Proceedings of the

11th International Workshop on Automated Verification of Critical Systems

Newcastle
September 12 - 14, 2011

Preface

AVOCS, the workshop on Automated Verification of Critical Systems, is an annual meeting that brings together researchers and practitioners to exchange new results on tools and techniques relating to the verification of critical systems. Topics of interest include all aspects of automated verification, including model checking, theorem proving, abstract interpretation and refinement; application areas include various types of critical systems (safety-critical, security-critical, business-critical, performance-critical, etc.). Contributions that describe different techniques or industrial case studies are encouraged.

This volume contains the pre-proceedings of the 11th workshop on Automated Verification of Critical Systems that was hosted by Newcastle University and took place during September 1214, 2011 in Newcastle upon Tyne, UK.

Previous AVOCS workshops were held at the University of Oxford (2001 and 2007), the University of Birmingham (2002), the University of Southampton (2003), The Royal Society in London (2004), the University of Warwick (2005), LORIA, Nancy (2006), the University of Glasgow (2008), Gregynog (organized by Swansea University) and Heinrich-Heine-Universität Düsseldorf (2010). AVOCS 2012 will take place in Bamberg, Germany.

AVOCS 2011 received 18 submissions (with authors from 13 countries) for Full Papers, out of which 12 papers were selected for presentation at the workshop. Furthermore, AVOCS received 11 submissions for Short Contributions out of which 8 were accepted for presentation. The selection process was carried out by the Program Committee, taking into account the originality, quality, and relevance of the material presented in each submission. The selected preliminary Papers are included in this volume, together with the contributions from the invited speakers Janet Barnes and Tom Maibaum. All full papers will subsequently appear in an Electronic Communications of EASST.

We wish to thank all authors who submitted their papers to AVOCS 2011, Jodi Hossbach for help with workshop organization, the Program Committee for its excellent work and the reviewers who supported the Program Committee in the evaluation and selection process.

We are grateful to the School of Computing Science at Newcastle University for hosting the event and thank CSR, Formal Methods Europe and Microsoft for sponsoring AVOCS 2011. We also gratefully acknowledge the use of EasyChair, the conference management system developed by Andrei Voronkov.

Jens Bendisposto
Cliff Jones
Michael Leuschel
Alexander Romanovsky

AVOCS 2011 Program Committee:

Jens Bendisposto (co-chair)
Antonio Casimiro
Michael Goldsmith
Ian Hayes
Cliff Jones (co-chair)
Michael Leuschel (co-chair)
Felix Loesch
Gerald Luettgen
Ursula Martin
Stefan Merz
Alice Miller
Markus Roggenbach
Alexander Romanovsky (co-chair)
Thomas Santen
Sebastian Wieczorek
Jim Woodcock

AVOCS 2011 Referees:

Names will be published in the final EASST Proceedings

Content

Janet Barnes
Experiences in the Industrial use of Formal Methods

Franz Weitl, Shin Nakajima
Integrated Model Checking of Static Structure and Dynamic Behavior using Temporal Description Logics

Marco Bozzano, Alessandro Cimatti, Oleg Lisagor, Cristian Mattarei, Sergio Mover, Marco Roveri, Stefano Tonetta
Symbolic Model Checking and Safety Assessment of Altarica models

Michael Jastram, Stefan Hallerstede, Lukas Ladenberger
Mixing Formal and Informal Model Elements for Tracing Requirements

Mohammad Reza Sarshogh, Michael Butler
Specification and refinement of discrete timing properties in Event-B

Nicolas Chausse, Helen Xu, Juergen Dingel, Karen Rudie
Combining Model Checking and Discrete-Event Supervisor Synthesis

Pieter Philippaerts, Frederic Vogels, Jan Smans, Bart Jacobs, Frank Piessens
The Belgian Electronic Identity Card: a Verification Case Study

Qiuzi Lu, Tianhua Xu, Tao Tang, Haifeng Wang, Yan Cao, Gengqin Chen
A Visualization Framework for the Modeling and Formal Analysis of a Computer Based Interlocking System

Thai Son Hoang, Alexei Iliasov, Renato A Silva, Wei Wei
A Survey on Event-B Decomposition

Sabina Akhtar, Stephan Merz
Partial-Order Reduction for Verifying PLUSCAL-2 Algorithms

Sanaz Yeganefard, Michael Butler
Structuring Functional Requirements of Control Systems to Facilitate Refinement-based Formalisation

Xiang Gan, Jori Dubrovin, Keijo Heljanko
A Symbolic Model Checking Approach to Verifying Satellite Onboard Software

Michael Fisher
Verifying Autonomous Systems

Lukas Ladenberger, Aryldo G Russo Jr.
Towards an automatic formal model generation and verification derived from a graphical model

Yassin Chkouri, Jose Esteves, Elie Soubiran
Designing synchronous to asynchronous model translations for interlocking systems verification

Brijesh Dongol, Ian J. Hayes
Approximating idealised real-time specifications using time bands

Christophe Ponsard, Jean-Christophe Deprez, Renaud De Landtsheer
Is my Formal Method Tool Ready for the Industry?

Jan Tobias Mühlberg and Leo Freitas
Verifying FreeRTOS: from requirements to binary code

Marc Dragon, Andy Gimblett, Markus Roggenbach
A Simulator for Timed CSP

James Sharp, Helen Treharne and Steve Schneider
Assessing the Applicability of SVA in Analysing VHDL Models

Alexei Iliasov
Generation of certifiably correct programs from formal models

Experiences in the Industrial use of Formal Methods

Janet Barnes

janet.barnes@altran-praxis.com, http://www.altran-praxis.com/
Altran Praxis Ltd, 20 Manvers Street, Bath, UK.

Abstract: Altran Praxis has used formal methods within its high integrity development approach, Correctness by Construction (CbyC), for a number of years. The Tokeneer ID Station (TIS) developed for the US National Security Agency (NSA) is one example of a development using formal methods and the CbyC approach. This project used a number of rigorous techniques including formalisation of the specification using the Z Notation, refinement of the specification to a formal design, software development in SPARK with proof of absence of run-time errors of the software and proof of system properties. The project has stood up well to the intense scrutiny it has been subject to since it became available to the wider community in 2008, with only five errors being found. Despite the general success of the approach there are challenges to using formal methods in an industrial context. By looking at a number of key properties that affect the success of deployment of tools and techniques in industry we attempt to put the challenges of industrial deployment of formal methods into perspective.

Keywords: Correctness by Construction, Formal Methods, SPARK, Tokeneer, Z

1 Introduction

The application of formal methods to the development of software has long been considered by industry as niche; only applicable to the development of core functions in particularly critical domains, where safety or security is paramount. Industry in general perceives the application of formal methods to be prohibitive for a number of reasons: cost, familiarity and maturity often being cited [Hal90].

Altran Praxis has applied formal methods in a number of its development projects [Hal96, HC02, KHCP00, TIS]. This paper looks at the way that Altran Praxis approaches software development via its Correctness by Construction approach [Ame06], considering how formal methods support the fundamental goals of the approach. It then explores the Tokeneer project as an example of a CbyC implementation where formal methods were adopted at every point in the lifecycle. Taking the view of an experienced industrial user of formal methods this paper takes a critical look at some of the criteria that impact the actual and perceived success of the adoption of formal methods. In conclusion, this paper questions whether industry is in a position to drop long held prejudices that Formal Methods are too challenging to use in practice and considers what changes are needed to fully overcome such prejudices.

2 Correctness by Construction

Over 20 years Altran Praxis has distilled the essence of best practice, captured from observation and experiences, into a principle of software development referred to as Correctness by Construction (CbyC). The key philosophy of CbyC is to avoid the introduction of errors; but where errors are injected, to find and remove them as early as possible; and to gather certification evidence efficiently as a natural by-product of the process.

2.1 Applying Correctness by Construction

Correctness by Construction does not prescribe particular tools or techniques in order to achieve its aims. However, it does propose a number of characteristics to be applied across the development lifecycle.

- **Use unambiguous notations.** Ambiguity makes it difficult to determine whether or not errors exist and misinterpretation is a source of error introduction. Using a notation that has a well defined and well understood semantics removes ambiguity. Such notations often benefit from tool support, which can assist in verification.

- **Take small steps.** By taking small semantic steps between stages of the lifecycle it is easy to demonstrate that one development stage has been correctly refined from its predecessor.

- **Use appropriate notations.** Accept that a given notation may be powerful at expressing certain system properties but clumsy for expressing others. The aim is to use notations that allow the system properties or behaviour to be expressed simply. Don't attempt to use a single notation if this results in key system properties being difficult to express. Awkward expressions can be difficult to interpret or verify. Similarly, use the most appropriate verification techniques at each stage. Expect the outputs at each stage in the lifecycle to be clear and simple to understand.

- **Don't repeat information.** Each stage of the lifecycle should have a well defined purpose and focus on the new detail being introduced rather than repeat information. It is then clear what information has been introduced and what needs to be verified — rather than wasting energy verifying that information has been correctly copied from one source to another. Duplication can also be expensive during maintenance as it may become inconsistent and thus a source of error and confusion.

- **Check each stage before progressing.** Each design step should be verified as soon as possible to eliminate errors introduced in that stage. Effective reviews are crucial; reviews should clearly identify what an artefact is being reviewed against and the purpose of the review. Where review checks can be automated — such as coding style checks — then tool support should be used early.

- **Justify decisions.** Document the justifications for why design decisions were made, why they are appropriate, and any arguments demonstrating correctness of the decision. Such justifications support future analysis — especially in the event of implementing changes to

a system, but more importantly the process of documenting what you do is highly effective at driving out errors during development.

- Solve difficult problems first. Manage development risks by solving difficult problems early. This also drives down the level of internal change that might otherwise be introduced if risks mature later.

Many of the approaches advocated here also contribute to the provision of strong verification evidence that, if collected appropriately, can contribute positively to the construction of a certification argument, demonstrating that the system has been built respecting safety or security needs. None of the concepts are new or radical; if anything it is the careful application of sound engineering practices using understood tools and techniques that has made this approach successful.

2.2 Using formal methods within the CbyC framework

The CbyC approach is particularly powerful when instantiated with formal methods and approaches. Formal methods have precise semantics and often have an associated language of reasoning that enables the user to unequivocally demonstrate the truth or otherwise of a property. Specification languages such as the Z Notation [Spi85] benefit from a richness of notation that allow the application to be described in terms of real world entities and relationships; Z supports both the concepts of refinement and encapsulation. In Z, data and operation refinement allow an abstract specification to be refined toward a concrete, executable realisation. Z's schema notation allows detail to be hidden except at the point of introduction and makes complex specifications manageable, giving focus to the aspects of interest at a given point in a specification and allowing the problem to be decomposed into small, manageable fragments. Notations such as CSP [Hoa85] are powerful for modelling and reasoning about concurrency problems, especially when used in conjunction with model checkers such as FDR [FDR]. SPARK is a subset of the Ada programming language enhanced with contracts that has a formal semantics and is supported by a suite of tools: the Examiner, Simplifier and Proof Checker, that allow conformance to language and program properties to be proven. All these notations (Z, CSP and SPARK) provide points in the development lifecycle prior to the production of object code, when there are artefacts with a clear semantics. This enables these artefacts, specification and design documents, or source code, to be formally verified, either as a refinement of a previous lifecycle phase, or more commonly, as possessing key properties.

Interestingly, many of the benefits of formal notations do not come from the application of verification techniques, tool supported or otherwise, but from the additional attention to detail imposed on the author when applying the techniques. Although tools can help to demonstrate (partial) completeness or correctness it is often before the point of application of such tools that benefits are first realised as the very act of expression within a formal notation causes the author to explore the problem domain with a logical mindset — thereby detecting and investigating incompleteness in the requirements early in the lifecycle.

Having said that, the ability to use tool support to automatically check properties of the system and even simulate aspects of the system under development is extremely powerful at detecting

early lifecycle errors and demonstrating properties of the final system to the customer or key stakeholders.

3 The Tokeneer ID Station Experiment

The aim of the Tokeneer ID Station (TIS) Experiment [TIS], commissioned by the US National Security Agency (NSA), was to determine whether it was possible to write software to the standards imposed by EAL5 of the Common Criteria [ISO99] in a cost effective manner.

The method by which the experiment was undertaken was for Altran Praxis to redevelop a well defined component of the existing Tokeneer System [RL98] using the CbyC approach applied using formal notations at every stage of the development lifecycle. Tokeneer was a system previously developed by the NSA as an unclassified demonstration of the use of smart cards and biometrics. CbyC was applied in the redevelopment of the core functions of one component of the Tokeneer system. The development was assessed against EAL5 of the Common Criteria to determine whether the approach achieved the necessary assurance evidence to certify a security system to EAL5. By monitoring the skills needed to perform each stage of the development approach and the effort involved it was also possible to establish whether the approach was cost effective.

The experiment was time boxed and some activities were not completed but an estimate of the cost to complete the activity was provided in all cases to allow the true cost of the approach to be determined.

3.1 The Tokeneer system

Tokeneer provides protection to secure information held on a network of workstations situated in a physically secure enclave. The Enrolment Station issues tokens to users. To do this it relies on a Certificate Authority (CA) to generate user ID Certificates and an Attribute Authority (AA) to generate attribute certificates containing clearance and privilege information and biometric information. The TIS provides protection to the enclave by checking whether the user is authorized to enter the enclave and adding a certificate to the user token that authorizes the user to operate on the workstations within the enclave. The workstations check the certificate added by the TIS station to determine whether the user is authorized to use the facilities it provides.

Once initialised, the TIS holds public keys for the CA and AA. The primary function of the TIS is controlling user entry. The entry process being as follows: the user presents a token to the TIS containing three certificates, the user ID certificate, a biometric certificate containing fingerprint data, and a privilege certificate containing the role and privileges held by the owner of the token; the TIS checks the validity of these certificates and ensures they are signed by known authorities. The user then presents their finger to a fingerprint reader and the TIS authenticates the user by comparing the biometric data on the token with a scan of the user's finger. If this data matches and the user privileges allow them access to the enclave then a further authentication certificate is added to the token, (this is a certificate of relatively short duration) and then unlocks the enclave door, permitting access. If at any point the TIS deems there to be a breach of security an alarm is raised. There are also a number of administrator functions that TIS offers to users

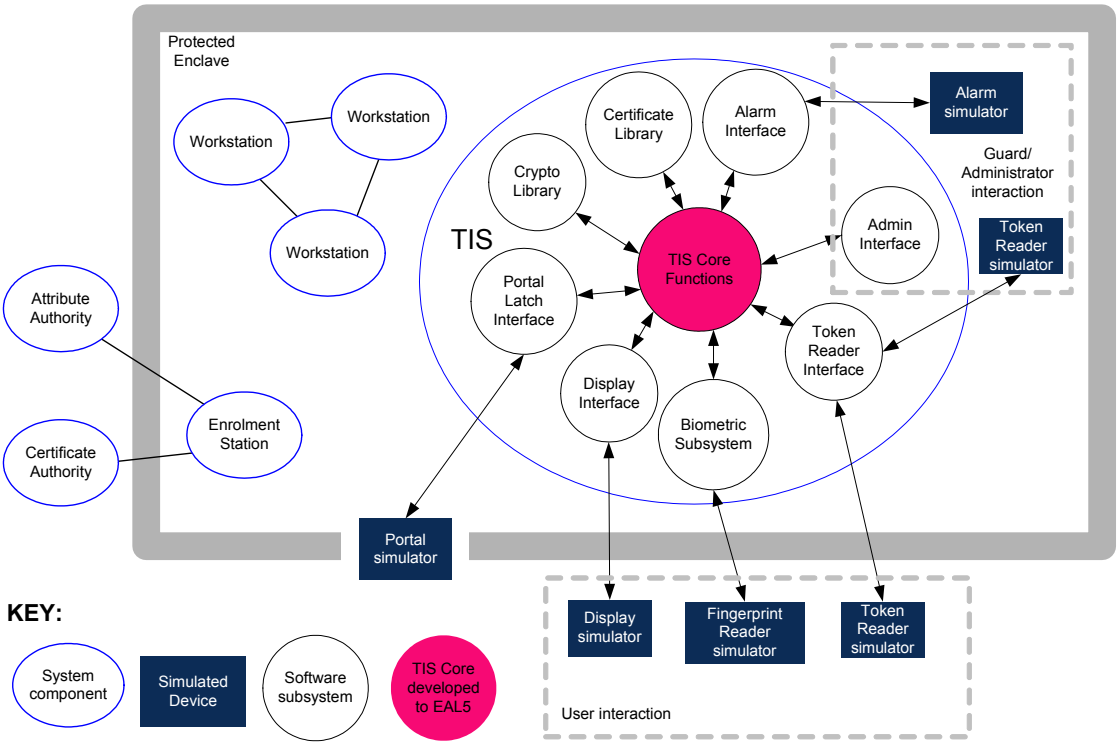


Figure 1: Overall Tokeneer System

with the appropriate roles. These are archiving log data of all transactions, overriding the door lock, and updating the configuration data which controls properties of the particular installation such as operating hours and security classification of the enclave.

Only the core functions of the TIS were developed using the full high integrity Correctness by Construction approach. Biometric and cryptographic components were simulated as were all external devices. The interfaces to external devices were developed using industry good practice but without the application of formal methods.

The customer introduced a change to the requirements part way through the design as a test of the robustness of the process. They added a requirement for the system to permit entry to the enclave to a user who had a valid authentication certificate on their token without needing to repeat the biometric checks.

3.2 The lifecycle

The TIS development lifecycle is depicted in Figure 2, it comprised six distinct phases: requirements analysis, security analysis, specification, design, implementation, and test.

Requirements analysis followed Altran Praxis' requirements engineering approach REVEAL [HRH01]. Key to this process was clear identification of the system boundary — important in this experiment was a clear understanding of boundaries between core functionality, to be developed to EAL5 criteria, supporting software, and functionality out of scope of the experiment

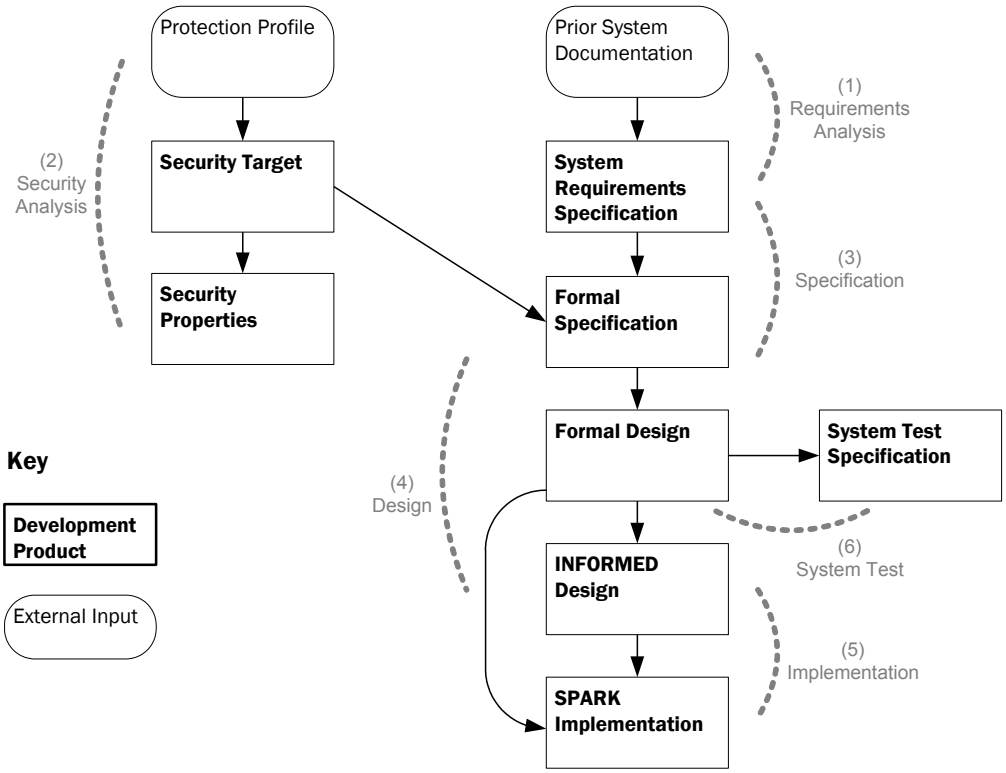


Figure 2: The development process

— for instance the original Tokeneer system additionally used a password in the authentication sequence. The context in which the TIS operated was also analysed giving a clear understanding of the TIS environment, such as the certificates generated externally and the way in which the door and its locking mechanism operated. Scenarios representing successful and erroneous interactions with TIS were developed with the customer to gain a clear understanding of the required behaviour of the system.

