revise our models so that the in-mission state-machine was independent of (i.e. parallel with) the connected state-machine.

Refinement Strategy. The refinement strategy provides a plan for how we intend to build the model, choosing abstractions, adding details in refinement steps and introducing invariant properties at appropriate stages. We considered two alternative approaches, (a) start from an abstract safe system or (b) start from an unsafe system and make it safe. For this example we chose the second approach. While the first approach is perhaps more traditional, in this case, the safety properties were not so obvious and were complicated by unsafe, albeit mitigated, scenarios. So we wanted to capture the essence of train movement before introducing assumptions and progressing towards details that can distinguish between safe scenarios and mitigated unsafe scenarios. Again, the refinement strategy evolved as we discovered difficulties and adapted our approach.

Modelling. In modelling we used iUML-B for its diagrammatic notation which follows on from the diagrams used in our analysis and review stages. As usual, we used the provers to verify models and when they fail, and we cannot be sure why, the ProB model checker helps to find counter examples. We also animated the models to check that the model behaves as expected.

Review. We held regular reviews to discuss problems with the modelling. As indicated in the previous steps, the reviews led to significant iterations to our understanding of the system, revisions to our refinement plan and consequent changes to the model. Problems fell into the following categories:

- We cannot prove this PO - look for a better modelling approach. Example: contiguity of *next_VSS* relationship - We found it difficult to prove contiguity properties about *Virtual Sub-Section* (VSS) using abstract properties. While this should, in principle, be possible, we decided it was not worth the effort and introduced numeric indexing of VSS (relying on the contiguity of a range of integers). We retained the *next* function for elegance of expression in guards and actions.
- This is not a useful refinement - change refinement strategy. Example: We wished to introduce features such as timers as soon as it was possible to do so (i.e. when the triggering functionality was available). However, we had not yet introduced the relevant VSS state changes to utilise the timeout. To rectify this we altered our refinement strategy to introduce abstract versions of VSS states and associated transitions.

– This is not a true data refinement - change systems analysis. Example: As we modelled the flow of information through the control components we found it difficult to reconcile the reported train positions and controlled MA with the safety properties of the abstract environment. It seemed that we would need to introduce some form of responsiveness assumptions to limit the difference between actual and control variables. However, the specification implied that the VSS states were asynchronously updated. As our understanding of the MA principle improved we realised that the position inaccuracy is of no consequence and we adjusted our systems description.

4 System Analysis

The HLIII specification is a detailed description of one component (the VBD) of a wider system that controls train movements. The other components involved in the system are the trains and trackside equipment, which we refer to as *environment* (ENV), and the RBC that calculates movement authorities limiting the movement of trains.

The VBD receives messages from trains and train detectors. It also receives information about the output of the RBC. It calculates a set of sections that it believes to be free of any trains and sends these to the RBC. The RBC sends to each train, a movement authority consisting of a set of sections that the train may move into. The train is either instructed that the sections are all free or that they might not be free. We wish to model and verify item 3, the VBD. To do this we also need to consider (and model) the other 2 items.

The **environment** consists of a linear track divided into fixed sections (*Virtual Sub-Section* (VSS)) with trains moving in one direction on the track. Detectors (*Trackside Train Detection* (TTD)) report when a train is present. However, there is only one TTD for a group of VSS. There are 2 kinds of trains; those that communicate with the control system, and those that do not. Trains that communicate send three items of information to the VBD:

– their current position (in finer granularity than track sections),
– the length of the train,
– whether the train is confirmed as integral.

Communicating trains are able to receive information about the range of sections they are allowed to move through and whether the authorised track is guaranteed to be free (full-supervision) or not (on-sight). For the purpose of this description we partition trains into three kinds: ghost trains (not communicating), controlled trains (communicating with guaranteed free sections authorised) and trusted[2] trains (communicating with possible non-free sections authorised). Trains that do not communicate can only be detected by TTD and may move freely according to some assumptions concerning physical limitations and those imposed by train design regulations.

[2] *Controlled* and *trusted* (trains) are terms that we have introduced, they are not terms from the specification.

The **RBC** grants movement authority (permissions) to the communicating trains. The RBC uses information it receives from the VBD about which VSS are free. An MA consists of a set of track sections that the train is allowed to move through. The train is also instructed as to whether it needs to be responsible for avoiding collisions with trains in front (*On-Sight Movement Authority* (OSMA)) or whether it can assume the track sections are free (*Full Supervision Movement Authority* (FSMA)). We assume the RBC always issues safe FSMA in accordance with the information it receives from the VBD. I.e all sections in an FSMA are ones that the VBD has calculated to be free.

The **VBD** is responsible for deciding which VSS are free based on information it receives from the TTD and from *Positive Train Detection* (PTD) communications received from communicating trains. It sends information about which VSS it believes are free to the RBC. Since PTD reports may be intermittent or interrupted and some trains do not communicate at all, the estimate of free VSS is cautious in these circumstances.

The positions of trains that are communicating are known fairly accurately (subject to some lag in communications) from the PTD data sent by the train (position, length and integrity) as well as physical limits on possible train movement in between communications. The position of the train may cover a range of sections from that occupied by the rear to that occupied by the front. Some robustness is necessary to accommodate limitations of the communication mechanisms such as temporary loss of communication etc.

The position of a train that is not communicating (i.e. a ghost train) is difficult to determine. The possible positions of a ghost train are estimated as a range of sections based on the following:

- its last known position (from a PTD or a loss of integrity),
- how far it could possibly have travelled since its position was known,
- information from trains and free TTD that delimits its movement range.

A ghost train is created in the VBD by one of the following means: a communicating train stops communicating, a TTD spontaneously and unexpectedly detects a train, or a communicating train reports that it has lost integrity.

For loss of integrity, a ghost train is created just behind the communicating train to represent the detached section of carriages. A communicating train is converted to a ghost train if the train's mute timer expires (after communication is lost) or if it sends a mission end message and terminates communication. A ghost train is removed (i.e. destroyed) by sweeping. Sweeping is the movement of a trusted train (with OSMA) through the sections where the ghost train may be. If the trusted train is able to pass through the sections the ghost train does not exist. A ghost train may also be converted to a communicating train if it starts communicating with the VBD (either by sending a mission start communication or by re-starting previous communication).

5 Refinement Strategy

Through system analysis and iterative modelling, the original outline refinement strategy evolved into the following. The target VBD model interacts with the physical environment of trains and trackside: the first refinement layers ENV. Next is the RBC component, followed by lower layers which elaborate the VBD.

ENV-M-1 Trains: Defines a linked list of trains to keep track of train order and prevent overtaking. Trains are created at the rear of the linked list and removed from its front. We also allow adding a new train in the middle of the linked list as a result of train split.

ENV-M0 Train movement, VSS: Introduces the train movement in terms of VSS section updates, where a VSS section is either free or occupied by a train. The train movement is modelled as an independent update of the position of the train front and rear.

ENV-M1 Ghost vs connected trains: Distinction between connected and ghost (i.e., non-connected) trains, where all new trains join as ghost.

ENV-M2 TTD: Introduces TTD sections which can be either free (no train on any of its VSS) or occupied (a train on at least one of its VSS). The TTD state is immediately updated by train movement events.

RBC-M3 RBC: RBC can grant trains MA. We call trains with MA inMission, where the RBC may extend or shrink their MA while connected.

VBD-M4 Position reporting: Presents the VSS four states (free, occupied, ambiguous, unknown). Also introduces the reported versus actual train position with the associated MA trimming. Disconnection related timers are also introduced.

VBD-M5 Controlled vs trusted trains: Fully supervised FS (controlled) vs on-sight OS (trusted) trains are introduced. An OS train has unsafe MA and is assumed not to crash into the back of other trains. An FS train has safe MA and therefore cannot crash into the back of other trains. In addition to Ghost train timers.

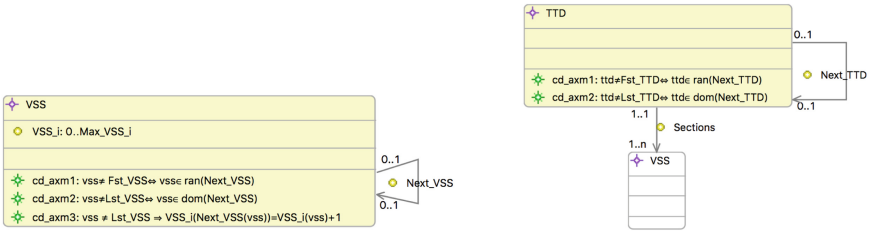
VBD-M6 Integrity loss: If a train reports either integrity loss or changed length, the train is split. Additionally, integrity loss propagation timers to control availability of adjacent VSS are introduced.

VBD-M7 Lower levels: Full VSS state transition as per specification, including all timers.

6 Modelling

The model consists mainly of two parts: the ENV and the VBD. The RBC provides an intermediate layer for moving from the ENV to the VBD.

Modelling the Environment. In the first part of the model, we focus on modelling the ENV and the possible trackside events, such as train movement, splitting and loss of communication.



(a) VSS Properties (b) TTD Properties

Fig. 1. Class diagram representing the track in the context

In the context, we model the network topology using iUML-B class diagrams (Fig. 1). First we introduce the *TRAIN* class (not shown in figures), then the *VSS* with their linear layout enforced by indexing via attribute, *VSS_i*, Fig. 1a.

At the abstract level, we introduce how trains can join and leave the network or in other words how trains can be created and destroyed. The variable class *train*, with superset *TRAIN*, represents the trains that currently exist in the network. There are two cases for creating trains, either a train can join from the beginning of the network or in the middle as a result of splitting behind an existing train. An important property at this level is: *trains cannot overtake*, which is why we introduce the relative ordering of the trains, represented by the variable association *next_train* in Fig. 2. Therefore, a train can only leave the network if there is no train in advance, this is represented by the guard $tr \notin dom(next\_train)$ added to the method ENV_leave_network of class *train*.

In the next refinement, we model train movement. A train's position is given by the VSS that it occupies: variable association *occupiedBy* in Fig. 2. We only model trains moving forward, hence a train can only leave a VSS if it occupies the next one. In order to ensure the no overtaking property, a train can only move forward if it doesn't share a VSS with its next train. Apart from splitting, a train can only join the network from the first VSS and trains can only leave from the last VSS. Since the no-overtaking property is fundamental to the safety of the system, we ensure the model does not break it by introducing the following invariant, which states that a train cannot occupy a vss with an index higher than the lowest indexed VSS of the next train:

$$\forall tr1, tr2 \cdot (tr2 \mapsto tr1) \in next\_train \implies max(VSS\_i[occupiedBy \sim [\{tr2\}]]) \leq min(VSS\_i[occupiedBy \sim [\{tr1\}]])$$

To distinguish between trains that are communicating and those that are not, we introduce sub-states *connected* and *ghost*, of *train* (Fig. 3).

Next, we introduce the *TTD* which group sets of contiguous *VSS* via association *Sections* (Fig. 1b). Class *occupiedTTD*, which is a sub-class of *TTD*, represents those TTD that have at least one of their VSS occupied by a train. At this level, we distinguish two cases when a train is leaving the last VSS of the TTD: (i) no other train occupies the TTD and the TTD becomes free (and is removed from *occupiedTTD*) or (ii) it remains occupied and not free. The same applies to a train leaving the network which can also free a TTD.

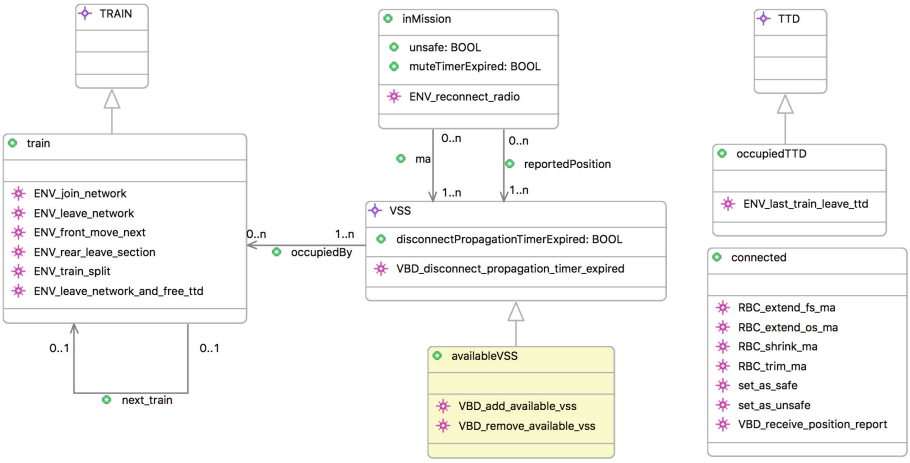


Fig. 2. Class diagram representing dynamic aspects of the environment

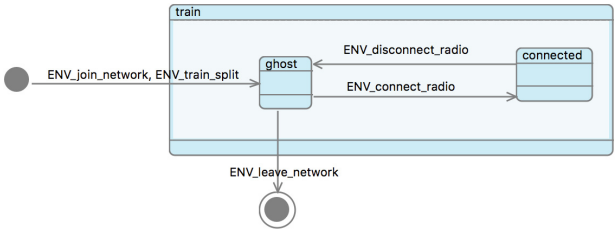


Fig. 3. Train communication statemachine

In the final environment model, we introduce the RBC role which paves the way for the VBD part. The RBC provides movement authorities (MA) which we assume trains will respect. The MA is modelled as a variable association *ma* between *train* and *VSS*. We refine the train statemachine further by introducing a parallel state-machine (Fig. 4). The sub-states, *inMission* and *noMission*, distinguish the mission status of trains. *inMission* represents trains that have performed a *Start of Mission* (SoM) (transition ENV_start_of_mission), while *noMission* represents trains that either did not start or performed an *End of Mission* (EoM) (transition ENV_end_of_mission). The mission state-machine was introduced as a parallel state-machine to the communication state-machine so that trains that lose communication retain their mission status. All connected trains have a mission. This is ensured by the invariant: $connected \subseteq inMission$.

We also split the radio connection/disconnection transitions in Fig. 3 into two cases to distinguish between SoM and reconnection and connection loss and EoM. The transitions ENV_start_of_mission and ENV_end_of_mission are common to the two statemachines. Note that when a train first joins a network, it joins as a ghost train with no mission, and when leaving the network it also has to leave as a ghost train with no mission.

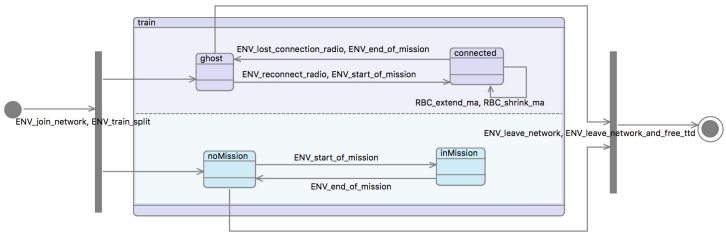


Fig. 4. Parallel statemachines for communication and movement authority

When a train performs SoM, it is immediately granted an MA for the VSS it occupies. However, this does not allow the train to move to new VSS sections. In order to move forward, the RBC should extend the MA as shown by the self transition RBC_extend_ma of the *connected* state in Fig. 4. Our assumption that trains with a mission respect their MA is enforced by the *inMission* class invariant: $occupiedBy \sim [\{tr\}] \subseteq ma[\{tr\}]$[3]. However, when the RBC shrinks the *ma* (e.g. due to propagation of an unknown VSS state) the actual train position may have progressed sufficiently to violate this invariant. We believe this is a limitation of the system and therefore introduce a boolean attribute *unsafe* of class *train* to indicate that the train has entered an unsafe scenario (to be detailed in later refinements), and the invariant can be violated in this scenario: $occupiedBy \sim [\{tr\}] \subseteq ma[\{tr\}] \vee unsafe(tr) = TRUE$. In Fig. 2, RBC_trim_ma in the *connected* class plays the role of a garbage collector, removing the VSS the train has left behind.

Modelling the VBD. The VBD cannot see directly what is happening in the ENV; it depends on periodic reports (PTD) sent by the train and it then asynchronously updates the VSS states. Similarly, the RBC receives information about VSS state from the VBD. This asynchronous behaviour relies on the fact that the actual position cannot be behind the reported position and is somewhere within the MA. I.e. reported position is only used to free VSS after a train has passed. This is embodied in the following invariants of class *inMission* which relate the actual position *occupiedBy* with the *reportedPosition* seen by the VBD.
$min(VSS\_i[reportedPosition[\{tr\}]]) \leq min(VSS\_i[occupiedBy \sim [\{tr\}]])$
$max(VSS\_i[reportedPosition[\{tr\}]]) \leq max(VSS\_i[occupiedBy \sim [\{tr\}]])$
We also refine loss of connection with mute timer expiry. We model time abstractly without introducing a clock, giving timeouts a non-deterministic opportunity to expire. When a mute timer expires this will enable the disconnect propagation timer whose expiry will affect the VSS state in later refinements.

In the next refinement of the VBD, we distinguish between the two different modes of MA: FSMA and OSMA. In FSMA mode the RBC only uses free VSS to extend *ma*. In OSMA mode, the RBC can extend *ma* with any VSS since we trust the OSMA trains not to crash. This behaviour is modelled in Fig. 5

[3] Note that class invariants are implicitly quantified over instances of the class, hence the antecedent $\forall tr \cdot tr \epsilon inMission$ is added automatically.

by partitioning *inMission* into two different sub-states, *controlled* and *trusted* representing FSMA and OSMA modes respectively. The choice between the two transitions, RBC_extend_os_ma and RBC_extend_fs_ma, is non-deterministic and determines the mode of the train.

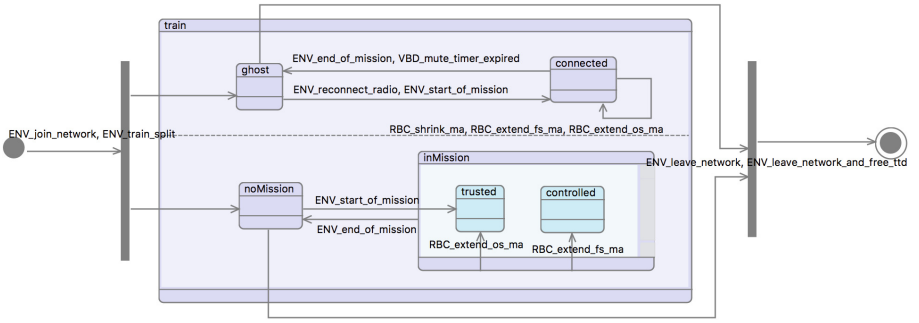


Fig. 5. Introducing sub-states to represent FSMA and OSMA modes

We can now introduce a safety invariant concerning the separation of controlled trains; the *ma* of controlled trains do not overlap:
$\forall tr1, tr2 \cdot tr1 \in controlled \wedge tr2 \in controlled \setminus \{tr1\} \implies ma[\{tr1\}] \cap ma[\{tr2\}] = \varnothing$

Hence, the RBC can only extend the *ma* of *controlled* trains using VSS sections that are free and not part of any *ma*. We introduce a sub-class *availableVSS* of *VSS* to represent the free vss sections. This will be refined to the VSS state *free* in future refinements as we introduce the state-machine of the specification. However, extending the *ma* for trusted trains does not have these restrictions.

At this level we add and remove *availableVSS* abstractly with the general conditions that apply for all cases. Therefore for adding a new *availableVSS*, it either belongs to a free TTD or no train has reported its position in this VSS, while for now the only condition for removing a VSS from *availableVSS* is that it belongs to an occupied TTD.

Next we introduce the concept of train integrity. We partition *connected* into two sub-states: *integral* and *nonIntegral*. We also refine the PTD position reports to include integrity information. Therefore, we split the method VBD_receive_position_report into different cases for confirming integrity, integrity loss, integrity not available and train length change. We also introduce the integrity waiting and propagation timers.

At this stage, it became clearer to us that *availableVSS* is insufficient, and it would have been better to introduce different sub-states of VSS as soon as we started the VBD part. Most timers result in a change to state *unknown* when they expire, hence there is no great benefit from having the timers without showing their effect. Moreover, this would have the advantage of introducing the different transitions of the VSS statemachine earlier, with the four states

(*free*, *unknown*, *ambiguous*, *occupied*) and gradually building towards the specification. Such decision requires a new iteration of the Event-B modelling in accordance with the refinement strategy described in Sect. 5.

Proof Statistics. The current modelling approach (before the next iteration in accordance with Sect. 5) contains 8 machines. Our modelling resulted in 246 proof obligations, where about 66% (162) were proved automatically with the default Rodin prover configuration. However, most of the proof obligations that were not discharged automatically were related to well-definedness of min/max operators. We then changed our Rodin configuration to include SMT solvers, this increased the number of automatically discharged proofs to 226 (92%). Finally, we added the relevance filter (but excluding newPP), which is a meta prover that improves the efficiency of the predicate prover by selecting relevant theorems. This improved our automatic percentage to about 99%. However, when recalculating auto-status, we found that this sometimes dropped to 97%. Presumable this is due to fluctuations in the processor resources available to the prover. We managed to get the auto-status back to 99% by increasing the timeout limits of the provers. This high percentage of automation depends on the modelling style applied. For example, we used indexing to avoid abstract models of sequences whose transitive properties are difficult to prove. In our models, we used iUML-B class diagrams and state-machines. The iUML-B state-machines plugin provides two alternative translations, one representing the states as an enumerated set and the other representing states as subsets of the statemachine instances. We used the latter translation, lifting the state-machine to a set of instances (*train*). Therefore, the generated state-machine type invariants are based on subsets of the instance set (*train*). In future work we will assess whether the use of iUML-B and the choice of state-machine translation affect the degree of automatic proof.

7 Related Work

Various approaches have been made during the development of the Event-B method, to integrating it into the broader Software Engineering process. The original interpretation of UML class diagrams and statemachines in classical B [17] have been presented - and tool-supported - as iUML-B [16] for Event-B. More recently Event-B refinement has been extended [14] to this diagrammatic modelling method. Example applications - of which this work is one - include [11]. CODA [3] is a tool-supported framework extending iUML-B for component-based embedded systems.

Train control is a familiar domain for Formal Methods, and specifically for B and Event-B-based approaches. Butler et al. [4] give a methodical treatment of the diagrammatic modelling of the rail interlocking system Railground with both iUML-B and Event Refinement Structures [15]. In [7], the authors present the Event-B development of a *Communications-based Train Control* (CBTC) system from Hitachi Ltd. Their focus is on the use of *Abstract Data Types* (ADTs) to manage the complexity of modelling a graph-based rail network and its dynamics. This example is comparable to ERTMS Level 3 and uses moving blocks. The

authors further proposed [10] the extension of iUML-B to support diagrammatic modelling of ADTs, using the same Railground case study as [4].

Other related work such as [13] on Hybrid ERTMS Level 3 is based on moving blocks. These models are hybrid, being concerned with continuous modelling of exact train position and speed reporting. This ABZ2018 case study is the first formal examination of fixed virtual blocks that we are aware of.

8 Conclusion

The specification is a rich and detailed source of information but is written as a functional specification of the VBD component rather than a systems requirements document. While trying to formalise and abstract a model of the system, we discovered several ambiguities. For example, when modelling the mute and disconnect propagation timers we found that Sect. 3.4.2.2 describes the start event of the disconnect propagation timer to be expiry of the mute timer. However, in scenario 4, EoM also starts the disconnect propagation timer. One possible explanation is that the mute timer also operates for trains when they perform EoM. On the other hand, transition 7 A in the VSS statemachine distinguishes between mute timer expiry and EoM, implying that these are two different cases. In addition, Sect. 3.4.2.2 states that the mute timer is stopped once the train re-connects, but doesn't describe the EoM case. Does the VBD need to keep a history of train positions with ended mission? Or is the mute timer not stopped in this case? This example illustrates how formal modelling can reveal ambiguities in the specification. Collaboration and interaction with domain experts is crucial to resolve such questions as it would be dangerous to model our own assumptions.

Formal modelling and the need to make abstractions and refine them, helped to develop our understanding of the system and to gain insights into the principles of the design. The main example of this is the link between: *what it does* - prevents certain kinds of collisions; *how it does it* - allocates movement authorities; *why it works* - movement authorities cannot be entered by another train. It also made us very aware of limitiations to the safety of the system such as the case where carriages could roll backwards which could break the *why it works*.

We intend to continue developing and improving the formal model as part of the Enable-S3 project[4]. The model will form a demonstrator for the Rail use case and will be used in conjunction with MoMuT for test case generation [12]. An acceptance test specification will be developed using 'Cucumber for iUML-B' [5] which is a formalised notation for describing test scenarios for iUML-B models. The acceptance tests provide a rigorous, repeatable validation accessible to domain-experts with limited formal methods expertise.

Some suggestions for improvements to the iUML-B notation and tools arose from modelling the HLIII. For example, it is often convenient to initialise class attributes/associations and have a complete mapping of their instances to values rather than specify a common value for each instance. Similarly, we often

[4] https://www.enable-s3.eu/.

needed to specify 'class-wide' invariants in which case the Event-B generator adds an unnecessary universal instance quantifier. These improvements will be incorporated in a future release.

Classes represent a set of instances with state represented by attributes and associations and behaviour described in methods. Lifted statemachines represent a set of instances with state represented by the statemachine state and behaviour described in transitions. It is often useful to use both visualisations for the same set of instances. While the diagrams can be linked to the same set of instances, the integration is not very strong and the tooling sometimes conflicts in Event-B generation. We experienced difficulties for example when modelling *connected* (trains) as both a class and a state. A first improvement would be to allow statemachines to be placed inside classes (an existing feature request) and rectify the problems with generation. However, a more fundamental integration might be possible: a common underlying record-based notation for the iUML-B model. In this case the diagrams would be alternative views of a common model. A text representation of the record-based model could also be provided. This would align well with our plans to provide a text based version of iUML-B to improve team-based development (where model diff and merge are essential).

Acknowledgements. The authors would like to thank Tomas Fischer of Thales Austria GmbH, for his helpful comments about the paper.

This work has been conducted within the ENABLE-S3 project that has received funding from the ECSEL Joint Undertaking under Grant Agreement no. 692455. This Joint Undertaking receives support from the European Union's HORIZON 2020 research and innovation programme and Austria, Denmark, Germany, Finland, Czech Republic, Italy, Spain, Portugal, Poland, Ireland, Belgium, France, Netherlands, United Kingdom, Slovakia, Norway.