Security Analysis was performed orthogonally to the remainder of the development process, it responded to the supplied Protection Profile with a security target and development of the security properties required of the TIS. These activities focussed on the security needs of the system without consideration of the required user functionality. A key output of this activity was a Formal specification of the security properties developed using the Z notation.

Specification of the TIS took the form of a formal behavioural specification developed using the Z notation. The specification provides an abstract model of the system, focusing on interactions of the system with its real world interfaces, ignoring internal details. By developing a behavioural model of the system it was possible for the details of the proposed behaviour of the system to be presented early — before code production. With the help of customer review we were confident that we were planning to build the right system.

Design was divided into two components. The Formal design, again developed in Z, is a refinement of the specification introducing the internal details of how the system works — in

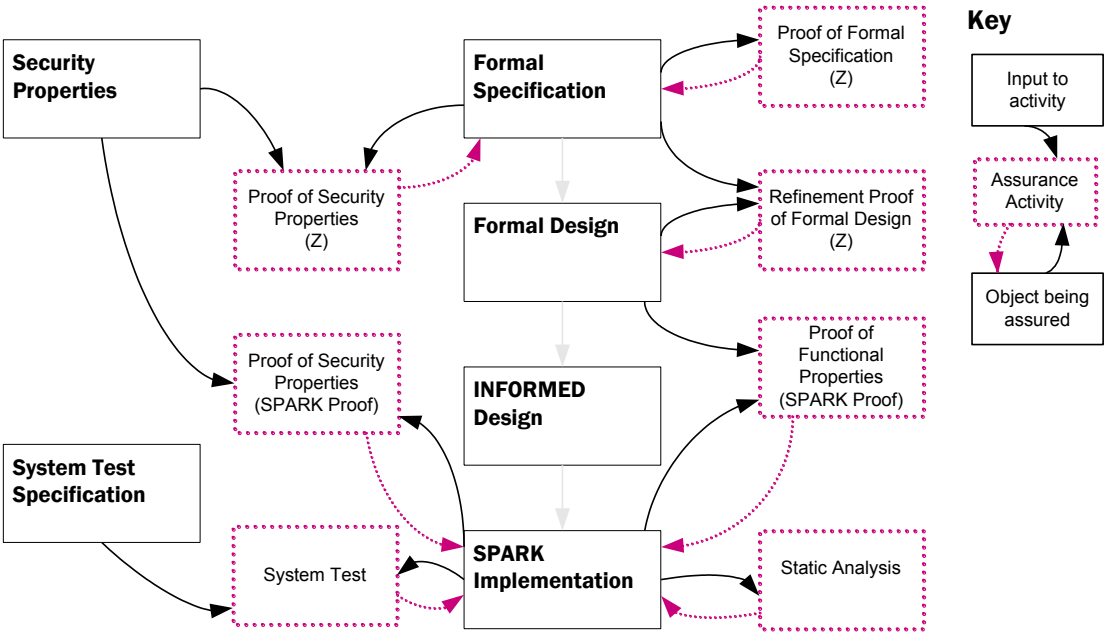


Figure 3: Assurance Activities

the case of TIS the design resolved some priority issues which led to the specification being potentially non-deterministic in its behaviour, additionally the details of logging and the structure of certificates as raw data streams were introduced.

The INFORMED design [Ame01] focused on developing a software architecture, it identifies implementation modules and the information flow between them, it apportions each component of the formal design to the program module that implements that component, it also covers file structures and constraints not covered formally.

Implementation of the core TIS is written in SPARK [Bar03] using both flow and proof contracts. Data and information flow analysis and proof of absence of run-time errors were done before code review and compilation. Implementation from the formal design was relatively straightforward — with simple mappings between predicates and code fragments.

Testing was limited to system testing, which was based on achieving a basic level of coverage of all the schemas in the Formal Design. Ordinarily this would have been undertaken with code coverage metrics being collected to ensure an adequate coverage of the source code had been achieved. The Formal Design provided a very clear definition of the required behaviour of the system on which to base tests.

The aspects of the implementation process that were more radical were the verification activities. These focused on verifying the correctness of each lifecycle phase early. Further, by using consistent Formal notations for the Security Properties, the Formal Specification and the Formal Design, it was possible to prove that the Formal Specification adhered to the Security Properties and that the Formal Design was a refinement of the Specification. The other area where proof was applied was in the code, in addition to proving the absence of run-time errors, some of the security properties were expressed as SPARK proof contracts, the code was then proven to con-

form to these properties. Figure 3 demonstrates the assurance activities undertaken, excluding review which occurs as each component is complete. Each assurance activity was undertaken as soon as all the inputs to the activity were complete and before proceeding to the next lifecycle activity allowing errors introduced at each phase to be driven out by more than just review scrutiny.

3.3 Results and subsequent scrutiny

The key outputs of the project were a 100 page behavioural Z specification; the core software comprised 9,939 lines of code with 6,036 lines of flow contracts and 1999 lines of proof contracts. The supporting software, written in Ada95, comprised 3,697 lines of code. The entire development required 260-man days, provided by three people working part time over 9 months. The productivity over the project as a whole was 38 lines of code per day, with the coding rate of the core software working out at 208 lines of code per day against a rate of 182 for the support software. Analysis [BC03] showed that the process had been developed to EAL5 and in some areas had exceeded the requirements of EAL5 particularly in the levels of formalism applied.

The whole project archive was donated to the Verified Software Repository in 2008 [TIS] and has subsequently been subjected to wide ranging scrutiny. To date, five defects have been found in the core software. These defects are fully documented in [WAC10] and were found through a combination of application of improved tools and critical review. Two of these are completely benign in the code as it stands, the other three represent potential insecurities in the software. Of these three, one would have been detected by the latest variants of the toolsets used on the project — assuming the most demanding levels of checks were selected, a further would have been detected by undertaking program proof of the remaining security properties and the last could have been detected following scrutiny of code coverage results.

These results are encouraging and suggest that, with the latest tools, the application of formal methods supports the development of high quality software suitable for critical domains. Of course, we can never be certain that every fault has been found but the level and variety of external scrutiny to date gives considerable confidence in the state of the Tokeneer core software.

Further, the results presented in [MW10] show that following extensive review of the whole code base and the use of CodePeer the most significant errors were found in the support software. This was written by the same engineers as the core software, but without the application of formal techniques such as SPARK and development from a formally specified design, giving a fair indication that the development process used on the core software did indeed produce higher quality software.

4 Challenges using formal methods in industry

It is clear from the results of Tokeneer that the application of formal methods can result in the efficient delivery of high quality software. However, the uptake of many of the approaches on an industrial scale has been limited. From a technical and commercial viewpoint this seems like a missed opportunity on the part of industry in general. To try and understand the reasons behind the apparent lack of industrial enthusiasm, the remainder of the paper seeks to establish

more abstract qualities of development and verification approaches which impact their successful adoption, taking as read that any formal approach will offer unambiguous notations and the opportunity for analysis of the system.

We propose that the following list is a representative, but not necessarily exhaustive, characterisation of desirable properties of any development notation, regardless of whether it constitutes a formal notation:

- scalable,
- notation approachable to all stakeholders,
- expressive (ease of capturing the problem),
- tool supported.

It is often the ability to satisfy these demands that influences the adoption of an approach, rather than the more obvious technical questions of whether the method or tools fulfil the goals of expressing the desired functionality and contributing towards a correct software implementation. In the following sections we consider these attributes in more detail and measure the success of the notations used in the development of Tokeneer against these criteria.

4.1 Scalable

This is a property that is well understood as being key to industrial applicability. There are two aspects to scalability, first whether the notation allows large problems to be expressed in a way that is still manageable to the authors and consumers of the artefact; secondly whether tooling associated with the notation is able to perform efficiently when processing representations of large problems. We look in more detail at the former problem. The problem of scalability is constant across the development lifecycle — a system that is complex is likely to have many requirements, a large design and a considerable code base. Effective notations offer encapsulation and modularisation which aid the presentation of information in manageable portions.

Tokeneer is small as industrial applications go. It has Altran Praxis' smallest Z specification covering full functional behaviour. Altran Praxis' most recent Z specification contains over 3000 schemas, the final developed system being of the order of 150KLOC of SPARK Ada demonstrating that the Z notation and SPARK are scalable. Larger SPARK developments have been undertaken outside of Altran Praxis.

In Z we can decompose the system state into logically cohesive components, developing structure within the system data model and allowing system behaviour to be decomposed into operations acting on a particular partition of the state. Overall system behaviour is achieved through composition of partial behaviours. This allows the participants of the specification to be able to contemplate the system using a divide and conquer approach, only ever needing to consider a small fragment in detail at any one time.

SPARK similarly allows the system to be analysed in fragments — making use of a rich package specification to allow components to be analyzed in isolation. Data abstraction also allows detail to be hidden from public contracts of a package and prevents contract proliferation.

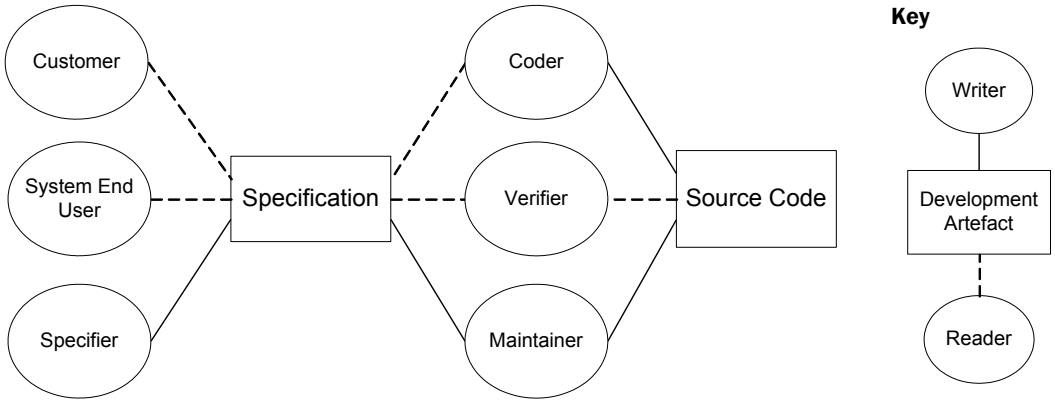


Figure 4: Artefacts and stakeholders

4.2 Approachable notation

A notation is considered approachable if it is usable by all those stakeholders who need to interact with it. The usability of a notation will depend on the familiarity of the notation — this familiarity can be acquired through use, although the ability to make such a transition to a notation will often be influenced by the underlying skills of the individual who needs to acquire the notation. To this end there are two things that influence the success of the notation to be approachable: the diversity of stakeholders who need to be involved with the notation and the difference between the notation and the languages already familiar to the stakeholders.

A system specification is likely to have a large number of stakeholders with diverse expertise, from end-users and customers to coders. The end-user and the customer are unlikely to be experts in the specification notation, although for the specification to be truly effective both the customer and the end-user will need to understand the system that is being specified — by doing so they will gain confidence that the system that is about to be built will offer the desired functionality. In the case of Tokeneer we were privileged to have a Z expert as our customer. However, where the customer and end-users are not experts in the notation we introduce a potential language barrier at a crucial early stage in the development lifecycle. It is at the point of developing the specification that we are first likely to uncover omissions from the requirements, details of corner case behaviours that the requirements don't define. Finding and resolving these at the point of specifying the system is highly efficient and reduces surprises in the system behaviour and increases the likelihood that we construct the desired product.

There are that can be employed to reduce the language barrier — Altran Praxis has a policy of supplying a high level of English language description alongside the formal notation although reading just the (imprecise) English text will loose the value of the precise formal notation. Provision of training can be effective where there is not too great a disparity between customer, end-user skills and the selected notation, However, training requires a high level of customer commitment and can be problematic where the customer or end-user representation is large. Animation and scenario modelling are powerful as they allow demonstration of features of the system based on the specification, however a large specification can result in state space explosions and exploring all cases exhibited by the animation could be prohibitive in terms of time.

Even relatively simple aspects such as the documentation environment can prove significant hurdles in terms of familiarity of notation. For example the predominant text preparation method for Z is via the use of LaTeXwhile the industrial norm for document production is Microsoft Word or the like. In recent years tools have been developed to support the direct incorporation of Z paragraphs into Word documents [Hal08] thereby simplifying the process of generating documentation which incorporates textual descriptions, diagrams and formal paragraphs.

It is attractive from a commercial supplier perspective to obtain agreement to the specification and deliver to the specification; however this is only a practical proposition where the customer is truly engaged in the notation. A more realistic goal is for the specification to be viewed as an artefact internal to the development which allows pertinent questions to be asked of the customer or end-user; the questions being asked in a language familiar to the customer. Taking this approach we need to accept that it is highly possible that when producing a specification there will be differences of interpretation and that these differences may not be realised until the system is validated — this feels like a lost opportunity although it is no less powerful than using informal or semi-structured notations to deliver the system specification — where the notation would be insufficiently precise to detect many of the points of clarification that are uncovered when writing a formal specification.

Altran Praxis' experience with the use of Z as a specification language is that Z reading skills are easily acquired by coders and verifiers alike, suggesting that software engineers typically possess the necessary logical deductive skills appropriate to interpreting the Z notation.

By contrast the number of stakeholders involved with the source code, who might be required to understand notations associated with formal code analysis or proof are fewer. Furthermore, it is often possible to express the proof language in a semantics which represents a modest extension from the code semantics. There is a small semantic gap between the SPARK language and Ada making it a relatively painless transition for an Ada programmer to be able to correctly express and interpret SPARK contracts and the verification conditions generated by the associated tools.

4.3 Expressiveness

One of the fundamental characteristics of the CbyC approach to software development is to take small steps between lifecycle stages so that at no point is there a large semantic gap during the refinement from specification to code. Taking this idea back a stage further it is important to be able to describe the system in its real-world context as easily as possible in the specification. Often to achieve this we need to express complex properties of the system's interaction with the environment. To this end a highly expressive notation can be extremely effective, allowing a wide range of concepts to be captured without significant overhead of constructing building blocks that take the specifier's attention away form the problem domain and the task of expressing the behaviour of the system within that domain. Formal refinement techniques can then be used to transition from an abstract representation toward a design that can be simply implemented. However, the richer the language the harder it is to become an expert in the full language — this seems to be a true dilemma, not only to humans as users of the notation but to the provision of tool support to provide automatic verification.

Our experience in the development of industrial scale specifications is that the use of Z as a

highly expressive notation is extremely powerful in allowing the engineer to focus on capturing the correct description of the system's behaviour, without excessive distractions from having to find a way of encoding the relationships with a restrictive language.

Expressiveness becomes less of a critical characteristic of the notation as we move through the lifecycle toward code. Industrially used programming languages such as Ada and C and their language subsets such as SPARK Ada and MISRA C are sufficiently expressive to implement the system.

4.4 Tool support

One of the benefits of using formal notations is that they have sound semantics which make them amenable to tool supported verification, from the most basic syntax checking to automated or semi-automated proof. Without the underlying semantic definition it is difficult to make anything but basic checks on an artefact.

Automated verification is a highly powerful way of finding errors and inconsistencies in the outputs of the development lifecycle. Furthermore it is typically repeatable and should not be subject to human error. However, for automated verification to be cost effective, that is detect a sufficiently high density of faults in a sufficiently short period of time, there are a number of characteristics that need to be exhibited by the automated verification technique. The tools that support the technique need to be

- fast,
- trustworthy and supported,
- easily interpretable.

4.4.1 Speed

An effective verification tool must be sufficiently fast that the checks to be run repeatedly in a cycle of develop – check – correct – check. The speed of a tool is highly dependant on the modularity of the notation; the class of checks being undertaken and the amount of the system that the tool needs to interpret to enable it to perform its analysis. The speed of the Examiner is achieved by the analysis of one package body only being reliant on the enriched specifications of the other packages that are used by the package under analysis. The fuzz type checker [Spi] is fast due to the limited scope of its analysis. Both are sufficiently fast that they can be repeatedly run during development to ensure that the development output is being constructed correctly. Any checks that need to be run overnight cannot easily be used effectively as development is undertaken — although they can be used in the performance of final verification activities.

4.4.2 Correctness and Support

It is important to discuss correctness and support together as it is unlikely that any software product is completely fault free, but if support is readily available to handle faults found then the product can be considered fit for purpose. When a method and associated tools are selected for use on a project in industry the answers to the following questions will be fundamental to whether the tools are selected for use:

Can I get help in using the product?

Will the product be fixed promptly if I find a fault with it?

Will the product still be supported in 10 or 20 years time?

Will the product be considered appropriate by any certifying body?

If a development programme has chosen to include a tool in its development or verification strategy then training of personnel in the use of the tool and technology will be paramount, not knowing how to use a product to its best effect is expensive in time and a waste of the investment in the technology.

If a tool is found to be faulty in some respect then it is crucial for the development programme to either upgrade to a corrected version of the tool or fully understand the limitations otherwise there can be profound cost implications on the programme as a revised development or verification technique would need to be introduced. There is a widely held view that a product being open source means that it can be corrected, but this assumes that the source can be understood by the user. Even where the source for a tool is supplied there are significant costs and risks involved to anyone proposing modification to the tool.

Life expectancy of the tool suite is often of key concern to industrial developers. Many contracts include ongoing maintenance requirements and if the system is to be maintained then its development environment needs to be maintained and supported for the in service life of the software product. Although this is a risk with any tool, the risk is perceived to be greater where the tool is not itself available with a support contract.

Where the software under development is of a safety or security critical nature it is likely that a regulatory body will assess the processes, methods and tools used during development. Any tool where the output is used to gain verification credit will be expected to have an appropriate pedigree — either gained through a good history of use in the field, or by demonstration that the tool itself has been developed to a high standard.

4.4.3 Interpretation of output

Quality of the output of a verification tool dramatically impacts the time consumed analysing output and correcting inputs. Developments in tools to include hyperlinked renditions of the material analysed to aid navigation to the source of errors have been powerful at reducing analysis time. The Z Word tools [Hal08] do this to great effect allowing the user to run fuzz on the Word document and then jump from each error message to the source of the error in the Word document.

The level of false alerting of a tool can be crucial to its effectiveness, a tool that identifies a large number of potential problematic outcomes in the output will absorb a considerable amount of manpower in checking and justifying those cases that the tool could not provide a negative or affirmative outcome. One of the significant successes of the Examiner and Simplifier is the high percentage of verification conditions (VCs) generated through checking for absence of run-time errors that are automatically discharged. This makes the activity of checking the outstanding VCs manageable and has made the proof of absence of run-time errors in SPARK programs an option that is widely used.

5 Conclusion

Formal methods have a huge amount to offer industry in terms of providing unambiguous notations that are suited to formal verification that can in turn be automated. Many industrial standards for development of software at the highest integrity levels encourage the use of formal methods [ISO99, DEF97, EN 01] — to the point that it can be cheaper to conform to the standard by using a development approach that makes use of formal methods than relying on a test driven argument for certification. The results of the Tokeneer project are a clear demonstration that the application of formal methods is a cost effective route to the development of high integrity software. Despite this, the adoption of formal methods by industry is perceived as difficult. This paper has looked at some of the less technical aspects that influence decisions about the process by which software is developed and has considered why these aspects rather than the technical merits of the approach are likely to be significant barriers to acceptance of formal methods.

Of the four key industrial indicators for the acceptability of a general development notation considered in this paper, scalability and expressiveness are being addressed by formal methods. The approachability of the notation is more challenging where the notation becomes exposed to a wide range of stakeholders, so this indicator is most applicable to early lifecycle activities such as systems specification, where interaction with the customer or end user becomes necessary to establish the desired behaviour. A number of tactics have been explored that suggest that approachability of the notation can be addressed by careful choice of the manner of presentation.

This suggests that the most significant barrier to industrial acceptance is the availability of supported tools — there is a relative plethora of tools available open source that provide the desired levels of automation, however, this is insufficient. In an industrial context, the need for tool qualification, fitness for purpose arguments, training and ongoing support make the adoption of open source tools without support contracts too high a risk on exactly the classes of project that would most benefit from automated verification. To overcome this hurdle, formal methods tools need committed maintenance — this requires collaboration between industry and academia to place supported products in the marketplace at a price that allows adoption on both modest and large scale applications.

Acknowledgements: My gratitude goes to John Barnes, Rod Chapman and Neil White for their comments on the draft of this paper.

Bibliography

[Ame01] P. Amey. The INFORMED Design Method for SPARK. 2001. Available on request from Altran Praxis. http://www.altran-praxis.com/

[Ame06] P. Amey. Correctness by Construction. S.P8001.11.1. 2006. Available on request from Altran Praxis. http://www.altran-praxis.com/

[Bar03] J. G. P. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[BC03] J. E. Barnes, D. Cooper. EAL5 Demonstrator: Summary Report. S.P1229.81.1. Dec. 2003. in [TIS].

[DEF97] DEFSTAN 00-55 (Part 1). Requirements For Safety Related Software in Defence Equipment. Aug. 1997.

[EN 01] CENELEC BS EN 50128. Railway applications — Communications, signalling and processing systems — Software for railway control and protection systems. 2001.

[FDR] FDR2 refinement checker. Formal Systems (Europe) Ltd. http://www.fsel.com/

[Hal90] A. Hall. Seven Myths of Formal Methods. *IEEE Software* 7(5), 1990.

[Hal96] A. Hall. Using Formal Methods to Develop an ATC Information System. *IEEE Software* 13(2), 1996.

[Hal08] A. Hall. Integrating Z Into Large Projects: Tools and Techniques. In Börger et al. (eds.), *Short Papers of the ABZ 2008 Conference*. 2008.

[HC02] A. Hall, R. Chapman. Correctness by Construction: Developing a Commercial Secure System. *IEEE Software* 19(1), Jan. 2002.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[HRH01] J. Hammond, R. Rawlings, A. Hall. Will it Work? In *RE'01, 5th IEEE International Symposium on Requirements Engineering*. 2001.

[ISO99] ISO 15408. Common Criteria for Information Technology Security Evaluation. 1999. Version 2.1.

[KHCP00] S. King, J. Hammond, R. Chapman, A. Pryor. Is Proof More Cost Effective than Testing? *IEEE Transactions on Software Engineering* 26(8), 2000.

[MW10] Y. Moy, A. Wallenburg. Tokeneer: Beyond Formal Program Verification. 2010. http://www.open-do.org/wp-content/uploads/2010/04/ERTS2010_final.pdf

[RL98] L. Reinert, S. Luther. TOKENEER User Authentication Techniques Using Public Key Certificates, Part 3: An Example Implementation. Technical report, NSA Central Security Service INFOSEC Engineering, 1998.

[Spi] J. M. Spivey. The fuzz type-checker for Z. http://Spivey.oriel.ox.ac.uk/mike/fuzz

[Spi85] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 2nd edition, 1985.

[TIS] Tokeneer ID Station EAL5 Demonstrator Project. http://www.altran-praxis.com/security.aspx

[WAC10] J. Woodcock, E. G. Aydal, R. Chapman. The Tokeneer Experiments. In Jones et al. (eds.), *Reflections on the work of C.A.R. Hoare*. Springer-Verlag, 2010.

Integrated Model Checking of Static Structure and Dynamic Behavior using Temporal Description Logics

Franz Weitl and Shin Nakajima

National Institute of Informatics,
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, 101-8430 Japan

Abstract: This paper presents a new notation for the formal representation of the static structure and dynamic behavior of software, based on description logics and temporal logics. The static structure as described by UML class diagrams is represented formally by description logics while the dynamic behavior is represented by linear temporal logic and state transition systems. We integrate these descriptions of static and dynamic aspects into a single formalism called LTL_{DL}. LTL_{DL} enables a concise and natural yet precise definition of the behavior of software w.r.t. UML class diagrams and state transition diagrams. We demonstrate our approach on the sake warehouse problem. Further, we describe how properties of finite LTL_{DL} models can be analyzed based on bounded model checking and SMT (satisfiability modulo theory) solving. We implemented a restricted SMT solver for finite sets and relations. This SMT solver helped to reduce the model checking runtime significantly as compared to bounded model checking with SAL.

Keywords: Bounded Model Checking, Temporal Description Logics, SMT

1 Introduction

UML class diagrams and state transition diagrams are widely adopted for modeling software. It is desirable to detect flaws in these models as early as possible prior to implementation. We propose a new integrated approach on representing and checking consistency criteria for system models consisting of class diagrams and state transition diagrams. We base our approach on description logic, temporal logic, bounded model checking, and satisfiability modulo theory (SMT) solving.

Description logics are expressive for representing the static structure of some application domain. Their expressiveness is closely related to UML class diagrams [BCG05]. Temporal logics are well-suited to describe the behavior of processes in a formal yet abstract way. We propose to combine these formalisms in a family of temporal description logics called LTL_{DL}, to be able to address both the static and dynamic aspects of modeled systems. This goes beyond existing approaches such as Alloy [Jac02] or Spin [Hol97] which focus either on the static structure or on the dynamic behavior of the modeled system.

For the formal verification of LTL_{DL} properties, we propose a new approach based on bounded model checking and SMT solving. In a first step, LTL_{DL} models and formulae are transformed for a certain bound k into a non-temporal SMT(DL) formula which is a Boolean formula over a restricted theory of finite sets and relations. We implemented a solver for this theory based on OpenSMT [Bru09]. Experimental results show a higher performance as compared to Boolean encodings of relational models and SAT solving.

The contributions of the paper are:

1. Definition of the family of temporal description logics LTL_{DL} as a generalization of $ALC-$LTL proposed in [BGL08].
2. Demonstration of the usefulness of LTL_{DL} for representing static and dynamic properties of software models w.r.t. UML class and state transition diagrams.
3. Approach on model checking LTL_{DL}, based on bounded model checking and SMT solving.

The rest of the paper is organized as follows: first, we introduce the sake warehouse problem as a demonstration case, and model its static structure and dynamic behavior. Next, we define LTL_{DL} and discuss its application to the sake warehouse scenario. In the sequel, we present our approach on bounded model checking LTL_{DL} using SMT solving. Finally, we compare our approach with existing work and conclude the paper.

2 Sake Warehouse Scenario

We demonstrate our approach using the sake (Japanese liquor) warehouse scenario which has been published in 1984 [Yam84] as a shared scenario for comparing different modeling and programming methods. In Japan, it has been used extensively to evaluate modeling and analysis methods [NF97]. We summarize the scenario as follows: A sake shop has a warehouse in which containers are stored. A container contains bottles of one or more brands of sake. Customers place orders to the shop. Each order may include one or more brands of sake. If all ordered brands are on stock, the order is delivered immediately to the customer. Otherwise, the customer is notified and the order is put on a list of pending orders. Whenever new containers enter the warehouse, pending orders are checked and delivered in case of sufficient stock.

We use this scenario to illustrate the following steps of our approach:

1. Modeling the static structure in terms of a UML class diagram.
2. Modeling the dynamic behavior in terms of a state transition diagram.
3. Representing target properties w.r.t. the models of step 1) and 2).
4. Checking target properties, using SMT-based bounded model checking.

2.1 Sake Warehouse – Static Structure

Figure 1 depicts a UML model of the static structure of the sake shop scenario.

A *sake shop* keeps a *stock* and maintains a *list of pending orders* (Figure 1 top). The stock consists of a number of *containers* each of which may contain bottles of several *sake brands* (Figure 1 lhs). The sake shop receives new containers at regular intervals (Figure 1 lhs top).

The sake shop handles *orders* which are placed by *customers* (Figure 1 center). Each order contains one or more requested *sake brands* (Figure 1 lhs). During the order handling process, an order may become *delivered*, or *pending* if it cannot be delivered immediately because of insufficient stock (Figure 1 bottom). Pending orders are put on the *pending list* (i.e., list of

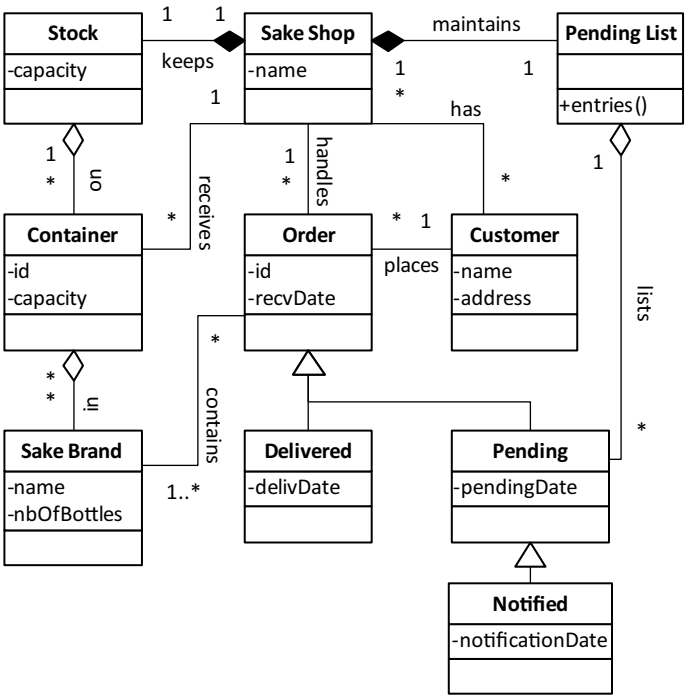


Figure 1: class diagram modeling the static structure of the sake shop.

pending orders) (Figure 1 rhs) and become *notified* (Figure 1 bottom) as soon as the shop keeper issues a notification about the delayed order to the customer.

2.2 Sake Warehouse – Behavior

Figure 2 models the basic behavior of the sake shopkeeper.

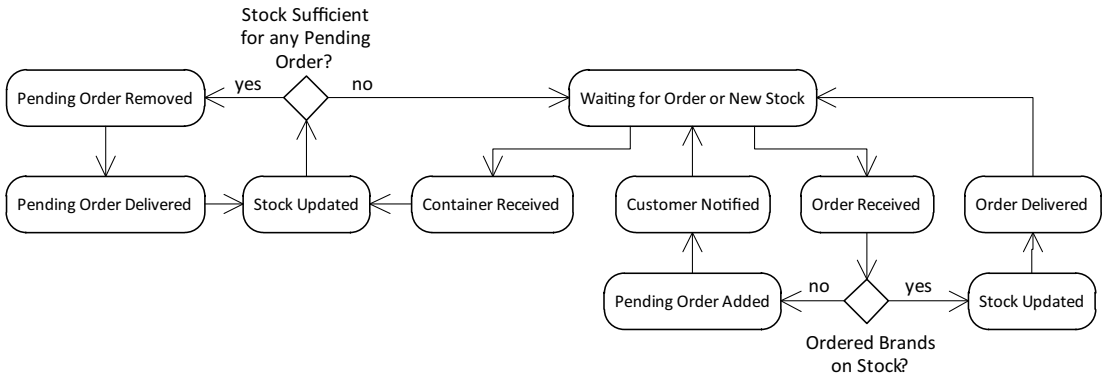


Figure 2: state transition diagram modeling the behavior of the sake shopkeeper.

Initially, the shopkeeper *waits for an order or new incoming stock* (Figure 2 rhs top). When an *order is received*, it is checked, whether all *ordered brands are on stock* (Figure 2 rhs bottom). If this is the case, the *stock is updated* and the *order is delivered* (Figure 2 rhs). Otherwise, the order is *added to the list of pending orders* and the *customer is notified* (Figure 2 center).

If the sake shop *receives a container*, it is put on the stock and the *stock is updated* (Figure 2 lhs center). Next, it is checked, if there are any pending orders and if the updated *stock is sufficient for delivering any of them* (Figure 2 lhs top). If this is the case, an appropriate order will be picked, *removed from the list of pending orders* and *delivered* (Figure 2 lhs). Further pending orders may be delivered as long as there is sufficient stock (Figure 2 lhs).

3 Sake Warehouse – Representation of Target Properties

We aim at representing properties w.r.t. both the static model and the behavior model of some application domain. In the case of our sample scenario, the following properties may be important to meet:

P1 Whenever a customer places an order, the customer will receive some response which may either be the delivery of the order or a notification that the order is pending because of insufficient stock (cf. [Nak08]).

P2 Orders may not be pending forever, i.e., orders delayed due to insufficient stock will be delivered eventually.

P3 If orders are pending then repeatedly incoming stock will eventually cause an order to be delivered.

P4 Pending orders will be handled with higher priority, i.e., a pending order of some brand X will be delivered before new orders of brand X (cf. [Nak08]).

We propose LTL_{DL} for the formal representation of such criteria. LTL_{DL} is a modular composition of linear temporal logic and description logic (*DL*). This allows for the representation of properties that address both the static structure and dynamic behavior since the semantics of UML class diagrams can be represented well by *DL*, and properties of state transition diagrams can be expressed by LTL. Before we define syntax and semantics of LTL_{DL}, we briefly review LTL and description logics.

3.1 Preliminaries – LTL

LTL (linear temporal logics) [Eme90] is supported by many model checking tools for the specification of requirements that should be met by automata-based models of the system's behavior.

Definition 1 (LTL syntax)
Let P be a set of symbols representing atomic propositions and $a \in P$ an atomic proposition. Then LTL *formulae* p, q are built according to the following rules:

$$p, q \;\rightarrow\; a \text{ (atomic proposition)} \mid \neg p \text{ (not)} \mid p \wedge q \text{ (and)} \mid p \vee q \text{ (or)} \mid p \rightarrow q \text{ (implies)} \mid$$
$$\mathsf{X}p \text{ (next)} \mid \mathsf{F}p \text{ (future/eventually)} \mid \mathsf{G}p \text{ (globally/always)} \mid p \;\mathsf{U}\; q \text{ (until)} \qquad \square$$

LTL formulae are interpreted w.r.t. state transition systems $M = (S,R,L)$ where S is a non-empty, finite set of states, $R \subseteq S \times S$ is a left-total transition relation and $L : S \to \mathscr{P}(P)$ is a labeling of states $s \in S$ with sets of atomic propositions $L(s) \subseteq P$ that hold at s.

Definition 2 (LTL semantics)
Let $M = (S,R,L)$ be a finite state transition system and $x = (s_0, s_1, ...)$ an infinite path in M, i.e., $s_i \in S$ and $(s_i, s_{i+1}) \in R$ for each $i \in \mathbb{N}$. Let $x_i = (s_i, s_{i+1}, ...)$ denote the tail of x starting from state s_i. Let a be an atomic proposition and p, q LTL formulae. Then

$$
\begin{array}{ll}
x \models a & \text{iff } a \in L(s_0) \\
x \models \neg p & \text{iff } x \not\models p \\
x \models p \ [\wedge \vee \rightarrow] \ q & \text{iff } x \models p \text{ [and or implies] } x \models q \\
x \models \mathsf{X}p & \text{iff } x_1 \models p \\
x \models \mathsf{F}p & \text{iff there is } i \in \mathbb{N} : x_i \models p \\
x \models \mathsf{G}p & \text{iff for all } i \in \mathbb{N} : x_i \models p \\
x \models p \ \mathsf{U} \ q & \text{iff there is } i \in \mathbb{N} : x_i \models q \text{ and for all } j \in \{0, ..., i-1\} : x_j \models p
\end{array}
$$

$x \models p$ expresses that path x satisfies p (or p holds on path x, respectively). An LTL formula p is considered to hold at a state $s \in S$, denoted as $s \models p$, iff for all paths $x = (s, s_1, s_2, ...)$ in (S,R) starting at s, it holds: $x \models p$. □

3.2 Preliminaries – Description Logics

Description logics is a family of fragments of first order predicate logics that are well-suited for formalizing the meaning of UML class diagrams (cf. [BCG05]).

As for this paper, we choose the description logic *ALC* for further illustration. However, the modularity of our approach allows for adopting any other decidable description logics depending on expressiveness and performance requirements. We briefly review the syntax and semantics of *ALC* as defined, for instance, in [BN03].

Definition 3 (*ALC* syntax)
Let $\mathscr{C}$ be a set of symbols called *atomic concepts* representing sets, and $\mathscr{R}$ be a set of symbols disjoint from $\mathscr{C}$ called *atomic roles* representing binary relations.

Let $A \in \mathscr{C}$ be an atomic concept and $R \in \mathscr{R}$ an atomic role. Then *ALC concepts* C, D and *ALC formulae* f, respectively, are built according to the following rules:

$$
\begin{array}{rcl}
C, D & \rightarrow & A \text{ (atomic concept)} \mid \neg C \text{ (complement)} \mid C \sqcap D \text{ (intersection)} \mid C \sqcup D \text{ (union)} \mid \\
 & & \exists R.C \text{ (existential quantification)} \mid \forall R.C \text{ (universal quantification)} \\
f & \rightarrow & C \sqsubseteq D \text{ (subsumption)} \mid C \doteq D \text{ (equality)}
\end{array}
$$

$\top$ (universal concept) abbreviates $A \sqcup \neg A$ and $\bot$ (empty concept) abbreviates $A \sqcap \neg A$. □

Example 1 (*ALC* syntax)
Consider the atomic concepts *Order*, *Delivered*, *Pending*, *Notified*, *SakeBrand*, *Container*, *PendingList* representing classes, and the atomic roles *contains*, *lists*, *in* representing binary

relations according to Figure 1. Then the following are *ALC* formulae:

$a_1 : Delivered \sqcup Pending \sqsubseteq Order$	Every *delivered* or *pending* thing is an *order*.
$a_2 : Order \doteq \exists contains.SakeBrand$	*Orders contain* at least one *sake brand*.
$a_3 : PendingList \sqsubseteq \forall lists.Pending$	Each *pending list contains pending* orders, only.
$a_4 : Order \sqcap \neg Delivered \sqsubseteq Notified$	Every *order* that is not *delivered* is *notified*.
$a_5 : Order \sqcap \forall contains.\exists in.Container \sqsubseteq Delivered$	*Orders*, which *contain* sake brands, only, that are... ...available *in* some *container*, are *delivered*.
$a_6 : Order \sqcap \neg\forall contains.\exists in.(Container \sqcap \exists on.Stock)) \doteq Pending)$	The set of *orders*, the sake brands of which are... ...not all available *in* some *container on stock*, are... ... equal to the set of *pending* orders. □

Formulae a_1 through a_3 represent some but not all properties expressed by the class diagram in Figure 1. Formulae a_4 through a_6, in turn, specify complex properties that are not represented in the class diagram of Figure 1. For a general discussion of the relationship between description logics and UML class diagrams, we refer to reader to [BCG05].