References

1. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Abrial, J.-R., Butler, M., Hallerstede, S., Hoang, T., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. Softw. Tools Technol. Transf. **12**(6), 447–466 (2010)
3. Butler, M., Colley, J., Edmunds, A., Snook, C., Evans, N., Grant, N., Marshall, H.: Modelling and refinement in CODA. In: Refine@IFM 2013, EPTCS, Turku, Finland, vol. 115, pp. 36–51 (2013)
4. Butler, M., Dghaym, D., Fischer, T., Hoang, T., Reichl, K., Snook, C., Tummeltshammer, P.: Formal modelling techniques for efficient development of railway control products. In: Fantechi, A., Lecomte, T., Romanovsky, A. (eds.) RSSRail 2017. LNCS, vol. 10598, pp. 71–86. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-319-68499-4_5
5. Fischer, T., Snook, C., Hoang, T.: Formal model validation through acceptance tests. Technical report, University of Southampton, UK, March 2018
6. Furness, N., van Houten, H., Arenas, L., Bartholomeus, M.: ERTMS Level 3: the game-changer. IRSE News **232**, 2–9 (2017)

7. Fürst, A., Hoang, T.S., Basin, D., Sato, N., Miyazaki, K.: Large-scale system development using Abstract Data Types and refinement. Sci. Comput. Program. **131**, 59–75 (2016)
8. EEIG ERTMS Users Group. Principles: Hybrid ERTMS/ETCS Level 3. http://www.southampton.ac.uk/assets/sharepoint/groupsite/Academic/ABZ-Coneference-2018/Public%20Documents/ABZ2018/16E0421A_HL3.pdf. Accessed 18 Jan 2018
9. Hoang, T.: An introduction to the Event-B modelling method. In: Industrial Deployment of System Engineering Methods, pp. 211–236. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-33170-1
10. Hoang, T., Snook, C., Dghaym, D., Butler, M.: Class-diagrams for abstract data types. In: Hung, D., Kapur, D. (eds.) ICTAC 2017. LNCS, vol. 10580, pp. 100–117. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-319-67729-3_7
11. Hoang, T.S., Snook, C., Ladenberger, L., Butler, M.: Validating the requirements and design of a hemodialysis machine using iUML-B, BMotion studio, and co-simulation. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) ABZ 2016. LNCS, vol. 9675, pp. 360–375. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33600-8_31
12. Krenn, W., Schlick, R., Tiran, S., Aichernig, B., Jobstl, E., Brandl, H.: MoMut::UML model-based mutation testing for UML. In: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), pp. 1–8 (2015)
13. Platzer, A., Quesel, J.-D.: European train control system: a case study in formal verification. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 246–265. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10373-5_13
14. Said, M., Butler, M., Snook, C.: A method of refinement in UML-B. Softw. Syst. Model. **14**(4), 1557–1580 (2015)
15. Salehi, A., Butler, M., Rezazadeh, A.: Language and tool support for event refinement structures in Event-B. Formal Asp. Comput. **27**(3), 499–523 (2015)
16. Snook, C.: iUML-B statemachines. In: Proceedings of the Rodin Workshop 2014, Toulouse, France, pp. 29–30 (2014). http://eprints.soton.ac.uk/365301/
17. Snook, C., Butler, M.: UML-B: formal modeling and design aided by UML. ACM Trans. Softw. Eng. Methodol. **15**(1), 92–122 (2006)

An Event-B Model of the Hybrid ERTMS/ETCS Level 3 Standard

Amel Mammar[1], Marc Frappier[2], Steve Jeffrey Tueno Fotso[2,3(✉)], and Régine Laleau[3]

[1] Télécom SudParis, SAMOVAR-CNRS, Évry, France
amel.mammar@telecom-sudparis.eu
[2] Laboratoire GRIL, Département d'informatique, Faculté des sciences, Université de Sherbrooke, Québec, Canada
marc.frappier@usherbrooke.ca
[3] LACL, Université Paris-Est Créteil, Créteil, France
steve.tuenofotso@univ-paris-est.fr, laleau@u-pec.fr

Abstract. This paper presents an Event-B model of the ABZ2018 case study on the European Rail Traffic Management System (ERTMS) standard. The case study focusses on the management of fixed virtual subsections (VSS). We model the hybrid level 3 of the standard, which assumes that trains may be either equipped with an on-board train integrity monitoring system (TIMS) and that they report their position and integrity, ERTMS trains not fitted with TIMS that report only their front position or non-ERTMS trains that do not report any information about their position. We take into account most of the main features of the case study. Our model is decomposed into four refinements. All proof obligations have been discharged using the Rodin provers, except those related to the computation of the VSS state machine, which was found to be ambiguous (nondeterministic). Our model has been validated using ProB. The main safety property, which is that ERTMS trains do not collide, is proved.

Keywords: Hybrid ERTMS/ETCS level 3 · Event-B · ProB
Control system

1 Introduction

This paper proposes an Event-B model of the hybrid ERTMS/ETCS level 3 case study [5] proposed for ABZ2018. The case study concerns the European Rail Traffic Management System (ERTMS), the system of standards for management and interoperation of signalling for railways by the European Union. For the sake of concision, we only provide a brief overview of the case study. The reader is referred to [2] for more details.

This paper is structured as follows. In Sect. 2, we summarize the characteristics of the standard that we have taken into account in our model. In Sect. 3, we

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 353–366, 2018.
https://doi.org/10.1007/978-3-319-91271-4_24

describe our modelling strategy, explaining how we take into account controller and environment characteristics, while in Sect. 4, we present an overview of our Event-B model. We describe the refinement strategy, explaining the order in which the various features of the ERTMS were taken into account. In Sect. 5, we describe each refinement. In Sect. 6, we discuss how the requirements and our specification of it have been verified. We conclude in Sect. 7 with an appraisal of this work. In the sequel, we suppose that the reader can read the case study text, in order to avoid unnecessary repetitions.

2 Modelled Characteristics

We model the hybrid level 3 of the standard, which assumes that trains may be equipped with an on-board train integrity monitoring system (TIMS) and that they report their position and integrity to the train supervisor (the system controller, called the *trackside* in the case study), ERTMS trains not fitted with TIMS that report only their front position or non-ERTMS trains that do not report any information about their position. We assume that trains move on a single track, all in the same direction. We also take into account trains that can enter and move on the track without reporting their position to the supervisor (*i.e.*, non-ERTMS trains).

A track is divided into sections called TTD (Trackside Train Detection). A TTD is equipped with sensors that can detect the presence of an object, which can be a train, or something else; it cannot identify a train with this sensor. A TTD is further divided into subsections called VSS (virtual sub-section). The TIMS can be used to determine the VSS occupied by the train and the train's integrity. A train can lose its integrity by splitting into several parts.

The supervisor periodically computes an MA (Movement Authority) and sends it to ERTMS Trains. An MA specifies the VSS that the train can move up to, but never beyond, in order to avoid collision with another train ahead. As stated in the case study, the computation of MAs is out of scope; we simply nondeterministically choose an MA that avoids a collision with the trains ahead. Trains can be connected or disconnected with the supervisor. When connected, a train reports its position and integrity to the supervisor on a regular basis. Timers are used to detect disconnected trains and to manage ghost trains. A ghost train is either a physical object that is present on the track and detected by a TTD, but for which no position report has been received, or a failure of the TTD sensors which incorrectly report the presence of an inexistant object.

3 Modelling Conventions

We reuse the terminology introduced in [8]. A control system interacts with its environment using sensors and actuators. A sensor measures the value of some environment characteristic m, called a *monitored* variable (*e.g.*, train on a track), and provides this measure (*e.g.*, detection of an object on the track) to the software controller as an *input* variable i. In a perfect world, we have $m = i$,

but a sensor may fail. The software controller can influence the environment by sending commands, called *output* variable o to actuators. An actuator influences the value of some characteristics of the environment, call a *controlled* variable c. Variables m and c are called *environment variables*. Variables i and o are called *controller variables*. Finally, a controller has its own internal state variables to perform computations. In this case study, we use Event-B state variables to represent both environment and controller variables.

4 Model Overview

Event-B models are iteratively constructed using refinement. A model component can either be a context or a machine. A context contains constants declaration. A machine contains events that modify state variables. A machine can refine another machine; a context can extends another context. A machine can see contexts to have access to its constants. Each refinement adds new information to the model; these could be new state variables, data refinement of state variables, new events or new properties. Event-B refinement [1] allows for guard strengthening, non-determinism reduction, and new events introduction. New events of a machine M' that refines M are considered to refine the skip event of M, hence they cannot modify a variable introduced in M. Consequently, all events that need to modify a variable v are introduced where v is first declared.

Our model contains three contexts. Context C0 declares constants related to the track. We consider a single track which is represented by an interval of natural numbers $minTTD \mathinner{..} maxTTD$. A stronger typing using an abstract set TTD would be more type safe, but it makes the proofs more cumbersome, as we have experienced in the first drafts of our specification. This is why each TTD is represented by a natural number of this interval. TTDs are ordered using their number. The set of trains is partitioned into trains or cars (*i.e.*, cars that have accidentally split from a train). Constant *trainKind* indicates for each actual train whether it is a TIMS train, a ERTMS train or a non-ERTMS train. Only TIMS and ERTMS trains can connect and send their information to the supervisor.

CONTEXT C0
SETS
TRAINS *StateTTD* *TrainKind*
CONSTANTS
freeT *occupiedT* *Ttds* *minTTD* *maxTTD* *TimErtms* *Ertms* *NoErtms* *Trains* *Cars*
AXIOMS
axm1 : $finite(TRAINS)$
axm2 : $partition(StateTTD, \{freeT\}, \{occupiedT\})$
axm3 : $partition(TRAINS, Trains, Cars)$
axm4 : $minTTD \in \mathbb{N}_1 \land maxTTD \in \mathbb{N}_1 \land minTTD \leq maxTTD$
axm7 : $Ttds = minTTD \mathinner{..} maxTTD$

axm8 : $partition(TrainKind, \{TimErtms\}, \{Ertms\}, \{NoErtms\})$
axm9 : $trainKind \in Trains \rightarrow TrainKind$

Context C1 declares the VSSs, which are also modelled as an interval of naturals. We use a total, monotonic, surjective function *TtdOfVss* to associate a VSS to its TTD.

axm4 : $Vss = minVSS \mathinner{..} maxVSS$
axm5 : $TtdOfVss \in Vss \twoheadrightarrow Ttds$
axm6 : $\forall v_1, v_2 \cdot \{v_1, v_2\} \subseteq Vss \wedge v_1 < v_2 \Rightarrow TtdOfVss(v_1) \leq TtdOfVss(v_2)$

Context C2 declares an abstract set $StateVSS = \{freeV, occupiedV, unknown, ambiguous\}$ to represent the states of a VSS from the supervisor view point. A VSS in state *freeV* contains no train. A VSS in state *occupiedV* contains a single train. State *unknown* denotes a VSS for which it is unknown whether there is a train on it. State *ambiguous* denotes a VSS which contains at least one train; it is not sure whether there are more than one train.

The specification is structured into four refinement steps (*i.e.*, four machines). Machine M0 introduces the trains, the supervisor and the unsupervised movements of trains on TTDs. Machine M1 introduces the reporting of positions by trains to the supervisor, but still without supervision of their movement. Machine M2 introduces the VSS, still without supervision. Machine M3 introduces movement supervision with MAs, and the computation of VSS states using timers and other informations. A final refinement M4 is introduced to prove the main safety property, namely that trains do not collide when following MAs.

5 Refinements

In this section, we briefly describe each refinement. The complete archive of the Event-B project is available in [7].

5.1 Machine M0: Free Movement on TTDs

This machine contains five variables. Controller variable *stateTTD* faithfully represents the real state of TTDs (*i.e.*, the case study assumes $m = i$ for this variable). Environment variables *trainOccupationTTDRear* and *trainOccupationTTDFront* respectively denote the first and last TTD occupied by a given train. Environment variable *isConnected* denotes whether a train is connected to the supervisor. This variable denotes a total function including the trains that are not on track because some of them should be connected to receive the authorization to enter on the track. Boolean variable *trainMvt* is used to guard train movements to ensure that other events like train supervision are interleaved with train movements. The following invariants type these variables. Symbols "$\rightarrow$" and "$\nrightarrow$" respectively denote a total function and a partial function.

inv1 : $stateTTD \in Ttds \rightarrow StateTTD$
inv2 : $trainOccupationTTDFront \in TRAINS \nrightarrow Ttds$
inv3 : $trainOccupationTTDRear \in dom(trainOccupationTTDFront) \rightarrow Ttds$
inv4 : $\forall tr \cdot tr \in dom(trainOccupationTTDFront) \Rightarrow$
$trainOccupationTTDRear(tr) \leq trainOccupationTTDFront(tr)$
inv5 : $isConnected \in trainKind^{-1}[\{Ertms, TimErtms\}] \rightarrow BOOL$
inv6 : $trainMvt \in BOOL$

The set of trains on the track is represented by the domain of function *trainOccupationTTDFront* (*i.e.*, $dom(trainOccupationTTDFront)$). We consider events that model the sensing of all the TTD states by the supervisor, the entering and exiting of a train on the track, the movement of a train on the track, the connection and disconnection of a train. The movement of a train is decomposed into three events to distinguish between the cases where the train moves within the same TTD, the front of the train enters a new TTD and the rear of the train leaves a TTD. This decomposes the proofs for train movement into smaller ones. Trains move freely and collisions can occur at this level. The supervisor does not know the position of a train; it only knows the states of TTDs. Also, we have defined an event to split a train into two parts, the train with the engine and the cars left behind, to model the loss of integrity. As a simple illustration, we provide below the specification of event trainSupervisor.

Event trainSupervisor $\widehat{=}$
any *ttds active*
where
grd1 : $ttds = (\bigcup tr \cdot tr \in dom(trainOccupationTTDFront) \mid$
$trainOccupationTTDRear(tr) \mathinner{..} trainOccupationTTDFront(tr))$
grd2 : $active \in BOOL$
then
act1 : $stateTTD := (ttds \times \{occupiedT\}) \cup ((Ttds \setminus ttds) \times \{freeT\})$
act2 : $trainMvt := active$
end

Guard grd1 constrains event local variable *ttds* to the set of TTDS which are occupied by trains. Action act1 updates TTD states. Action act2 nondeterministically gives controls to either the trains or the supervisor using the choice made in guard grd2.

5.2 Machine M1: Trains Reporting Their Positions

This machine adds controller variables *trainLocationTTDRear* and *trainLocationTTDFront* to store in the supervisor train positions as reported by ERTMS trains. The case study assumes that reports are accurate. The following invariants provide the types of these variables. Note that the location of a train on a track may be unknown to the supervisor. Thus, *trainLocationTTDFront* is modeled as a partial function of the domain of *trainOccupationTTDFront*, which

denotes the real train position. Invariant inv3 states that the rear is known only for TIMS ERTMS trains that have already provided their front positions.

$$\begin{aligned}
&\texttt{inv1} : trainLocationTTDFront \in \\
&\quad dom(Trains \lhd trainOccupationTTDFront) \pfun Ttds \\
&\texttt{inv2} : trainLocationTTDRear \in dom(trainLocationTTDFront) \pfun Ttds \\
&\texttt{inv3} : trainKind^{-1}[\{TimErtms\}] \cap dom(trainLocationTTDFront) \\
&\quad \subseteq dom(trainLocationTTDRear) \\
&\texttt{inv4} : \forall tr \cdot tr \in dom(trainLocationTTDFront) \Rightarrow \\
&\quad trainLocationTTDRear(tr) \leq trainLocationTTDFront(tr)
\end{aligned}$$

This refinement introduces a new event, trainSnd, to report train positions. Existing events are refined (extended) to take into account the new variables. Event trainSnd reports the position of a train by modifying controller variables *trainLocationTTDRear* and *trainLocationTTDFront* using the environment variables *trainOccupationTTDRear* and *trainOccupationTTDFront*. Train integrity is nondeterministically chosen to reflect the possibility of loosing it at any point. When train integrity is lost, the rear position of a train is not updated, in order to ensure that its last known rear position remains and to avoid collision with the preceding train when computing the MA. However, there is no provision in M1 to avoid collision; this is introduced in M3.

The specification of event trainSnd is provided below. Action act2 simulates an if-then-else by using a set containing two tuples of the form $\{TRUE \mapsto e_1, FALSE \mapsto e_2\}$; hence this set is a function and it is evaluated with the value of *integ*, acting like **if** *integ* **then** e_1 **else** e_2. Guard grd6 ensures that the reported position does not decrease, since a train cannot move backward.

$$\begin{aligned}
&\textbf{Event}\ \ \mathsf{trainSnd} \mathrel{\widehat{=}} \\
&\quad \textbf{any}\ \ tr\ \ integ\ \ lengch \\
&\quad \textbf{where} \\
&\qquad \texttt{grd2} : tr \in dom(trainOccupationTTDFront) \\
&\qquad \texttt{grd3} : tr \in dom(isConnected) \land isConnected(tr) = TRUE \\
&\qquad \texttt{grd4} : integ \in BOOL \\
&\qquad \texttt{grd5} : tr \in trainKind^{-1}[\{TimErtms\}] \land tr \notin dom(trainLocationTTDRear) \Rightarrow \\
&\qquad\quad integ = TRUE \\
&\qquad \texttt{grd6} : tr \in dom(trainLocationTTDFront) \Rightarrow \\
&\qquad\qquad trainOccupationTTDFront(tr) \geq trainLocationTTDFront(tr) \\
&\quad \textbf{then} \\
&\qquad \texttt{act2} : trainLocationTTDRear := \\
&\qquad\qquad \{TRUE \mapsto trainLocationTTDRear \lhd\!\!+ \\
&\qquad\qquad\quad \{tr \mapsto trainOccupationTTDRear(tr)\}, \\
&\qquad\qquad\ FALSE \mapsto trainLocationTTDRear\}\ (integ) \\
&\qquad \texttt{act3} : trainLocationTTDFront(tr) := trainOccupationTTDFront(tr) \\
&\quad \textbf{end}
\end{aligned}$$

5.3 Machine M2: Introducing VSSs

Recall that a TTD is divided into VSSs. This refinement data replaces (*i.e.*, data refines) the train position variables based on TTDs (*i.e.*, *trainOccupationTTDx*

and *trainLocationTTDx*) with position variables based on VSSs. New environment variables *trainOccupationVSSRear* and *trainOccupationVSSFront* represent the real VSS position of a train. New controller variables *trainLocationVSSRear* and *trainLocationVSSFront* represent the VSS positions computed by the supervisor using train reports.

$$\texttt{inv5} : trainLocationVSSFront \in dom(trainLocationTTDFront) \rightarrow Vss$$
$$\texttt{inv6} : trainLocationVSSRear \in dom(trainLocationVSSFront) \rightarrow Vss$$
$$\texttt{inv7} : trainOccupationVSSFront \in dom(trainOccupationTTDFront) \rightarrow Vss$$
$$\texttt{inv8} : trainOccupationVSSRear \in dom(trainOccupationVSSFront) \rightarrow Vss$$

Four gluing invariants stating that the VSS positions and the TTD positions are consistent, for both the controller and the environment, using function *TtdOfVss*, are also added, like the following one.

$$\texttt{inv11} : \forall tr \cdot tr \in dom(trainOccupationVSSFront) \Rightarrow TtdOfVss(trainOccupationVSSFront(tr)) = trainOccupationTTDFront(tr)$$

No new event is added. The existing events are refined to take into account these new variables. As in M1, train collisions can occur in M2.

5.4 Machine M_3: Computing VSS States and Assigning MAs

Introducing New Variables. This refinement is the most complex one. The state of each VSS is computed and MAs are assigned to trains. At this level, the integrity and the length information of a train are stored by two Boolean variables since they are used in the VSS computation. Timers are introduced to detect disconnected trains, the loose of integrity and ghost trains. New variables are introduced and typed using the following invariants.

$$\texttt{inv1} : MATrainRear \in dom(trainLocationVSSFront) \nrightarrow Vss$$
$$\texttt{inv2} : MATrainFront \in dom(MATrainRear) \rightarrow Vss$$
$$\texttt{inv3} : \forall tr \cdot tr \in dom(MATrainRear) \Rightarrow MATrainRear(tr) \leq MATrainFront(tr)$$
$$\texttt{inv4} : \forall tr1, tr2 \cdot tr1 \in dom(MATrainFront) \wedge tr2 \in dom(MATrainFront) \wedge tr1 \neq tr2 \Rightarrow MATrainRear(tr1) \mathinner{..} MATrainFront(tr1) \cap MATrainRear(tr2) \mathinner{..} MATrainFront(tr2) = \varnothing$$

Controller variables *MATrainRear* and *MATrainFront* define the MA of each train that the supervisor knows. An MA is an interval of VSSs. Invariant `inv4` states that the MAs of trains are disjoint, to avoid collisions.

The next invariants introduce timers; timers related to trains may be *running* or *expired* while those associated with the VSS and TTD may be *running* or *expired* but also *inactive*.

$\texttt{inv5}$: $muteTimer \in dom(trainLocationVSSFront) \rightarrow \{running, expired\}$
$\texttt{inv6}$: $integrityTimer \in dom(trainLocationVSSFront) \rightarrow \{running, expired\}$
$\texttt{inv7}$: $disconnectTimer \in Vss \rightarrow \{inactive, running, expired\}$
$\texttt{inv8}$: $ghostTimer \in Ttds \rightarrow \{inactive, running, expired\}$

The *muteTimer* is used to detect that a train has failed to report its position within the required time frame; in that case, the state of the VSSs in front of that train and within the train's MA becomes *unknown*.

Finally, the following variables are introduced to compute the VSS states.

$\texttt{inv9}$: $currentStateVSS \in Vss \rightarrow StateVSS$
$\texttt{inv10}$: $previousFront \in dom(trainLocationVSSFront) \nrightarrow Vss$
$\texttt{inv11}$: $previousFrontState \in dom(previousFront) \nrightarrow StateVSS$

Variables *previousFrontState* and *previousFront* respectively record the previous value of *currentStateVSS* and the previous front position of the trains. They are respectively updated when the supervisor computes the states of the VSS and when the train reports its position; they are needed in the computation of some VSS state transitions.

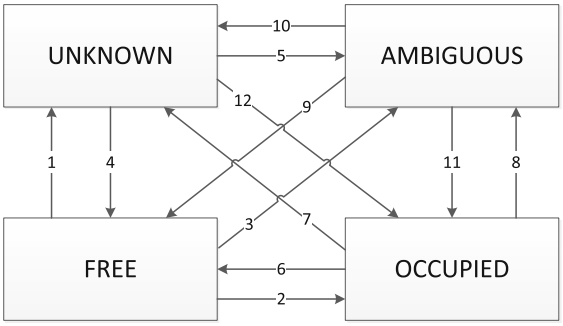


Fig. 1. The state machine of VSS reproduced from Fig. 7 of [2]

Modelling VSS State Machine Transitions. The main complexity of this refinement is to compute the VSS states, which depend on several conditions. These conditions are described by a state machine in Fig. 7 of [2] and reproduced here in Fig. 1. The guards of its transitions are described, using natural language, in Table 2 of [2]. This table spans 3.5 pages (pp. 24–28). Figure 2 provides an excerpt of this table. The guard of a transition i in Fig. 1 is given by the disjunction of the guards labeled $\texttt{\#}iX$ in Fig. 2. For example, the guard of transition 1 is $\texttt{\#1A} \vee \texttt{\#1B} \vee \ldots \texttt{\#1F}$; only `#1A` and `#1B` are shown in Fig. 2. Some transitions have priority over others (*e.g.*, guards `#2A` and `#2B` have precedence over transition 3).

Ideally, the computation of the state of each VSS should be done in a single event, because the states must be all computed before assigning MAs. It also ensures that Table 2 of [2] is deterministic, *i.e.*, well-defined. Furthermore, it

allows for taking into account the priority between transitions for a given VSS. We have coded the state machine of Fig. 1 into a single event, namely trainSupervisor. We use guard numbers (*e.g.*, `#1A`) to name local variables of the event (*e.g.*, *vss1A*). Such a variable is constrained to contain the new state values for the VSSs satisfying the corresponding guard. For instance, set *vss1A* contains the VSSs satisfying guard `#1A` and their state will change from FREE to UNKNOWN. The union of sets *vss iX* is used to update state variable *currentState VSS* in event trainSupervisor.

To illustrate our approach, we provide in Fig. 3 an excerpt of the guards of event trainSupervisor that models guards `#1A` and `#1B` of Fig. 2. Guard `grd4` of Fig. 3 represents guard `#1A`. We use a quantified union to identify the VSSs satisfying `#1A`. It reads as follows: a VSS must currently be free, since transition 1 start from state FREE; it must also be on an occupied TTD (first conjunct of guard `#1A`) and any VSS of this TTD must not be within an MA or occupied by a train (second conjunct of `#1A`). The resulting state of these VSSs is UNKNOWN as given by transition 1, which is represented by taking the Cartesian product of the VSSs returned by the quantified union with the singleton set {*unknown*}. In summary, guard `#1A` says that the TTD sensor detected an object, but the supervisor has no record of a train on a VSS of that TTD, thus it's status is unknown.