Note that, in our application scenario, the truth of formulae a_1 through a_6 may or may not depend on time. Since a_1, a_2, and a_3 formalize static properties expressed in the class diagram of Figure 1, they are expected to hold regardless of time. In contrast, the truth of a_4, a_5 and a_6 may vary throughout the order handling process. For instance, a_4 may be false at the time a new order is received. However, a_4 should become true *shortly after* an order becomes pending because of insufficient stock. In the case of a_5, orders of brands, which are on stock, may not be delivered immediately but at *some later time*. As for a_6, an order of some brand that is not on stock may become pending not immediately but *eventually*. *ALC* and any other standard description logic cannot capture such time dependencies. To solve this problem we will combine *ALC* with LTL in section 3.3.

ALC formulae are interpreted w.r.t. an interpretation domain Δ and an interpretation function $\cdot^I$ of atomic concepts and roles such that $A^I \subseteq \Delta$ and $R^I \subseteq \Delta \times \Delta$ for each atomic concept $A \in \mathscr{C}$ and atomic role $R \in \mathscr{R}$.

Definition 4 (*ALC* semantics)
Let $I = (\Delta, \cdot^I)$ be an interpretation of atomic concepts and roles, C, D *ALC* concepts and R an atomic role. Let $R^I(a) = \{b \in \Delta \mid (a,b) \in R^I\}$ denote the image of relation R^I for some $a \in \Delta$. Then

$$
\begin{aligned}
(\neg C)^I &= \Delta \backslash C \\
(C \sqcup D)^I &= C^I \cup D^I \\
(C \sqcap D)^I &= C^I \cap D^I \\
(\exists R.C)^I &= \{a \in \Delta \mid \exists b \in R^I(a) : b \in C^I\} \\
(\forall R.C)^I &= \{a \in \Delta \mid \forall b \in R^I(a) : b \in C^I\} \\
I \models C \sqsubseteq D &\text{ iff } C^I \subseteq D^I \\
I \models C \doteq D &\text{ iff } C^I = D^I
\end{aligned}
$$

□

3.3 LTL_{DL}

We propose the family of temporal logics LTL_{DL} for the representation of properties w.r.t. models of both the static structure and the dynamic behavior. LTL_{DL} is similiar to $ALC-$LTL as introduced in [BGL08]. Section 5 contains a detailed comparison of LTL_{DL} with $ALC-$LTL and other temporal description logics.

Definition 5 (LTL_{DL} syntax)
Let P be a set of symbols representing atomic propositions and **DL** be the set of formulae of some decidable description logic DL. Let $a \in A \cup \mathbf{DL}$ be an atomic proposition or DL formula. Then LTL_{DL} *formulae* p,q are built according to the following rules:

$$p,q \rightarrow a \text{ (atomic prop. or } DL \text{ formula)} \mid \neg p \text{ (not)} \mid p \wedge q \text{ (and)} \mid p \vee q \text{ (or)} \mid p \rightarrow q \text{ (implies)} \mid \mathsf{X}p \text{ (next)} \mid \mathsf{F}p \text{ (future/eventually)} \mid \mathsf{G}p \text{ (globally/always)} \mid p \,\mathsf{U}\, q \text{ (until)}$$ □

Remark 1 (LTL_{DL} syntax)
LTL_{DL} extends LTL by allowing DL formulae *in addition to* atomic propositions at locations where only atomic propositions are allowed in LTL. Hence both LTL and DL are contained in LTL_{DL}. □

Example 2 (LTL_{DL} syntax)
Consider the logic LTL_{ALC}, i.e., let DL in Definition 5 refer to ALC. Since LTL_{ALC} subsumes ALC, the formulae of Examples 1 are also LTL_{ALC} formulae. However, the following LTL_{ALC} formulae are neither in LTL nor in ALC.

$la_0 : \mathsf{F}(PendingList \sqsubseteq \neg\exists lists.Pending)$	The list of pending orders will eventually be empty.
$la_1 : \mathsf{G}(\neg(\exists places.Order \sqsubseteq \bot) \rightarrow$	Always if somebody places an order then...
$\mathsf{F}(Order \sqsubseteq Delivered \sqcup Notified))$	...eventually any order will be delivered or notified.
$la_2 : \mathsf{GF}(Pending \sqsubseteq Delivered)$	Always, eventually pending orders are delivered.
$la_3 : \mathsf{G}(\neg(Pending \sqsubseteq \bot) \rightarrow (\mathsf{GF}$	Always, if there is some pending order then...
$(SakeShop \sqsubseteq \exists receives.Container)$	...if the sake shop receives some container infinitely
$\rightarrow \mathsf{F}\neg(Delivered \sqsubseteq \bot)))$	...often then eventually there will be a delivered order.
$la_4 : \mathsf{G}((Order \sqcap \exists contains.BrandX$	Always, non-pending orders of brand X...
$\sqcap \neg Pending \sqsubseteq \neg Delivered)$	...will not be delivered...
$\mathsf{U}(Pending \sqcap \forall contains.BrandX$	...until all pending orders, which contain nothing...
$\sqsubseteq Delivered))$	...but BrandX, are delivered.

la_1 through la_4 are formal representations of properties **P1** through **P4** listed in the introduction of section 3. □

LTL_{DL} formulae are interpreted w.r.t. *finite relational state transition systems* $M = (S,R,L,\Delta,I)$ where S is a non-empty, finite set of states, $R \subseteq S \times S$ is a left-total transition relation, $L : S \rightarrow \mathscr{P}(A)$ is a labeling of states $s \in S$ with sets of atomic propositions $L(s) \subseteq A$ that hold at s, Δ is a finite set representing some domain of objects, and $I : S \rightarrow \{.^{I(s)}\}$ is a state-dependent interpretation function such that $A^{I(s)} \subseteq \Delta$ and $R^{I(s)} \subseteq \Delta \times \Delta$ for each state $s \in S$, atomic concept $A \in \mathscr{C}$, and atomic role $R \in \mathscr{R}$, respectively.

Definition 6 (LTL_{DL} semantics)
Let $M = (S, R, L, \Delta, I)$ be a finite relational state transition system and $x = (s_0, s_1, \ldots)$ an infinite path in M. Let d be a DL formula. Then

$$x \models d \text{ iff } I(s_0) \models d$$

The semantics of all other cases (atomic proposition a, Boolean connectives $\neg, \wedge, \vee, \rightarrow$, and temporal connectives $\mathsf{X}, \mathsf{F}, \mathsf{G}, \mathsf{U}$ is identical to the semantics of LTL (Definition 2). □

Example 3 (LTL_{DL} semantics)
Consider the formula $\mathsf{GF}(Order \sqsubseteq Delivered)$, i.e., "always it holds eventually that any order is delivered". Consider the path $x = (s_0, s_1, s_2, s_0, s_1, s_2, s_0, \ldots)$ where

$$\begin{array}{ll} Order^{I(s_0)} = \{o1\} & Delivered^{I(s_0)} = \emptyset \\ Order^{I(s_1)} = \{o1, o2\} & Delivered^{I(s_1)} = \{o1\} \\ Order^{I(s_2)} = \{o1, o2\} & Delivered^{I(s_2)} = \{o1, o2\} \end{array}$$

i.e., there are two orders $o1$ and $o2$ which appear in state s_0 and s_1, respectively, and which will be delivered in state s_1 and s_2, respectively. Then $x \not\models \mathsf{G}(Order \sqsubseteq Delivered)$ because, for instance, $Order^{I(s_0)} \not\subseteq Delivered^{I(s_0)}$. However, $x \models \mathsf{GF}(Order \sqsubseteq Delivered)$ because in each state s_i of x eventually s_2 will be reached and $Order^{I(s_2)} \subseteq Delivered^{I(s_2)}$.

□

4 Model Checking LTL_{DL}

Definition 7 (LTL_{DL} model checking)
Let $M = (S, R, L, \Delta, I)$ be a finite relational state transition system, $s \in S$ a state, and f a LTL_{DL} formula. Then the LTL_{DL} model checking problem for M, s, and f is to decide if $x \models f$ for all infinite paths $(s, s_1, s_2, \ldots)$ in (S, R) starting from s. □

Theorem 1 (LTL reduction)
Let $M = (S, R, L, \Delta, I)$ be a finite relational state transition system and f be a LTL_{DL} formula. Let $D = \{d_1, \ldots, d_n\}$, $n \in \mathbb{N}$, be the set of DL formulae in f. Let $A = \{a_1, \ldots, a_n\}$ be a set of atomic propositions not appearing in f such that there is a bijection $d : A \leftrightarrow D : d(a_i) = d_i$. Let $f' = f[d_1/a_1][d_2/a_2]\ldots[d_n/a_n]$ be the formula derived from f by substituting all description logics formula in f with atomic propositions.
Let $M' = (S, R, L')$ be such a transition system that $L'(s) = L(s) \cup \{a \in A \mid I(s) \models d(a)\}$.
Then f' is a LTL formula and M' a LTL transition system and it holds for each $s \in S$: $M, x \models f$ for all paths x in M' starting from state s iff $M', x \models f'$ for all paths x in M starting from s.

Proof. This is a direct consequence of the syntax and semantics definition of LTL and LTL_{DL}. □

Remark 2 (LTL reduction)
By theorem 1, a model checking algorithm for LTL_{DL} can be constructed by composing a LTL model checker and DL model checker as follows: First, using the DL model checker to

calculate the labeling function L' in Theorem 1, and then check for $M',x \models f'$ using the LTL model checker. This straight forward approach, however, is not efficient in the case of systems with many states. Hence, we strive for a more tight interaction between the LTL and *DL* model checker, using SMT-based bounded model checking. □

4.1 Bounded LTL_{DL} Model Checking

In bounded model checking [BCC+03], a transition system M, an initial state s and a LTL formula f is transformed for a given bound $k \in \mathbb{N}$ into such a non-temporal formula of the form $T_{M,s,k} \wedge \neg(f_k)$ that the following holds: if $T_{M,s,k} \wedge \neg(f_k)$ is satisfiable then there is a counterexample for $M,s \models f$ the length of which is less or equal to k and hence $M,s \not\models f$. We illustrate the approach of bounded model checking and its application to LTL_{DL} in the following example.

Example 4 (bounded LTL_{DL} model checking)

Consider the following scenario in an order handling process. Initially, there is no order. Next, a new order $o1$ is received and the reception of the order is notified to the customer. Next, another order $o2$ is received and the previously received order $o1$ is delivered. The following state transition system M models this scenario, adopting set type variables $order, notified, delivered$ for representing the set of orders, notified, and delivered orders, respectively:

state s_0	$order = notified = delivered = \emptyset$;	no orders, no deliveries, no notifications.
state s_1	$order \leftarrow order \cup \{o1\}$;	new order $o1$,
	$notified \leftarrow notified \cup \{o1\}$;	reception of $o1$ is notified to the customer.
state s_2	$order \leftarrow order \cup \{o2\}$;	new order $o2$,
	$delivered \leftarrow delivered \cup \{o1\}$;	$o1$ is delivered.
state s_3	$= s_0$	return to state s_0.

Let the *DL* concepts *Order*, *Delivered*, *Notified* represent the set of orders, deliveries, and notifications as used above. Consider the property "At any time, any order, which is not delivered, is notified":

$$f = \mathsf{G}(Order \sqcap \neg Delivered \sqsubseteq Notified)$$

We attempt to find a counterexample for f of a certain maximum length k in the state transition system M starting at s_0. As for the given scenario, a sensible bound is $k = 2$. First, we represent paths in M with maximum length k by a formula $T_{M,s_0,k}$ in which all variables are indexed by state (static single assignment form). For $k = 2$ we get:

$$\begin{aligned} T_{M,s_0,2} \;=\; & (order_0 = \emptyset) \wedge (notified_0 = \emptyset) \wedge (delivered_0 = \emptyset) \wedge \\ & (order_1 = order_0 \cup \{o1\}) \wedge (notified_1 = notified_0 \cup \{o1\}) \wedge (delivered_1 = delivered_0) \wedge \\ & (order_2 = order_1 \cup \{o2\}) \wedge (notified_2 = notified_1) \wedge (delivered_2 = delivered_1 \cup \{o1\}) \end{aligned}$$

Next, f is transformed into a non-temporal formula f_k equivalent to f in the scope k. In the given scenario, if f holds in M then $Order \sqcap \neg Delivered \sqsubseteq Notified$ holds in each state s_0, s_1, and s_2.

Adopting the semantics definition of the *ALC* connectives $\sqcap$, $\neg$, and $\sqsubseteq$ we get:

$$\begin{aligned} f_2 \;=\; & (order_0 \backslash delivered_0 \subseteq notified_0) \wedge \\ & (order_1 \backslash delivered_1 \subseteq notified_1) \wedge \\ & (order_2 \backslash delivered_2 \subseteq notified_2) \end{aligned}$$

Finally, we check if $T_{M,s_0,2} \wedge \neg f_2$ is satisfiable. From $T_{M,s_0,2}$, we get:

$$\begin{aligned} order_2 &= order_1 \cup \{o2\} = order_0 \cup \{o1\} \cup \{o2\} &&= \{o1, o2\} \\ delivered_2 &= delivered_1 \cup \{o1\} = delivered_0 \cup \{o1\} &&= \{o1\} \\ notified_2 &= notified_1 = notified_0 \cup \{o1\} &&= \{o1\} \\ order_2 \backslash delivered_2 &= \{o2, o1\} \backslash \{o1\} &&= \{o2\} \end{aligned}$$

and thus $order_2 \backslash delivered_2 \not\subseteq notified_2$ which violates f_2. Hence $T_{M,s_0,2} \wedge \neg f_2$ is satisfied and we conclude $M, s_0 \not\models f$. □

4.2 SMT(DL)

As illustrated by Example 4, we transform LTL_{DL} models and formulae into formulae that contain set-type variables and operations corresponding to the semantics of *DL* connectives. These formulae can be interpreted as SMT formulae with sets and relations as background theory. We define the language SMT(DL) for the representation for such formulae. The concrete (i.e., machine processible) syntax of SMT(DL) is defined by the following rules:

formula	→	NOT formula | formula AND formula | formula OR formula | term
term	→	TRUE | FALSE | *boolvar* | set = set | rel = rel | subset(set,set)
set	→	EMPTYSET | *setvar* | insert(set,*int*) | remove(set,*int*) |
		union(set,set) | intersect(set,set) | minus(set,set) | some(rel,set) | all(rel,set)
rel	→	EMPTYREL | *relvar* | insertrel(rel,*int,int*) | removerel(rel,*int,int*)

Table 1: SMT(DL) syntax definition

The basic symbols are composed by the disjoint sets of Boolean variables *boolvar*, set variables *setvar*, variables for binary relations *relvar*, and integer numbers *int* serving as elements of sets and relations. Formulae are built using Boolean connectives NOT, AND, OR. Basic formulae are terms, which may be either Boolean atoms or set expressions corresponding to the *DL* connectives $\doteq$ and $\sqsubseteq$. Besides the constant "EMPTYSET", set variables *setvar* may be used to represent sets. Further, "insert(set,*int*)" represents a function inserting a single integer value into a set and "remove(set,*int*)" removes an element from a set. Line 4 of Table 1 defines set operators corresponding to the syntax of the *DL* expressions $C \sqcup D$, $C \sqcap D$, $\neg C$, $\exists R.C$, $\forall R.C$. Finally, binary relations may be manipulated by "insertrel", which inserts a pair of integer values into a relation, and "removerel", which removes a pair of integer values from a relation.

Example 5 (SMT(DL) concrete syntax)
Formula $T_{M,s_0,2}$ of Example 4 reads in SMT(DL) syntax as follows:

(order0 = EMPTYSET) AND (notified0 = EMPTYSET) AND (delivered0 = EMPTYSET) AND
(order1 = insert(order0, 1)) AND (notified1 = insert(notified0, 1)) AND (delivered1 = delivered0) AND
(order2 = insert(order1, 2)) AND (notified2 = notified1) AND (delivered2 = insert(delivered1,1))

Note that orders $o1$ and $o2$ in formula $T_{M,s_0,2}$ are represented by integer values 1 and 2, respectively. This is valid in general because we assume a finite interpretation domain (cf. Definition 6) which can be mapped onto integer numbers without loss of information.

Formula f_2 of Example 4 reads in SMT(DL) syntax as follows:

subset(minus(order0,delivered0),notified0) AND
subset(minus(order1,delivered1),notified1) AND
subset(minus(order2,delivered2),notified2)

□

4.3 Prototypical Implementation and Experimental Results

We implemented a partial solver for SMT(DL) based on OpenSMT [Bru09] which is an open source SMT solver implemented in C++. For the representation of SMT(DL) formulae, we use the standard format SMT-LIB 1.2. The current implementation is limited to SMT(DL) formulae, the set and relation expressions of which are bound to finite domains and do not contain cyclic definitions such as "s = insert(s,1)". The latter is not a restriction in our application because, in bounded model checking, LTL_{DL} models are transformed into static single assignment form (cf. Example 4) which do not contain any cyclic definitions by construction.

The aim of the subsequent experiment is to determine the runtime of model checking LTL_{DL} as compared to existing bounded model checkers. The runtime of bounded model checking is dominated by checking the satisfiability of the generated formula $T_{M,s,k} \wedge \neg f_k$ (cf. Example 4). To determine the scaling of runtime w.r.t. the input size, we use a parameterized scenario similar to that in Example 4, as follows:

state s_0	$order = notified = delivered = \emptyset$;
state s_1	$order \leftarrow order \cup \{o_1\}$; $notified \leftarrow notified \cup \{o_1\}$;
state s_2	$order \leftarrow order \cup \{o_2\}$; $delivered \leftarrow delivered \cup \{o_1, o_2\}$;
state s_3	$order \leftarrow order \cup \{o_3\}$; $notified \leftarrow notified \cup \{o_3\}$;
state s_4	$order \leftarrow order \cup \{o_4\}$; $delivered \leftarrow delivered \cup \{o_3, o_4\}$;
...	
state s_{2n-1}	$order \leftarrow order \cup \{o_{2n-1}\}$; $notified \leftarrow notified \cup \{o_{2n-1}\}$;
state s_{2n}	$order \leftarrow order \cup \{o_{2n}\}$; $delivered \leftarrow delivered \cup \{o_{2n-1}, o_{2n}\}$;
state s_{2n+1}	$order \leftarrow order \cup \{o_{2n+1}\}$;

As a property, we check, if each undelivered order is notified at any time (cf. Example 4):

$$f = \mathsf{G}(Order \sqcap \neg Delivered \sqsubseteq Notified)$$

The only state violating f is s_{2n+1}. To detect the error by bounded model checking, the bound k must be chosen greater or equal to $2n+1$, making the case increasingly challenging for larger n. Moreover, the maximum sizes of the sets for representing received, notified, and delivered orders grow linearly in n.

To compare the performance of our approach with existing ones, we chose the SAL tool [MOR+04] since it integrates a variety of state-of-the-art model checking algorithms, including SAT and SMT-based bounded model checking. SAL uses the SMT solver Yices 1.03 [DM06] as a backend engine for bounded model checking. The scenario above can be described compactly in terms of the SAL input language by representing the characteristic function $\mathbf{1}_S : S \to \{false, true\} : \{x \in S \mid \mathbf{1}_S(x) = true\} = S$ of each set S as a Boolean array (cf. [KRW09]). The bounded model checker of SAL translates an input file for a given bound k into a SAT or SMT formula which is then solved by Yices. For our experiment, we chose the transformation into SAT because this yielded higher performance.

An alternative SMT(DL)-based representation (cf. Example 5) for different problem sizes n and bounds k has been generated. Generally, we distinguish two cases. 1) $k = 2n+1$: in this case, the generated SAT and SMT(DL) formulae are satisfiable, i.e., the property violation is detected; 2) $k = 2n$: the generated SAT and SMT(DL) formulae are not satisfiable.

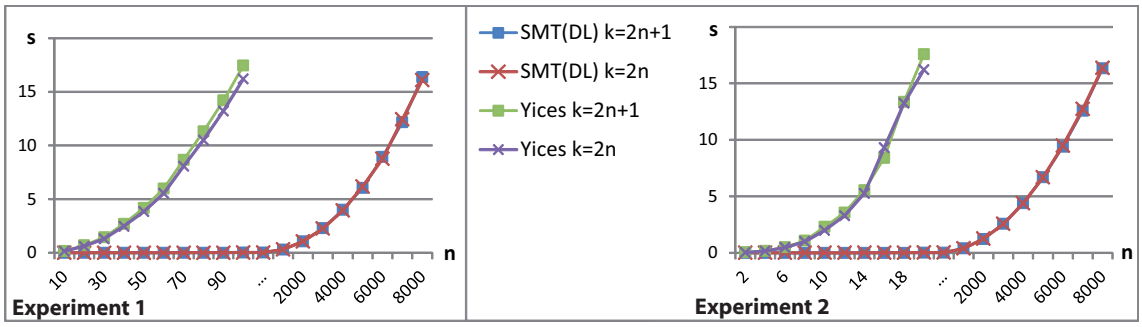


Figure 3: execution time of SMT(DL) solving as compared to SAT solving with Yices for different input sizes n in Experiment 1 and 2.

Figure 3 (lhs: **Experiment 1**) shows the runtime of Yices and our SMT(DL) solver for the two cases $k = 2n+1$ and $k = 2n$ and increasing input sizes n, obtained on a desktop computer with and 6 GB RAM and Intel Core i7 processor at 3.8 GHz. While the runtime of Yices for $k = 2n$ is slightly lower than in the case of $n = 2n+1$, the runtime of SMT(DL) is identical for both cases. In the case of $n = 100$, Yices takes 17.5 seconds for $k = 201$ and 16.2 seconds for $k = 200$. In about the same time, the SMT(DL) solver processes a formula 80 times as large ($n = 8000$, $k = 16000/16001$).

Figure 3 (rhs: **Experiment 2**) shows the runtime of Yices and our SMT(DL) solver for checking the formula

$$f' = \mathsf{G}(\neg(\exists places.Order \sqsubseteq \bot) \to \mathsf{XX}(Order \sqcap \neg Delivered \sqsubseteq Notified))$$

in a LTL_{DL} model corresponding to the state transition diagram of Figure 2. f' reads: "Always (G), if someone places an order ($\neg(\exists places.Order \sqsubseteq \bot)$) then two states later (XX) each order that has not been delivered is notified ($Order \sqcap \neg Delivered \sqsubseteq Notified$)".

In this scenario, Yices takes 17.6 seconds for $n = 20$ if a counterexample is found, and 16.2 seconds if no counterexample is found. We suppose that the Boolean encoding of the binary relation *places* in formula f' is the major source of additional complexity. In contrast, the runtime of

the SMT(DL) solver is hardly affected by the presence of a binary relation in Experiment 2. This indicates that supporting sets and relations in SMT solving can significantly speed up bounded model checking of relational models as compared to SAT-based bounded model checking.

5 Related Work

Description logics are well-known to be appropriate for the formal representation of conceptual data models such ER diagrams and UML class diagrams. For instance, [CLN98] proposes a unifying description logics for the logical representation of class-based data models such as ER and object-oriented data models. [BCG05] presents an encoding of UML class diagrams in the description logic *ALCQI* to discover inconsistencies in models by means of description logic reasoning. We extend these approaches by combining a description logic with a temporal logic to support the representation of properties related to both state transition diagrams and class diagrams.

In the past, several combinations of description logics and temporal logic have been suggested [AF01, LWZ08]. A first temporal extension of the description logic *ALC* called *ALCT* was suggested by Schild [Sch93]. In *ALCT*, the temporal connectives G, F, and U can be applied to concepts but not to axioms. A similar combination of LTL and *ALC* is called LTL_{ALC} in [LWZ08]. In contrast, *ALC*−LTL, as introduced in [BGL08], supports the application of temporal connectives to *ALC* axioms but not to *ALC* concepts.

LTL_{DL}, as proposed in this paper, follows the latter approach because, this way, a higher degree of modularity between the temporal and non-temporal part of the logic is achieved. This simplifies the formalization of properties in close correspondence with UML class diagrams (*DL* component) and state transition diagrams (LTL component), as well as the implementation of a model checker. However, LTL_{DL} is different from *ALC*−LTL in the following aspects:

- LTL_{DL} is a family of logics, obtained by a modular combination of some *DL* with LTL, rather than a single logic.
- While in *ALC*−LTL, atomic propositions are replaced by *ALC* axioms, LTL_{DL} supports *DL* formulae *in addition* to atomic propositions. This ensures compatibility with propositional LTL widely adopted in model checking.
- In contrast to *ALC*−LTL, we do not consider ABox assertions in LTL_{DL} since they seem to be dispensable for formalizing general domain models represented by UML class diagrams.
- As opposed to *ALC*−LTL, we do not consider *rigid* symbols, i.e., concepts and roles the interpretations of which do not depend on states. Incorporating rigid symbols to LTL_{DL} may be an interesting topic of future research.

[BGL08] focusses on the satisfiability problem of *ALC*−LTL and the impact of rigid symbols on the complexity of solving the satisfiability problem. In this paper, we do not consider the satisfiability problem but the model checking problem of LTL_{DL}. A thorough investigation of complexity properties will be an issue of future work.

[BBL09] proposes runtime verification based on $ALC-$LTL. In runtime verification, a monitor constantly observes the behavior of a system in execution and determines if the observed prefix of an execution trace conforms to a temporal formula. Each state of the execution trace is represented in a potentially incomplete way by a set of *ALC* ABox assertions (open world assumption). In our work, we adopt a model checking approach, i.e., all possible behaviors of a system described by a state transition system are considered. However, the information about each single state is assumed to be complete (closed world assumption).

An algorithm for model checking the temporal description logics *ALC*CTL has been proposed in [Wei08]. In this paper, we consider the bounded model checking problem of LTL_{DL} and reduce it to SMT solving which we believe is a new approach that simplifies the integration of LTL_{DL} model checking into an existing model checking environment such as SAL and helps to increase the performance of model checking for bounded sets and relations.

State-of-the-art model checkers supporting linear temporal logic are Spin [Hol97], SAL [MOR+04], and NuSMV [CCG+02]. However, the input languages of these model checkers do not support set and relation data types and hence are inefficient for representing properties w.r.t. relational models.

Alloy is a declarative object-oriented modeling notation, the semantics of which is based on sets and relations [Jac02]. The notation supports the formulation of assertions. Dynamic aspects may be addressed in terms of pre- and post-conditions or by explicitly representing time as a linearly ordered set of states. However, temporal logic for the representation of behavioral properties is not supported. A tool based on SAT solving automatically analyzes whether assertions hold in models where the sizes of all sets and relations are bounded by some user chosen constants [JSS00]. In [GT11], an alternative approach is presented which is not limited to bounded sets: Alloy relational specifications are translated into first order quantified SMT formulae which are passed on to the SMT solver Z3 [MB08]. However, since the Alloy specification language is undecidable, the SMT solver may fail to prove assertions.

Event B [Abr10] is a formal specification language for the required behavior of a system, based on set theory and logic. A central concept is the refinement-based modeling for system requirements. Consistency and refinement checking of specifications, based on theorem proving, is supported by the Rodin tool [ABH+10] which generates and manages the necessary proofs. However, user interaction may be required for certain types of proofs. ProB [LB08], an animation and model checking tool for (Event) B specifications, supports model checking of properties expressed in LTL. Similar to Alloy, data types such as sets and relations must be restricted to small sizes for exhaustive analysis. LTL_{DL} is less expressive than the temporal logic supported by ProB but the supported constructs are closely related to UML class and state transition diagrams. We believe that this simplifies the identification and formalization of relevant consistency properties which is usually considered as a rather difficult task.

The syntax definition for SMT(DL) (Table 1) is inspired by [KRW09] which suggests a format for representing finite lists, sets, and maps as part of the SMT-Lib 2.0 format. As for solving formulae over finite sets, a mapping onto Boolean arrays is suggested. We have adopted this approach in our experiments with SAL and Yices (see section 4.3). To the best of our knowledge, none of the currently available SMT solvers implements dedicated decision procedures for sets and relations.

6 Conclusion

We have presented a new integrated approach on representing both static and dynamic aspects of software models. We defined LTL_{DL} as a modular composition of linear temporal logic LTL and a description logic *DL*. LTL_{DL} supports representing properties w.r.t. both UML class diagrams and state transition diagrams. We believe that the close correspondence of LTL_{DL} formulae to these commonly used diagram notations facilitates the identification and formalization of important consistency requirements at an early development stage. Further, we have demonstrated how LTL_{DL} formulae can be checked by SMT-based bounded model checking. We have implemented a prototypical SMT solver for formulae containing set-type expressions corresponding to the semantics of LTL_{DL} connectives. As compared to reducing set-type expressions to Boolean arrays, about two orders of magnitude as large problems could be solved in the same execution time.

In this paper, we discussed LTL_{DL} from an application-oriented perspective and demonstrated its usefulness and performance by a case study. Fundamental properties of LTL_{DL} such as expressiveness and runtime complexity of model checking and deciding satisfiability are left to be studied in future work.

In our current experiments, we use the input language of SAL for representing LTL_{DL} models, adopting a Boolean encoding for sets and relations. A more adequate representation language for LTL_{DL} models offering explicit support for sets and relations is a major issue of ongoing work. Ongoing is also the improvement of the implemented SMT solver in terms of supported types of formulae and performance. Issues are, for instance, the support of cyclic expressions and negation in unbounded domains (cf. section 4.3). To this end, a mapping of SMT(DL) formulae onto either first order quantified SMT formulae or description logic knowledge bases seems to be promising and calls for further examination. Finally, further case studies to compare our approach with existing approaches such as Event-B and Alloy are necessary. In addition, the comparison with existing state-of-the-art model checkers such as CBMC [CKL04] and SMT solvers, which support quantified formulae such as Z3 [MB08], is an important issue of future work.

Acknowledgements: This work is funded by the program "Research at International Science and Technology Centers" of the German Academic Exchange Service (DAAD). We thank the reviewers for their detailed comments which helped to improve the paper significantly and gave directions for future work.

Bibliography

[ABH+10] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT* 12(6):447–466, 2010.

[Abr10] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[AF01] A. Artale, E. Franconi. A Survey of Temporal Extensions of Description Logics. *Annals of Mathematics and Artificial Intelligence (AMAI)* 30(1-4):171–210, 2001.

[BBL09] F. Baader, A. Bauer, M. Lippmann. Runtime Verification Using a Temporal Description Logic. In Ghilardi and Sebastiani (eds.), *Frontiers of Combining Systems*. LNCS 5749, pp. 149–164. Springer-Verlag, 2009.

[BCC+03] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu. Bounded Model Checking. In Zelkowitz (ed.), *Highly Dependable Software*. Advances in Computers 58, pp. 118–149. Academic Press, 2003.

[BCG05] D. Berardi, D. Calvanese, G. D. Giacomo. Reasoning on UML Class Diagrams. *Artificial Intelligence* 168(1-2):70–118, 2005.

[BCM+03] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, P. Patel-Schneider (eds.). *The Description Logic Handbook - Theory, Implementation and Applications*. Cambridge University Press, 2003.

[BGL08] F. Baader, S. Ghilardi, C. Lutz. LTL over description logic axioms. In *Proceedings of the 11th Inernational Conference on Principles of Knowledge Representation and Reasoning (KR 2008)*. Pp. 684–694. Morgan Kaufmann, Sydney, Australia, 2008.

[BN03] F. Baader, W. Nutt. Basic description logics. In *[BCM+03]*. Chapter 2, pp. 47 – 100. 2003.

[Bru09] R. Bruttomesso. An Extension of the Davis-Putnam Procedure and its Application to Preprocessing in SMT. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories (SMT2009)*. Montreal, Canada, 2009.

[CCG+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *Proceedings of Computer Aided Verification (CAV 02)*. LNCS 2404. Springer, 2002.

[CKL04] E. Clarke, D. Kroening, F. Lerda. A Tool for Checking ANSI-C Programs. In Jensen and Podelski (eds.), *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*. LNCS 2988, pp. 168–176. Springer-Verlag, 2004.

[CLN98] D. Calvanese, M. Lenzerini, D. Nardi. Logics for databases and information systems. In Chomicki and Saake (eds.). Chapter 8 Description logics for conceptual data modeling, pp. 229–263. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[DM06] B. Dutertre, L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proceedings of the 18th Computer-Aided Verification Conference (CAV'06)*. LNCS 4144, pp. 81–94. Springer-Verlag, 2006.

[Eme90] E. Emerson. Temporal and Modal Logic. In Leeuwen (ed.), *Handbook of Theoretical Computer Science: Formal Models and Semantics*. Pp. 996–1072. Elsevier, 1990.

[GT11] A. A. E. Ghazi, M. Taghdiri. Relational Reasoning via SMT Solving. In *17th International Symposium on Formal Methods (FM)*. Limerick, Ireland, 2011.

[Hol97] G. J. Holzmann. The Model Checker Spin. *IEEE Transactions on Software Engineering* 23(5):279–295, 1997.

[Jac02] D. Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology (TOSEM'02)* 11(2):256–290, 2002.

[JSS00] D. Jackson, I. Schechter, I. Shlyakhter. Alcoa: the alloy constraint analyzer. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*. Pp. 730–733. ACM Press, 2000.

[KRW09] D. Kröning, P. Rümmer, G. Weissenbacher. A Proposal for a Theory of Finite Sets, Lists, and Maps for the SMT-Lib Standard. Published on http://www.cprover.org/SMT-LIB-LSM/, 2009. Visited 9 Jan 2010.

[LB08] M. Leuschel, M. Butler. ProB: An Automated Analysis Toolset for the B Method. *Journal Software Tools for Technology Transfer* 10(2):185–203, 2008.

[LWZ08] C. Lutz, F. Wolter, M. Zakharyaschev. Temporal Description Logics: A Survey. In *Proceedings of the 15th International Symposium on Temporal Representation and Reasoning (TIME '08)*. Pp. 3–14. IEEE Computer Society, Washington, DC, USA, 2008.

[MB08] L. de Moura, N. Bjorner. Z3: An Efficient SMT Solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*. LNCS 4963, pp. 337–340. Springer-Verlag, 2008.

[MOR+04] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, A. Tiwari. SAL 2. Tool description presented at CAV 2004. LNCS 3114, pp. 496–500. Springer-Verlag, 2004.

[Nak08] S. Nakajima. *Model Checking with SPIN*. Chapter 9: Case Study(4). Kindaikagakusha, Tokyo, Japan, 2008.

[NF97] S. Nakajima, K. Futatsugi. An object-oriented modeling method for algebraic specifications in CafeOBJ. In *Proceedings of the 19th international conference on Software engineering (ICSE '97)*. Pp. 34–44. Boston, Massachusetts, United States, 1997.

[Sch93] K. Schild. Combining terminological logics with tense logic. In *Proceedings of the 6th Portuguese Conference on Artificial Intelligence*. Pp. 105–120. Porto, 1993.

[Wei08] F. Weitl. *Document Verification with Temporal Description Logics*. PhD thesis, University of Passau, 2008.

[Yam84] T. Yamasaki. Surveys of Program Design Methods Using a Common Example Problem. *Journal of IPS Japan* 25(9):934, 1984. In Japanese.

Symbolic Model Checking and Safety Assessment of Altarica models

Marco Bozzano [1], Alessandro Cimatti [1], Oleg Lisagor [2], Cristian Mattarei [1], Sergio Mover [1], Marco Roveri [1] and Stefano Tonetta [1]

[1]Fondazione Bruno Kessler, Trento, Italy
[2]The University of York, York, United Kingdom
{bozzano,cimatti,mattarei,mover,roveri,tonettas}@fbk.eu
oleg.lisagor@cs.york.ac.uk

Abstract: Altarica is a language used to describe critical systems. In this paper we present a novel approach to the analysis of Altarica models, based on a translation into an extended version of NuSMV. This approach opens up the possibility to carry out functional verification and safety assessment with symbolic techniques. An experimental evaluation on a set of industrial case studies demonstrates the advantages of the approach over currently available tools.

Keywords: Model Checking, Safety Assessment, Fault Tree Analysis, Altarica

1 Introduction

The dramatic increase in complexity of safety-critical systems in recent years has motivated a growing interest in model-based techniques for system verification. Such techniques must be able to verify functional correctness, but also to carry out safety assessment, that is, assess system behavior in the presence of faults [Ba03, ÅBB06, BV10]. In particular, there has been a growing interest in formal verification tools that can automate the generation of artefacts such as Fault Trees and Failure Mode and Effects Analysis (FMEA) tables [FSA, BV07, BCK+10].

One of such tools is Cecilia OCAS [BBC+04] – a model-based safety assessment platform developed by Dassault Aviation, based on the Altarica [Alt, AGPR00] language. Altarica has been used in the past for safety assessment of industrial systems, see, e.g., [BCS02, BBC+04]. Moreover, OCAS is being used at an industrial level for architectural safety assessment of avionics systems. For example, the Flight Control System of Falcon 7x aircraft has been certified on the basis of the OCAS analysis. OCAS is equipped with different model analysis tools, the main ones are a trace simulator, and a sequence generator to generate minimal cut sets. However, these tools are neither able to perform an *exhaustive* space examination, nor they are able to model check *temporal* properties; even reachability analysis is bounded in depth. Furthermore, developed as an in-house tool, the OCAS sequence generator does not correctly implement language features that are not used within Dassault Aviation. In particular it is unable to adequately explore non-deterministic instantaneous transitions, potentially leading to incomplete analysis results (although the tool can be configured to provide a warning). Finally, the OCAS sequence generator is based on explicit state techniques, hence it suffers from the state-explosion problem.

In this paper we propose a fully symbolic approach that overcomes these limitations, and allows for the industrial usage of advanced symbolic verification and safety assessment techniques. Our approach is based on the translation to an extended version of NuSMV [NuS], and

is tightly integrated with the OCAS environment. NuSMV is a state-of-the art symbolic model checker providing cutting-edge model checking technologies such as BDD-based [Bry92] and SAT-based Bounded Model Checking (BMC) [BCCZ99] techniques. It supports both temporal model checking (CTL and LTL temporal logics) and safety assessment, e.g., Fault Tree Analysis (FTA) and FMEA, through its add-on NuSMV-SA. NuSMV has been used in several industrial contexts, for instance for verification and validation of aerospace systems [BCK+10].

More specifically, our contribution is as follows. First, we have isolated a fragment of Altarica in the Dataflow formulation. This choice has been dictated by what is being made available through the OCAS interface. As the semantics for this fragment is not fully documented, an additional effort has been required to provide a formal definition for its semantics, by adaptation from the general definition of [AGPR00], and to validate its correctness with respect to the behavior shown by OCAS and user expectations. In the course of our work, we have identified model features that are not correctly managed in OCAS, clarified their intended semantics, and reflected it in our tool. Based on the semantics, we have implemented a translator to convert Altarica models into NuSMV. The translation uses *HyDI* [CMT11] as an intermediate language. The use of *HyDI* proved to be convenient as it provides primitives to deal with networks of automata, and different mechanisms for synchronizing them. The translator has been incorporated as a plugin, named the NuSMV/OCAS plugin, into the OCAS environment, and it provides the following functionalities: invariant checking, temporal model checking, and fault tree generation.

The NuSMV/OCAS plugin has been developed within the MISSA project [MIS] (More Integrated Systems Safety Assessment), an EC-sponsored project involving various research centers and industries from the avionics sector. We evaluated the plugin on a set of industrial-size case studies developed in MISSA, and compared it with existing tools available in OCAS. The results of the evaluation clearly show a significant advantage of symbolic techniques over explicit-state techniques currently provided by OCAS, in terms of performance.