#1A		(TTD is occupied)
	AND	(no FS MA is issued or no train is located on this TTD)
#1B		(TTD is occupied)
	AND	(VSS is part of the MA sent to a train for which the mute timer is expired)
	AND	(VSS is located in advance of the VSS where the train was last reported)

Fig. 2. An excerpt of Table 2 in [2]

6 Requirements Verification and Model Validation

This section describes the verifications carried out using the provers of Rodin (Event-B's development platform) and the model checker/animator ProB [6] plug-in for Rodin. ProB is an explicit state-based model checker for the B methods (classic B and Event-B) and several others (TLA, CSP, Alloy). Our strategy to verify the development and the requirements is as follows. We used ProB mainly to discover possible invariant violations prior to the proof phase that may be long and complex. ProB has proved to be a useful and effective tool to check the sequencing of the events. We have also used it to play the scenarios provided in the case study to validate our specification.

$$
\begin{aligned}
&\texttt{grd4} : vss1A = \\
&\quad (\bigcup vs \cdot currentStateVSS(vs) = freeV \quad \wedge \\
&\qquad vs \in ttds \quad \wedge \\
&\qquad ((\forall tr \cdot tr \in dom(MATrainFront) \Rightarrow TtdOfVss(vs) \notin \\
&\qquad\quad TtdOfVss(MATrainRear(tr)) .. \\
&\qquad\quad TtdOfVss(MATrainFront(tr))) \quad \vee \\
&\qquad\quad (\forall tr \cdot tr \in dom(trainLocationVSSFront) \Rightarrow TtdOfVss(vs) \notin \\
&\qquad\quad TtdOfVss(trainLocationVSSRear(tr)) .. \\
&\qquad\quad TtdOfVss(trainLocationVSSFront(tr)))) \\
&\qquad \mid \{vs\}) \times \{unknown\} \\
&\texttt{grd5} : vss1B = \\
&\quad (\bigcup vs \cdot currentStateVSS(vs) = freeV \quad \wedge \\
&\qquad vs \in ttds \quad \wedge \\
&\qquad (\exists tr \cdot (tr \in dom(muteTimer) \quad \wedge \\
&\qquad\quad muteTimer(tr) = FALSE \quad \wedge \\
&\qquad\quad vs \in MATrainRear(tr) .. MATrainFront(tr)) \quad \wedge \\
&\qquad\quad vs \geq trainLocationVSSFront(tr)) \\
&\qquad \mid \{vs\}) \times \{unknown\}
\end{aligned}
$$

Fig. 3. An excerpt of the guards of trainSupervisor corresponding to Fig. 2

6.1 Proving Safety Properties

We have stated one main safety property, which is that two TIMS/ERTMS trains cannot be on the same VSS, and thus TIMS/ERTMS trains should not collide, but non-ERTMS trains could. This property is expressed using the environment variables *trainOccupationVSSRear* and *trainOccupationVSSFront*, which represent the real position of the trains (not the position as known by the supervisor). This proof was conducted in a new refinement machine M4, for the sake of modularity.

$$
\begin{aligned}
&\texttt{inv1} : \forall tr1, tr2 \cdot tr1 \in Trains \wedge tr2 \in Trains \wedge tr1 \neq tr2 \wedge \\
&\qquad tr1 \in dom(trainOccupationVSSFront) \quad \wedge \\
&\qquad tr2 \in dom(trainOccupationVSSFront) \quad \wedge \\
&\qquad trainKind(tr1) \in \{TimErtms, Ertms\} \wedge \\
&\qquad trainKind(tr2) \in \{TimErtms, Ertms\} \\
&\quad \Rightarrow \\
&\qquad trainOccupationVSSRear(tr1) .. trainOccupationVSSFront(tr1) \\
&\qquad \cap \\
&\qquad trainOccupationVSSRear(tr2) .. trainOccupationVSSFront(tr2) \\
&\qquad = \varnothing
\end{aligned}
$$

The guards of events that modify these variables are based solely on the controller variables, and thus represent the fact that trains move according to their MAs computed by the supervisor. If the invariant holds, it means that trains following their MAs should not collide.

To prove this property, we needed to add and prove the following invariants, which can be seen as lemmas required for the main proof.

$$\texttt{inv2} : \forall tr \cdot tr \in dom(trainOccupationVSSFront) \quad \wedge$$
$$tr \in Train \wedge trainKind(tr) \in \{TimErtms, Ertms\} \Rightarrow$$
$$tr \in dom(MATrainFront) \wedge$$
$$trainOccupationVSSRear(tr) \mathinner{..} trainOccupationVSSFront(tr)$$
$$\subseteq$$
$$MATrainRear(tr) \mathinner{..} MATrainFront(tr)$$
$$\texttt{inv3} : \forall tr \cdot tr \in dom(trainOccupationVSSFront) \wedge$$
$$tr \in dom(trainLocationVSSRear) \Rightarrow$$
$$trainOccupationVSSRear(tr) \geq trainLocationVSSRear(tr)$$
$$\texttt{inv4} : \forall tr \cdot tr \in dom(trainOccupationVSSFront) \wedge$$
$$tr \in dom(trainLocationVSSRear) \Rightarrow$$
$$trainOccupationVSSFront(tr) \geq trainLocationVSSFront(tr)$$

Invariant inv2 tates that a TIMS/ERTMS train can occupy only the VSS included in its MA. Invariants inv3 and inv4 state that the position of a train known by the supervisor is behind the real position of the train. Recall that the case study assumes that the position reported by trains are accurate.

6.2 Proving the Determinacy of the VSS State Machine

Recall that state variable *currentStateVSS* is typed as a function. The proof obligation generated by this typing invariant ensures that each VSS state has a single new value, hence there is a single transition that updates it. This is equivalent to proving that the VSS state machine described in the case study is deterministic. This turns out to be fairly complex. For each VSS state value (*e.g.*, FREE), there are three outgoing transitions to the other three possible VSS state values (*e.g.*, transitions 1, 2 and 3 of Fig. 1). To ensure determinacy, we must prove that the guards of these three transitions are mutually disjoint. Let n_i be the number of disjuncts in the disjonctive normal form of the guard of transition i. Then we have to consider $n_i * n_j$ cases in the proof of disjointness of transitions i and j. Luckily, transitions priorities eliminate a few cases to consider. In total, there are 47 high-level cases to consider, which is a significant proof effort.

One way to simplify the handling of this proof in Rodin would be to decompose event trainSupervisor into four events, one for each VSS state value. That would still allow us to prove the determinacy of the VSS state machine, but we would lose the atomicity of VSS state computation. We would then have to control the ordering of events to ensure that these four events are computed before assigning MAs. For the sake of simplicity and to ease the construction of the overall specification, we have chosen to use a single event.

Using ProB to Check the Determinacy of the VSS State Machine. ProB can be used to find invariant violations with counterexamples. We have used this feature extensively. The counter-examples provided help in identifying the missing guards and invariants required to prove invariant preservation. However, the state space of machine M3 is huge, with its 22 variables, most of them typed as functions. ProB will only check the reachable states, and when

it does not terminate in a reasonable time, one cannot determine which interesting conditions have been explored, for instance among the 47 cases of guard disjointness discussed earlier.

An alternative way to check the determinacy of the VSS state machine is to use the constraint satisfier of ProB, which can find models for a formula. ProB uses it to find values of constants in an Event-B context. To specifically check one case among the 47 cases for the determinacy of the VSS state machine, we construct a new context that declares the state variables, used in the guards of the VSS state machine, as constants, and their related invariants as axioms. We finally add to this context the local variables of event trainSupervisor that computes new sets of VSS states and we check that these two sets are not disjoint (*e.g.*, check that $dom(vss1A) \cap dom(vss2A) \neq \{\}$). If ProB finds a model for this context, it means that the corresponding transition guards in the VSS state machine are not disjoint, given the invariants used in our machine. It thus means that the invariants are insufficient to prove the determinacy of the VSS state machine and that they must enriched or strengthened.

Dealing with Inconsistencies of the VSS State Machine. We have found several cases where the guards are not disjoint, which means that one of the following three alternatives holds: (i) our representation of the guards are incorrect, (ii) the case study text is incorrect, (iii) invariants are missing to rule out these counterexamples (*i.e.*, these Event-B states are not reachable from the initial state of the system). Since we are not expert of the ERTMS standard, it is hard for us to determine which alternative holds. In a first model of the system ProB finds several counterexamples when searching for invariant violations, that leads to a state where two transitions are not disjoint. Such traces are due, for instance, to the expiration of several timers reported at the same moment as the reporting of the train position. Thus, we do not know if the case study is wrong, or if this trace is impossible in the real world where the timers represent actual clocks with different values or perhaps there are implicit assumptions in the case study that we missed or we could not figure out by simply reading it. To rule them out, we assume that the transitions depending on the timers are dealt with last; priority is given to those depending on the train position. From the Event-B point of view, we use the overload operator to express it. Moreover, as for the representation of the guards, we have used a straight forward translation of the phrasal terms of the natural language text into state variables, to simplify as much as possible the translation of the guards. However, there is still the possibility of misinterpreting the natural language text. For instance, consider the following conjunct of `#2A`.

`#2A`		...
	AND	(VSS where the estimated front end of the train was last reported, was "occupied" after the processing of this previous position report)

This conjunct can be interpreted as an implication, which means that the guard holds even when only one position report has been issued for the train. Or, it can be interpreted as a conjunction, which means that at least two position reports must have been issued for the train, for the guard to hold. Given the length of the case study, our limited expertise in the domain and the number of ambiguities or missing (implicit) assumptions, we decided not to elicit further these aspects, because there is no point in making hypothetical (as opposed to "realistic") assumptions in order to prove the determinacy of the VSS state machine. The key issue is more to be able to identify ambiguities, thanks to formalisation, validation and verification. In a real context, they can be resolve in a systematic manner using domain experts. Moreover, since proof obligations can be independently discharged, not proving the determinacy of the state machine does not prevent us from proving the main safety property; we can assume that the VSS state machine can be made deterministic. In addition, the four VSS states can be reduced to only two (free or occupied), since the other two are used to manage potentially hazardous situations, as noted in the case study (paragraph 3.2.1.1.1 of [2]).

7 Conclusion

Our model covers the essential parts of the case study. We were able to prove the safety of TIMS/ERTMS trains. It only remains to prove the determinacy of the VSS state machine, which could not be completed because of the ambiguities of the case study text. Understanding the case study itself was a challenge, because of the difficulty to identify missing assumptions. Determining the ordering of events was anything but trivial. Domain experts typically write for other domain experts; it is not natural for them to think of all the details that a non-expert does not know.

We have found Event-B to be adequate to model this case study. In this paper, we deliberately chose not to use any Event-B plugins (*e.g.*, [3,9]) in order to be able to compare our solution with solutions based on them (and assuming that a paper using them will be submitted to ABZ2018). In a companion paper, we explore the use of ontologies and SysML/KAOS to model this case study [4].

Acknowledgements. This research was supported in part by NSERC (Natural Sciences and Engineering Research Council of Canada) and the FORMOSE project funded by the French National Research Agency (ANR).

References

1. Abrial, J.: Modeling in Event-B. Cambridge University Press, Cambridge (2010)
2. EEIG Ertms Users Group: Hybrid ERTMS/ETCS Level 3: Principles. Technical report, Brussels, Belgium, July 2007
3. Fathabadi, A.S., Butler, M.J., Rezazadeh, A.: Language and tool support for event refinement structures in Event-B. Formal Asp. Comput. **27**(3), 499–523 (2015)

4. Fotso, S.J.T., Frappier, M., Laleau, R., Mammar, A.: Modeling the Hybrid ERTMS/ETCS Level 3 Implementation through Goal Diagrams and Ontologies Using the FORMOSE Approach, February 2018. http://info.usherbrooke.ca/mfrappier/abz2018-ERTMS-Case-Study-Formose
5. Hoang, T.S., Butler, M., Reichl, K.: The hybrid ERTMS/ETCS level 3 case study. Technical report, ECS, University of Southampton, U.K. (2007)
6. Leuschel, M., Butler, M.: ProB: a model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45236-2_46
7. Mammar, A., Frappier, M., Fotso, S.J.T., Laleau, R.: An Event-B Model of the Hybrid ERTMS/ETCS Level 3 Standard, February 2018. http://info.usherbrooke.ca/mfrappier/abz2018-ERTMS-Case-Study
8. Parnas, D.L., Madey, J.: Functional documents for computer systems. Sci. Comput. Program. **25**(1), 41–61 (1995)
9. Said, M.Y., Butler, M.J., Snook, C.F.: A method of refinement in UML-B. Softw. Syst. Model. **14**(4), 1557–1580 (2015)

Short Papers

AsmetaA: Animator for Abstract State Machines

Silvia Bonfanti[1(✉)], Angelo Gargantini[1], and Atif Mashkoor[2,3]

[1] Department of Economics and Technology Management, Information Technology and Production, University of Bergamo, Bergamo, Italy
{silvia.bonfanti,angelo.gargantini}@unibg.it
[2] Software Competence Center Hagenberg GmbH, Hagenberg, Austria
atif.mashkoor@scch.at
[3] Johannes Kepler University Linz, Linz, Austria
atif.mashkoor@jku.at

Abstract. In this paper, we present AsmetaA – a graphical animator for Abstract State Machines integrated within the ASMETA framework. The execution of formal specifications through animation provides several advantages, e.g., it provides an immediate feedback about system behavior, it helps understand system evolution, and it increases the overall acceptability of formal methods.

1 Introduction

One important feature of the Abstract State Machines (ASM) method [3] is that it allows to *execute* specifications that represent the evolution of a system by a *sequence of states*. An important advantage of the state-based execution is that it helps users understand through experimentation the behavior of the system being designed [7].

The ASMETA framework [4] provides an environment for systems development using ASMs including a simulator. During simulation, the user can interact with the simulator by inserting monitored values when required and observe the system evolution. The user can drive and follow the ASM execution and understand whether the specification really captures the intended system behavior. However, the simulation engine currently available for the ASMETA platform provides only a textual interface that prints the states as strings with some optional messages on the console. Observing system evolution in this fashion can be difficult. For this reason, the need for a graphical animator for the ASMETA platform has been felt for a long time.

The graphical animation of specifications consists in showing by means of graphical elements, e.g., tables and colors, the evolution of the system state.

The research reported in this paper has been partly supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the Province of Upper Austria in the frame of the COMET center SCCH.

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 369–373, 2018.
https://doi.org/10.1007/978-3-319-91271-4_25

This provides several advantages: the user can perform a rapid validation, it helps in understanding the system behavior, and it shows concrete scenarios in which abstract states can be instantiated. For this reason, many formal notations and tools (see Sect. 4) support this kind of validation technique.

In this paper, we present AsmetaA – a graphical animator for the ASMETA platform, which shows the system execution and state evolution using graphical elements. It is integrated in the framework and it can be downloaded and installed as an eclipse plug-in[1]. The rest of the paper is organized as follows: In Sect. 2, we discuss the main goal of this work. In Sect. 3, we briefly present the AsmetaA tool. A brief comparison of AsmetaA with similar tools is presented in Sect. 4. The paper is concluded in Sect. 5.

2 Animation of ASMs: Requirements and Goals

In the wake of a recent effort for providing visual information to users of the ASMETA framework, we have already developed a visualizer that provides a graphical view of ASMETA models [2]. The visualizer provides information about the structure of the machine, in terms of a set of construction rules and schemas that give a graphical representation of an ASM and its rules. However, in current settings, information about system dynamics is missing. For this reason, we started to work on the concept of *animation* of ASM specifications. In our tool, *animation* has the following main objectives:

1. Providing a user with complete information about all the locations in one *state*. In this way, the user can understand the system state at every step.
2. Showing the *evolution* of an ASM during the execution. In this way, the user can understand the behavior of the specification.
3. Using colors, tables, and figures over simple text to convey information about states and their evolution.

In order to achieve these goals, we decided to structure the animator as shown in Fig. 1. The table captures the two dimensions of locations in one state and their evolution. On the horizontal axis, we want to represent the evolution of the execution, by showing the sequence of states. On the vertical axis, we want to show the states, i.e., locations and their values at each state.

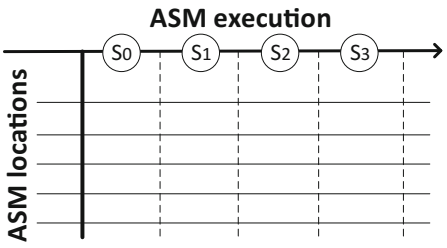


Fig. 1. Animation of an ASM

As an auxiliary goal of the animator, we use graphical elements also for user interaction and avoid the use of textual consoles whenever possible. Additionally, we cope with the specification complexity by allowing the user to highlight some locations of interest.

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

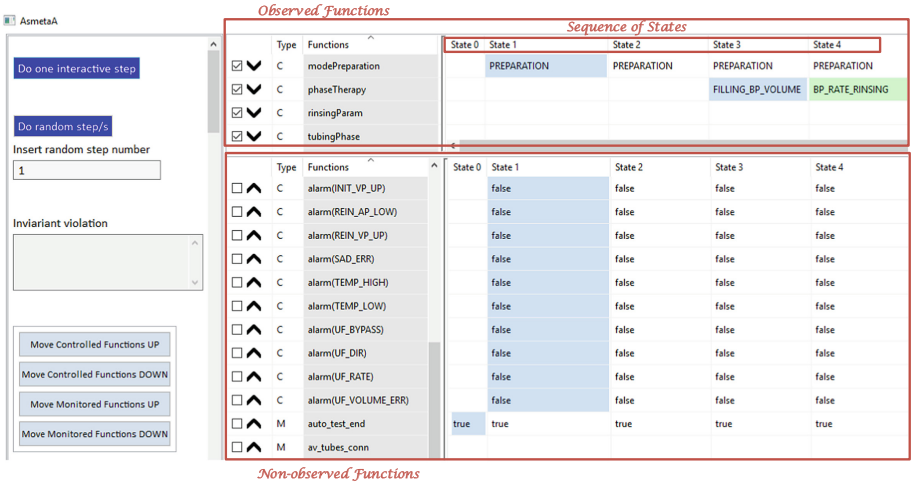


Fig. 2. AsmetaA tool

3 AsmetaA: Animator for ASM Specifications

A graphical view of the AsmetaA tool is shown in Fig. 2. We now describe various characteristics of this tool.

Random vs interactive animation. Random and interactive animations are two modalities of execution provided by the animator. Random animation runs the ASM specification and the values of monitored functions are chosen by the animator. The number of steps is selected by the user and it can be changed dynamically. Interactive animation runs one step at a time and the value of monitored functions are selected by the user. These two modalities of animation can be mixed within the same run. This means that one state can be reached using random animation and the next state can be reached using interactive animation. The two modalities of animation are executed using the following corresponding buttons.

- *Do one interactive step*: asks the user about the value of monitored functions and runs one step. The monitored functions are inserted through a dialog box.
- *Do random step/s*: runs one or more steps based on the number inserted in the field *Insert random step number*.

In case of invariant violation, the message is shown in the dedicated text box.

Random simulation: multiple steps. With complex specifications, running one random step each time is tedious. To overcome this limit, we have added a field in which the user inserts the number of steps to be performed and the tool performs the random simulation accordingly.

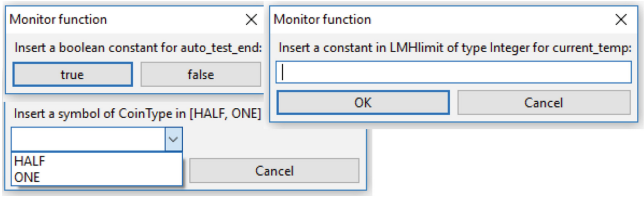


Fig. 3. GUI dialogs allow the user to input new values for monitored functions

Use of tables. The first version of the animator was realized using one table for all functions. However, it was difficult for the user to follow the functions of his/her interest in complex specifications. To cope with this difficulty, we have now added two tables. The upper table contains the functions observed by the user, while the lower table contains all other functions. When the function appears for the first time during the simulation, the animator inserts it in the lower table. If the user is interested to follow this function, he/she has to move it to the upper table using the check box display in the first column. When the user is no longer interested in observing a specific function, he/she can move it into the lower table and still follow other observed functions. The user can also move functions (from one table to the other) that belong to a specific type (controlled functions or monitored functions) using the buttons in the lower left corner shown in Fig. 2. Moreover, the content of the tables can be sorted alphabetically based on the type or name of the functions.

Colors for easy reading. One of the main features of AsmetaA is the usage of multiple colors to facilitate the readability of tables. Multiple colors help a user to identify particular events during the ASM execution: the initial value of the function (light blue cells) and when the function changes the value compared to the previous state (light green cells).

Dialog box. The insertion of monitored functions is achieved through different dialog boxes (see Fig. 3) depending on the type of function to be inserted. For example, in case of a boolean function, the box has two buttons: one if the answer is true and one if the answer is false. By pushing the button, the user assigns the corresponding value to the function. In case of functions with the enumerative domain, the dialog box shows all the possible assignable values and the user selects the chosen value from a combo box. At the moment, for other data types, the user inserts the value in a text box.

4 Related Work

Several formal methods support the animation of specifications. For the B family methods, one of the main tools that provides such facility is ProB [6]. In this tool, the user can select the events to fire while the state is constantly updated

and shown to the user. Differently from our animator, ProB displays the history of events but the history of the system states does not appear.

Visualization of traces is used in TLA+ to show counter examples in case of errors while model checking a specification [8]. Also for the NuSMV model checker, traces can be shown in tables by the NuSeen toolbox [1]. As compared to model checkers, our animator is intended to be used in an interactive way to allow the validation of specifications.

There is already one tool with the goal of animating ASM specifications [5]. In this work, the authors extend CoreASM with some plug-ins to show the state evolution of the ASM specifying a flash file system. As compared to AsmetaA, it is an application specific work focusing on a particular case study. AsmetaA, on the other hand, is a generic animator capable of animating *any* ASM. As a future work, we plan to introduce a method allowing the definition of special graphical widgets for application specific animations.

5 Conclusion

In this paper, we have presented AsmetaA – an animation engine for ASM specifications. The tool supports users in the validation process and concretizes the abstract states. The users can run two types of animation: random and interactive. In initial tests, the graphical interface of AsmetaA has been proved intuitive, simple, and user friendly.

References

1. Arcaini, P., Gargantini, A., Riccobene, E.: NuSeen: a tool framework for the NuSMV model checker. In: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 476–483 (2017)
2. Arcaini, P., Bonfanti, S., Gargantini, A., Riccobene, E.: Visual notation and patterns for abstract state machines. In: Milazzo, P., Varró, D., Wimmer, M. (eds.) STAF 2016. LNCS, vol. 9946, pp. 163–178. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50230-4_12
3. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-642-18216-7
4. Gargantini, A., Riccobene, E., Scandurra, P.: A metamodel-based language and a simulation engine for abstract state machines. J. UCS **14**(12), 1949–1983 (2008)
5. Haneberg, D., Junker, M., Schellhorn, G., Reif, W., Ernst, G.: Simulating a flash file system with CoreASM and eclipse. In: Informatik 2011: Informatik schafft Communities, Beiträge der 41. Jahrestagung der Gesellschaft für Informatik e.V. (GI). LNI, vol. 192, p. 355. GI (2011)
6. Leuschel, M., Butler, M.: ProB: an automated analysis toolset for the B method. J. Softw. Tools Technol. Transf. **10**(2), 185–203 (2008)
7. Mashkoor, A., Jacquot, J.-P.: Validation of formal specifications through transformation and animation. Requir. Eng. **22**(4), 433–451 (2017)
8. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA+ specifications. In: Pierre, L., Kropf, T. (eds.) Correct Hardware Design and Verification Methods, pp. 54–66. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48153-2_6

Formal Specification of the Semantics of Control State Diagrams

Markus Leitz and Alexander Raschke(✉)

Institute of Software Engineering and Programming Languages,
Ulm University, Ulm, Germany
{markus.leitz,alexander.raschke}@uni-ulm.de

Abstract. Control State Diagrams (CSD) are a graphical representation of Control State Abstract State Machines, a subclass of Abstract State Machines (ASM). We extend the existing semi-formal specification of this diagram type by a concrete syntax and its formal semantics. The semantics is given by a translation approach that transforms combinations of nodes into ASM snippets which are inserted into a textual ASM. This node-by-node translation is not only the basis for a code generation tool, but it also allows users to capture the behavior of a CSD more easily.

1 Introduction and Goals

Abstract State Machines (ASMs) are a rigorous system engineering method which guides the developer seamlessly from requirements capture to their implementation [3]. Although ASMs have a mathematical foundation, a developer can correctly understand them as pseudo-code without any special mathematical knowledge.

Many systems inherently split their behavior in finitely many phases or modes, between which can be switched. This general architecture is captured by control state ASMs, a subclass of ASMs. For an easier understanding, control state diagrams (CSDs), a flowchart-like graphical representation of control state ASMs, has been introduced [3]. Besides a simple formal definition in [3], with some extensions described in natural language, there is currently no complete formal specification of such diagrams. As a consequence, different kinds of representation and semantics are used, sometimes without any explanation (e.g. the rhomb with an inner circle in [6]). This can lead to misunderstandings and contradicts the original intention of CSDs.

The goal of our work is to give a precise formal semantics and syntax of CSDs by defining a meta model and the transformation from an instance of it into textual ASMs. More details and a complete formalization are given in [5].

2 Related Work

One of the first definitions of CSDs is given in [3]. As stated above, this definition is given in a schematic and less formal way. Based on this definition an eclipse

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 374–379, 2018.
https://doi.org/10.1007/978-3-319-91271-4_26

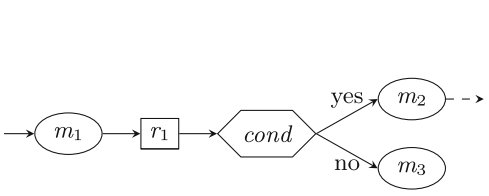
based graphical editor has been developed [4]. This tool is able to generate textual ASMs, but a formal definition of the translation is missing.

A more recent work considers the other way round: In [1] the authors try to create a graphical representation out of textual ASMs. Although the idea is completely different, the used notation and some of the translations are similar to our work. The general usefulness of ASMs to describe the semantics of diagrams has been proven in many cases, e.g. for UML state diagrams in [2].

3 Structure

A CSD is a directed graph where the set of *Node*s is a disjoint union of three sets: *Mode* (contains all control states, including one *initialMode*), *Condition* (conditional rules), and *Rule* (all remaining (basic) ASM rules). The set of directed *Edge*s consists of ordered pairs (*source*, *target*) of *Node*s and each edge has one of the three *edgeType*s {*yes*, *no*, *empty*}.

The concrete representation of the nodes is based on [3]: *Mode*s are circles or ellipses, *Rule*s are rectangles, and *Condition*s are stretched hexagons.



CSDasText =
if $mode = m_1$ **then**
r_1
seq
if *cond*
then $mode := m_2$
else $mode := m_3$
if $mode = m_2$ **then**
...

Fig. 1. Excerpt of an example CSD and its translation to ASM.

Figure 1 shows a small excerpt of an example CSD and its corresponding representation as ASM code (textual ASM). It starts in mode m_1 and executes the rule r_1. The successive conditional node is executed sequentially. If the boolean expression *cond* is true, the system's new mode is m_2 else m_3. As for m_1, for each mode with at least one outgoing edge, such a guarded command (e.g. **if** $mode = m_2$ **then** ...) is added to the (main) ASM (here CSDasText) and thus executed in parallel to all other guarded commands.

A *ValidCSD* must fulfill the following constraints (here given as text):

1. All *Node*s (except the *initialMode*) must have at least one incoming edge.
2. All *Condition*-nodes must have at least one outgoing edge with an arbitrary *edgeType* (*empty* is treated as *yes*). Outgoing edges of all other nodes must have the type *empty*.
3. Loops are only allowed between *Mode*s. Loops among *Rule*s or *Condition*s between two *Mode*s are prohibited.