The paper is organized as follows. In Section 2 we give a short overview of the Altarica syntax and semantics. In Section 3 we present the design of the translation. In Section 4 we describe the integration into OCAS. In Section 5 we discuss the experimental evaluation. Finally, in Section 6 we present some related work, and in Section 7 we conclude and discuss future work.

2 Overview of Altarica

In this section we briefly describe the syntax of the Altarica language (Dataflow dialect implemented in Cecilia OCAS) and its semantics - we refer the reader to [Alt, AGPR00] for additional details. A simple example of Altarica model is presented in Figure 1. It consists of two counters modulo 4 and an adder. The base component of an Altarica model is called *node*. Its structure may comprise the following sections:

- *sub*: used to describe the hierarchy of the Altarica nodes; in this section, it is possible to instantiate the *subnodes* which are the children of the current node;
- *state*: this section is used to declare the state variables of the (basic) node; the value of these variables may change only upon firing of an event; this implies that their value does not change in between two consecutive event firings (while other components are executing);

```
node adder
  flow
    input1:[0,3]:in;
    input2:[0,3]:in;
    value_out:[0,7]:out;
  state
    value:[0,7];
  event
    add,
    fault_add;
  trans
    value < 7 |- add -> value := input1 + input2;
    true |- fault_add -> value := 7;
  init
    value := 0;
  assert
    value_out = value;
  extern
    law <event fault_add> = Exponential(0.1);
edon

node observer
  flow
    out_ok:bool:out;
    input1:[0,3]:in;
    input2:[0,3]:in;
    inputS:[-1,6]:in;
  assert
    out_ok = (inputS = (input1 + input2));
edon
```

```
node counter
  flow
    value_out:[0,3]:out;
  state
    value:[0,3];
  event
    inc, reset;
  trans
    value < 3 |- inc -> value := value + 1;
    value = 3 |- reset -> value := 0;
  init
    value := 0;
  assert
    value_out = value;
edon

node main
  event
    total_reset;
  sub
    c1: counter;
    c2: counter;
    add: adder;
    obs: observer;
  sync
    <total_reset, c1.reset, c2.reset>;
  assert
    c1.value_out = add.input1,
    c2.value_out = add.input2,
    c1.value_out = obs.input1,
    c2.value_out = obs.input2,
    add.value_out = obs.inputS;
edon
```

Figure 1: An example Altarica model

- *init*: this section is used to specify the initial value of state variables;
- *event*: used for defining the events that can be fired and, thus, trigger a state transition;
- *flow*: this section declares flow variables, used to describe the connections with the other components; flow variables are linked to state variables by means of assertions; there are two types of flow variables, namely *input* and *output* flow variables;
- *trans*: this section is used to describe the transitions of the system; each transition consists of a guard, the firing event, and a list of assignments; the assignments specify how the system state changes when the corresponding event is fired; the guard is a precondition that has to be satisfied for the transition to be taken;
- *assert*: used to establish links from a flow variable to a state variable or another flow variable; more specifically, it declares a set of equalities either between an output flow variable and an expression over input flow and state variables (*internal assert*), or between an input flow of a subnode and the output flow of another subnode (*in-out assert*), or between an input flow of the node and an input flow of a subnode (*in-in assert*), or between an output flow of the node and an output flow of a subnode (*out-out assert*);
- *sync*: used to define the synchronizations; a synchronization associates an event of the node to the events of the subnodes; there are three types of synchronizations, namely *strong sync*, *weak sync*, and *Common Cause Failure* (CCF) (cf. end of this section);
- *extern*: used to associate events with *priorities* and optional *laws*; priorities and some of the laws constrain permissible order of event firing.

An Altarica model is a hierarchical graph composed of nodes. At the same level of the hierarchy, nodes communicate through flows and synchronizations. The hierarchy yields a tree structure, where two types of nodes are possible:

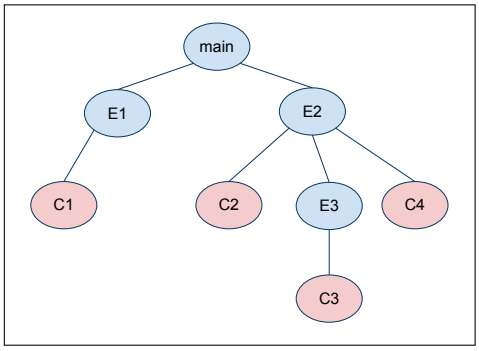


Figure 2: Altarica hierarchy

- *component*: a component represents a single process of the system, it cannot contain definition of subnodes or synchronizations;

- *equipment*: an equipment node represents a container for nodes; it may contain declarations of subnodes and synchronizations, but it cannot have state variables.

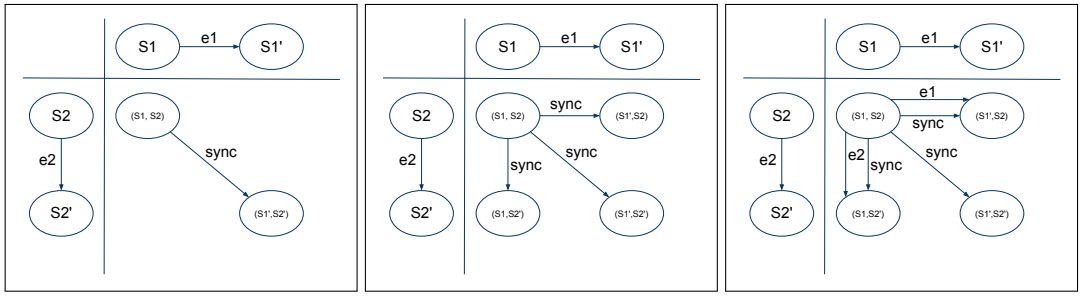
As shown in Figure 2, this structure imposes that the *component* nodes represent the leafs, whereas the *equipment* nodes are containers for the components. Moreover, there is a special equipment node called *main*, which represents the root of the full Altarica model.

The semantics of the Altarica model is defined in terms of Interfaced Transition Systems (ITSs) (cf. [AGPR00, Mat11]). Intuitively, the ITS associated with a component is given straightforwardly by the state variables (that define the states), the initial condition, the transitions, the events and flow variables (which define the observations) of the node. The ITS associated to an equipment node is given by the composition of the ITSs associated with the subnodes taking into account synchronizations. The mechanisms for the different synchronizations are illustrated in Figs. 3a, 3b and 3c, and explained in more detail in the following:

- *strong sync* (see example in Figure 3a): if we have a strong sync between the events e_1 and e_2, the corresponding processes (components) p_1 and p_2 must move synchronously on such events. This means that the transitions of p_1 fired by e_1 and the transitions of p_2 fired by the event e_2 happens at the same time, and that e_1 is fired if and only if e_2 is fired; as an example, the system in Figure 1 declares a strong synchronization, called $total\_reset$, synchronizing the reset on the two counters;

- *weak sync* (see Figure 3b): this type of synchronization represents a broadcast; participating events happen synchronously as in the strong sync, but only if the corresponding transitions are enabled; this means that if the event e_1 of p_1 is fired and there exists a transition t_2 of p_2 on the event e_2 whose guard is true, then e_2 is fired at the same time as e_1; otherwise (if the guard is false) e_1 is fired and p_2 does not change state; similarly, if e_2 is fired and the guard on e_1 is false, p_1 does not change state;

- *CCF sync* (see Figure 3c): short for Common Cause Failure, this kind of synchronization is similar to a weak synchronization, with the difference that individual processes are also



(a) Strong synchronization (b) Weak synchronization (c) CCF synchronization

Figure 3: Synchronization examples

allowed to move on the events independently; this means that either we have a CCF sync involving e_1 and e_2 (with the same rules of the weak sync) or e_1 is fired or e_2 is fired.

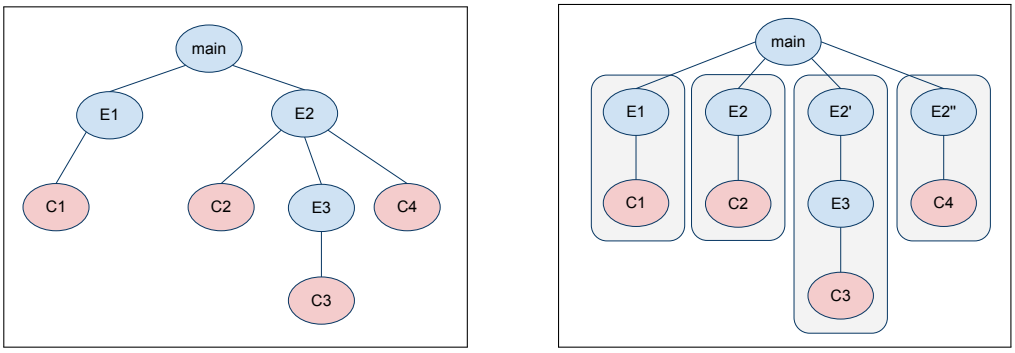
The evolution of an Altarica system can be further constrained by associating events with special *laws* and *priorities*. By default, events are considered *stochastic*. These events are typically used to model component failures and can be optionally associated with a probability distribution law (e.g., $Exponential(\lambda)$ law). These laws are used to establish interoperability with commercial RAMS (Reliability, Availability, Maintainability and Safety) analysis tools and do not affect qualitative behaviour of the system. However, a special law – $Dirac(x)$ – is used to mark *instantaneous* and *temporal* events (with $x = 0$ and $x > 0$ respectively). These events fire deterministically x time steps after the guard of the corresponding transition becomes $true$. Whenever more than one transition is possible at the same time, instantaneous events take precedence. The precedence of transitions can be further constrained by event priorities (events with higher priority are fired first). For the sake of brevity, we do not describe the semantics of priorities in detail – we refer to Section 3 for their encoding.

3 Translation

In this section we describe the encoding of the Altarica language into NuSMV. The formal translation [Mat11] has been designed using *HyDI* [CMT11] as an intermediate language. In the following, we first introduce the *HyDI* language and then we focus on the translation of the main characteristics of Altarica into *HyDI*- we refer to [CMT11] for a discussion of the translation from *HyDI* to NuSMV. In particular, we discuss the management of:

- *hierarchy*: unlike Altarica, *HyDI* does not support hierarchical process definitions;
- *flow variables and assertions*: these definitions cannot be directly mapped into *HyDI*;
- *event priorities*: *HyDI* does not support the definition of event priorities;
- *synchronizations*: Altarica supports three kinds of synchronizations: *strong*, *weak* and *CCF*, whereas *HyDI* supports only the first two.

Finally, we briefly discuss how to model the leaf nodes.



(a) Altarica structure

(b) *HyDI* structure

Figure 4: Hierarchy translation

3.1 The *HyDI* language

HyDI is an extension of *SMV* [McM93] that supports the definition of networks of hybrid automata with different kinds of synchronizations. We restrict our presentation to the finite state case, thus ignoring continuous variables and their evolution – see [CMT11] for a complete description. A *HyDI* program is given by a set of *modules*, a set of *processes* and a set of *synchronization* constraints. A *HyDI* module extends *SMV* modules allowing one to specify synchronization constraints. A module contains a set of declarations which define: a set of variables (*VAR*); a set of input variables (*IVAR*); a set of initial constraints (*INIT*) defining the initial states; a set of invariant conditions (*INVAR*) which restricts the valid assignments to the variables; a set of transition constraints (*TRANS*), defining the state transitions. A module can be instantiated in the *VAR* section of another module. The main module is the top-level module of a program and cannot be instantiated. The *HyDI* language allows one to define a network of processes which run asynchronously on private events while they synchronize on shared events. The processes are instantiated in the main module. The network is not hierarchical, since the synchronizations are declared between processes. However, the definition of a single process may be hierarchical, since it can contain the instantiation of sub-modules. The module used to instantiate a process contains the definition of the set of discrete events (*EVENT* section) used to define its synchronization with other processes. In the *HyDI* language a synchronization declares that two events of two processes must be fired at the same time. A variant of this type of synchronization, called "weak" synchronization, allows one to specify a guard which forces the synchronization only if the guard evaluates to true. Finally, the order of occurrence of events can be further constrained with a scheduler, modeled in *HyDI* by variables and constraints in the main module.

3.2 Hierarchy translation

The network of processes defined by Altarica is hierarchical in that the synchronizations may be specified at the different levels of the Altarica tree structure. Thus, in order to encode the Altarica specification into *HyDI* we perform a flattening of the Altarica hierarchy as depicted in Figure 4b. Each Altarica equipment node is split into several new instances in order to create a

hierarchy corresponding to the paths from the root to each leaf. This flattening is possible since the instances of the equipment nodes cannot have definition of state variables.

For the flattening it is necessary to perform some additional transformations on the resulting structure because of the constraints imposed by the *HyDI* language. In Altarica synchronization definitions can be specified at all levels of the hierarchy (i.e., in the equipment nodes). In *HyDI* they must be in the main module. Thus, we need to move all the synchronization definitions in the top level *HyDI* main module. Another difference between *HyDI* and Altarica concerns the definition of discrete events used in the synchronizations. In *HyDI* the declaration of discrete events is done in the module definition of each instance and, thus, new events cannot be declared in a submodule. Altarica, on the other hand, requires them to be specified within the leafs (i.e., in the component nodes). Our solution restructures the Altarica hierarchy in such a way all the events present in the original Altarica structure are declared in the definition of an instance in *HyDI*, and passed as parameters to the submodules. The drawback of this encoding consists in the possible growth in terms of resulting model size. However, this solution does not increase the complexity and also it permits to greatly simplify the translation from Altarica to *HyDI*.

3.3 Variables and assertions translation

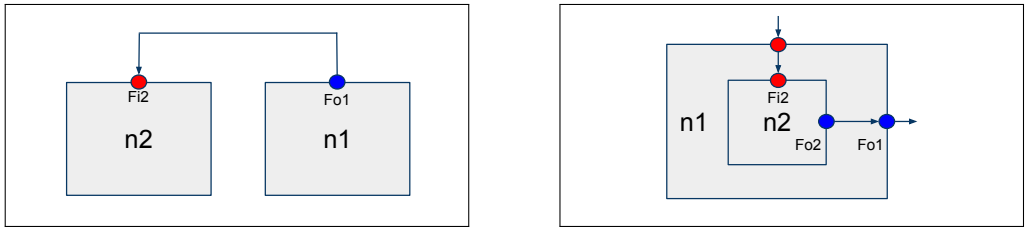
Altarica allows one to define two types of variables: state variables (which represent the internal state of the system) and flow variables (used to expose the internal state and to link the different components). The translation of the state variables is straightforward, as they also become state variables in *HyDI*. The translation of the flow variables is carried out as follows:

- *Internal assert*: the link between output flow and state variables is expressed by an assertion. In this case the flow variable is represented as a NuSMV *define* on the state variable;
- *In-Out* (Figure 5a): in this case we have a link connecting an input flow of one component with an output flow of another component. In this case the direction is explicitly expressed by the flow labels. This is translated by passing the state variable referred to by the output flow as a parameter to the module translating the component with the input flow;
- *In-In* (Figure 5b): this situation is represented by the direct forwarding of an input flow to a subcomponent. In this case the solution is analogous to the previous case, with the difference that the external component plays the writer role;
- *Out-Out* (Figure 5b): this case is similar to the previous one with the difference that the subcomponent plays the role of writer.

3.4 Priority, synchronization and leaf node translation

Event priorities and *Dirac(x)* laws in Altarica impose a partial order on the firing of the events. We distinguish between events with *Dirac(0)* law (which have higher priority) and events with *Dirac(x)* with $x > 0$[1]. Within each of these two classes, events are ordered by the explicit defini-

[1] Temporal events, i.e. those with *Dirac(x)* law for $x > 0$, in OCAS are given an operational semantics based on event queues and recursive evaluation; in this work, we have used a simplified semantics, that was sufficient for our purposes, and reduces to the original semantics under suitable hypotheses.



(a) Case *In-Out*

(b) Case *In-In* and *Out-Out*

Figure 5: Flow translation cases

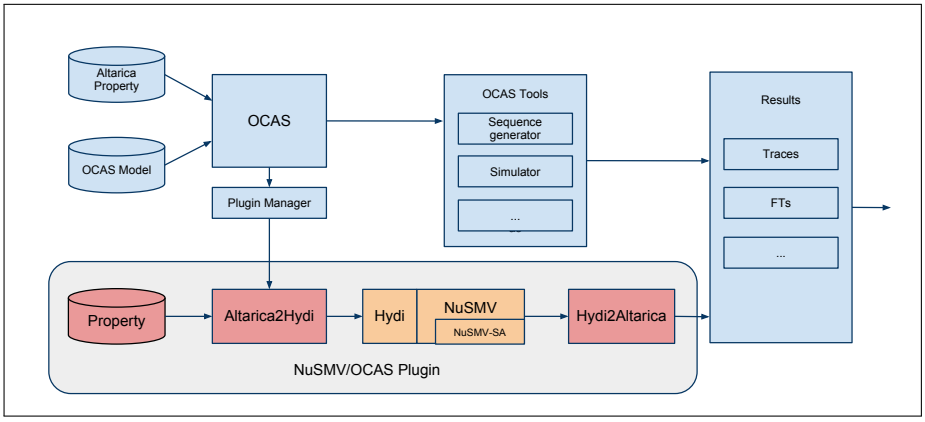


Figure 6: The NuSMV/OCAS plugin and its integration into OCAS

tion of the priority (an integer number). The induced partial order among the events is encoded as a scheduler in the main module of the *HyDI* translation.

The Altarica language permits the definition of three possible kinds of synchronizations between events: strong, weak, and CCF (see Figure 3 and Section 2). *HyDI* has native support for the weak and strong synchronizations, while there is no support for the CCF synchronization. We encode the CCF synchronization taking into account its semantics: a CCF involving two events e_1 and e_2 is either a weak synchronization among e_1 and e_2, or simply event e_1 or event e_2 in isolation. Thus, we duplicate events e_1 and e_2 in e'_1 and e'_2, respectively, to enable for the two events to occur in isolation, and we add a new weak synchronization between e_1 and e_2.

The translation of the leaf nodes is straightforward. Each leaf node maps to an SMV module. Each state variable is encoded into an SMV state variable of the same type. The Altarica init and trans sections directly translate into SMV INIT and TRANS formulas, respectively.

4 Tool Integration and Functionalities

In the following we describe the architecture of the NuSMV/OCAS plugin and its functionalities.

4.1 The NuSMV/OCAS plugin

The NuSMV/OCAS plugin has been developed in Python. It is composed of four main components, as illustrated in Figure 6:

- *Property*: this block provides a GUI to specify the (temporal) properties to be verified and the analysis parameters, and to invoke the verification and safety assessment routines; it extends the 'Altarica property' block, which allows only to compare a variable with a value. In the example of Altarica model presented in Figure 1 OCAS needs an observer that internally evaluates if the output of the adder is the sum of the two counters (see *out_ok*). With our plugin this check is possible directly from the GUI;
- *Altarica2HyDI*: this module is responsible for the translation of the Altarica model into the equivalent *HyDI* specification to be given as input to the extended version of NuSMV (the NuSMV model checker extended with the NuSMV-SA and *HyDI* plugins);
- *HyDI/ NuSMV* : the verification engine;
- *HyDI2Altarica*: this module is responsible for the back conversion of the results generated by NuSMV to a format that can be visualized or executed within OCAS. In particular, it is responsible for the conversion of the traces generated by NuSMV (corresponding to a simulation or to a counterexample to a property) into the *XML* format accepted by OCAS.

The translation from Altarica to *HyDI*, provided by the *Altarica2HyDI* component, is performed in three main steps (see Figure 7):

1. *Parsing*: this module generates an abstract syntax tree (*AST*) of the Altarica design. This module relies on the ANTLR[2] parser generator;
2. *Preprocessing*: this module analyzes the *AST* generated at parsing time to build a new *AST* corresponding to the flattened Altarica model. Moreover, it collects *common information* about the structure of the design, that is re-used in the following steps of the translation;
3. *Translation*: this module, based on the new *AST* and on the structural information previously gathered, generates an in-memory Python structure corresponding to the *HyDI* model. This structure is then dumped into a textual file to be given as input to NuSMV.