4. To avoid obvious inconsistent updates (see translation below), edges from a *Mode* or a *Rule* node to more than one *Mode* node are not permitted. For *Conditions*, this constraint applies to each subset of edges with the same *edgeType*. Note that this constraint only avoids simple inconsistent updates, since the consistency problem in general is undecidable.

4 Translation Approach

The general translation approach is to create a guarded command for the initial mode node (in Fig. 1 m_1) and translate all paths until an other or the same mode node is hit. All combinations of nodes can be translated by a fixed scheme which is defined using ASMs again. To avoid arbitrary many combinations, we only consider combinations of a node with its successor nodes and translate them into textual ASM snippets. See Fig. 2 for some examples of these combinations. The translated textual snippets are inserted at specific *positions* in the main ASM.

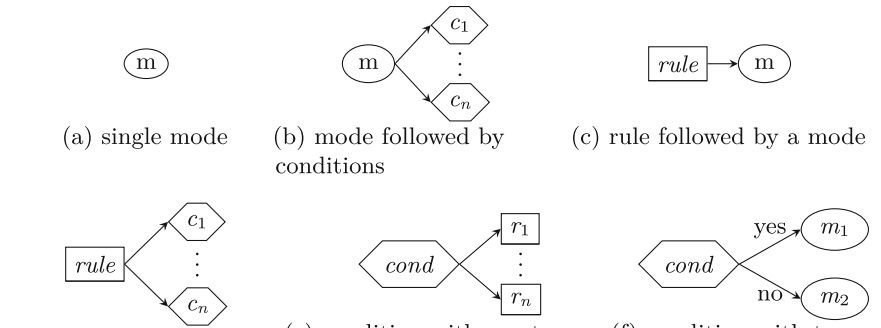


Fig. 2. Examples of possible combination patterns of nodes in a CSD

For each mode node with at least one outgoing edge, one guarded command is created that is executed in parallel to all other guarded commands. To deal with this special case, a location *nextModePos* always points to the next top level position in parallel to all other guarded commands (e.g. position β in the example at the end of this section).

Other combinations of nodes can occur at different positions, depending on the incoming edges. Hence, a function *positions* provides for each rule and condition node a set of *Position*s in the textual ASM (e.g. position α in the example at the end of this section). The *Position*s remain abstract because they can be implemented in different ways, e.g. by placeholders or offsets in a text file or nodes in a syntax tree of the ASM.

The *status* of each node controls if (a) the node can not be translated *undefined*), because there is no position defined yet, (b) can be translated (*active*),

because at least one position is set for this node or (c) if its translation is finished for all positions (*closed*). As initial state, the *status* of all nodes is *undefined* and the set of *position*s of each node is initialized by an empty set. Only the *initalMode* is *active* and *nextModePos* is set to the beginning of the main ASM.

During the translation process, one can think of the textual ASM as a fragmentary ASM that contains some placeholders at different positions. A *node* only becomes *active* if at least one position where its translation can be inserted into is defined for this node, which again can introduce new positions for newly activated *nodes*.

The compile process is specified by the ASM TranslateCSD. As long as there are nodes whose *status* is *active*, one of them is chosen and applied with one of its *positions* to a list of patterns describing a subgraph consisting of the selected node and its successor nodes.

```
TranslateCsd =
  choose node ∈ Node with status(node) = active
    if node ∈ Mode then
      TranslatePattern(node, nextModePos)
      status(node) := closed            -- a mode node is immediately closed
    else
      choose pos ∈ positions(node)      -- translate pos by pos
        TranslatePattern(node, pos)
        positions(node) := positions(node) \ {pos}
        if positions(node) \ {pos} = ∅ then
          status(node) := closed        -- if no positions left, node is closed
```

The general scheme of the TranslatePattern rule is as follows in which for each matching *pattern*, the corresponding *actions* are executed.

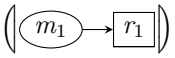
```
TranslatePattern(node, pos) =
  (| pattern_1(node) |) ⇒ actions_1(node, pos)
  ⋮
  (| pattern_n(node) |) ⇒ actions_n(node, pos)
```

Each *pattern* is graphically defined by e.g. $\left(\!\left| \textcircled{m_1} \rightarrow \boxed{r_1} \right|\!\right) \Rightarrow actions(node, pos)$ which is a short form for an appropriate conditional rule on the sets of *Node*s and *Edge*s that describes the drawn combination. The equivalent ASM notation of this example is:

```
if node = (m1) and (m1) ∈ Mode and [r1] ∈ Rule
  and ((m1), [r1]) ∈ Edge then
    actions(node, pos)
```

Each action contains at least a call to the rule AddTextualASM(*pos*, *textsnippet*) that inserts the *textsnippet* at the position *pos* into the to be created main ASM. The *textsnippet* can also assign concrete positions to newly

introduced position labels. In the translation rules, we mark the *textsnippets* by a frame and name the positions with Greek letters. Besides the generation and insertion of the textual ASM, the *status* and new *positions* of affected nodes (resp. *nextModePos*) are set. This generic behavior is illustrated in the following examples when a mode node is followed by a rule:



$\Rightarrow$ **let** $\alpha, \beta =$ **new** (*Position*) **in**
AddTextualASM(*pos*,
if *mode* $= m_1$ **then**
α
β
) -- add framed ASM snippet to main ASM at position *pos* and define new positions α, β
status($\boxed{r_1}$) := *active*
positions($\boxed{r_1}$) := *positions*($\boxed{r_1}$) $\cup \{\alpha\}$
nextModePos := β

In the translation, we use e.g. m_1 as a variable of the node's actual label. At position α, the translation of the combination with $\boxed{r_1}$ as source node will be inserted during one of the next steps. At position β the text snippet of the translation of the next mode node will be inserted.

5 Conclusion and Outlook

We have developed a formal definition of the semantics of CSDs, which is defined by a transformation into textual ASMs. This, in combination with the breakdown of the CSD into simple subgraphs consisting of nodes and their successors, facilitates the use of this semantics also for human readers of CSDs. In [5] the complete specification is presented together with a proof of completeness and consistency. The implementation of a prototype tool for the creation and translation of CSDs (similar to [4]) is also part of this thesis.

The next steps of our work are to evaluate the legibility of the specification by user studies and a detailed research of the aspects resulting from the refinement of arbitrary parts of a CSD by an other CSD.

Acknowledgments. We want to express our thanks to Egon Börger for many fruitful discussions about this topic which improved the work significantly.

References

1. Arcaini, P., Bonfanti, S., Gargantini, A., Riccobene, E.: Visual notation and patterns for abstract state machines. In: Milazzo, P., Varró, D., Wimmer, M. (eds.) STAF 2016. LNCS, vol. 9946, pp. 163–178. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50230-4_12
2. Börger, E., Cavarra, A., Riccobene, E.: Modeling the dynamics of UML state machines. In: Gurevich, Y., Kutter, P.W., Odersky, M., Thiele, L. (eds.) ASM 2000. LNCS, vol. 1912, pp. 223–241. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44518-8_13

3. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-642-18216-7
4. Jackson, P.: CSDe: control state diagram editor. https://sourceforge.net/p/coreasm/code/HEAD/tree/eclipse-tools/CSDe (2008). Accessed 05 Feb 2018
5. Leitz, M.: Definition of the formal semantics of control state diagrams and implementation of a graphical editor. Master's thesis. Ulm University (2018)
6. Mukala, P., Cerone, A., Turini, F.: An abstract state machine (ASM) representation of learning process in FLOSS communities. In: Canal, C., Idani, A. (eds.) SEFM 2014. LNCS, vol. 8938, pp. 227–242. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-15201-1_15

Capturing Membrane Computing by ASMs

Klaus-Dieter Schewe[1(✉)], Loredana Tec[2], and Qing Wang[3]

[1] Laboratory for Client-Centric Cloud Computing, Linz, Austria
kdschewe@acm.org
[2] FAW GmbH, Hagenberg, Austria
loredana.tec@gmail.com
[3] The Australian National University, Canberra, Australia
qing.wang@anu.edu.au

Abstract. Natural computing is a field of research that tries to imitate the ways of "computing" in nature. Membrane computing is a branch of natural computing that exploits hierarchically nested membrane structures that are associated with multisets of objects. The key notion is the P-system, which describes the transitions by rules for the creation, elimination and wandering of objects through membranes as well as manipulation of the membrane structure as such. In this short paper we sketch how P-systems can be captured by parallel ASMs. We further give a glimpse of further generalisations in several directions.

1 Membrane Structures and P-Systems

Natural computing is an umbrella for various computing paradigms taking processes in nature as models [10]. It captures neural networks, genetic programming, DNA computing and membrane computing. Membrane computing exploits hierarchically nested membrane structures as in cells, each associated with multisets of objects [9]. These associations are manipulated by so-called P-systems, which associate rules with each membrane. Rules are executed in parallel leading to the creation, elimination and wandering of objects through membranes as well as the manipulation of the membrane structure as such [7].

Membrane structures are equivalence classes of well-formed words over an alphabet of brackets: [] is well-formed, and whenever $\mu_1, \ldots, \mu_n$ are well-formed, then this also holds for $[\mu_1, \ldots, \mu_n]$. The equivalence relation $\sim$ on the set MS of these words is defined by $\mu_1\mu_2\mu_3\mu_4 \sim \mu_1\mu_3\mu_2\mu_4$ if $\mu_1\mu_4, \mu_2, \mu_3 \in MS$ and building the reflexive, transitive closure.

Each pair of matching brackets μ in a membrane structure ms is called a *membrane*, and the *degree* $\deg(ms)$ of a membrane structure is the number of its membranes. Membranes of the form [] are called *elementary*. We usually use an index set I to denote the membanes, so $M_{ms} = \{\mu_i \mid i \in I\}$ denotes the set of membranes of the membrane structure ms.

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 380–385, 2018.
https://doi.org/10.1007/978-3-319-91271-4_27

Let $\mathcal{O}$ denote a denumerable set, elements of which are called *objects*. A *super-cell* comprises a membrane structure ms and an association o of multisets $o(\mu)$ to each membrane $\mu \in M_{ms}$.

An *evaluation rule* is an expression $\ell \to r$ with $\ell \in \mathcal{O}^*$ and either $r \in \hat{\mathcal{O}}^*$ or $r = r'\delta$ with a special symbol $\delta \notin \mathcal{O}$ and $r' \in \hat{\mathcal{O}}^*$, where

$$\hat{\mathcal{O}} = \mathcal{O} \times (\{\text{here}, \text{out}\} \cup \{\text{in}_\mu \mid \mu \in M_{ms}\}). \tag{1}$$

If (ms, o) is a super-cell with objects in $\mathcal{O}$ and a distinguished *output membrane* $\mu_0 \in M_{ms}$ and ϱ assigns a partially-ordered set of evaluation rules to each membrane μ of ms, then $(ms, \mathcal{O}, o, \mu_0, \varrho)$ is a *P-system*.

The semantics of a single evaluation rule is defined as follows. If $\ell \to r \in \varrho(\mu)$, then ℓ is treated as a multiset with elements in $\mathcal{O}$, and if $\ell \subseteq o(\mu)$ holds (for inclusion of multisets), then the rule $\ell \to r$ can *fire*.

The application of a rule $\ell \to r$ that can fire results in replacing $o(\mu)$ by the multiset difference $o(\mu) - \ell$, and

- adding o to $o(\mu)$ for all (o, here) in r,
- adding o to $o(\mu')$ for all (o, out) in r, where μ' is the unique membrane that contains μ such that all other membranes containing μ also contain μ',
- adding o to $o(\mu')$ for all $(o, \text{in}_{\mu'})$ in r, where μ' is a membrane contained in μ such that any membrane contained in μ and containing μ' is equal to either μ or μ', and
- dissolving μ for δ in r, i.e. to discard all rules in $\varrho(\mu)$ and to add all $o \in o(\mu)$ to $o(\mu')$ for the unique minimal membrane containing μ.

A subset $R \subseteq \varrho(\mu)$ of rules can *fire simultaneously* iff $\biguplus_{\ell \to r \in R} \ell \subseteq o(\mu)$ holds. If R_1 and R_2 are two sets of rules that can fire simultaneously, then R_1 is *preferred* iff for all rules $\rho_1 \in R_1 - R_2$ and $\rho_2 \in R_2 - R_1$ we have $\rho_1 \geq_\mu \rho_2$ and at least once $\rho_1 >_\mu \rho_2$ holds, where $\geq_\mu$ is the partial order on $\varrho(\mu)$.

The semantics of the P-system is defined by selecting for each membrane μ a subset of rules $R_\mu \subseteq \varrho(\mu)$ that can fire simultaneously and is maximally preferred and to apply all selected rules in parallel.

Example 1.1. Let us take the membrane structure $ms = [[][[][]]]$ with $\deg(ms) = 5$. We take the index set $I = \{0, 00, 01, 010, 011\}$, so the set of membranes is $M_{ms} = \{\mu_i \mid i \in I\}$, and the corresponding matching brackets are indexed as follows: $ms = [_0[_{00}]_{00}[_{01}[_{010}]_{010}[_{011}]_{011}]_{01}]_0$. Here the membranes $\mu_{00}, \mu_{010}, \mu_{011}$ are elementary.

We obtain a super-cell (ms, o) with the object set $\mathcal{O} = \{a, b, c, d, e\}$ using the association $o(\mu_0) = \langle a, a, b, c\rangle$, $o(\mu_{00}) = \langle a, c, d, d\rangle$, $o(\mu_{01}) = \langle b, c, d, e, e\rangle$, $o(\mu_{010}) = \langle c, d, e\rangle$, and $o(\mu_{011}) = \langle b, b, b, e\rangle$.

This is further extended to a P-system $(ms, \mathcal{O}, o, \mu_0, \varrho)$ with the following sets of evaluation rules:

$$\varrho(\mu_0) = \{aa \to (c, \text{here})(b, \text{in}_{\mu_{00}}), ab \to (b, \text{here})(a, \text{in}_{\mu_{01}}), bc \to (a, \text{in}_{\mu_{01}})\}$$
$$\varrho(\mu_{00}) = \{bd \to (a, \text{here}), d \to (a, \text{out}), bbc \to (a, \text{out})\delta\}$$
$$\varrho(\mu_{01}) = \{bce \to (b, \text{here})(c, \text{here})(a, \text{in}_{\mu_{010}})(b, \text{in}_{\mu_{011}}), cde \to (a, \text{out})\}$$
$$\varrho(\mu_{010}) = \{ac \to (d, \text{here}), ae \to (b, \text{out})(e, \text{here})\}$$
$$\varrho(\mu_{011}) = \{bb \to (d, \text{here})(e, \text{here}), be \to (c, \text{out}), e \to (a, \text{out})\delta\}$$

Let the rules in these sets be ordered from left to right with decreasing priority. On the super-cell above we can fire simultaneously the first and the third rule in $\varrho(\mu_0)$, twice the second rule in $\varrho(\mu_{00})$, the first rule in $\varrho(\mu_{01})$, and the first two rules in $\varrho(\mu_{011})$, which results in a new super-cell with $o(\mu_0) = \langle a, a, c\rangle$, $o(\mu_{00}) = \langle a, b, c\rangle$, $o(\mu_{01}) = \langle a, b, c, d, e\rangle$, $o(\mu_{010}) = \langle a, c, c, d, e\rangle$, and $o(\mu_{011}) = \langle b, d, e\rangle$.

On this super-cell the first rules in $\varrho(\mu_0)$, $\varrho(\mu_{01})$ and $\varrho(\mu_{010})$ and the second rule in $\varrho(\mu_{011})$ fire simultaneously to yield the new super-cell with $o(\mu_0) = \langle c, c\rangle$, $o(\mu_{00}) = \langle a, b, b, c\rangle$, $o(\mu_{01}) = \langle a, b, c, d\rangle$, $o(\mu_{010}) = \langle a, c, c, d, d, e\rangle$, and $o(\mu_{011}) = \langle b, d\rangle$. Continuing this way the next super-cell will be $o(\mu_0) = \langle a, a, c, c\rangle$, $o(\mu_{01}) = \langle a, b, c, d\rangle$, $o(\mu_{010}) = \langle c, d, d, d, e\rangle$, and $o(\mu_{011}) = \langle b, d\rangle$, while the membrane μ_{00} has been dissolved. The next step will only change $o(\mu_0)$ to $\langle b, c, c, c\rangle$—here the creation of b in the no longer existing membrane μ_{00} is treated as a creation in μ_0, followed by a step that changes $o(\mu_0)$ to $\langle c, c\rangle$ and $o(\mu_{01})$ to $\langle a, a, b, c, d\rangle$, where no more rules can be applied.

2 An ASM Model for Membrane Computing

In [8] it was shown that P-systems can be simulated in Cardelli's ambient calculus, which itself can be captured by ASMs [3], though in view of the parallel ASM thesis [6] it does not come as a surprise that P-systems can be simulated step-by-step by ASMs. While we give a specification of P-systems by ASMs we lay the foundations for extending the computation paradigm as such.

In order to model a P-system by a parallel ASM [6] we use index trees. An *index tree* is a non-empty, finite set I of words over $\mathbb{N}$ such that the following conditions hold: (1) whenever $\langle i_1, \ldots, i_k\rangle \in I$ with $i_k = n + 1$ holds, then also $\langle i_1, \ldots, i_{k-1} i'_k\rangle \in I$ with $i'_k = n$, and (2) whenever $\langle i_1, \ldots, i_k\rangle \in I$, then also $\langle i_1, \ldots, i_{k-1}\rangle \in I$.

We can identify a membrane structure ms with an index tree I, and use I as set of indices for the membranes. For the signature of an ASM $\mathcal{M}_P$ capturing a P-system $P = (ms, \mathcal{O}, o, \mu_0, \varrho)$ we use a unary function symbol *cell*, which is defined on I and takes multisets as values, where the elements of the multisets can be arbitrary values. That is, we take a finite subset of the universe as representative of the object set $\mathcal{O}$ and model $o(\mu_i)$ by $cell(i)$ in every state. Let $\langle\rangle$ be the index of the output membrane μ_0.

In order to capture the rules in $\varrho(\mu)$ the signature further contains a binary function symbol *rule*, which is defined on a finite subset of $I \times \mathbb{N}$. Here $rule(i,j)$ is a pair (ℓ, r), where ℓ is a multiset containing arbitrary values in the universe U, and r is a multiset with elements in $\hat{U} \cup \{dissolve\}$, where $\hat{U}$ is defined analogously to $\hat{O}$ in Eq. (1) using constants here, out and in_i (for $i \in I$). We further use derived functions $stimulator(i,j) = \pi_1(rule(i,j))$ and $action(i,j) = \pi_2(rule(i,j))$. The multiplicity of *dissolve* in r must be at most 1. Note that if in_k appears in $action(i,j)$, we must have $k = i \cdot \langle n \rangle$ for some $n \in \mathbb{N}$.

Finally, let $\leq_i$ denote a partial order on $\mathbb{N}$ that is used to capture the partial order on the rule set $\varrho(\mu_i)$. For a set $R \subseteq \{j \in \mathbb{N} \mid stimulator(i,j) \neq undef\}$ we define $fire(i,R) \equiv \biguplus_{j \in R} stimulator(i,j) \subseteq cell(i)$ capturing that the rules in $\varrho(\mu_i)$ with index in R can simultaneously fire in the current state. A *preference* relation $\preceq_i$ is defined by

$$R_2 \preceq_i R_1 \equiv fire(i,R_1) \wedge fire(i,R_2) \wedge \forall j_1, j_2.(j_1 \in R_1 \wedge j_2 \in R_2 \wedge j_1 \notin R_2 \wedge j_2 \notin R_1 \Rightarrow j_2 \leq_i j_1) \quad (2)$$

Then the main rule in $\mathcal{M}_P$ is given by

FORALL $i \in I$ CHOOSE R_i WITH $max\text{-}fire(i,R_i)$ DO $apply_i(R_i)$ ENDDO

where $max\text{-}fire(i,R) \equiv fire(i,R) \wedge \forall R'.(fire(i,R') \Rightarrow R \not\preceq_i R')$ and the rules $apply_i$ (using partial updates [12]) are defined as follows:

$apply_i(R) \equiv$
FORALL $j \in R$
DO $cell(i) \leftleftarrows^{-} stimulator(i,j)$
SEQ FORALL $x \in action(i,j)$ WITH $x \neq dissolve$
DO IF $x = (v_{\text{here}}, \text{here})$ THEN $cell(i) \leftleftarrows^{\uplus} \{\!\{v_{\text{here}}\}\!\}$
IF $x = (v_{\text{out}}, \text{out})$
THEN CHOOSE $i' \in I$ WITH $\exists n \in \mathbb{N}.i = i' \cdot \langle n \rangle$
DO $cell(i') \leftleftarrows^{\uplus} \{\!\{v_{\text{out}}\}\!\}$ ENDDO
IF $x = (v_{\text{in}}, \text{in}_{i'}) \wedge \exists n \in \mathbb{N}.i' = i \cdot \langle n \rangle$
THEN $cell(i') \leftleftarrows^{\uplus} \{\!\{v_{\text{in}}\}\!\}$
ENDDO ;
IF $dissolve \in R$
THEN CHOOSE $i' \in I$ WITH $\exists n \in \mathbb{N}.i = i' \cdot \langle n \rangle$
DO $cell(i') \leftleftarrows^{\uplus} cell(i)$
$omit\_and\_reorder(i)$
ENDDO
ENDSEQ

We omit here the tedious definition of the rule $omit\_and\_reorder(i)$, which removes i from the index-tree and re-adjust all indices, cell values and rule sets accordingly.

3 Extending the Scope

The main rule of an ASM $\mathcal{M}_P$ is defined in a way that selected rule sets for the different membranes are executed in parallel and synchronously. However,

it appears more natural to assume that the different membranes act more independently and the rule sets are applied in an asynchronous way. This could be captured using separate agents for all selected rule sets and to use a concurrent ASM [2]. That is, if a rule $\ell \to r$ is selected to be fired, then instead of applying directly the partial updates as in the rule $apply_i(R)$ the rule activate an agent to execute such an update rule in an asynchronous way. This further permits to refine such agents in a way that an execution may be interrupted or changed[1].

While P-systems manipulate the objects associated with a membrane and allow membranes to be dissolved or created, the rules remain unchanged. If rules are allowed to be changed as well, this can be captured by concurrent reflective ASMs [11], which capture distributed adaptive systems [5].

While objects in membrane computing are just elements of an alphabet, the capture by ASMs allows arbitrary Tarski structures to be taken into consideration [4], which would add further power to the computations.

As most processes in nature are continuous rather than discrete, it makes sense to consider also continuous functions as in hybrid ASMs [1]. This would extend membrane computing from a discrete computation model that is somehow inspired by nature to a model that can truly describe natural processes on cell level.

In this way ASMs can be used for a natural capture of membrane computing, thus making nature-inspired computing paradigms an application field for rigorous methods. While this is so far only a sketch of a new research direction for rigorous methods, details will be explored in further research.

References

1. Banach, R., Zhu, H., Su, W., Wu, X.: A continuous ASM modelling approach to pacemaker sensing. ACM Trans. Softw. Eng. Methodol. **24**(1), 2:1–2:40 (2014)
2. Börger, E., Schewe, K.D.: Concurrent abstract state machines. Acta Informatica **53**(5), 469–492 (2016)
3. Börger, E., Cisternino, A., Gervasi, V.: Ambient abstract state machines with applications. J. Comput. Syst. Sci. **78**(3), 939–959 (2012)
4. Börger, E., Stärk, R.: Abstract State Machines. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-642-18216-7
5. Ferrarotti, F., Schewe, K.-D., Tec, L.: A behavioural theory for reflective sequential algorithms. In: Petrenko, A.K., Voronkov, A. (eds.) PSI 2017. LNCS, vol. 10742, pp. 117–131. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-74313-4_10
6. Ferrarotti, F., Schewe, K.D., Tec, L., Wang, Q.: A new thesis concerning synchronised parallel computing - simplified parallel ASM thesis. Theor. Comput. Sci. **649**, 25–53 (2016)
7. Martín-Vide, C., Păun, G., Rodríguez-Patón, A.: On P systems with membrane creation. Comput. Sci. J. Moldova **9**(2), 134–145 (2001)
8. Petre, I., Petre, L.: Mobile ambients and P-systems. J. UCS **5**(9), 588–598 (1999)

[1] In particular, if we think about the rules expressing chemical reactions on molecules as objects, such asynchronous behaviour much better the time-consuming nature of such processes.

9. Păun, G.: Membrane computing. In: Meyers, R.A. (ed.) Encyclopedia of Complexity and Systems Science, pp. 5523–5535. Springer, Heidelberg (2009). https://doi.org/10.1007/978-0-387-30440-3_328
10. Păun, G., Rozenberg, G., Salomaa, A.: DNA Computing - New Computing Paradigms. An EATCS Series. Springer, Heidelberg (1998). https://doi.org/10.1007/978-3-662-03563-4
11. Schewe, K.D.: Concurrent reflective abstract state machines. In: Jebelean, T., et al. (eds.) 19th Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2017). IEEE (2018, to appear)
12. Schewe, K.D., Wang, Q.: Partial updates in complex-value databases. In: Information Modelling and Knowledge Bases XXII (EJC 2010). Frontiers in Artificial Intelligence and Applications, vol. 225, pp. 37–56. IOS Press (2010)

Towards Creating a DSL Facilitating Modelling of Dynamic Access Control in Event-B

Inna Vistbakka[1(✉)], Mikhail Barash[1], and Elena Troubitsyna[1,2]

[1] Åbo Akademi University, Turku, Finland
{inna.vistbakka,mikhail.barash,elena.troubitsyna}@abo.fi
[2] KTH, Stockholm, Sweden

Abstract. Role-Based Access Control (RBAC) is a popular authorization model used to manage resource-access constraints in a wide range of systems. The standard RBAC framework adopts a static, state-independent approach to define the access rights to the system resources. It is often insufficient for correct implementation of the desired functionality and should be augmented with the dynamic, i.e., a state-dependant view on the access control. In this paper, we present a work in progress on creating a domain-specific language and the tool support for modelling and verification of dynamic RBAC. They support a tabular representation of the static RBAC constraints together with the graphical model of the scenarios and enable an automated translation of them into an Event-B model.

Keywords: Access control · DSL · JetBrains MPS · Event-B Verification

1 Introduction

Role-Based Access Control (RBAC) [2] is one of the main mechanisms for ensuring data integrity in a wide range of computer-based systems. The authorisation model defined by RBAC regulates users' access to computer resources based on their role in an organisation.

RBAC is built around the notions of users, roles, rights and protected system resources. A resource is an entity that contains some information. A user can access a resource based on an assigned role, where a role is usually seen as a job function performed by a user within an organisation. In their turn, rights define the specific actions that can be applied to the resources.

Usually RBAC gives a static view on the access rights associated with each role, i.e., it defines the permissions to manipulate certain resources "in general", i.e., without referring to the system state. RBAC can be defined as a table that relates roles with the permissions over the resources. However, such a static view is often insufficient for a correct implementation of the intended functionality.

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 386–391, 2018.
https://doi.org/10.1007/978-3-319-91271-4_28

An explicit definition of the dynamic state-dependant view could significantly facilitate system development.

A dynamic view of the access policy reflects the workflow to be supported by the system. Typically, it is described by the *scenarios*. A scenario defines a sequence of operations (often called *use cases*) that should be performed over the resources to implement the desired functionality. The dynamic and static views of RBAC are intrinsically interwoven. The permissions defined by the static view constitute the constraints on the operations execution. Therefore, we need to verify that the scenarios are feasible, i.e., not deadlocked by the (static) RBAC constraints. Similarly, we also need to check that if an operation is valid from the static point of view, it can be executed according to the workflow logic.

Domain experts while creating the informal descriptions of different RBAC views, are reluctant to formalise them by themselves. Yet they appreciate the feedback that can be provided by formal modelling and verification. To address this issue, in this paper, we propose a domain-specific language (DSL) and the corresponding tool support – *PapeRBACk* – that integrates tabular and graphical description of RBAC with formal modelling in Event-B [1]. We present the envisaged development process to be supported by the *PapeRBACk* approach and explain the main ideas behind it. We argue that by creating a DSL framework, we combine an expressiveness of informal domain-specific descriptions with rigour and verification feedback of Event-B.

Our ultimate goal here is to bridge the gap between highly-expressive RBAC models and specification languages. To achieve it, we create a DSL for dynamic RBAC. Using this language, a domain expert can describe main system elements, their relationships and the desired system workflow (in terms of scenarios on operations). After the intended workflow is described an initial Event-B specification can be generated and verified using a model checker ProB [6,9] to detect conflicting operations and inconsistent operation definitions.

2 Approach for Modelling Dynamic RBAC

A domain-specific language is a programming or modelling language specifically designed for working within a particular area of interest. There are different types of DSLs including visual diagramming language, e.g., the ones created by the Generic Eclipse Modeling System; programmatic abstractions, e.g., the Eclipse Modeling Framework, or textual languages [11]. Using a DSL improves development productivity and allows the domain experts to get involved in the development process. It is expected that the domain experts would be able to write and read models or code written in DSL.

Implementing a DSL traditionally requires defining a parser for it, often using a parser generator. A powerful Integrated Development Environment (IDE) for a DSL is vital for an adoption and success of the language. Recently, tools designed to define DSLs together with their IDEs have appeared under the name of *language workbenches*. In this work, we use the language workbench *JetBrains MPS* to implement a DSL for modeling RBAC.

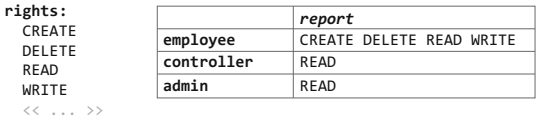


Fig. 1. RBAC table

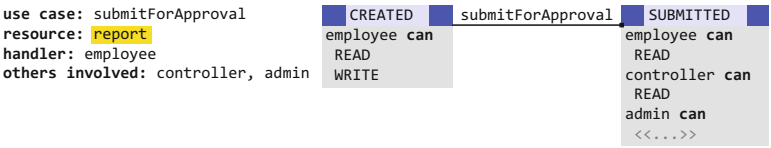


Fig. 2. `submitForApproval` use case

JetBrains MPS supports *projectional editing*, i.e., the key representation of a program is its *abstract syntax tree* (AST) that can be projected into different representations: textual, tabular, and graphical. Projectional editing does not require parsing, as the user edits the AST directly.

We will now briefly introduce our DSL *PapeRBACk* for representing both static and dynamic views on RBAC. The requirements description in *PapeRBACk* is a combination of textual, tabular and graphical elements. It starts with a specification that statically defines the rights of the roles to access system resources. Such a specification is represented by a table, with columns corresponding to the resources, and the rows to the roles.

An example of such a table is given in Fig. 1. Here we consider a periodic reporting system often used in organisations. Once per certain period, an `employee` should create and fill in a report and submit it for an approval to his/her boss (`controller`). The `controller` can approve a report and submit it to the `admin` for archiving or return to an `employee` for edits. The table in Fig. 1 shows the rights that each role has over the resource `report`.

The tabular format allows an engineer to immediately see whether all rights have been specified by visually controlling that all corresponding cells of the table are filled in. Rows and columns can be added, edited, and removed in the manner similar to the standard text processors.

To define the dynamic view of RBAC, an engineer can create a representation of the workflow. In *PapeRBACk* it consists of a textual and graphical representation of the use cases included in the workflow. Let us consider a simple use case of submitting a report for approval. Its representation in *PapeRBACk* is shown in Fig. 2. Since the main goal of the dynamic access control is to guarantee that the system resources are not accessed or manipulated by the unauthorised users, in our description we explicitly define the resource and the roles that access it in the given use case. We distinguish between the `handler` – the role that should initiate the execution of the use case, and `others` – the roles whose dynamically defined access rights will be changed as a result of executing the use case.

The dynamic view should also take into account the current system state. In *PapeRBACk* such a state is defined by the state of the resource. In our example, the use case is represented graphically as a "state machine" with only two "states" and one "transition" labelled with the name of the use case. The "entry state" contains information about the state of the resource (in our example, it is `CREATED`), and the required rights that the handler has to have to perform the use case. This set of rights is a subset of handler's statically defined rights specified in the table in Fig. 1. The "exit state" specifies the state that the resource will have after the use case is completed (in the example, it is `SUBMITTED`). In our description of the states, we explicitly define how the rights of the `handler` and `others` change as a result of reaching the corresponding state. Here again, these rights are the subsets of the corresponding roles' rights defined in Fig. 1.

The specification of static and dynamic RBAC views continues until all the required use cases are defined in the similar manner. After the description of the dynamic access control is completed, *PapeRBACk* generates an Event-B specification where the defined use cases (operations) are represented as the model events. For example, an Event-B event modelling the operation `submitForAproval` is presented below:

submitForApproval $\widehat{=}$
any rep, h
where $report\_state(rep) = CREATED$ // the state of the report is *CREATED*
$US\_ASSIGN(h) = \texttt{Employee}$ // handler's role is Employee
$\{R, W\} \subseteq dPerm(\texttt{Employee} \mapsto rep)$ // required rights to perform the operation
then
$report\_state(rep) := SUBMITTED$ // the state of the report is updated
$dPerm := dPerm \Leftarrow (\{\texttt{Employee} \mapsto rep \mapsto \{R\}\} \cup \{\texttt{Controller} \mapsto rep \mapsto \{R\}\})$ // rights are updated
end

Upon generation an Event-B specification, for every scenario (represented as a sequence of operations in the Event-B context) we run a model checker ProB [6] to detect the unfeasible scenarios or inconsistently defined constraints.

3 Discussion

In this paper we present the ongoing work on defining a DSL *PapeRBACk*. The language provides an integrated flexible support for the domain experts to describe static and dynamic aspects of role-based access control. To provide the domain engineers with the immediate verification feedback, we have also experimented with generation of the corresponding Event-B model. The Pro-B model checker is used to verify feasibility of the defined scenarios and possible mistakes or contradictions in the RBAC descriptions.

In this work, our primary goal was to explore feasibility of the proposed approach. There are several lessons that we have learnt. Firstly, working with a DSL is much easier than working with any "general purpose" informal modelling frameworks. This is due to the fact that we can restrict (via the GUI) the user's input and avoid an interpretation of the inputs incompatible with the defined AST. Secondly, while verification using model checking fitted well the

development approach, an integration of the proof-based verification is less obvious. Finally, to exploit the full power of Event-B, we need to discover a way to support the refinement-based development. We are experimenting with defining a domain-compliant refinement strategy, by employing a pattern-based approach proposed in [3,4,7] to maintain a connection between *PapeRBACk* and Event-B.

Our future plans focus on creating a powerful automated tool support for integrated DSL and formal modelling. We are working in two directions: enriching the *PapeRBACk* language with such concepts as role hierarchy and cardinality policies and separation of duty constraints as well as defining the facilities to support invariant definition and refinement.

Related Work. The policy analysis and verification issues related to RBAC model got a wide attention in last decades. The paper [8] presents a methodology to specify access control policies starting with a set of graphical diagrams: UML for the functional model, SecureUML for static access control and ASDT for dynamic access control. Then these diagrams are translated into a set of B machines. Moreover, the authors present the formal specification of an access control filter that actually regulates access control to the data. However, in this work the dynamics is mainly considered with the respect to an execution order of operations, while in our work a dynamic view on the access policies depends on the system state, in particular, on the state of a resource.

A DSL for RBAC based on UML diagrams and OCL constrains is discussed in [5]. However, it might be difficult for a domain expert who is not familiar with OCL to define such constrains.

Event-B is employed to specify the dynamic semantics of an industrial DSL as presented in [10]. Dynamic semantics map each DSL model to the corresponding execution behaviour. In contrast, in our work we use Event-B to verify the correctness of the informal descriptions and clarify contradicting requirements.

References

1. Abrial, J.R.: Modeling in Event-B. Cambridge University Press, Cambridge (2010)
2. Ferraiolo, D.F., Sandhu, R.S., Gavrila, S.I., Kuhn, D.R., Chandramouli, R.: Proposed NIST standard for role-based access control. ACM Trans. Inf. Syst. Secur. **4**(3), 224–274 (2001)
3. Iliasov, A., Troubitsyna, E., Laibinis, L., Romanovsky, A., Varpaaniemi, K., Ilic, D., Latvala, T.: Developing mode-rich satellite software by refinement in event-B. Sci. Comput. Program. **78**(7), 884–905 (2013)
4. Iliasov, A., Troubitsyna, E., Laibinis, L., Romanovsky, A.: Patterns for refinement automation. In: de Boer, F.S., Bonsangue, M.M., Hallerstede, S., Leuschel, M. (eds.) FMCO 2009. LNCS, vol. 6286, pp. 70–88. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17071-3_4
5. Kuhlmann, M., Sohr, K., Gogolla, M.: Employing UML and OCL for designing and analysing role-based access control. Math. Struct. Comput. Sci. **23**(4), 796–833 (2013)
6. Leuschel, M., Butler, M.: ProB: an automated analysis toolset for the B method. Int. J. Softw. Tools Technol. Transf. **10**(2), 185–203 (2008)

7. Lopatkin, I., Iliasov, A., Romanovsky, A., Prokhorova, Y., Troubitsyna, E.: Patterns for representing FMEA in formal specification of control systems. In: HASE 2011, pp. 146–151. IEEE Computer Society (2011)
8. Milhau, J., Idani, A., Laleau, R., Labiadh, M., Ledru, Y., Frappier, M.: Combining UML, ASTD and B for the formal specification of an access control filter. ISSE **7**(4), 303–313 (2011)
9. Rodin: Event-B platform. http://www.event-b.org/
10. Tikhonova, U., Manders, M., Boudewijns, R.: Visualization of formal specifications for understanding and debugging an industrial DSL. In: Milazzo, P., Varró, D., Wimmer, M. (eds.) STAF 2016. LNCS, vol. 9946, pp. 179–195. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50230-4_13
11. Voelter, M.: DSL Engineering (2013). dslbook.org

State-Based Formal Methods in Scientific Computation

John Baugh(✉) and Tristan Dyer

Civil, Construction, and Environmental Engineering,
North Carolina State University, Raleigh, NC, USA
{jwb,atdyer}@ncsu.edu

Abstract. Control systems, protocols, and hardware design are among the most common applications of state-based formal methods, and yet the types of modeling and analysis they enable are also well-suited to problems in scientific computation, where quality, reproducibility, and productivity are growing concerns. We survey the challenges faced by developers of scientific software, characterize the nature of the programs they write, and offer some perspective on the role that state-based methods can play in scientific domains.

1 Introduction

Called a third pillar of science, computation is an indispensable tool not only for scientists, but for engineers who simulate physical and natural processes to evaluate design alternatives. Recent studies on reliability, reproducibility of results, and productivity have cast concern over what many have suspected or experienced firsthand, that existing practices of constructing scientific software are inadequate and limiting the pace of technological advancement. A disconnect between modern software engineering practice and scientific computation is apparent, and yet the unique challenges facing developers of scientific software must also be recognized: the lack of test oracles, software lifetimes and evolving needs that span decades, and the competing objectives of performance, maintainability, and portability.

We seek to address fundamental design and quality assurance challenges that are intrinsic to scientific computation and related types of numerical software. While numerous directions might be taken, our premise and motivating viewpoint is the central role that modeling can and must play in the process of designing and working with complex artifacts, including scientific programs. Culturally, the fit may be a natural one: scientists and engineers are accustomed to working with models anyway, and with the kind of automatic, push-button analysis supported by some state-based formalisms, those who develop software can focus on modeling and design instead of theorem proving.

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 392–396, 2018.
https://doi.org/10.1007/978-3-319-91271-4_29

2 Background

Despite broad and recognized impacts, the field of scientific computation faces a number of challenges. Meeting quality and reproducibility standards is a growing concern [10], as is productivity [6]. Not merely anecdotes, numerous empirical studies of software "thwarting attempts at repetition or reproduction of scientific results" have been cataloged in a recent article by Storer [9], along with their concomitant effects, including a widespread inability to reproduce results and subsequent retractions of papers in scientific journals. Productivity problems are also reported, which Faulk et al. [6] refer to as a *productivity crisis* because of "frustratingly long and troubled software development times" and difficulty achieving portability requirements and other goals.

Sources of difficulty may stem from fundamental characteristics of the problem domain, along with cultural and development practices within it. For instance, projects are often undertaken, as one might imagine, for the purpose of advancing scientific goals, so results may constitute novel findings that are difficult to validate. In the absence of test oracles, developers may have to settle for plausibility checks based on, say, conservation laws or other principles that are expected to hold. Then, if the software is successful, its lifetime may span a 20 or 30 year period, starting with development and then moving through hardware upgrades and evolving requirements that are intended to keep up with ongoing scientific advancements. Development priorities are such that traditional software engineering concerns, like time to market and producing highly maintainable code, may receive relatively less attention compared with performance and hardware utilization [6].

Proposals to address quality and productivity concerns are varied. Storer [9] places new and suggested approaches into broad categories of (a) software processes, including agile methods, (b) quality assurance practices, including testing, inspections, and continuous integration, and (c) design approaches, including component architectures and design patterns. In the category of quality assurance practices, he adds formal methods, noting a couple of experience reports, but also observing that such approaches have received considerably less attention in the scientific programming community, possibly due to "the additional challenge of verifying programs that manage floating point data."

3 Approach

Although the tools and techniques most identified with scientific computation are those of numerical analysis—where error prediction, stability, and convergence are central concerns—such an enterprise offers little guidance in the development process, where early decisions about decomposition and organization establish program structure. We suggest separating concerns, and lay out an approach informed by numerical analyses that allows scientists and engineers to represent and reason about the essential structure and behavior of the programs they create. The ideas are well-suited for lightweight tools like Alloy [7], a state-based formalism that combines declarative modeling and bounded model checking.

3.1 About Scientific Programs

We consider the application of state-based methods in a relatively uncharted domain, scientific computation, for which there is little community experience in working with formal methods. We might ask about the essential complexities, what they are, and whether formal methods might help. By way of contrast, when computer engineers model systems, they already have some experience in getting at these questions. So, for instance, when specifying a two-phase handshake protocol they know whether they can ignore what's going through the pipe: they generally have some sense of how and what to specify, and what to ignore. There is far less of this kind of experience with programs in scientific areas, so it is helpful to characterize what they are like.

When we refer to scientific computation, we think primarily of problems expressed as mathematical models, where approximate solutions are sought for differential or integral equations that have no closed form solution. As a result, they must be discretized to produce a finite system of equations that can then be solved by algebraic methods. Ocean circulation models, for instance, may be expressed as a system of partial differential equations of the hyperbolic type, and solved by finite element [11] or other numerical schemes. Because they represent aspects of the physical and natural world, the terms and parameters appearing in the equations capture rich state in the form of spatial, geometric, material, topological, and other attributes. The types of discretizations that may be employed in both time and space are varied, and each has its own performance, accuracy, and ease-of-development implications.

3.2 Separating Concerns

What we propose is something akin to the two-phase handshake protocol analogy where the data going through the pipe are, in this case, numerical expressions. We cannot ignore them, of course, but we aim to consider them separately, so we advance the following perspective:

$$\text{scientific programs} = \text{numerical expressions} + \text{interstitial machinery}$$

By *interstitial machinery* we mean the discrete data structures and algorithms throughout which numerical expressions are embedded. In many cases, the interstitial machinery is itself a complex apparatus, as we find in the class of problems above, and these are aspects of a program that warrant increased scrutiny and care. Correctness arguments for this part of scientific programs can be made without simultaneously reproducing the sometimes deep, semantic proofs of numerical analysis [8]. Instead, pertinent results may be brought into the modeling process in the form of invariants and other structural properties.

Beyond appealing to experience, a supporting idea for this claim is the following: the numerical analyses performed for scientific computations often apply, unchanged, throughout a broad range of implementation choices and modifications, changes in libraries and solvers, and diverse hardware upgrades, over the life of the program.

3.3 Examples

Applying this perspective, the following studies show how finite state models can be used to draw useful conclusions about scientific software:

Hurricane Storm Surge. Used in production by the U.S. Army Corps of Engineers and others, ADCIRC is a large-scale ocean circulation model that simulates hurricane storm surge. In this study [3], we consider implementation choices for a performance enhancement made by our group, and use models developed in Alloy to make guarantees about them, in particular that they are equivalence preserving. The study is motivated by complex interactions between the enhancement and ADCIRC's discrete wetting and drying algorithm, which operates on a finite element mesh to accommodate advancing and receding flood waters.

Coupled Earth Models. Numerical models of the earth capture interactions between atmospheric, ocean, land surface, sea ice, and other components, which execute concurrently and exchange data during runtime. By modeling read-write behavior and the timestamps associated with updates, race-free phasing arrangements can be generated, thereby preventing data from either being overwritten too soon or becoming stale. This approach is applied to a research prototype of simultaneously executing ocean circulation models for which the exchange of data must be coordinated [2].

Structural Analysis. Moment distribution [5] is an iterative technique, well-known among civil engineers, for finding the internal member forces that develop in building structures when external forces are applied to them. In its most general form, the method is similar to asynchronous, chaotic relaxation algorithms, where portions of a building structure converge numerically at differing rates as the computation unfolds, depending on process scheduling. The nondeterminism available here is also inherent in methods used to solve elliptic partial differential equations, which may exploit nondeterminism in different ways depending on problem characteristics and hardware features. In an unpublished specification that appears online [1], we make use of a numerical study [4] and predicate abstraction in a modeling approach that facilitates refinement checking.

The examples above span a range of scales from production to research software to what might be considered a toy problem, moment distribution, and yet the problems share features that suggest a role for state-based methods:

- Structure: by supporting *implicitness* in a specification, Alloy allows arbitrary spatial discretizations to be considered in the analysis, e.g., the varied topological relationships that exist in real building structures.
- Behavior: by not imposing fixed idioms, it can accommodate specifications of different styles and with different approaches to parallelism that may be encountered, e.g., in library interfaces like MPI, OpenMP, and OpenCL.

While other approaches might be considered, state-based methods like Alloy seem particularly appropriate for the types of modeling and analysis we describe, and for the support it provides for conceptual design.

4 Conclusions

Numerical concerns figure prominently in scientific computation, and yet the major sources of complexity in actual software, from our perspective, have more to do with the interstitial machinery that ties them together. Separating concerns, along the lines we have suggested, should allow state-based methods to find productive use in a domain that could benefit from the kind of modeling and push-button analysis they provide. Invariants and other structural properties often follow directly from numerical analyses, both for algorithms and for data structures, facilitating safety, liveness, and fairness checks that can be put together in a variety of ways beyond the ones we mention.

Given the fundamental role of computation in the conduct of modern science, the development and adoption of better design practices could have far-reaching benefits. Toward that end, we suggest a focus on essential complexities and scientifically relevant computational abstractions, as advocated by Faulk et al. [6], using precise and expressive notations that support exploration and analysis. Future work in this direction may lead to new insights and deeper understanding, as well as auxiliary tools and instructional materials that make these advances more accessible to scientists and engineers in traditional areas.

References

1. Alloy models from the paper. http://www4.ncsu.edu/~jwb/alloy/
2. Altuntas, A., Baugh, J.: Verifying concurrency in an adaptive ocean circulation model. In: Proceedings of the First International Workshop on Software Correctness for HPC Applications, Correctness 2017, pp. 1–7. ACM (2017)
3. Baugh, J., Altuntas, A.: Formal methods and finite element analysis of hurricane storm surge: a case study in software verification. Sci. Comput. Program. (2017, in press). https://doi.org/10.1016/j.scico.2017.08.012
4. Baugh, J., Liu, S.: A general characterization of the hardy cross method as sequential and multiprocess algorithms. Structures **6**, 170–181 (2016)
5. Cross, H.: Analysis of continuous frames by distributing fixed-end moments. In: Proceedings of the American Society of Civil Engineers, pp. 919–928 (1930)
6. Faulk, S., Loh, E., Van De Vanter, M.L., Squires, S., Votta, L.G.: Scientific computing's productivity gridlock: how software engineering can help. Comput. Sci. Eng. **11**(6), 30–39 (2009)
7. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press, Cambridge (2012)
8. Linz, P.: A critique of numerical analysis. Bull. Am. Math. Soc. **19**(2), 407–416 (1988)
9. Storer, T.: Bridging the chasm: a survey of software engineering practice in scientific programming. ACM Comput. Surv. (CSUR) **50**(4), 47:1–47:32 (2017)
10. Wilson, G.V.: Where's the real bottleneck in scientific computing? Am. Sci. **94**(1), 5–6 (2006)
11. Zienkiewicz, O.C., Taylor, R.L., Nithiarasu, P.: The Finite Element Method for Fluid Dynamics, 7th edn. Butterworth-Heinemann, Oxford (2013)

Proposition of an Action Layer for Electrum

Julien Brunel[1], David Chemouil[1(✉)], Alcino Cunha[2], Thomas Hujsa[1], Nuno Macedo[2], and Jeanne Tawa[1]

[1] ONERA/DTIS, Université Fédérale Toulouse Midi-Pyrénées, Toulouse, France
david.chemouil@onera.fr
[2] INESC TEC, Universidade do Minho, Braga, Portugal

Abstract. Electrum is an extension of Alloy that adds (1) mutable signatures and fields to the modeling layer; and (2) connectives from linear temporal logic (with past) and primed variables à la TLA$^+$ to the constraint language. The analysis of models can then be translated into a SAT-based bounded model-checking problem, or to an LTL-based unbounded model-checking problem. Electrum has proved to be useful to model and verify dynamic systems with rich configurations. However, when specifying events, the tedious and sometimes error-prone handling of traces and frame conditions (similarly as in Alloy) remained necessary. In this paper, we introduce an extension of Electrum with a so-called "action" layer that addresses these questions.

1 Introduction

The specification and verification of software and systems are crucial tasks at early development phases. Indeed, the later the detection of an error happens in the development cycle, the more costly it is. This calls for expressive formal specification languages, ideally supported by automatic verification tools. Then, an important issue is the trade-off between the expressiveness of the specification language and the automation degree of the verification. Alloy [4], one of the main propositions in *lightweight* formal methods, does not favor one concern over the other. Instead, it gives up on the completeness of the verification: it performs an exhaustive exploration of the system states up to a user-specified depth.

Alloy is based on an extension of first-order logic and offers a rich way to specify structural properties over a system. In [5], we proposed Electrum, an extension of Alloy with support for dynamic features based upon linear temporal logic (LTL). Electrum preserves the flexibility of Alloy while easing the specification

This work is financed by the ERDF - European Regional Development Fund - through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 - and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project POCI-01-0145-FEDER-016826, and the French Research Agency project FORMEDICIS ANR-16-CE25-0007.

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 397–402, 2018.
https://doi.org/10.1007/978-3-319-91271-4_30

of behavioral properties and enabling verification over an unbounded temporal horizon. With Electrum, the system behavior is specified using FOLTL formulas. Electrum thus preserves the fully declarative feature of Alloy: there is no "constructive" description of the system, but only the constraints that the system satisfies. However, in practice, it is often convenient to specify the basic *actions* of the system (which needs little expressiveness in terms of temporal logic) separately from other behavioral requirements, such as the way these actions are ordered (which may need the full expressive power of temporal logic). Relying on this kind of idioms can have several advantages: (1) some part of the behavior, such as the frame conditions or the time model, can be specified in a systematic way; (2) such a description of the evolution of the system is more likely to be exploited by a verification procedure that relies on a model checker.

Thus, still pursuing the goal of allowing the straightforward specification and verification of models featuring rich structure and behavior, we propose here an extension of Electrum with an action layer.

The remainder of the article is organized as follows. In Sect. 2 we present the Electrum framework. In Sect. 3, we define the syntax and semantics of the action layer and illustrate it on an example.

2 Electrum

Following Alloy, structure in an Electrum specification is introduced through the declaration of *signatures*, which represent sets of uninterpreted atoms, and *fields* of arbitrary finite arity, which relate atoms belonging to different signatures. Each of these signatures and fields can be declared as *static* (by default) or *variable* (keyword `var`): the former have the same valuation throughout a given time trace, while the latter are mutable and hence may evolve in time. Hierarchy between signatures (which can additionally be declared as `abstract`) can be introduced through *extension* (`extends` keyword) or *inclusion* (`in`). Finally, both signatures and fields may be restricted by simple *multiplicity* constraints. Notice that for variable elements, these restrictions are applied globally in time.

Additional restrictions can be imposed through *facts*, axioms that every instance of the specification is required to conform to. Those may rely on reusable *predicates* and *functions*. Relational expressions are built by composing signatures and fields (and some built-in constants) with common set-theoretic operators and relational operators like *join* `.` or transitive closure `^`.

Every relational expression can be *primed*, referring to its valuation in the succeeding state. Atomic formulas are then built as inclusion (or equality) tests of relational expressions, which can be composed through the common Boolean operators, first-order quantifications and future and past LTL operators.