The plugin calls NuSMV, waits for the results, and then converts them back into a format that can be imported into OCAS (e.g., simulation traces to be given as input to the sequence generator).

4.2 Functionalities

The NuSMV/OCAS plugin relies on NuSMV, that provides standard BDD-based (CTL and LTL) model checking techniques [McM93], and SAT-based LTL Bounded Model Checking (BMC) techniques [BCCZ99]. It allows one to perform guided and random simulation, and to re-execute partial traces. Moreover, it provides optimized model checking algorithms, developed in the

[2] ANother Tool for Language Recognition (ANTLR), http://www.antlr.org.

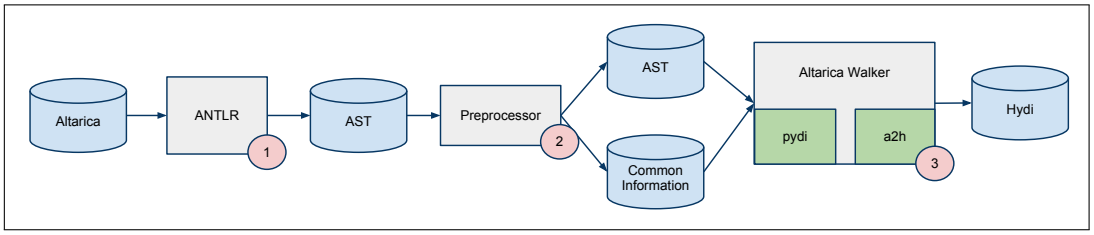


Figure 7: The *Altarica2HyDI* component

MISSA project, which aim at reducing the state explosion problem with techniques that combine BDD and SAT for the verification of invariants. For formal safety assessment the NuSMV/OCAS plugin relies on an extended version of the NuSMV model checker, comprising NuSMV-SA [BV07]. NuSMV-SA allows one to investigate the behavior of a system in degraded conditions (that is, when some parts of the system are not working properly, due to malfunctioning). Key techniques in this area are (dynamic) FTA (Fault Tree Analysis), (dynamic) FMEA (Failure Modes and Effects Analysis), fault tolerance evaluation, and criticality analysis. NuSMV-SA provides advanced and very optimized techniques for the generation of (dynamic) FT and of (dynamic) FMEA tables. NuSMV-SA provides three main engines for safety assessment. The first two are based on classical BDD-based or on SAT-based techniques. The BDD-based engine is complete, but if the model is huge may not scale well. The SAT-based approach is incomplete but allows one to handle very large domains. These two basic approaches are complemented with a third complete approach, developed in the MISSA project, that combines BDD and SAT. It first uses BMC techniques, up to a given depth, to prune the search space, and then it performs an exhaustive analysis on the reduced model using BDD-based model checking algorithms.

5 Experimental Evaluation

5.1 Validation of the translation

As the formal semantics of the Altarica dialect used in OCAS is not fully documented, before starting an experimental evaluation on realistic case studies, we were confronted with the issue of validating the semantics we implemented with respect to the one implemented in OCAS. For the validation we focused on trace simulation generation and trace execution functionalities that are common to both tools. We used several small handcrafted models developed for checking some specific conditions. Then, we used some realistic case studies developed within MISSA.

The validation of the tool was done using the possibility offered by OCAS to re-execute a simulation trace on the Altarica model, using the internal trace simulator. We generated a simulation trace with the NuSMV/OCAS plugin, and then we re-executed it in the OCAS environment. The validation flow we used can be summarized as follows (compare Figure 8):

1. we translate the Altarica model provided by OCAS into *HyDI*, and then into *SMV*;
2. we either verify properties known to be not satisfied, or we generate random simulation traces in order to obtain an execution trace, that we save in the NuSMV *XML* format;
3. we translate the trace provided by NuSMV into the OCAS *XML* format;

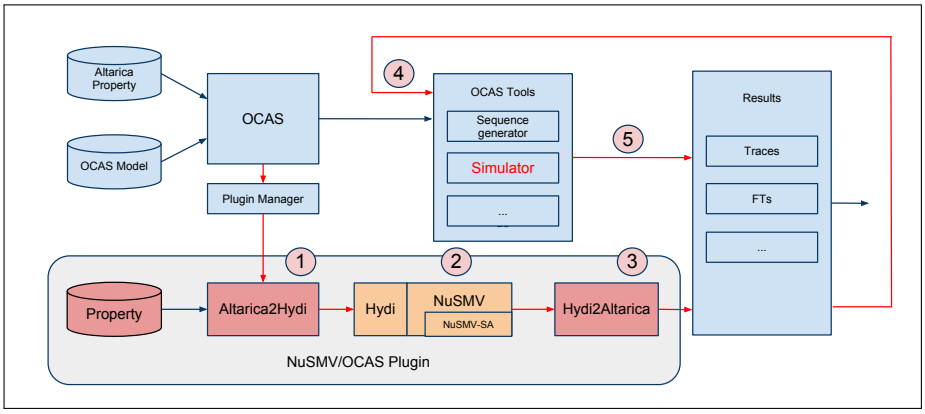


Figure 8: Trace-based validation

4. we load the trace generated in the previous step into the trace simulator of OCAS;

5. we verify that the state reached at the end of the trace execution is compatible with the property, and with the state reported as final in the simulation trace.

Whenever a discrepancy was detected, a thorough analysis of the simulation execution in OCAS was carried out to identify the cause of the discrepancy and - if needed - come up with a fix in the translation to capture OCAS semantics. In a few cases, the behavior shown by OCAS was found to be incorrect by the users, hence not reflected in the translator (cf. Section 1).

5.2 Verification and safety assessment on industrial case studies

In this section we discuss the comparison between the common functionalities provided by the OCAS sequence generator and the NuSMV/OCAS plugin. The sequence generator of OCAS is able to perform Fault Tree Analysis (generation of minimal sequences) up to a bounded depth. For a fair comparison, we then compared this feature with the Fault Tree Analysis provided by NuSMV-SA that relies on the SAT and the mixed BDD+SAT approaches[3].

For the experimental evaluation we used four industrial models developed in MISSA. The ELEC_1, ELEC_2, and ELEC_3 models describe a simplified electrical power distribution system (that resembles that of the A320 aircraft), at different levels of detail. The BRSYS model is a realistic model of the braking system of an aircraft. The properties to be analyzed formalize different failure conditions (e.g., "Loss of deceleration capability during landing" for the BRSYS model). The characteristics of the models are reported in Table 1. This table also shows the time and memory requirements needed to translate the model into an equivalent *HyDI* specification. Note that time and memory increase with the model complexity (however, the translation is performed only once for each given model, whenever several properties have to be verified).

The experimental results are presented in Figure 9. We executed the tests on a laptop equipped with an Intel 3GHz CPU, and with 4GB of RAM running Windows 7. We used a memory limit of 1GB and a timeout of 1000 seconds. The plots report the time needed by OCAS and by the

[3] We also used the NuSMV/OCAS plugin to verify temporal properties of the Altarica design; as this functionality is not available in OCAS, we do not report the results here.

Model	# States	# Nodes	Translation time	Translation memory
ELEC_1	$1.49x10^5$	41	1.127s	27MB
ELEC_2	$2.64x10^5$	44	2.782s	38MB
ELEC_3	$2.0x10^7$	51	2.811s	37MB
BRSYS	$3.8x10^{25}$	135	9.820s	69MB

Table 1: Characteristics of the industrial case studies and translation requirements

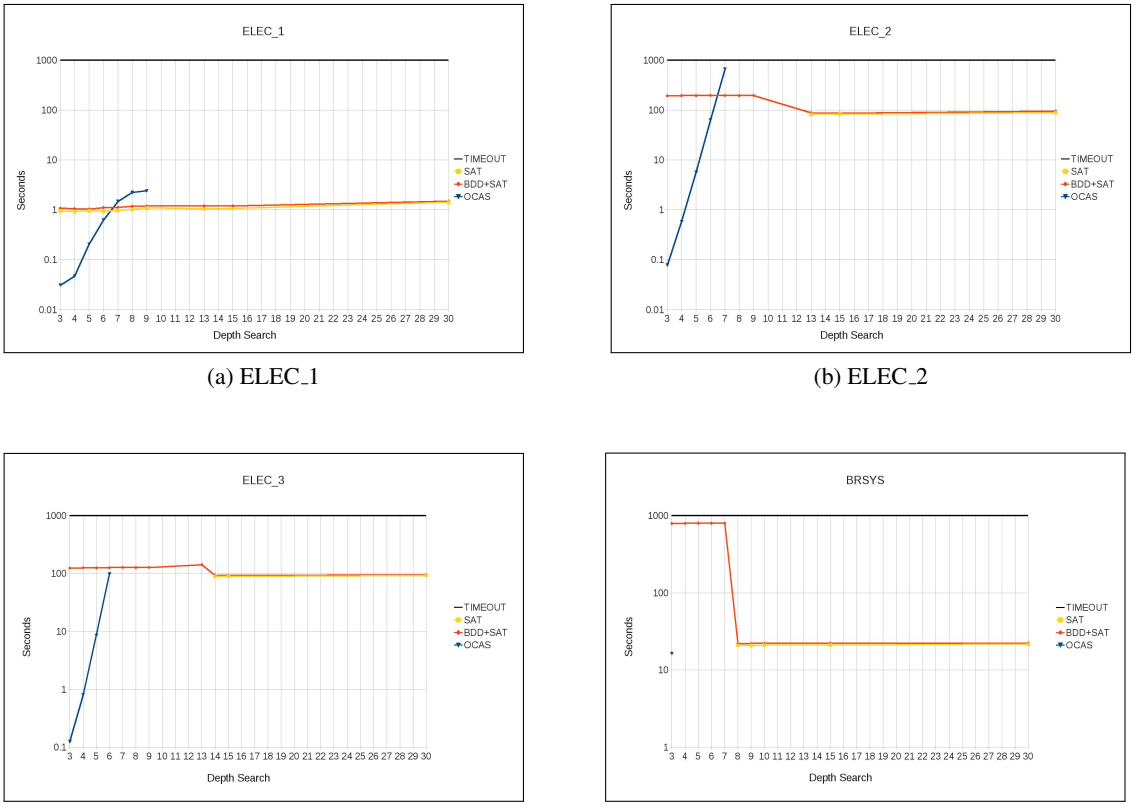
SAT (BMC) and BDD+SAT algorithms provided by the extended version of NuSMV to perform an exhaustive search at increasing depths.

The results on the smallest model (ELEC_1) are reported in Figure 9a. The plots clearly show that the sequence generator is not able to perform the verification with a bound greater than 9, while NuSMV has a behavior nearly independent of the bound. When the complexity grows (models ELEC_2 and ELEC_3, Figures 9b and 9c) OCAS shows a very fast degradation – the sequence generator timeouts with bounds bigger than 7 and 6, respectively – whereas the performance of NuSMV degrades only marginally. The sequence generator performs better than NuSMV for sufficiently low depths – this is due to some internal overhead NuSMV incurs while reading and converting the *HyDI* model, and encoding the verification problem. The results on the BRSYS model (Figure 9d) show a similar trend – OCAS timeouts at depth 3. Notice that the 'step' behavior which is visible in some BDD+SAT plots is due to the fact that for higher depths, SAT may be able to find additional results, that are used to prune the search space before BDD is run. Concerning memory, NuSMV uses up to 36 MB (with bound 30), whereas OCAS allocates up to 100MB (with bound at most 9) on these models. A detailed comparison is difficult, as it is not possible to trace precisely how OCAS uses the allocated memory.

We remark that, in all the examples, the SAT BMC approach outperforms the OCAS explicit state approach by orders of magnitude. This enables analyses that were out of the scope of the previous version of OCAS without the NuSMV/OCAS plugin. Moreover, in all the examples, we were able to run NuSMV to convergence, using the *complete* BDD+SAT approach, with a running time which is only slightly worse than the SAT BMC approach. Being complete, BDD+SAT is guaranteed not to miss cut sets, as a difference with OCAS sequence generator. We also remark that, although not shown in the experimental evaluation, the BDD+SAT approach performed consistently better than the pure BDD approach on these case studies.

6 Related Work

The original language of Altarica, developed by LaBRI, is based on the notion of interfaced constraint automata. A restricted dialect - Altarica Dataflow - was later developed to restrict the complexity of the models and, under certain constraints, permit synthesis of the fault trees [BDRS06, Rau02]. Dialects of Altarica are supported by a number of tools ranging from the academic toolset developed and maintained at the University of Bordeaux [Alt] to SIMFIA [SIM], a modelling, simulation and RAMS analysis environment developed by EADS APSYS - that supports a Dataflow dialect similar to that implemented by OCAS. Another workbench, COMBAVA, has been previously developed by ARBoost Technologies but is now obsolete. To our knowl-



(a) ELEC_1

(b) ELEC_2

(c) ELEC_3

(d) BRSYS

Figure 9: Performance comparison NuSMV vs OCAS

edge, OCAS is the most industrially mature of existing toolsets. OCAS is tightly integrated with Cecilia ARBOR - a Fault Tree Analysis software. Quantitative and Qualitative analysis of fault trees performed in both Cecilia ARBOR and SIMFIA Safety modules are based on Aralia [Rau01]. Whilst there also exists a plugin for synthesis of fault trees (implementing the algorithm of [Rau02]), such functionality is only available for a very restricted subset of Altarica Dataflow.

There are other model checkers that support altarica, in particular MEC 5 [MEC] and Arc [Arc]. MEC 5 is a somewhat outdated model checker that is now superseded by Arc. Arc is a more recent, BDD-based model checker based on the Altarica language, which supports CTL* temporal logics and μ-calculus. Arc is not currently linked to OCAS and the interoperability with a MEC 5 plugin has not been supported in newer versions of OCAS. Moreover, neither Arc nor its predecessor MEC support safety assessment functionalities. Altarica studio [GPV11] is a prototypical toolset, based on Arc, for model-based formal analyses. To our knowledge, safety assessment functionalities are not available in Altarica studio, yet. A thorough comparison of the model checking engines is hard because of differences in the dialects (and flavours thereof) of Altarica supported by the different tools. This work has been focused on (a variant of) Altarica Dataflow - a more extended comparison will be targeted for future work.

7 Conclusions and Future Work

In this work we have presented a novel encoding of Altarica models into NuSMV, which enables verification and safety assessment of Altarica models using state-of-art symbolic model checking and formal safety assessment techniques. We have integrated the encoder as a plugin into the OCAS environment, and we have experimentally demonstrated the feasibility of the approach by evaluating the plugin on a set of industrial case studies. As part of our future work, we plan to address the semantics of Altarica temporal events, which was simplified in the current implementation. Finally, we plan to investigate a timed extension of Altarica, along the lines of [CPR04]. This extension fits very naturally in our framework, given that the *HyDI* language provides a native support for encoding networks of timed (more in general, hybrid) systems.

Acknowledgements: This work has been supported by the E.C. project MISSA, contract no. ACP7-GA-2008-212088. We would like to thank Chris Papadopoulos (Airbus UK), Pierre Bieber and Christel Seguin (Onera), Xavier Leduc and Valerie Sartor (Dassault Aviation), Laurent Sagaspe (EADS APSYS) and Antonella Cavallo (Alenia Aeronautica) for their precious support and advice for the development, integration and evaluation of the plugin. The ELEC models used for evaluation have been originally developed by ONERA and, in some cases, expanded by Alenia Aeronautica. The BRSYS model has been developed by EADS APSYS.

Bibliography

[ÅBB06] O. Åkerlund, P. Bieber, E. Böede et al. ISAAC, a framework for integrated safety analysis of functional, geometrical and human aspects. In *Proc. ERTS*. 2006.

[AGPR00] A. Arnold, A. Griffault, G. Point, A. Rauzy. The AltaRica formalism for describing concurrent systems. *Fundamenta Informaticae* 40:109–124, 2000.

[Alt] The Altarica language. `http://altarica.labri.fr/forge`.

[Arc] The Arc model checker. `http://altarica.labri.fr/forge/projects/arc/wiki`.

[Ba03] M. Bozzano, et al. ESACS: An Integrated Methodology for Design and Safety Analysis of Complex Systems. In *Proc. ESREL*. Pp. 237–245. Balkema Publisher, 2003.

[BBC+04] P. Bieber, C. Bougnol, C. Castel, J.-P. Christophe Kehren, S. Metge, C. Seguin. Safety Assessment with Altarica. In *Building the Information Society*. IFIP International Federation for Information Processing 156, pp. 505–510. 2004.

[BCCZ99] A. Biere, A. Cimatti, E. M. Clarke, Y. Zhu. Symbolic Model Checking without BDDs. In *Proc. TACAS*. LNCS 1579, pp. 193–207. Springer, 1999.

[BCK+10] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, M. Roveri. Safety, Dependability, and Performance Analysis of Extended AADL Models. *The Computer Journal* doi: 10.1093/com, March 2010.

[BCS02] P. Bieber, C. Castel, C. Seguin. Combination of Fault Tree Analysis and Model Checking for Safety Assessment of Complex System. In *Proc. EDCC-4*. LNCS 2485, pp. 19–31. Springer, 2002.

[BDRS06] M. Boiteau, Y. Dutuit, A. Rauzy, J.-P. Signoret. The AltaRica Data-Flow Language in Use: Modelling of Production Availability of a Multi-State System. *Reliability Engineering and System Safety* 91(7):747–755, 2006.

[Bry92] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys* 24(3):293–318, 1992.

[BV07] M. Bozzano, A. Villafiorita. The FSAP/NuSMV-SA Safety Analysis Platform. *Software Tools for Technology Transfer* 9(1):5–24, 2007.

[BV10] M. Bozzano, A. Villafiorita. *Design and Safety Assessment of Critical Systems*. CRC Press (Taylor and Francis), an Auerbach Book, 2010.

[CMT11] A. Cimatti, S. Mover, S. Tonetta. HYDI: a language for symbolic hybrid systems with discrete interaction. Technical report, Fondazione Bruno Kessler, 2011. https://es.fbk.eu/people/mover/hydi

[CPR04] F. Cassez, C. Pagetti, O. Roux. A timed extension for AltaRica. *Fundamenta Informaticæ* 62(3–4):291–332, 2004.

[FSA] The FSAP/NuSMV-SA platform. `http://es.fbk.eu/tools/FSAP`.

[GPV11] A. Griffault, G. Point, A. Vincent. Altarica-studio : the easier way to do model checking. In *Proc. MBSAW 2011*. 2011.

[Mat11] C. Mattarei. Definizione e sviluppo di una traduzione formale da Altarica ad Hydi per la verifica di sistemi avionici. Master's thesis, Università degli studi di Trento, 2011.

[McM93] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[MEC] The MEC model checker. `http://altarica.labri.fr/forge/projects/mec/wiki`.

[MIS] The MISSA Project. http://www.missa-fp7.eu.

[NuS] The NuSMV model checker. `http://nusmv.fbk.eu`.

[Rau01] A. Rauzy. Mathematical Foundations of Minimal Cutsets. *IEEE Transactions on Reliability* 50(4):389–396, 2001.

[Rau02] A. Rauzy. Mode Automata and Their Compilation into Fault Trees. *Reliability Engineering and System Safety* 78(1):1–12, 2002.

[SIM] SIMFIA. `http://www.apsys.eads.net/en/17/Software`.