Execution instructions consist of `run` and `check` commands restricted by *scopes* that determine the maximum (or `exactly` the) number of atoms of each signature that will be considered by the analyses: (1) `run` instructs the Analyzer to search for an instance satisfying a given constraint; (2) `check` instructs the Analyzer to prove a given assertion valid (in practice: by checking that it

```
open util/ordering[Key]
sig Key {}
sig Room {
  keys: set Key,
  var current: one keys }
fact DisjointKeySets {
  Room<:keys in Room lone→ Key }
one sig Desk {
  var lastKey: Room → lone Key,
  var occupant: Room → Guest }
sig Guest { var gkeys: set Key }
...
fun nextKey[k: Key, ks: set Key] : set Key {
  min[nexts[k] & ks] }

act checkin[g: Guest, r: Room, k: Key]
  modifies gkeys, occupant, lastKey {
  no r.(Desk.occupant)
  k = nextKey[r.(Desk.lastKey), r.keys]
  gkeys' = gkeys + g → k
  Desk.occupant' = Desk.occupant + r→g
  Desk.lastKey' = Desk.lastKey ++ r→k }
act checkout[g: Guest] modifies occupant {
  some Desk.occupant.g
  Desk.occupant' = Desk.occupant - Room→g }
...

fact init {
  no Guest.gkeys
  no Desk.occupant
  all r: Room | r.(Desk.lastKey) = r.current }

pred consistent {}
run consistent for 4 but 10 Time
assert BadSafety  {
  always { all r: Room, g: Guest, k: Key |
    entry[g, r, k] and
    some r.(Desk.occupant)
      ⇒ g in r.(Desk.occupant) } }
check BadSafety for 4 but 10 Time
```

Fig. 1. Hotel example in Electrum with actions (syntax additions are underlined).

cannot find a counter-example). A protected keyword `Time` restricts the size of the traces when analysis is performed by *bounded* model checking (BMC). Note that *unbounded* model checking (UMC) is still bounded on the atoms in the valuations of signatures. The complete semantics of Electrum can be consulted in [5].

3 Extending Electrum with Actions

In this section, we present the syntax and semantics of the action layer. The layer is actually syntactic sugar on top of plain Electrum, therefore the semantics is defined by translation into Electrum. For the sake of readability, we illustrate this translation over an example (Fig. 1) inspired by the classic Alloy Hotel example. The latter specifies a system handling entries in the rooms of a hotel with disposable key-cards carrying cryptographic keys that must match other keys stored in room-door locks to release these and open the rooms.

Specifying behavior in Electrum is completely unrestricted. However, *in practice* (and as the Hotel example shows), many models are specified using actions (represented as predicates or using the event idiom [4]) that only relate two consecutive instants. Said otherwise, a large class of Electrum models does not rely on the full power of LTL to specify the valid traces: this logic is mainly useful when specifying additional facts (*e.g.* fairness properties) or stating properties to evaluate on a model using a `run` or a `check` command. Besides, the frame conditions and the time model could be described in a systematic way. Their generation could be automated, depending only on a few parameters (*e.g.* allowing, or not, simultaneous actions).

In practice, we add to plain Electrum an action syntactic sugar that is optional but committing: if no action is present in a model, then its semantics is fully unrestricted, as usual, but as soon as an action is present, the semantics associated with actions applies. The sugar thus introduces a notion of action. Frame

conditions are automatically generated out of a specific parameter of actions. Traces are automatically generated, forcing a specific time model. Finally, the occurrence of an action can be referred to in the syntax of constraints.

Actions and Time Model. We add an `act` keyword that introduces a named action, possibly with parameters. Parameters may only be singletons (*i.e.*, no set may be passed as an argument: if this is needed, then a new signature pointing to the said set should be introduced first and then passed as an argument). An action executes *atomically* and relates two consecutive instants, therefore the only temporal constructs allowed are the `after` keyword and the prime operator. As in plain Electrum, formulas (from the action body) relating to the "current" instant represent a *guard* (necessary condition) for the action to occur, and the ones talking about the next instant stand for the post-condition.

The most important semantic constraint in our action layer is that the time model imposes an *interleaving semantics*: *exactly one action is executed at every instant*. Also, it does not feature stuttering steps by default: if this is needed, the user may define an *ad hoc* action: `act skip {}` (with an empty `modifies` clause).

Actions are translated into a structure of signatures and fields encoding the possible *events* (action occurrences). We introduce first an `_Action` enumeration for all action *names*. Then we add a relation encoding all possible events by taking the union of all possible valuations of actions (as actions may differ in arity, we pad them to the highest arity with a dummy signature). Simultaneously, we specify the time model by forcing exactly one event to occur at every instant. Finally, a fact states the effect of every action when it is `fired` (cf. p. 4):

```
enum _Action { checkin, checkout, ... } // action names
one sig _Dummy {}
one sig _E { // simply an enclosing signature for _event
  var _event:   (checkin → Guest → Room → Key)
              + (checkout → Guest → _Dummy → _Dummy)
              + ... // other possible events
  } { one _event } // time model
fact { always { // effect of every action when it is fired:
  all g : Guest, r : Room, k : Key {
    fired[checkin, g, r, k] implies ... /* checkin body */ }
  ... /* the same for other actions */ } }
```

Frame Conditions. An action can specify, using a `modifies` clause, which variable signatures and fields it *controls*. In practice, this allows the automatic generation of frame conditions under a simple rule saying that *any variable signature or field that is not controlled by an action is left unchanged by this action. E.g.*, the `checkout` action (see Fig. 1 l. 23–25) controls the `occupant` field only, inducing that `current`, `lastKey` and `gkeys` do not change when this action fires. On the other hand, notice every action is responsible for handling the frame conditions for the variable constructs declared in its `modifies` clause.

Referring to Actions in Constraints. Any occurrence of an action can be referred to in a constraint, with actual parameters (*e.g.* as `entry[g,r,k]` in

Fig. 1 l. 37) or without (in which case, there is an implicit existential quantification over all parameters). For instance, `after checkout` actually means: `after (some g: Guest | checkout[g])`. To allow this, we generate a `fired` predicate saying whether an action is indeed fired. As actions may take parameters of different types, the `fired` predicate profile accepts arguments in the union of all these types. Again, its arity is the highest arity for actions.

```
var sig _Arg = _Dummy + Guest + Room + Key {} // union of all types
pred fired [a : _Action, x1, x2, x3 : _Arg] { // if max arity = 3
    a→x1→x2→x3 in _E._event }
```

This way, `entry[g,r,k]` (Fig. 1 l. 37) translates to `fired[entry, g, r, k]`.

4 Related Work and Conclusion

TLA$^+$ inspired Electrum in general, and its action layer in particular. However, significant differences between TLA$^+$ and plain Electrum have already been pointed out in [5]. Moreover, our proposition slightly differs as Electrum is stuttering sensitive and the time model is forced. The enhancement of Alloy with behavior [1–3,6,7] has been widely studied. Among these propositions, DynAlloy [3] defines a syntax for actions similar to ours, but the semantics differs in the time model and in the firing of actions. Besides, all these frameworks propose in the end a translation into plain Alloy and thus, they only offer verification over a bounded temporal horizon. In our experience, using the Electrum action layer makes the behavior specification both easier (specifying the actions, and reasoning about their occurrence, is quite natural) and less error-prone because part of the behavior specification is automatically generated. We benchmarked (not shown due to lack of space) the action layer on examples coming from the Alloy literature: w.r.t. plain Electrum, the efficiency of analyses is often reduced for valid properties, but still acceptably. In the future, we intend to assess several new compilation strategies (and perhaps semantics) to improve the efficiency.

References

1. Chang, F.S., Jackson, D.: Symbolic model checking of declarative relational models. In: ICSE 2006, pp. 312–320. ACM (2006). https://doi.org/10.1145/1134329
2. Cunha, A.: Bounded model checking of temporal formulas with Alloy. In: Ait Ameur, Y., Schewe, K.D. (eds.) ABZ 2014. LNCS, vol. 8477, pp. 303–308. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43652-3_29
3. Frias, M.F., Galeotti, J.P., Pombo, C.L., Aguirre, N.: DynAlloy: upgrading Alloy with actions. In: ICSE 2005, pp. 442–451. ACM (2005). https://doi.org/10.1145/1062455.1062535
4. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2012). Revised edn.
5. Macedo, N., Brunel, J., Chemouil, D., Cunha, A., Kuperberg, D.: Lightweight specification and analysis of dynamic systems with rich configurations. In: SIGSOFT FSE, pp. 373–383. ACM (2016). https://doi.org/10.1145/2950290.2950318

6. Near, J.P., Jackson, D.: An imperative extension to Alloy. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 118–131. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11811-1_10
7. Vakili, A., Day, N.A.: Temporal logic model checking in Alloy. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) ABZ 2012. LNCS, vol. 7316, pp. 150–163. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30885-7_11

Insulin Pump: Modular Modeling of Hybrid Systems Using Event-B

Wen Su(✉), Jinxin Chen, and Shehroz Khan

School of Computer Engineering and Science, Shanghai University, Shanghai, China
wsu@shu.edu.cn

Abstract. This case study of an insulin pump is to describe our solution of the following difficulties. Firstly, how to model features to obtain a family of products. Secondly, how to handle complex constraints and synchronization of components when composing features. Thirdly, how to construct the continuous environment for the individual features as well as for the composed system.

1 Introduction

Insulin pumps are safety-critical medical devices which are usually built as product families. It means that the various products of insulin pumps are built with shared common functionalities, but also with varieties to satisfy the diversity of patient needs. Moreover, features of an insulin pump need to collaborate and synchronize with each other. An insulin pump and its environment form a typical hybrid system that the discrete software controls directly the continuous physical injection. Therefore, it is a very interesting case study for formal development.

When tried to construct a family of insulin pump following the requirements in [1] using Event-B [2], we met the following problems. Firstly, how to configure the components to obtain a family of products based on the diverse requirements of patient needs is a problem. Secondly, the components of an insulin pump have complex constraints and synchronize with each other. Therefore, how to guarantee these relations after composition is a problem. Thirdly, if we model some features as independent hybrid systems, when we compose them, how to obtain a unique hybrid system is a problem.

In this paper, we present the modular approach of modeling and composition to form a product family, to solve the first problem. Then, we explain the approach that transforms guarded event into pre-condition operation when we compose models, to solve the second problem. Finally we show how to decompose and compose hybrid systems, to solve the third problem.

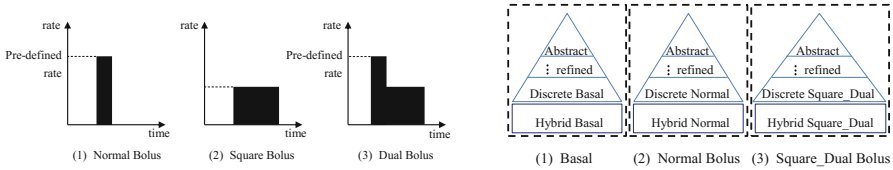
This research was supported by the NSFC (No. 61602293), and the STCSM (No. 15YF1403900).

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 403–408, 2018.
https://doi.org/10.1007/978-3-319-91271-4_31

2 Requirements and System Design

Due to the diversity of needs, insulin pumps are built as a family of products that have common and various functions. The requirements of a family of insulin pumps simplified from [1] are described as below. The family is made up of two products: both the products have a basal feature and a normal bolus feature, but only one product has an extra square/dual bolus feature.

1. Basal feature. (a) The insulin pump therapy provides a *basal* function that delivers continuously over 24 hours a day. The duration rates of the basal function are made up of start and stop times that cannot overlap.
2. Bolus features. (a) The insulin pump provides three types of *bolus* that are *normal bolus*, *square wave bolus*, and *dual wave bolus*. (b) Normal bolus is an infusion that pumps completely at the onset of the bolus. It calculates the delivery time according to the insulin quantity set by users, and delivers the bolus immediately at a pre-defined rate. (c) Square bolus is a slow infusion that spreads over time. Dual bolus combines a normal bolus with a square wave bolus. It provides a high dose up front, and extends the tail of infusion action. The wave diagrams of the three bolus are shown in Fig. 1(a).
3. Suspend and Resume. (a) Suspend function is always available. It stops all the deliveries that are in progress. (b) When the pump resumes from the suspended mode, the rate of basal bolus depends on the corresponding rate of the resume time. (c) A bolus that is suspended will not be resumed. The user must reprogram and activate the bolus to finish the delivery. (d) When the pump is suspended, only its resume function is available.
4. Constraints. (a) If a bolus feature (normal, square, or dual bolus) is working in parallel with a basal feature, the rate is a superimposition of both the basal feature and the bolus feature. (b) A normal bolus can deliver at any time except during another normal bolus is delivering. (c) During a normal bolus is delivering, both the square bolus and the dual bolus are disabled until the normal bolus finish its delivery. (d) A normal bolus can temporarily interrupt a square bolus or a dual bolus that is delivering. When the normal bolus is finished, the bolus delivery resumes to the rate that just before interruption. (e) The basal feature and normal bolus feature are mandatory, whereas the square/dual bolus feature is optional.



(a) Three bolus types (b) Bolus features modeling

Fig. 1. The modeling design for hybrid system

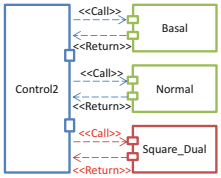
Due to complex constraints, we define two controllers for the communication and synchronization of features, which are Control1 and Control2 for Product1 and Product2, respectively. Thus, the family is described using Product Family Algebra (PFA) [3] as below:

$$Insulin\_Family = Basal \cdot Normal \cdot (Control1 + Square\_Dual \cdot Control2) \quad (1)$$

$$Product1 = Basal \cdot Normal \cdot Control1 \quad (2)$$

$$Product2 = Basal \cdot Normal \cdot Square\_Dual \cdot Control2 \quad (3)$$

where $+$ is a choice between components, and $\cdot$ is a composition or merging operation on components. The system development follows the following steps.

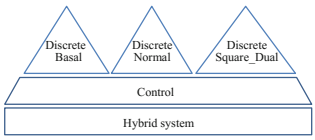


(a) Design of product2

Model	Requirements
Basal	1(a),3(a)p, 3(b)
Normal Bolus	2(a)p,2(b),3(a,c,d)p
Square_Dual	2(a)p,2(c),3(a,c,d)p
Control1	3(a,c)p, 4(a,b)
Control2	3(a,c)p, 4(a,b,c,d)
Product Family	4(e)

p means PARTIALLY taken.

(b) Modeling strategy



(c) Hybrid system

Fig. 2. The design of modules for constructing product family

Firstly, we construct the individual features of Basal, Normal, Square_Dual, Control1 and Control2. Each feature is modeled independently by the Event-B refinement as shown in Fig. 1(b) with its encapsulated data used through interface. The features trace back to the requirements as presented in Fig. 2(b).

Secondly, we form the two products Product1 and Product2 following (2) and (3), respectively. The structure of Product2 is shown in Fig. 2(a). The related composition is detailed in Sect. 3.

Thirdly, we construct hybrid systems. Since the features Basal, Normal, and Square_Dual control independently the injection rates *brate*, *nrate*, and *sdrate*, respectively, we refine each of them to a hybrid system as shown in Fig. 1(b). The composed product has a merged environment, where the *pump rate* of Product2 is a superimposition of the rates *brate*, *nrate*, and *sdrate*. Thus, we refine the composed discrete model to a hybrid system with a merged environment as shown in Fig. 2(c). More details of the hybrid systems are shown in Sect. 4.

3 Model Composition

After studying various composition approaches [4–8], we choose the function calls composition approach proposed in [9], which transforms guarded events into preconditioned operations, to keep the function call process atomic. Moreover, in this section we also propose a refinement pattern for function calls. The formal model of this work can be found on [10].

3.1 Function Call Structure

Here we explain how to compose the features Control2 and Basal in the case study. The feature Basal has an event rate_update with the guards:

$$t \in dom(rate\_setting2 \rhd \{-1\}) \quad \wedge \quad basal\_mode = delivering, \tag{4}$$

where t is a formal parameter, and the guards in (4) are the pre-conditions of calling rate_update. Define a technical enumerated set as a carrier set:

$$PROG = \{call\_basal\_update,\ return\_basal\_update,\ null\}$$

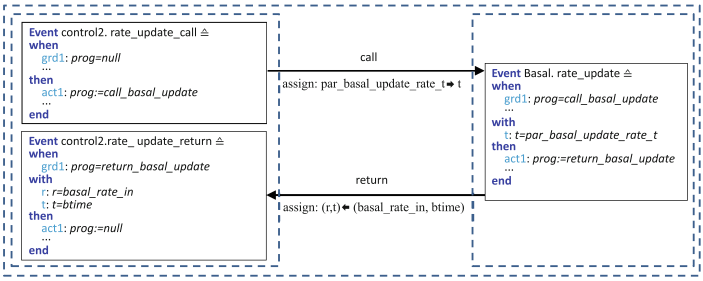


Fig. 3. function call structure

The part of the composed model that relates to the call from Control2 to the function rate_update of Basal is shown below in Fig. 3, where the variable $prog \in PROG$ is used for the atomic function call, and the formal parameter t is instantiated by the actual parameter $par\_basal\_update\_rate\_t$. The pre-conditions in (4) are removed from rate_update by adding the following invariant to guarantee that the pre-conditions are satisfied when the function is called:

$$prog = call\_basal\_update \Rightarrow par\_basal\_update\_rate\_t \in \mathrm{dom}(rate\_setting2 \rhd \{-1\}) \wedge basal\_mode = delivering$$

3.2 Refinement Pattern

A refinement pattern for introducing function calls gradually in refinements is proposed here. Given $PROG$ as a carrier set for the type of function calls, where $null \in PROG$. In the abstract, to call function $P1$ we define a constant $pg1$ that:

$$\textbf{axm1}: pg1 \subseteq PROG \ \wedge \ pg1 = \{call\_P1, return\_P1\} \ \wedge \ null \notin pg1$$

Then we define a variable $prog1$ that $prog1 \in pg1 \cup \{null\}$ (initialized to $null$) to call $P1$. If we need to call another function $P2$ that is different from $P1$ in a refinement, we introduce a constant $pg2$ that:

$$\textbf{axm2}: pg2 = \{call\_P2, return\_P2\} \ \wedge \ null \notin pg2 \ \wedge \ pg1 \cap pg2 = \varnothing$$

In the refinement we define a variable $prog2$ to call both functions $P1$ and $P2$, where $prog2 \in pg2 \cup pg1 \cup \{null\}$. The variable $prog1$ can be replaced by $prog2$ based on the following invariant.

$$\textbf{inv1}: prog2 \in pg1 \cup \{null\} \ \Rightarrow \ prog1 = prog2 \quad \textbf{inv2}: prog2 \in pg2 \ \Rightarrow \ prog1 = null$$

where the usage of $prog1$ and $prog2$ follows the approach proposed in [9]. This pattern is followed in the composition of the case study, where $prog1$, $prog2$ and $prog3$ are used in the models Control2_Basal, Control2_Basal_Normal and Control2_Basal_Normal_Square_Dual, respectively. In the composition, firstly, the models Control2 and Basal are composed to form the model Control2_Basal with

the variable *prog*1, then it is further composed with the model Normal to obtain the model Control2_Basal_Normal, where *prog*1 is replaced by *prog*2 following the pattern above to call the functions from either the model Basal or Normal. The composed model Control2_Basal_Normal refines the models Control2, Basal, Normal, and Control2_Basal.

4 Transforming Discrete Systems to Hybrid Systems

The insulin pump controls the continuous injection rate. Thus, we would like to analyze both discrete controller and continuous environment together. We refine the discrete models to hybrid systems following the hybrid approach proposed in [11]. For the modeling of Product2 defined in formula (3), firstly we refine the three independent features Basal, Normal, and Square/Dual to hybrid systems, to control the rates *brate*, *nrate*, and *sdrate*, respectively. Then we refine the composed model Control2_Basal_Normal_Square_Dual to a hybrid system with a merged pump rate.

To animate the hybrid systems, we developed a prototype to translate the hybrid Event-B to Matlab Simulink. Using this prototype, the simulation rates *brate*, *nrate*, and *sdrate* of the model Basal, Normal, and Square/Dual are shown in Fig. 4, respectively. The *pump rate* of the composed model which is a superimposition of *brate*, *nrate*, and *sdrate* are shown in Fig. 4 as well.

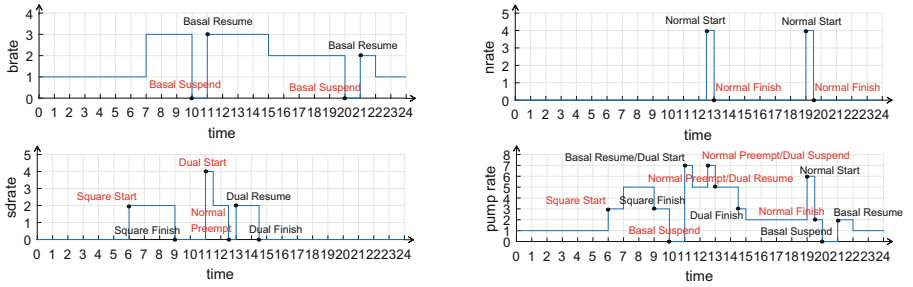


Fig. 4. Pump rate

5 Conclusion

The statistics of Proof Obligations (POs) of Product2 is listed in Table 1, where the first four models are independent models with refinements, and the last three models show that the composition is done gradually. The POs are proved 100% automatically. The extra POs of the composed model are generated for

Table 1. Statistics of the POs

Model	Total
Basal	344
Normal	88
Square_Dual	141
Control2	102
Control2_Basal	634
Control2_Basal_Normal	794
Control2_Basal_Normal_Square_Dual	1858

verifying the function calls, which result in an increase of the POs and make the provers work heavily. In the future work, we will propose a systematic approach that gradually introduces function calls when composing models, as well as fits paralleling system.

References

1. Medtronic: Paradigm Real-Time Insulin Pump and Continuous Glucose Monitoring System Insulin Pump User Guide. Medtronic MiniMed Inc. (2008)
2. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
3. Höfner, P., Khédri, R., Möller, B.: An algebra of product families. Softw. Syst. Model. **10**(2), 161–182 (2011)
4. Hoang, T.S., Dghaym, D., Snook, C.F., Butler, M.J.: A composition mechanism for refinement-based methods. In: ICECCS, pp. 100–109 (2017)
5. Abrial, J., Hallerstede, S.: Refinement, decomposition, and instantiation of discrete models: application to Event-B. Fundam. Inform. **77**(1–2), 1–28 (2007)
6. Silva, R., Butler, M.: Shared event composition/decomposition in Event-B. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 122–141. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25271-6_7
7. Poppleton, M.: The composition of Event-B models. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 209–222. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87603-8_17
8. Iliasov, A., Troubitsyna, E., Laibinis, L., Romanovsky, A., Varpaaniemi, K., Ilic, D., Latvala, T.: Supporting reuse in Event B development: modularisation approach. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 174–188. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11811-1_14
9. Abrial, J.R., Su, W.: Transforming guarded events into pre-conditioned operations (2014). http://wiki.event-b.org/images/Abrial2RUDW2014.pdf
10. Formal Models. http://www.cas.mcmaster.ca/khedri/?page_id=1219
11. Su, W., Abrial, J., Zhu, H.: Formalizing hybrid systems with Event-B and the Rodin platform. Sci. Comput. Program. **94**, 164–202 (2014)

An Automation-Friendly Set Theory for the B Method

Guillaume Bury[1], Simon Cruanes[2], David Delahaye[3(✉)], and Pierre-Louis Euvrard[3]

[1] LSV, ENS Paris-Saclay, Inria, Cachan, France
Guillaume.Bury@inria.fr
[2] Aesthetic Integration, Austin, TX, USA
simon@aestheticintegration.com
[3] LIRMM, Université de Montpellier, CNRS, Montpellier, France
{David.Delahaye,Pierre-Louis.Euvrard}@lirmm.fr

Abstract. We propose an automation-friendly set theory for the B method. This theory is expressed using first order logic extended to polymorphic types and rewriting. Rewriting is introduced along the lines of deduction modulo theory, where axioms are turned into rewrite rules over both propositions and terms. We also provide experimental results of several tools able to deal with polymorphism and rewriting over a benchmark of problems in pure set theory (i.e. without arithmetic).

Keywords: B method · Set theory · Automated deduction
Polymorphic types · Rewriting

1 Introduction

In this paper, we present the set theory of the B method [1] using polymorphic types and rewriting. Expressed this way, this theory has the benefit of being quite automatable for several reasons. In particular, the use of polymorphism allows us to make the theory more synthetic by removing some typing predicates, which therefore improves proof search. As for rewriting, it is introduced along the lines of deduction modulo theory [5], where axioms are turned into rewrite rules over both propositions and terms. Deduction modulo theory has proved to be also very useful to improve proof search when integrated to usual automated proof techniques. In this paper, we also aim to advertise that more and more automated tools are able to deal with polymorphic types and rewriting, and we provide some experimental results involving the latest versions of these tools.

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 409–414, 2018.
https://doi.org/10.1007/978-3-319-91271-4_32

Axioms of Set Theory

$(x,y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} s \times_{\alpha_1,\alpha_2} t \longrightarrow x \in_{\alpha_1} s \wedge y \in_{\alpha_2} t$
$s \in_{\mathsf{set}(\alpha)} \mathbb{P}_\alpha(t) \longrightarrow \forall x : \alpha.x \in_\alpha s \Rightarrow x \in_\alpha t$
$s =_{\mathsf{set}(\alpha)} t \longrightarrow \forall x : \alpha.x \in_\alpha s \Leftrightarrow x \in_\alpha t$

Set Inclusion

$s \subseteq_\alpha t \longrightarrow s \in_{\mathsf{set}(\alpha)} \mathbb{P}_\alpha(t) \qquad s \subset_\alpha t \longrightarrow s \subseteq_\alpha t \wedge s \neq_{\mathsf{set}(\alpha)} t$

Derived Constructs

$x \in_\alpha s \cup_\alpha t \longrightarrow x \in_\alpha s \vee x \in_\alpha t$ $\qquad$ $x \in_\alpha s \cap_\alpha t \longrightarrow x \in_\alpha s \wedge x \in_\alpha t$
$x \in s -_\alpha t \longrightarrow x \in_\alpha s \wedge x \notin_\alpha t$ $\qquad$ $x \in_\alpha \emptyset_\alpha \longrightarrow \bot$
$x \in_\alpha \{a\}_\alpha \longrightarrow x =_\alpha a$ $\qquad$ $\mathbb{P}_{1\,\alpha}(s) \longrightarrow \mathbb{P}_\alpha(s) -_\alpha \{\emptyset_\alpha\}_{\mathsf{set}(\alpha)}$

Binary Relation Constructs: First Series

$p \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} u \leftrightarrow_{\alpha_1,\alpha_2} v \longrightarrow$
$\quad \forall x : \alpha_1.\forall y : \alpha_2.(x,y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} p \Rightarrow x \in_{\alpha_1} u \wedge y \in_{\alpha_2} v$
$(y,x) \in_{\mathsf{tup}(\alpha_2,\alpha_1)} p^{-1}_{\alpha_1,\alpha_2} \longrightarrow (x,y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} p$
$x \in_{\alpha_1} \mathsf{dom}_{\alpha_1,\alpha_2}(p) \longrightarrow \exists b : \alpha_2.(x,b) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} p$
$x \in_{\alpha_2} \mathsf{ran}_{\alpha_1,\alpha_2}(p) \longrightarrow \exists a : \alpha_1.(a,x) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} p$
$(x,y) \in_{\mathsf{tup}(\alpha_1,\alpha_3)} p;_{\alpha_1,\alpha_2,\alpha_3} q \longrightarrow \exists b : \alpha_2.(x,b) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} p \wedge (b,y) \in_{\mathsf{tup}(\alpha_2,\alpha_3)} q$
$q \circ_{\alpha_1,\alpha_2,\alpha_3} p \longrightarrow p;_{\alpha_1,\alpha_2,\alpha_3} q$
$(x,y) \in_{\mathsf{tup}(\alpha,\alpha)} \mathsf{id}_\alpha(u) \longrightarrow x \in_\alpha u \wedge x =_\alpha y$
$(x,y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} s \lhd_{\alpha_1,\alpha_2} p \longrightarrow (x,y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} p \wedge x \in_{\alpha_1} s$
$(x,y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} p \rhd_{\alpha_1,\alpha_2} t \longrightarrow (x,y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} p \wedge y \in_{\alpha_2} t$
$(x,y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} s \lhd\!\!\!\cdot\, _{\alpha_1,\alpha_2} p \longrightarrow (x,y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} p \wedge x \notin_{\alpha_1} s$
$(x,y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} p \rhd\!\!\!\cdot\, _{\alpha_1,\alpha_2} t \longrightarrow (x,y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} p \wedge y \notin_{\alpha_2} t$

Fig. 1. Rewriting rules of the B set theory (part 1)

2 A Set Theory with Polymorphism and Rewriting for B

In the following, we consider the pure set theory part of the B method, i.e. the material introduced in Chap. 2 of the B-Book [1]. This part of the B theory is suitable as it can be easily turned into a theory that is compatible with deduction modulo theory, i.e. where a large part of axioms can be turned into rewrite rules. We therefore transform whenever possible the axioms and definitions into rewrite rules. The resulting theory is summarized in Figs. 1 and 2, where we omit the set BIG and the sets defined in extension.

As can be seen, the proposed theory is typed, using first order logic extended to polymorphic types à la ML, through a type system in the spirit of [2]. This extension to polymorphic types offers more flexibility, and allows us to deal with

Binary Relation Constructs: Second Series

$$x \in_{\alpha_2} p[w]_{\alpha_1,\alpha_2} \longrightarrow \exists a : \alpha_1 . a \in_{\alpha_1} w \wedge (a, x) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} p$$
$$(x, y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} q <+_{\alpha_1,\alpha_2} p \longrightarrow$$
$$((x, y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} q \wedge x \notin_{\alpha_1} \mathsf{dom}_{\alpha_1,\alpha_2}(p)) \vee (x, y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} p$$
$$(x, (y, z)) \in_{\mathsf{tup}(\alpha_1,\mathsf{tup}(\alpha_2,\alpha_3))} f \otimes_{\alpha_1,\alpha_2,\alpha_3} g \longrightarrow (x, y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} f \wedge (x, z) \in_{\mathsf{tup}(\alpha_1,\alpha_3)} g$$
$$((x, y), z) \in_{\mathsf{tup}(\mathsf{tup}(\alpha_1,\alpha_2),\alpha_1)} \mathsf{prj}_{1\ \alpha_1,\alpha_2}(s, t) \longrightarrow$$
$$((x, y), z) \in_{\mathsf{tup}(\mathsf{tup}(\alpha_1,\alpha_2),\alpha_1)} (s \times_{\alpha_1,\alpha_2} t) \times_{\mathsf{tup}(\alpha_1,\alpha_2),\alpha_1} s \wedge x =_{\alpha_1} z$$
$$((x, y), z) \in_{\mathsf{tup}(\mathsf{tup}(\alpha_1,\alpha_2),\alpha_2)} \mathsf{prj}_{2\ \alpha_1,\alpha_2}(s, t) \longrightarrow$$
$$((x, y), z) \in_{\mathsf{tup}(\mathsf{tup}(\alpha_1,\alpha_2),\alpha_1)} (s \times_{\alpha_1,\alpha_2} t) \times_{\mathsf{tup}(\alpha_1,\alpha_2),\alpha_1} t \wedge y =_{\alpha_1} z$$
$$((x, y), (z, w)) \in_{\mathsf{tup}(\mathsf{tup}(\alpha_1,\alpha_3),\mathsf{tup}(\alpha_2,\alpha_4))} h||_{\alpha_1,\alpha_2,\alpha_3,\alpha_4} k \longrightarrow$$
$$(x, z) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} h \wedge (y, w) \in_{\mathsf{tup}(\alpha_3,\alpha_4)} k$$

Function Constructs: First Series

$$f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \mathrel{+\!\!\!\rightarrow}_{\alpha_1,\alpha_2} t \longrightarrow f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \leftrightarrow_{\alpha_1,\alpha_2} t \wedge$$
$$\forall x : \alpha_1 . \forall y, z : \alpha_2 . (x, y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} f \wedge (x, z) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} f \Rightarrow y =_{\alpha_2} z$$
$$f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \rightarrow_{\alpha_1,\alpha_2} t \longrightarrow$$
$$f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \mathrel{+\!\!\!\rightarrow}_{\alpha_1,\alpha_2} t \wedge \mathsf{dom}_{\alpha_1,\alpha_2}(f) =_{\mathsf{set}(\alpha_1)} s$$
$$f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \rightarrowtail_{\alpha_1,\alpha_2} t \longrightarrow$$
$$f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \mathrel{+\!\!\!\rightarrow}_{\alpha_1,\alpha_2} t \wedge f^{-1}_{\alpha_1,\alpha_2} \in_{\mathsf{set}(\mathsf{tup}(\alpha_2,\alpha_1))} t \mathrel{+\!\!\!\rightarrow}_{\alpha_2,\alpha_1} s$$
$$f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \rightarrowtail_{\alpha_1,\alpha_2} t \longrightarrow$$
$$f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \rightarrowtail_{\alpha_1,\alpha_2} t \wedge f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \rightarrow_{\alpha_1,\alpha_2} t$$
$$f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \mathrel{+\!\!\!\twoheadrightarrow}_{\alpha_1,\alpha_2} t \longrightarrow$$
$$f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \mathrel{+\!\!\!\rightarrow}_{\alpha_1,\alpha_2} t \wedge \mathsf{ran}_{\alpha_1,\alpha_2}(f) =_{\mathsf{set}(\alpha_2)} t$$
$$f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \twoheadrightarrow_{\alpha_1,\alpha_2} t \longrightarrow$$
$$f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \mathrel{+\!\!\!\twoheadrightarrow}_{\alpha_1,\alpha_2} t \wedge f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \rightarrow_{\alpha_1,\alpha_2} t$$
$$f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \rightarrowtail\!\!\!\!\twoheadrightarrow_{\alpha_1,\alpha_2} t \longrightarrow$$
$$f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \rightarrowtail_{\alpha_1,\alpha_2} t \wedge f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \mathrel{+\!\!\!\twoheadrightarrow}_{\alpha_1,\alpha_2} t$$
$$f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \rightarrowtail\!\!\!\!\twoheadrightarrow_{\alpha_1,\alpha_2} t \longrightarrow$$
$$f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \rightarrowtail_{\alpha_1,\alpha_2} t \wedge x \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \twoheadrightarrow_{\alpha_1,\alpha_2} t$$

Fig. 2. Rewriting rules of the B set theory (part 2)

theories that rely on elaborate type systems, like the B set theory. The complete type system used here can be found in [3]. The type constructors, i.e. tup for tuples and set for sets, and type schemes of the considered set constructs are provided in Fig. 3 of Appendix A, where Type is the type of types and o the type of formulas. Type arguments are subscript annotations of the construct, and to improve readability, we remove the type annotations in tuples when they are redundant with the membership construct.

Table 1. Experimental results over the B set theory benchmark

319 problems	Zenon Modulo	ArchSAT	Zipperposition	Alt-Ergo
Proofs	138	272	306	232
Rate	43.3%	85.3%	95.9%	72.7%
Time(s)	2.86	268.69	109.88	8.42

3 Experimental Results

To test the previous theory, we consider 319 lemmas[1] coming from Chap. 2 of the B-Book [1]. As tools, we consider automated theorem provers able to deal with polymorphic types and rewriting natively. Our set of tools includes: Zenon Modulo (version 0.4.2), a tableau-based prover that is an extension of Zenon to deduction modulo theory; ArchSAT (development version[2]), a prover that combines a SAT solver with tableau calculus and rewriting; and Zipperposition (version 1.5), a prover based on superposition and rewriting. To show the impact of rewriting over the results, we also include the Alt-Ergo SMT solver (version 1.01), which deals with polymorphic types but not rewriting.

The experiment was run on an Intel Xeon E5-1650 v3 3.50 GHz computer, with a timeout of 90 s and a memory limit of 1 GiB. The results are summarized in Table 1. These results show the high performances, in terms of proved problems, obtained by the provers extended to rewriting, Zipperposition and ArchSAT in particular, compared to the SMT approach of Alt-Ergo. Looking at the cumulative times, Alt-Ergo is not really faster than Zipperposition and ArchSAT, which take more time to find few more difficult problems (with a timeout of 3 s, they respectively find 303 and 260 proofs in 17.61 s and 16.61 s, while Alt-Ergo finds the same number of proofs). The low results of Zenon Modulo are probably due to the fact that it uses a heuristic to transform the axioms into rewrite rules.

4 Conclusion

In light of the previous experimental results and as perspectives, we aim to apply our approach, consisting of a B set theory using polymorphic types and rewriting together with appropriate tools (Zenon Modulo, ArchSAT, and Zipperposition), to proof obligations coming from the formalization of real-world applications. In particular, we plan to use the benchmark provided by the industrial partners of the BWare project [4], which gathers about 13,000 proof obligations.

[1] The benchmark is available at: https://github.com/delahayd/bset.
[2] Git version 7720d8c, available at: https://gforge.inria.fr/projects/archsat.

A Typing of the B Set Theory

Type Constructors

$\mathsf{tup} : \Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{Type}$ $\quad$ $\mathsf{set} : \Pi\alpha : \mathsf{Type}.\mathsf{Type}$

Type Schemes of the Set Constructs

$\mathsf{-} \in \mathsf{-}$: $\Pi\alpha : \mathsf{Type}.\alpha \to \mathsf{set}(\alpha) \to o$

$(\mathsf{-},\mathsf{-})$: $\Pi\alpha_1, \alpha_2 : \mathsf{Type}.\alpha_1 \to \alpha_2 \to \mathsf{tup}(\alpha_1, \alpha_2)$

$\mathsf{-} \times \mathsf{-}$: $\Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{set}(\alpha_1) \to \mathsf{set}(\alpha_2) \to \mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2))$

$\mathbb{P}(\mathsf{-})$: $\Pi\alpha : \mathsf{Type}.\mathsf{set}(\alpha) \to \mathsf{set}(\mathsf{set}(\alpha))$

$\mathsf{-} = \mathsf{-}$: $\Pi\alpha : \mathsf{Type}.\alpha \to \alpha \to o$

BIG : $\Pi\alpha : \mathsf{Type}.\mathsf{set}(\alpha)$

$\mathsf{-} \subseteq \mathsf{-}$, $\mathsf{-} \subsetneq \mathsf{-}$:
$\Pi\alpha : \mathsf{Type}.\mathsf{set}(\alpha) \to \mathsf{set}(\alpha) \to o$

$\mathsf{-} \cup \mathsf{-}$, $\mathsf{-} \cap \mathsf{-}$, $\mathsf{-} - \mathsf{-}$:
$\Pi\alpha : \mathsf{Type}.\mathsf{set}(\alpha) \to \mathsf{set}(\alpha) \to \mathsf{set}(\alpha)$

$\{\mathsf{-}\}$: $\Pi\alpha : \mathsf{Type}.\alpha \to \mathsf{set}(\alpha)$

$\emptyset$: $\Pi\alpha : \mathsf{Type}.\mathsf{set}(\alpha)$

$\mathbb{P}_1(\mathsf{-})$: $\Pi\alpha : \mathsf{Type}.\mathsf{set}(\alpha) \to \mathsf{set}(\mathsf{set}(\alpha))$

$\mathsf{-} \leftrightarrow \mathsf{-}$: $\Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{set}(\alpha_1) \to \mathsf{set}(\alpha_2) \to \mathsf{set}(\mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)))$

$\mathsf{-}^{-1}$: $\Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \to \mathsf{set}(\mathsf{tup}(\alpha_2, \alpha_1))$

$\mathsf{dom}(\mathsf{-})$: $\Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \to \mathsf{set}(\alpha_1)$

$\mathsf{ran}(\mathsf{-})$: $\Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \to \mathsf{set}(\alpha_2)$

$\mathsf{-};\mathsf{-}$: $\Pi\alpha_1, \alpha_2, \alpha_3 : \mathsf{Type}.\mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \to \mathsf{set}(\mathsf{tup}(\alpha_2, \alpha_3)) \to \mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_3))$

$\mathsf{-} \circ \mathsf{-}$: $\Pi\alpha_1, \alpha_2, \alpha_3 : \mathsf{Type}.\mathsf{set}(\mathsf{tup}(\alpha_2, \alpha_3)) \to \mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \to \mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_3))$

$\mathsf{id}(\mathsf{-})$: $\Pi\alpha : \mathsf{Type}.\mathsf{set}(\alpha) \to \mathsf{set}(\mathsf{tup}(\alpha, \alpha))$

$\mathsf{-} \lhd \mathsf{-}$: $\Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{set}(\alpha_1) \to \mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \to \mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2))$

$\mathsf{-} \rhd \mathsf{-}$: $\Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \to \mathsf{set}(\alpha_2) \to \mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2))$

$\mathsf{-} \lhd \mathsf{-}$: $\Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{set}(\alpha_1) \to \mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \to \mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2))$

$\mathsf{-} \rhd \mathsf{-}$: $\Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \to \mathsf{set}(\alpha_2) \to \mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2))$

$\mathsf{-}[\mathsf{-}]$: $\Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \to \mathsf{set}(\alpha_1) \to \mathsf{set}(\alpha_2)$

$\mathsf{-} \lhd\!\!+ \mathsf{-}$: $\Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \to \mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \to \mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2))$

$\mathsf{-} \otimes \mathsf{-}$: $\Pi\alpha_1, \alpha_2, \alpha_3 : \mathsf{Type}.\mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \to \mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_3)) \to$
$\mathsf{set}(\mathsf{tup}(\alpha_1, \mathsf{tup}(\alpha_2, \alpha_3)))$

$\mathsf{prj}_1(\mathsf{-})$: $\Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{tup}(\mathsf{set}(\alpha_1), \mathsf{set}(\alpha_2)) \to \mathsf{set}(\mathsf{tup}(\mathsf{tup}(\alpha_1, \alpha_2), \alpha_1))$

$\mathsf{prj}_2(\mathsf{-})$: $\Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{tup}(\mathsf{set}(\alpha_1), \mathsf{set}(\alpha_2)) \to \mathsf{set}(\mathsf{tup}(\mathsf{tup}(\alpha_1, \alpha_2), \alpha_2))$

$\mathsf{-}||\mathsf{-}$: $\Pi\alpha_1, \alpha_2, \alpha_3, \alpha_4 : \mathsf{Type}.\mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \to \mathsf{set}(\mathsf{tup}(\alpha_3, \alpha_4)) \to$
$\mathsf{set}(\mathsf{tup}(\mathsf{tup}(\alpha_1, \alpha_3), \mathsf{tup}(\alpha_2, \alpha_4)))$

$\mathsf{-} \nrightarrow \mathsf{-}$, $\mathsf{-} \to \mathsf{-}$, $\mathsf{-} \rightarrowtail \mathsf{-}$, $\mathsf{-} \rightarrowtail \mathsf{-}$, $\mathsf{-} \twoheadrightarrow \mathsf{-}$, $\mathsf{-} \twoheadrightarrow \mathsf{-}$, $\mathsf{-} \twoheadrightarrow \mathsf{-}$, $\mathsf{-} \twoheadrightarrow \mathsf{-}$:
$\Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{set}(\alpha_1) \to \mathsf{set}(\alpha_2) \to \mathsf{set}(\mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)))$

$\mathsf{-}(\mathsf{-})$: $\Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \to \alpha_1 \to \alpha_2$

Fig. 3. Type constructors and type schemes of the set constructs

References

1. Abrial, J.-R.: The B-Book, Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996). ISBN 0521496195
2. Blanchette, J.C., Paskevich, A.: TFF1: the TPTP typed first-order form with rank-1 polymorphism. In: Bonacina, M.P. (ed.) CADE 2013. LNCS (LNAI), vol. 7898, pp. 414–420. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38574-2_29
3. Bury, G., Delahaye, D., Doligez, D., Halmagrand, P., Hermant, O.: Automated deduction in the B set theory using typed proof search and deduction modulo. In: Fehnker, A., McIver, A., Sutcliffe, G., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence and Reasoning (LPAR) - Short Presentations, vol. 35, pp. 42–58. EasyChair, Suva (Fiji), November 2015
4. Delahaye, D., Dubois, C., Marché, C., Mentré, D.: The BWare project: building a proof platform for the automated verification of B proof obligations. In: Ameur, Y.A., Schewe, K.-D. (eds.) ABZ 2014. LNCS, vol. 8477, pp. 126–127. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43652-3_26
5. Dowek, G., Hardin, T., Kirchner, C.: Theorem proving modulo. J. Autom. Reasoning (JAR) **31**(1), 33–72 (2003)

Teaching an Old Dog New Tricks

The Drudges of the Interactive Prover in Atelier B

Lilian Burdy and David Deharbe(✉)

ClearSy Systems Engineering, 320, av Archimède – Les Pléiades III,
13857 Aix-en-Provence CEDEX 3, France
{lilian.burdy,david.deharbe}@clearsy.com

Abstract. This paper presents an evolution of an industrial proof support framework that integrates state-of-the-art technologies without compromising the existing tool qualification status. Third-party provers produce proof rules that may be applied by the legacy system and verified using a certified approach. This approach has been implemented in Atelier B, a formal-methods based IDE for the development of software components and for the modeling of systems.

1 Introduction

The industrial applications of formal methods rely on the formal verification of conditions, e.g., invariant preservation. In case the specification logic is undecidable, human interaction is required to discharge proof obligations. This is the case of the B method and Event-B, two closely related formal methods, based on a first-order language with integer arithmetics and set theory. So even though automatic theorem provers are available, their application requires their users to interact with proof assistants. Atelier B, an integrated development environment for both the B method and Event-B, includes custom automatic and interactive provers. Increasing the success rate of automatic provers and decreasing the amount of user interaction are key to reducing the cost of formal methods and increasing their application in the industry. This paper addresses the former approach.

One main industrial application of formal methods in the industry is the development of safety-critical, software-based, systems. Mostly, formal methods are used when mandated or recommended by an industrial standard (e.g. EN50128 [3]). In that context, all the tool support must be qualified. Obtaining such a qualification has a significant cost and then usually only applies to a specific version of the tool. Atelier B has been qualified to formally develop software components by large industrial partners in the railway industry. This paper presents an extension to Atelier B that improves its support for interactive proof without compromising the certification obtained by the existing code base.

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 415–419, 2018.
https://doi.org/10.1007/978-3-319-91271-4_33

2 Technical Background

Atelier B and Proof. In Atelier B, proof obligations (POs) are produced automatically, two proof engines being available to check them. One solver, called *pr*, is a conditional term rewrite engine written in a Prolog-like notation, called the *theory language*, together with a default set of rules. Users may also write their own rules, which need also be verified eventually. The second solver, called *pp* for *predicate prover*, was developed to prove such rules and takes an (incomplete) tableaux-based approach. Both have been certified[1]. For rules that *pp* cannot handle, the certified verification procedure is to first prove the rule with a mathematical demonstration and then have a third-party independent expert validate this demonstration. Note that *pp* is also available to solve proof obligations.

In interactive mode, the user is presented with a PO composed of a goal, local hypotheses and global hypotheses. In the course of an interactive session, the goal and the hypotheses evolve, new POs are created and the current PO may be discharged. Examples of commands are: rewrite the goal using an equality from the hypotheses; call a built-in expression simplifier; apply either *pr* or *pp* on the current goal; instantiate a universally quantified hypothesis; apply a given rule (or set of rules); case split on a condition. The execution of some commands produce new PO, e.g., case splits and instantiations. A PO is discharged when a so called terminal rule is applied. An example of terminal rule is: `binhyp(a) => a or b`. Here `a` and `b` are so-called jokers and stand for terms, `=>` is a delimiter in the rule between the conditions (to the left) and the conclusion (to the right). Such a rule can be applied if the current PO goal matches `a or b`, i.e., it is a disjunction, and the term matching `a` is found in the hypotheses (that is the semantics of the `binhyp` operator).

Leveraging Automatic Provers in Interactive Proof Assistant. Since Atelier B has been originally developed, mechanical theorem proving has seen a lot of progress and powerful automatic provers are now available, such as SMT solvers [2,4]. The area of formal methods has also made efforts to use such tools to address its own verification challenges (see, e.g. [5,6]). Our goal has been to take advantage of some of these advances in Atelier B without compromising certification.

We follow the approach taken in the general purpose proof system Isabelle [7], extended with "sledgehammer" [8], a command to invoke external solvers on the current PO and, when successful, to reconstruct an Isabelle proof script from their output. Our take on applying this approach in the legacy proof system of Atelier B is an extension we named *the drudges of the interactive prover*.

3 The Drudges of the Interactive Prover: Principles

Even though the logic of POs in Atelier B is rich and undecidable, it is often the case that, during the course of an interactive proof, proving the current goal

[1] E.g., sanctioned for software development by RATP for line 14 (operated completely automatically).

only requires arguments that may be cast in a decidable logic, such as propositional logic, equality and integer arithmetics. For such goals, the layered solving architecture found in most SMT solvers is particularly fit. However this is not how Atelier B solvers function, and they sometimes fail to prove automatically POs that only require propositional reasoning or substitution of equals.

So our approach consists in applying SMT solvers to *an abstraction of the current goal*, i.e., a simplified version where the set operators are uninterpreted (i.e., their semantics is lost). Such simplified POs contain the declarations for the symbols from the original PO, then a series of assertions. There is one such assertion for each hypothesis (and so, all hypotheses are thus abstracted) and one assertion with the negation of the goal. All these assertions are labeled. If an SMT solver finds the simplified PO to be unsatisfiable, the original PO is valid. Then, all we have to do is build a rule that can be applied by the solver of Atelier B. This rule needs to be logically sound, applicable to the current PO, and as general as possible.

To this end, the SMT solver is then queried to obtain an unsatisfiable subset of the assertions, using the `get-unsat-core` functionality [2]. The solver then returns the set of the labels of the assertions it used to conclude unsatisfiability. Now all needs to be done is to build a rule from the logical formulas associated with these labels. Assuming the solver is sound, such a rule is valid. Also, by construction, it is applicable to the current PO. To make it more general, terms are replaced with jokers. The issue here is where jokers are introduced. To illustrate this point, consider the following PO:

$$\{\cdots; 0 \leq fn(s_3); s_1 = s_2 \vee s_1 = s_3; 0 \leq fn(s_2); \cdots \vdash fn(s_1) \leq fn(s_2) + fn(s_3)\},$$

where the hypotheses are to the left of $\vdash$ and the goal is to the right. Only the hypotheses returned by `get-unsat-core` are shown. Consider the proof rule, built with these formulas:

```
   binhyp(0<=fn(s3)) &
   binhyp(s1 = s2 or s1 = s3) &
   binhyp(0<=fn(s2)
=> fn(s1) <= fn(s2)+fn(s3)
```

This rule is sound but it only applies to POs where the goal and some hypotheses are identical to those in the rule.

To gain generality, we can replace (sub)-terms with jokers, by recursing over the structure of the given formulas, keeping the operators known to the SMT solver, and by introducing a joker for each different sub-term. The result is the following rule:

```
   binhyp(0<=a) &
   binhyp(b=c or b=d) &
   binhyp(0<=e)
=> f <= e+a
```

However, the abstraction here is too coarse and the resulting rule is no longer sound.

However, when jokers are introduced one level deeper in the syntactic structure, the rule obtained is sound, applicable to the original PO, and as general as possible given the initial problem.

```
   binhyp(0 <= a(b)) &
   binhyp(c=d or c=b) &
   binhyp(0 <= a(d))
=> a(c) <= a(d)+a(b)
```

This discussion illustrates the main principles in the generation of proof rules. Jokers are introduced at a given level, and if the rule is sound, the process stops, otherwise it is repeated one level deeper. This process is guaranteed to end at most when the level is the height of the original PO. The verification of the soundness is carried out by the same SMT solver that was applied to the original proof obligation.

4 The Drudges of the Interactive Prover: Application

The functionality is in release 4.5 of Atelier B. The user sets the preferences of the interactive prover to create drudges. A *drudge* is an SMT solver with settings to enable quantifiers and arithmetics reasoning. These settings guide the procedure that simplifies the POs for the SMT solver; e.g., disabling quantifiers forces all quantified sub-formulas to be abstracted.

The new functionality can be run in the interactive prover either with a command `smt` or with the click of a button. In case the interaction fails to prove the PO, a message is printed to the console. Otherwise, the following steps take place. First, a "most-general", sound, proof rule is created as described above. Second, the rule is added to the "pmm" file, a file containing the rules for the current component, in a section named `SMT_Rules` (actually, the rule is added only if no other equivalent rule is already present). Third, a command to apply the rules from `SMT_Rules` is issued to the Atelier B solver. The current PO is then proved. The interactive prover either loads the next proof obligation or signals completion of the session. In practice, the `smt` command is successful as soon as the proof only involves arithmetics, equalities and properties of Boolean operators. The pmm file of the component then contains all the rules that have been created through this command, thus guaranteeing compliance with certification requirements.

All the proof rules introduced in a development process should be verified. This includes the rules created by drudges. For this activity, the *pp* prover is applicable and practice shows that it is able to verify many such rules. For the rest, a manual proof must be performed.

5 Conclusion and Future Work

We extend the proof assistant of Atelier B with a command to run SMT solvers in a proof session. The main requirements to use a solver in our framework are

efficiency and a function returning the hypotheses used in a proof. This extension then creates and applies automatically proof rules.

For certification, these rules need to be verified. The *pp* solver of Atelier B is able to check only part of these rules. The remainder need to be verified and validated manually. We envision the following alternatives: to put a bridle to the function by only adding rules that are checked with *pp*; to develop a more efficient rule checker; to translate the proofs generated by the SMT solvers into a human-readable proof. Future work also includes: to evaluate axiomatizing set operators for the SMT solvers, to experiment with SMT solvers handling set operators (namely, CVC4 [1]), to use other classes of solvers (e.g., TPTP provers).

References

1. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
2. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa (2017)
3. CENELEC - EN 50128: Railway applications-communication, signaling and processing systems-software for railway control and protection systems (2011)
4. Cok, D.R., Déharbe, D., Weber, T.: The 2014 SMT competition. J. Satisf. Boolean Model. Comput. **9**, 207–242 (2016)
5. Couchot, J., Déharbe, D., Giorgetti, A., Ranise, S.: Scalable automated proving and debugging of set-based specifications. J. Braz. Comput. Soc. **9**(2), 17–36 (2004)
6. Déharbe, D., Fontaine, P., Guyot, Y., Voisin, L.: Integrating SMT solvers in Rodin. Sci. Comput. Program. **94**, 130–143 (2014)
7. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, vol. 2283. Springer Science & Business Media, Berlin (2002). https://doi.org/10.1007/3-540-45949-9
8. Paulson, L.C., Blanchette, J.C.: Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In: PAAR@ IJCAR, pp. 1–10 (2010)

Modelling Dynamic Data Structures with the B Method

Frédéric Badeau, Vincent Lacroix, Vincent Monfort, Laurent Voisin(✉), and Christophe Métayer

Systerel, Aix-en-Provence, France
{frederic.badeau,vincent.lacroix,vincent.monfort,
laurent.voisin,christophe.metayer}@systerel.fr
http://www.systerel.fr

The software B method has so far been mainly used in the industrial world to develop safety critical software with very basic memory management limited to arrays of fixed size defined at compilation time. We present here an alternative approach for modelling software based on a more classic memory management with dynamically allocated complex data structures accessed through pointers.

1 Context

Critical supervision systems are exposed to an increasing number of security threats, which have deep consequences in domains such as energy, transport, and defense. It is crucial to build components for the industry 4.0 and Industrial Internet of Things that can better resist such threats.

OPC-UA is a standard [6] for data exchange in industrial communications. It provides safe and secure means to connect supervision systems (SCADA) with programmable logic controller (PLC), actuators, and sensors.

INGOPCS is a research project that aims at developing an open source OPC-UA toolkit, named S2OPC[1]. This toolkit can be used for both server and client software. It has been designed to comply with both the EAL4 security level (Common Criteria [4]) and the SIL2 safety level (ISO/IEC 61508 [5]).

The toolkit is developed in C99. In order to ensure high assurance in the development, formal methods have been applied: B modelling of OPC-UA services, which is translated to C code; formal analysis of low-level manual C code using Frama-C and TrustInSoft Analyzer.

Systerel has strong experience in software B modelling for embedded railway applications. The toolkit however, does not address the same constraints:

– railway applications are cyclic (their entry point is a B operation called at each software cycle, that reads input messages, computes internal data and writes output messages), whereas the toolkit is triggered by events, either from the network or the application level,

This work has been supported by an FUI 19 grant of the French government.

[1] See https://gitlab.com/systerel/S2OPC.

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 420–424, 2018.
https://doi.org/10.1007/978-3-319-91271-4_34

– railway applications use static memory management, whereas the toolkit requires dynamic management, due to the wide diversity of exchanged objects.

This paper explains how Systerel adapted its methods and design patterns to build a B model of the OPC-UA services.

2 Modelling Dynamic Data

Historically, the software B method [1] has been used in an operational context for the development of safety critical railway software [2,3]. To meet these needs, the B language has evolved, B translators into classic programming languages, such as the C language, have been developed and methodological principles for software B development have been defined. This package B language/translator/software B methodology has reached a good level of maturity for the development of this type of software and has changed little since the early 2000s.

Although this approach is satisfactory, it is limited to the development of cyclic software where data is entirely managed within the model. All variables are statically allocated and contain simple data structures.

These features are not compatible with the development of software implementing the OPC-UA protocol that heavily uses record-based data structures, pointers to these data and dynamic allocation. We could try to get around these memory management issues by interfacing the C data structures with B arrays. However this approach is not satisfactory because it would imply a lot of data copying in between the C program and the B model.

We will see how to solve all these problems without changing either the B language or the B to C translators, thus bringing methodological solutions.

Classic B Data Structures. Implementable data structures supported by the software B language and the C translators are 1 or 2-dimensional scalar arrays and scalar types, where scalar types are the `int` type, the Boolean type, enumerated types and carrier sets. Attempts to introduce record types into the B-Language have raised major issues that made them impractical. For instance, modifying several fields of the same record in a row leads to very large and hard proof obligations. Therefore, with the classic B methodology, single records are usually replaced by several scalars and arrays of records are replaced by several arrays, one for each field of the record.

Carrier Sets. One of the great advances of the current software B methodology has been to successfully use carrier sets to provide strong typing at the B level, while at the C level everything is integer. If we were to use integer subtypes instead of carrier sets, we could misuse a variable for another one belonging to a different subset, even at proof level, since all variables would be regular integers. For example, in a rail system, trains, signals or points may be modeled by carrier sets. Thanks to the strong typing, one is sure not to confuse a signal and a point in the B formulas.

Pointer Management. To handle pointers, the key idea is to consider that a carrier set may represent a pointer type. For instance, to represent a pointer to a record of two Cartesian coordinates x and y, we define in a basic machine a pointer type `t_pos_i` denoting all possible pointer values, a subset `t_pos` denoting pointers valid at some point in time and constant `c_pos_undef` denoting the `NULL` pointer value. The fields of the pointed record are modelled by partial functions. In addition, we define operations for memory management (allocation and de-allocation) and access to field values. The basic machine is implemented straightforwardly in C with the usual semantics.

SETS
  $\mathtt{t\_pos\_i}$

ABSTRACT_CONSTANTS	VARIABLES
$\mathtt{t\_pos}$,	$\mathtt{f\_pos\_x}$,
$\mathtt{c\_pos\_undef}$	$\mathtt{f\_pos\_y}$
PROPERTIES	INVARIANT
$\mathtt{t\_pos} \subseteq \mathtt{t\_pos\_i} \wedge$	$\mathtt{f\_pos\_x} \in \mathtt{t\_pos} \nrightarrow \mathbb{Z} \wedge$
$\mathtt{c\_pos\_undef} \in \mathtt{t\_pos\_i} \wedge$	$\mathtt{f\_pos\_y} \in \mathtt{t\_pos} \nrightarrow \mathbb{Z} \wedge$
$\mathtt{c\_pos\_undef} \notin \mathtt{t\_pos}$	$\mathrm{dom}(\mathtt{f\_pos\_x}) = \mathrm{dom}(\mathtt{f\_pos\_y})$

INITIALISATION
  $\mathtt{f\_pos\_x}, \mathtt{f\_pos\_y} := \varnothing, \varnothing$
OPERATIONS
  $\mathtt{p} \leftarrow \mathtt{pos\_alloc} \mathrel{\widehat{=}}$
    CHOICE $\mathtt{p} := \mathtt{c\_pos\_undef}$ OR
      ANY $\mathtt{np}, \mathtt{nx}, \mathtt{ny}$
      WHERE $\mathtt{np} \in \mathtt{t\_pos} - \mathrm{dom}(\mathtt{f\_pos\_x}) \ \wedge \ \mathtt{nx} \in \mathbb{Z} \ \wedge \ \mathtt{ny} \in \mathbb{Z}$
      THEN $\mathtt{f\_pos\_x}(np) := \mathtt{nx} \parallel \mathtt{f\_pos\_y}(np) := \mathtt{ny} \parallel \mathtt{p} := \mathtt{np}$ END
    END
  $\mathtt{pos\_free}(\mathtt{p}) \mathrel{\widehat{=}}$
    PRE $\mathtt{p} \in \mathrm{dom}(\mathtt{f\_pos\_x})$
    THEN $\mathtt{f\_pos\_x} := \{\mathtt{p}\} \lhd \mathtt{f\_pos\_x} \parallel \mathtt{f\_pos\_y} := \{\mathtt{p}\} \lhd \mathtt{f\_pos\_y}$ END
  $\mathtt{x} \leftarrow \mathtt{get\_pos\_x}(\mathtt{p}) \mathrel{\widehat{=}}$ PRE $\mathtt{p} \in \mathrm{dom}(\mathtt{f\_pos\_x})$ THEN $x := \mathtt{f\_pos\_x}(p)$ END
  $\mathtt{set\_pos\_x}(\mathtt{p}, \mathtt{x}) \mathrel{\widehat{=}}$ PRE $\mathtt{p} \in \mathrm{dom}(\mathtt{f\_pos\_x})$ THEN $\mathtt{f\_pos\_x}(p) := \mathtt{x}$ END

As pointers are viewed as scalars at the B level, nothing prevents us from defining more complex structures where a field value is itself a pointer to another record. We even can use arrays of pointers.

3 Use Case: OPC-UA Message Manipulation

The decoding of incoming OPC-UA messages is an interesting application of this approach. Messages arrive on a network socket as a stream of bytes. These

bytes are stored in a byte buffer (whose size is dynamic and unknown at compile time) by low-level C code. They are then interpreted by the B model to create a structured message, that is a record containing itself other records and arrays. Again, this message structure is dynamic and depends on the contents of the buffer, so it cannot be pre-allocated.

A pointer to the byte buffer is passed as input to a B model entry point. This buffer is already allocated and passed as a valid pointer. The buffer state variable is initialized as not read yet. The buffer can then be read in a predefined order that is enforced by buffer states used as preconditions for decoding operations (message type, message header and message). These decoding operations allocate message structures. In case of failure the buffer becomes unreadable (undefined state). When reading is finished, the buffer is invalidated by setting its state to undefined to prevent misuse.

The input message is allocated by the decoding operation. It is thereon used as a valid message pointer. Then, messsage fields can be accessed as stated in Sect. 2. Once the message content has been consumed, it is deallocated and the pointer is set as invalid.

This example shows how the B model encapsulates the low-level byte buffer and message structures. It controls the structure manipulation and the possible field accesses. The memory management is also driven by the B model, even though the object lifecycle is partial in it (e.g., an output pointer remains valid out of the model but is considered invalid in it). Basic machines are only used for low-level and basic operations (actual decoding, structure accesses).

4 Conclusion

The new approach that we have described allows us to incorporate explicitly dynamic memory management in our B model, including allocation, deallocation and the validity of pointers. Since these properties are represented in the model, they can be used in B invariants and operations to provide guarantees when manipulating data referenced by pointers. As a consequence, operations can define pre-conditions to manipulate only valid pointers and have read/write access to the underlying data structures. Moreover the input and output data flow can also be treated in the model even if part of the memory management is done upstream or downstream of the model.

Another important aspect is that the data structures and their associated low-level services (implemented directly in C) can be encapsulated in the B model to add high-level properties on the services which are guaranteed by a formal proof.

Thanks to the methodological advances presented, we successfully developed a B model integrated in a piece of C software manipulating dynamically allocated data and pointers, notably on structures.

There are still a few issues that we did not address yet. Firstly, all our data structures contain a fixed number of elements whose size is also statically fixed. One could extend the approach to dynamically sized fields and variant records

(e.g., C unions). Secondly, for recursive data structures (e.g., linked lists, trees) one may also want to express some (inductive) property on the data structure as a whole, while our approach is currently limited to each record (e.g., cell) independently of the others.

References

1. Abrial, J.: The B-book - Assigning Programs to Meanings. Cambridge University Press, Cambridge (2005)
2. Badeau, F., Amelot, A.: Using B as a high level programming language in an industrial project: Roissy VAL. In: Treharne, H., King, S., Henson, M., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 334–354. Springer, Heidelberg (2005). https://doi.org/10.1007/11415787_20
3. Behm, P., Benoit, P., Faivre, A., Meynadier, J.-M.: Météor: a successful application of B in a large project. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 369–387. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48119-2_22
4. Common Criteria: Common criteria for information technology security evaluation. CCMB 2017–04-001, Common Criteria Portal (2017)
5. IEC: Functional safety of electrical/electronic/programmable electronic safety-related systems. IEC 61508:2010, International Electrotechnical Commission, Geneva, Switzerland (2016)
6. IEC: OPC unified architecture. IEC TR 62541:2016, International Electrotechnical Commission, Geneva, Switzerland (2016)

On the Importance of Explicit Domain Modelling in Refinement-Based Modelling Design. Experiments with Event-B

Yamine Aït-Ameur[1(✉)], Idir Ait-Sadoune[2], P. Casteran[3], Paul Gibson[4], K. Hacid[1], S. Kherroubi[5], Dominique Méry[5], L. Mohand-Oussaid[2], Neeraj K. Singh[1], and Laurent Voisin[6]

[1] INP-ENSEEIHT/IRIT, Université de Toulouse, Toulouse, France
yamine@enseeiht.fr
[2] CentraleSupelec/LRI/Paris Saclay University, Compus de Paris-Saclay, Paris, France
[3] LABRI, Université de Bordeaux, Bordeaux, France
[4] Télécom Sud Paris, Évry, France
[5] LORIA, Telecom Nancy, Université de Lorraine, Nancy, France
[6] Systerel, Aix-En-Provence, France

1 Context

Although several authors like Zave and Jackson [11,17], Bjørner [5], Van Lamsweerde [13] have drawn the attention of system designers on the necessity to handle domain knowledge, while designing systems, it is still a major concern nowadays. The IMPEX[1] project, funded by the French ANR national research agency, addresses the problem of making explicit domain knowledge in formal system developments using refinement and proof based formal methods. It advocates the use and formalisation of ontologies as models for domain knowledge. The Event-B [1] modelling technique has shown its usefulness to support the various developments. In this paper, we briefly describe the approach [3] and the case studies developed in the context of this project.

2 Formal Ontologies as Domain Models

Gruber defines an ontology as *an explicit specification of a conceptualization* [7]. Another definition relies on the notion of dictionary. [12] considers a domain ontology as a *formal and consensual dictionary of categories and properties of entities of a domain and the relationships that hold among them.* Here, an entity represents any concept belonging to the considered domain. *Dictionary* entails

This work was supported by grant from the French national research agency - ANR ANR-13-INSE-0001 (The IMPEX Project http://impex.gforge.inria.fr (or http://impex.loria.fr)).

[1] http://impex.gforge.inria.fr (or http://impex.loria.fr).

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 425–430, 2018.
https://doi.org/10.1007/978-3-319-91271-4_35

two major concepts. First, it makes explicit the existence, through a constructive definition or declaration, of entities in the domain under consideration and second any entity or relationship described in this ontology is directly referenceable independently of other entities or relationships. Reference is carried by a symbol defining an identifier. Each *description* of each entity or relationship is formally *stated* using an *ontology modelling language* equipped with a formal semantics. It allows automatic reasoning and consistency checking. Event-B is a good candidate to support such formal descriptions.

2.1 Ontologies as Theories in Event-B [2]

In the context of IMPEX, we have identified two approaches to define ontologies as formal theories. They use two different modelling processes: shallow and deep.

- The **shallow modelling** approach consists in formalising the ontology concepts directly in the target modelling language without keeping trace of the structure of the ontology modelling language concepts [16]. One way to integrate the ontology concepts into a specific formal method development process is to express the ontology modelling languages constructs into the target formal language by means of transformation rules. In our case, a shallow modelling approach consists in encoding the ontology concepts (classes, properties, ...) directly in an Event-B context using abstract sets, constants and axioms.
- The **deep modelling** approach consists in formalising the ontology concepts together with the concepts of the modelling language that were used to define the ontology concepts [8,9]. Here, ontologies are defined as instances of ontology models. Two steps are required. First, an ontology model is formalised and then ontologies are defined as specific models corresponding to the defined ontology model. In our approach, we consider that both ontology modelling concepts and ontologies are explicitly modelled. These concepts have been formalised in Event-B. More precisely, as we consider ontologies as theories, we have used Event-B contexts to formalise such concepts.

The OntoEventB Plug-In. The OntoEventB plug-in[2] [16] has been developed to automatically support the translation of ontologies models, described using ontology description languages such as OWL [4] or PLIB [10], into Event-B contexts. It takes as input an ontology description file and generates, according to the selected approach (shallow or deep), the corresponding Event-B contexts. The OntoEventB plug-in is integrated it into Rodin. To use the OntoEventB plug-in in your Rodin platform instance, you must install the plug-in by using the Install New Software menu item.

2.2 The IMPEX Approach [3]

As mentioned above, ontologies have been chosen as a framework for modelling domain knowledge. Additionally, annotation relationships have been set up to

[2] OntoEventB update site: http://wdi.supelec.fr/OntoEventB-update-site/.

link system models concepts to its semantic description unit provided by the ontologies. In this way, it becomes possible to consider properties of the domain in system models. As a consequence, domain knowledge is made explicit in such system models. Domain properties together with reasoning capabilities become accessible from the system models.

2.3 *A Priori* or *A Posteriori* Handling of Domain Models

When domain models are formalised, it is possible to take into account the expressed domain concepts and properties in system models. Two different situations depending on the availability of the considered system models may occur.

- In the **a priori** case, we consider that domain models are available before the system models are produced. In this case, when designing system models, domain concepts (axioms or theorems) are borrowed to define the system model concepts as being *subsumed* by domain concepts. Note that the subsumption mechanism available in ontology-based models allows a designer to borrow only relevant concepts and properties from an ontology. The two first examples of Sect. 3 report on an a priori approach.
- The **a posteriori** case occurs when system models are already designed. In this case, these models are *re-factored* using explicit references to ontology concepts. This mechanism uses specific references, based on explicit mapping definitions, to borrow ontology concepts inside the re-factored models. The last example of Sect. 3 reports on an a posteriori approach.

3 Case Studies

In this section, we briefly present some experiments we have conducted using both a priori and a posteriori approaches to explicitly handle domain knowledge.

Embedded Systems [14]**.** The embedded system under consideration is a nose gear velocity update function. It is responsible of estimating the velocity of an aircraft while moving on the ground. Hence, it is suitable to highlight the need for identifying and integrating explicit semantics. A single explicit, requirement is defined. `EXFUN-1`: *While the aircraft is on the ground, the estimated velocity shall be within 3* km/hr *of the true velocity of the aircraft at some moment within the past 3* s. Along with `EXFUN-1`, we have systematically extracted several other implicit/derived requirements from this requirements description. An a priori model [14] of the Nose Gear Velocity system has been developed with six Event-B refinements. The second refinement introduces the interrupt service routine (ISR) responsible for updating the rotation counter and recording the service request time. As per the system description, *NGRotations* counter is a 16-bit counter. However, it is observed that *NGRotations* counter can be modelled as an *always incrementing* counter – taking into account possible diameters of an aircraft wheel, a 16-bit rotations counter is more than enough for the

longest existing runway. The explanations are based on the strong requirement to avoid overflow during execution of the system: $\pi * WHEEL\_DIAMETER * NGRotations \leq LONGEST\_WORLD\_RUNWAY$ or equivalently the maximal distance of the aircraft is less than the longest world runway which is *Qamdo Bamda Airport, China, 5,500* m following the Internet. It means that the condition $NGRotations \leq 2^{15} - 1$ and a 16-bit counter is largely sufficient. The validation of the choice of the size is based on a knowledge borrowed from an ontology. The proof of absence of overflow is then obtained automatically as long as the prover is able to handle the fact.

Electronic Voting Systems [6]**.** In *Applying a Dependency Mechanism for Voting Protocol Models Using Event-B*' [6], the case study presents a method for re-using general concepts from an e-voting domain model in the formal development of specific systems within the same domain. The approach is refinement-based and thus the development is a sequence of models, moving from the abstract to the concrete. By following different refinement sequences, a family of e-voting systems can be produced that share common properties from the e-voting domain. This is illustrated in the study by a development which branches (through refinement) into two different e-voting system families. The e-voting domain knowledge is explicitly represented in the Event-B contexts. The first Event-B context introduces only the elements necessary to build an initial abstract machine for the phase of behavior associated with recording votes i.e.: sets, constants and static properties such as *Electors, Choices, Envelopes, Poll-Station, Representatives, Bulletins* .... As this abstract machine is refined it is necessary to extend the initial context with new conceptual elements from the e-voting domain. These correspond to specific features (increments of behavior) which the concrete system needs to offer; and they are added to the Event-B contexts as required.

Medical Systems [15]**.** Here, we describe the a posteriori approach for developing a medical protocol and we revisit the ECG interpretation protocol case study [15]. Our aim is to use domain knowledge explicitly in the development of a medical protocol including two different models: *domain model* and *system model*. The domain model describes the common medical concepts, relationships, properties and axioms related to biomedical, disease, diagnosis, anatomy, clinical procedure using several available medical ontologies (e.g. GALAN, OpenCyc, WordNet, UMLS, SNOMED-CT, FMA and Gene Ontology). The system model describes the stepwise clinical procedure for assessing the medical protocol. The Event-B models both domain and system. Domain knowledge is described in an Event-B context using ontology relations to capture the clinical procedure of the medical protocol. Note that both the domain model and system model are linked through *annotation*, in which the system model uses all axioms and theorems expressed in the domain ontology model. This model combination allows us to verify new properties related to domain knowledge within the enriched design medical protocol. We have used the ECG medical protocol for developing refinement-based formal models to systematically analyse whether the

formalisation complies with certain medically relevant protocol properties. Moreover, this approach allows us to identify possible anomalies and to improve the quality of the medical protocol.

4 Conclusion

Our results show that it is possible to handle formally and explicitly domain knowledge in formal system developments with Event-B and the Rodin platform. Ontologies have been formalised within Event-B as theories and a Rodin plug-in has been developed for this purpose. Moreover, the a priori and a posteriori scenarios have been set up to define system model annotations. For the future, we plan to investigate two research directions. The first one relates to the study of the properties of the annotation relationships, possibly modelled as Galois connections, and the second one concerns the study of domain knowledge for dynamic concepts like actions, events or transitions. Finally, experimenting the proposed approach in other application engineering areas is also planned.

References

1. Abrial, J.R.: Modeling in Event-B - System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Ait-Ameur, Y., Gibson, J.P., Méry, D.: On implicit and explicit semantics: integration issues in proof-based development of systems. In: Margaria, T., Steffen, B. (eds.) ISoLA 2014. LNCS, vol. 8803, pp. 604–618. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45231-8_50
3. Aït Ameur, Y., Méry, D.: Making explicit domain knowledge in formal system development. Sci. Comput. Program. **121**, 100–127 (2016)
4. Bechhofer, S., Van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D., Patel-Schneider, P., Stein, L., et al.: OWL Web Ontology Language Reference. W3C Recommendation (2004)
5. Bjørner, D.: Manifest domains: analysis and description. Form. Asp. Comput. **29**(2), 175–225 (2017)
6. Gibson, J.P., Kherroubi, S., Méry, D.: Applying a dependency mechanism for voting protocol models using event-B. In: Bouajjani, A., Silva, A. (eds.) FORTE 2017. LNCS, vol. 10321, pp. 124–138. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-60225-7_9
7. Gruber, T.R.: A translation approach to portable ontology specifications. Knowl. Acquis. **5**(2), 199–220 (1993)
8. Hacid, K., Ait-Ameur, Y.: Strengthening MDE and formal design models by references to domain ontologies. A model annotation based approach. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 340–357. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47166-2_24
9. Hacid, K., Ait-Ameur, Y.: Annotation of engineering models by references to domain ontologies. In: Bellatreche, L., Pastor, Ó., Almendros Jiménez, J.M., Aït-Ameur, Y. (eds.) MEDI 2016. LNCS, vol. 9893, pp. 234–244. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45547-1_19

10. ISO: Industrial automation systems and integration - parts library - part 42: Description methodology: Methodology for structuring parts families. ISO ISO13584-42, International Organization for Standardization, Geneva, Switzerland (1998)
11. Jackson, M., Zave, P.: Domain descriptions. In: Proceedings of IEEE International Symposium on Requirements Engineering, RE 1993, San Diego, California, USA, 4–6 January 1993, pp. 56–64 (1993)
12. Jean, S., Pierra, G., Ait-Ameur, Y.: Domain ontologies: a database-oriented analysis. In: Filipe, J., Cordeiro, J., Pedrosa, V. (eds.) Web Information Systems and Technologies. LNBIP, vol. 1, pp. 238–254. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74063-6_19
13. van Lamsweerde, A.: Requirements engineering in the year 00: a research perspective. In: Proceedings of the 22nd International Conference on Software Engineering, ICSE 2000, Limerick Ireland, 4–11 June 2000, pp. 5–19. ACM (2000)
14. Méry, D., Sawant, R., Tarasyuk, A.: Integrating domain-based features into event-B: a nose gear velocity case study. In: Bellatreche, L., Manolopoulos, Y. (eds.) MEDI 2015. LNCS, vol. 9344, pp. 89–102. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23781-7_8
15. Méry, D., Singh, N.K.: Medical protocol diagnosis using formal methods. In: Liu, Z., Wassyng, A. (eds.) FHIES 2011. LNCS, vol. 7151, pp. 1–20. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32355-3_1
16. Mohand-Oussaid, L., Ait-Sadoune, I.: Formal modelling of domain constraints in event-B. In: Ouhammou, Y., Ivanovic, M., Abelló, A., Bellatreche, L. (eds.) MEDI 2017. LNCS, vol. 10563, pp. 153–166. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66854-3_12
17. Zave, P., Jackson, M.: Four dark corners of requirements engineering. ACM Trans. Softw. Eng. Methodol. **6**(1), 1–30 (1997)

Author Index