Mixing Formal and Informal Model Elements for Tracing Requirements

Michael Jastram[1], **Stefan Hallerstede**[2], **Lukas Ladenberger**[3]

[1] michael@jastram.de
[2] stefan.hallerstede@wanadoo.fr
[3] lukas.ladenberger@gmx.de

Institut für Softwaretechnik und Programmiersprachen
Heinrich-Heine Universität Düsseldorf, Germany

Abstract: Tracing between informal requirements and formal models is challenging. A method for such tracing should permit to deal efficiently with changes to both the requirements and the model. A particular challenge is posed by the persisting interplay of formal and informal elements.

In this paper, we describe an incremental approach to requirements validation and systems modelling. Formal modelling facilitates a high degree of automation: it serves for validation and traceability. The foundation for our approach are requirements that are structured according to the WRSPM reference model. We provide a system for traceability with a state-based formal method that supports refinement. We do not require all specification elements to be modelled formally and support incremental incorporation of new specification elements into the formal model. Refinement is used to deal with larger amounts of requirements in a structured way.

We provide a small example using Problem Frames and Event-B to demonstrate our approach.

Keywords: Requirements, WRSPM, Event-B, Rodin, ProR

1 Introduction

We describe an approach for incrementally building a formal model from structured informal specifications providing a means of requirements validation. Our approach does not require all specification elements to be modelled formally, and the resulting system description provides traceability to both formal and informal model elements. The traceability allows us to detect which requirements are affected if the system implementation changes, and vice versa. Most elements of the structured specification are still stated in natural language. Our aim is to increase the confidence that the formal model represents what has been specified, and to ensure that specification elements that do not have a formal representation are validated at a different stage of the development by informal reasoning and tracing.

We identified the WRSPM reference model [GJGZ00] as the foundation for the informal structured specification. Many concrete approaches are consistent with this reference model, e.g., [Jac01, PM95]. A specification following the WRSPM approach can still be understood by stakeholders, while providing a good foundation for formalisation. These approaches define *phenomena* which describe the state space of the system and its environment, as well as artefacts that represent constraints on the state space and the state transitions. This structure makes a traceability to a state-based formalism doable.

A distinguishing feature of our approach is the incremental modelling of the specification using refinement, which the chosen formalism must support. Once modelled formally, the potential for automated verification is high. This is particularly useful for change management and requirements evolution, which are both important aspects for real-world systems. Also, we allow specification elements without formal representation. Those elements must be justified informally using techniques suggested in [Jac01], for instance.

1.1 Structure of this Paper

In the remainder of this section, we will provide a brief foundation of requirements and specifications, as well as state-based modelling. In Section 2 we present our main thesis, the traceability between formal and informal specification and model. We deepen the aspect of formal refinement in Section 3.

In Section 4, we provide a small example to demonstrate various aspects of our approach. The example uses the Problem Frames approach and the Event-B formal method.

We are actively working on tool support and present our progress in Section 5.

After describing some of the related work in Section 6, we conclude in Section 7, which also contains an outlook on future work.

1.2 Requirements and Specification

Our approach is based on WRSPM by Gunter et. al. [GJGZ00]. WRSPM is a reference model for applying formal methods to the development of user requirements and their reduction to a behavioural system specification.

WRSPM distinguishes between artefacts and phenomena. Phenomena describe the state space (and state transitions) of the domain and system, while artefacts represent constraints on the state space and the state transitions. The artefacts are broadly classified into groups that pertain mostly to the system versus those that pertain mostly to the environment. These are:

Domain Knowledge (W) describes how the world is expected to behave.

Requirements (R) describe how we would like the world to behave.

Specifications (S) bridge the world and the system.

Program (P) provides an implementation of S.

Programming Platform (M) provides an execution environment for P.

We distinguish phenomena by whether they are controlled by the system (belonging to set s) or the environment (belonging to set e). They are disjoint ($s \cap e = \varnothing$), while taken together, they represent all phenomena in the system ($s \cup e =$ "all phenomena"). Furthermore, we distinguish them by visibility. Environmental phenomena may be visible to the system (belonging to e_v) or hidden from it (belonging to e_h). Correspondingly, system phenomena belonging to s_v are visible to the environment, while those belonging to s_h are hidden from it. These classes of phenomena are mutually disjoint.

The distinction between environment and system is an important one; omitting it can lead to misunderstandings during the development. It is sometimes regarded as a matter of taste or convenience where the boundary between environment and system lies, but it has a profound effect on the problem analysis. It clarifies responsibilities and interfaces between the system and the world and between subsystems. If we require ourselves to explicitly make that distinction, we can avoid many problems at an early stage.

W and R may only be expressed using phenomena that are visible in the environment, which is $e \cup s_v$. Likewise, P and M may only be expressed using phenomena that are visible to the system, which is $s \cup e_v$. S has to be expressed using phenomena that are visible to both the system and the environment, which is $e_v \cup s_v$.

Once a system is modelled following WRSPM, a number of properties can be verified with regard to the model, one being *adequacy with respect to S*:

$$\text{FOR ALL } e\ s,\ W \text{ AND } S \text{ IMPLY } R \qquad \text{(Adequacy)}$$

This simply says that the specification constrains the world such that the requirements are realized. Obviously we are not interested in the trivial solution to (Adequacy), meaning that no e and s exist to satisfy (Adequacy).

We demonstrate our ideas using Problem Frames [Jac01], which is a concrete approach to software requirements analysis that is a manifestation of the WRSPM reference model. The central element in problem frames is the problem diagram that consists of exactly one machine domain, designed domains and given domains.

1.3 State-Based Modelling and Refinement

Our approach could be used with a wide range of formal methods for state-based modelling that have an associated notion of refinement. We find state-based formalisms such as ASM [BS03], VDM [Jon90], TLA+ [Lam02] or Event-B [Abr10] particularly suited because they permit straightforward specification of state, state invariants and state transitions for modelling dynamic behaviour. In this paper, we focus on state-based modelling and provide an example using Event-B, which we introduce in Section 3.1. Event-B is suitable for discussing the example that we introduce in Section 4. Using Event-B we can also discuss limitations of requirements tracing: not all requirements can be formalised within the core Event-B formalism. Formal and informal reasoning need to be combined in a sensible way. The boundary of formalisation in the example is given by temporal and real time properties. We have intentionally chosen a boundary that could be moved by using another formal method or extending Event-B because we think it is not fixed and may change depending on project characteristics. It also serves to illustrate that the boundary may be moved as a development progresses. We think of modelling and requirements validation as an incremental process: we permit the boundary to be moved as need arises.

We take advantage of the concept of refinement which is supported by Event-B. Other notions of refinement could be used without changing the approach fundamentally. Our approach allows us to account for additional requirements at later refinement stages, thereby providing a structuring mechanism for the introduction of requirements into the formal model.

2 From Formal to Informal and Back

The requirements engineering process can be broken down into requirements specification, system modelling, requirements validation and requirements management [Wie03].

We will briefly describe these process steps (shown in Figure 1) and how they relate to the work in this paper.

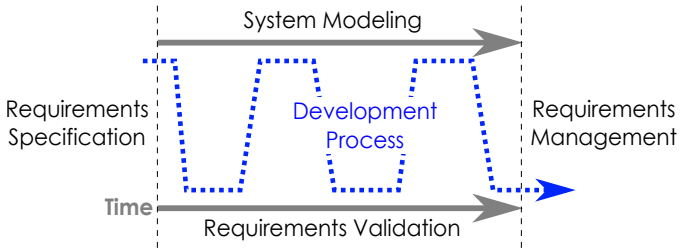


Figure 1: The Incremental Development Process

Requirements Specification – The requirements are structured according to the approach of choice, resulting in a specification that follows the WRSPM reference model. In the example that we introduce in Section 4, we use the Problem Frames approach.

System Modelling – The objective of this phase is the formal modelling of elements. Not all elements need to be modelled formally, which is one distinguishing feature of our approach. Also, this step can and should be performed incrementally. In general, any formalism can be used. The nature of the problem to be solved may suggest one formalism over another. In particular, it may be useful to select a formalism that makes it easy to model the safety critical aspects of the specification. It is also possible to user more than one formalism. In the example in Section 4, we use Event-B.

Requirements Validation – The validation of the requirements is a central aspect of this paper and is described in detail in Section 2.1.

Requirements Management – In practice, a specification is never "done". The ongoing work includes change management and requirement evolution. These tasks are supported by our approach. The amount of formality determines how effective this is. At one end of the spectrum, all elements are modelled formally, allowing us to prove (Adequacy). On the other end of the spectrum is an informal description.

These tasks, including elicitation, analysis and negotiation, are performed in parallel. We do not want to create the impression that this is a sequential process.

2.1 Requirements Validation

System modelling provides us with partly formalised elements as described by the requirements. We think of system modelling as an incremental process where more and more is formalised. However, we do not assume that necessarily everything is formalised. The methodology we propose allows for a mixture of formal and informal proof as a means of validation. As a consequence of frequent incremental changes we need effective support for tracing requirements: formal models change as they incorporate increasing detail, requirements change as a consequence of the validation itself. The transition to requirements management is considered fluent and the same techniques of traceability are applied.

Demonstrating (Adequacy) now involves dealing with formal and informal elements. In the following, we designate by Rf the formal requirements, by Wf the formal domain properties and by Sf the formal specification elements. The difference $R \setminus Rf$ of all requirements and formal requirements gives the informal requirements Ri, similarly for informal domain properties Wi and informal specification elements Si.

For the formal elements we can formally verify that

$$\forall e\ s \cdot Wf \wedge Sf \Rightarrow Rf\ , \tag{1}$$

assuming that sufficient of W and S have been formalised to cover Rf. For informal elements we allow informal arguments, for instance, of the kind used in the problem frames approach [Jac01] or not formalised mathematical proofs. Doing this, we show:

$$\text{FOR ALL } e\ s,\ W \text{ AND } S \text{ IMPLY } Ri\ . \tag{2}$$

We permit also using formal elements in the antecedent of (2) but only formal elements in the antecedent of (1). As many critical requirements as possible should be validated formally, giving high assurance of their satisfaction. Relying on formally verified facts in informal justification will also improve their quality.

2.1.1 Formal Tracing

To formalise artefacts A they need to be of a form that can be "translated" into a formula F so that we can state

$$A \text{ EQUIVALES } F\ . \tag{3}$$

This makes tracing from F to A and vice versa trivial. Formal proofs of (1) can provide information about which formal artefacts are used in order to validate specific requirements. Among others, this has been implemented in the proof support of the Rodin tool [ABH+10]. If formal artefacts $F_1, \ldots, F_k$ have been used to prove formal requirement Rf_n from $Wf \wedge Sf$, then we know that a change of the informal requirement R_n that equivales Rf_n affects the informal artefacts $A_1, \ldots, A_k$. The formal model provides a way to validate requirements rigorously and an efficient way to trace dependencies between informal artefacts. The latter is crucial for the maintenance of large numbers of requirements occurring in industrial practice. Support by proof tools means that this tracing can be automated to a large degree.

2.1.2 Informal Tracing

Artefacts that are not formalised can still be traced but the dependencies can only be checked manually by inspecting informal arguments. Changes of involved artefacts require corresponding human intervention. A known technique to limit the impact of changes is the identification of a *satisfaction base* [KJ10] for each informal artefact of Ri. A satisfaction base for a requirement R_n consist of those artefacts from S and W that are sufficient to justify it. Using the concept of a satisfaction base, (2) can be rephrased as

$$\text{FOR ALL } e\ s,\ SB(R_n) \text{ IMPLY } R_n\ . \tag{4}$$

where $SB(R_n)$ is a subset of W and S, representing a satisfaction base for the given requirement. The satisfaction base is used in the informal justification and for tracing dependencies, similarly to formal tracing. However, possibilities for automation are very limited. Also note that there may be multiple satisfaction bases.

3 Formal Refinement

Formula (1) can grow very large for a complex model. This can make it very difficult to verify any interesting property but also to compute a sufficiently small set of formal artefacts that are used to verify specific formal requirements Rf_n. Formal refinement alleviates this problem by introducing parts of the overall model in small increments. The original WRSPM approach sketch a notion of implementation based on the program P and the programming platform M:

$$\text{FOR ALL } e\ s,\ W \text{ AND } P \text{ AND } M \text{ IMPLY } R\ . \tag{5}$$

This can be achieved by relying on implication for implementation (see, e.g., [HJ98, Heh93, GJGZ00]),

$$\text{FOR ALL } e_v\ s,\ P \text{ AND } M \text{ IMPLY } S \tag{6}$$

providing a simple notion of refinement in a predicative specification style. Instead of formalising the refinement notion (6) we prefer a notion based on discrete transition systems that permits more direct specification of dynamic aspects of a model. For the purposes of this article we do not consider details of M such as the targeted programming language. We consider S as a collection of invariants and transitions of a discrete transition system which we specify by means of Event-B [Abr10]. The choice of Event-B over similar methods [BS03, Jon90] is mostly motivated by the built-in formal refinement support and the availability of a tool [ABH+10] for experimentation with our approach.

3.1 Event-B Machines

Event-B is a state-based modelling method whose models are characterised by *proof obligations*. Proof obligations serve to verify properties of the models. To a large degree, such properties originate in requirements that the model is intended to realise. Eventually, we expect that by verifying the formal model we have also established that the corresponding requirements are satisfied.

We only provide a brief summary of simplified Event-B in terms of proof obligations. A complete description can be found in [Abr10]. Variables v define the state of a machine. They are constrained by invariants $I(v)$. Possible state changes are described by means of events. Each event

$$\text{any } t \text{ when } G(t,v) \text{ then } x := E(t,v) \text{ end}$$

is composed of *parameters* t, a *guard* $G(t,v)$ and an *action* $x := E(t,v)$, where x are variables of the machine. The guard states the necessary condition under which an event may occur, and the action describes how the state variables evolve when the event occurs. Actions $x := E(t,v)$ are

characterised by before-after predicates $x' = E(v) \wedge y' = y$, where y are the remaining variables of the machine. In the presentation of the proof obligations we assume actions are of the form $v := E(t,v)$. The before-after predicate of an event is the formula $G(t,v) \wedge v' = E(t,v)$. A dedicated event $v := E_{\text{init}}$ without parameters or guards is used for initialisation. (The expression E_{init} also does not refer to any variables.)

3.2 Event-B Proof Obligations

In Event-B two main properties are proved about formal models: consistency and refinement.

Consistency. Consistency means that the invariant $I(v)$ is established by the initialisation

$$I(E_{\text{init}})$$

and maintained

$$I(v) \wedge G(t,v) \Rightarrow I(E(t,v))$$

by all other events of the machine. Usually, invariants are conjunctions, e.g., $I(v) = I_1(v) \wedge \ldots \wedge I_n(v)$. Hence, it suffices to prove $I(v) \wedge G(t,v) \Rightarrow I_j(E(t,v))$ for all $j \in 1..n$. The smaller predicates are easier to relate to informal artefacts and easier to trace in case artefacts correspond to theorems derived from the invariants.

Refinement. Refinement links abstract events to concrete events aiming at the preservation of properties of the abstract event when it is replaced by the concrete event. A concrete event with guard $H(u,w)$ and action $w := F(u,w)$ refines an abstract event with guard $G(t,v)$ and action $v := E(t,v)$ if, whenever the gluing invariant $J(v,w)$ is true:

i. the guard of of the concrete event is at least as strong as the guard of the abstract event, and
ii. for every possible execution of the concrete event there is a corresponding execution of an abstract event which simulates the concrete event such that the gluing invariant remains true after execution of both events.

Formally,

$$I(v) \wedge J(v,w) \wedge H(u,w) \Rightarrow \exists t \cdot G(t,v) \wedge J(E(t,v), F(u,w)) \ .$$

For initialisation we have to prove: $J(E_{\text{init}}, F_{\text{init}})$. To match the refinement notion of WRSPM described in Section 3 we have to void data-refinement where a variable is replaced by another. We think data-refinement could eventually serve to deal with abstractions of phenomena where in more abstract problem frame descriptions phenomena are bundled. The Event-B method derives proof obligations from these two properties that are easier to handle and can be efficiently generated by a tool [Abr06]. In particular, the conclusion is decomposed into small parts. To achieve this witnesses $t = T(u,w)$ for t are introduced for instantiating the existentially bound identifiers:

$$I(v) \wedge J(v,w) \wedge H(u,w) \Rightarrow G(T(u,w),v) \wedge J(E(T(u,w),v), F(u,w)) \ .$$

Usually, guards and (gluing) invariants are conjunctions and the proof obligation can be decomposed similarly to the consistency proof obligation above.

3.3 Tracing of Requirements with Event-B

The Event-B model contains formal artefacts as indicated by (3). The domain properties *Wf* and specification elements *Sf* can be represented by means of events and invariants. By consistency and refinement we get a collection of invariants *IA* that are preserved by all events *EA*. We can now partition events and invariants according to the artefacts they represent: $IA = IW \cup IS$ and $EA = EW \cup ES$. Making this distinction is standard in the Event-B method.[1] To fit into the shape

[1] Using model decomposition we could now decompose the two parts and focus on the refinement of the sub-model consisting of *IS* and *ES*. The interface to the other sub-model would act as a contract guaranteeing overall consistency.

of WRSPM adequacy we consider the before-after predicates of all events and identify $Wf = IW \wedge BA(EW)$ and $Sf = IS \wedge BA(ES)$, where $BA(EE)$ yields the disjunction of the before-after predicates of the events EE. In formal refinement Pf the formal program is usually considered a subset of Sf that is being gradually constructed during refinement. After some refinement steps we have $Pf = IS \wedge IP \wedge BA(EP)$ where the events EP are refinements of the events ES. Hence, $Pf \Rightarrow Sf$ by choosing suitable witnesses, obtaining the formal counterpart of (6). We have identified the formal domain properties Wf, specification element Sf and program Pf.

We can now turn to the formal requirements Rf, formal adequacy (1) and the formalised (5) not taking account of the programming platform M:

$$\forall e\ s \cdot Wf \wedge Pf \Rightarrow Rf\ . \tag{7}$$

Assuming we already have verified (1), adequacy of the implementation (7) follows by the discussion of the preceding paragraph, using

$$\forall s \cdot Pf \Rightarrow Sf\ . \tag{8}$$

Refinement allows this to be applied incrementally to deal with small more manageable sets of artefacts at each refinement step. Gradually, the set of satisfied refinements is extended until all requirements are covered,

$$\varnothing = Rf^0 \subseteq Rf^1 \subseteq Rf^2 \subseteq \ldots \subseteq Rf^n = Rf\ , \tag{9}$$

where the Rf^i correspond to the refinement steps of the model. Most of these refinement steps will involve the domain properties and specification elements:

$$\forall e\ s \cdot Wf^{i+1} \wedge Sf^{i+1} \Rightarrow Wf^i \wedge Sf^i\ . \tag{10}$$

Refinement steps for implementing the program will usually be less related to requirements. The refinement method, however, does not make a particular distinction between the two uses of refinement. Each refinement step can be used to verify adequacy of the specification gradually:

$$Wf^i \wedge Sf^i \Rightarrow Rf^i \setminus Rf^{i-1}\ . \tag{11}$$

Refinement theory guarantees that adequacy validated in earlier refinement steps is preserved. After n refinement steps (1) is verified.

Formula (11) suggests a method of stepwise tracing of requirements following the refinements. Often requirements can be identified with invariants, event guards or actions. In this case (11) holds trivially. Sometimes theorems can be stated [HL09] that are implied by the invariants. In this article we limit tracing to this level. However, this is not a fundamental limitation of the approach. For instance, one could also permit temporal formulas derived from $Wf \wedge Sf$ as supported by TLA+. Some of TLA+ is also implemented in the ProB tool [LB08] that has been integrated with the Rodin tool that we use. But for this article we contend ourselves with a less expressive notation relying only on invariants and possible transitions.

Problem frame diagrams do not use refinement, but techniques of decomposition like projection. They serve for structuring large sets of requirements. They correspond to the last refined model just before turning to implementation (by means of P). The problem frame diagram will always contain the entire set of formal and informal requirements R. We do not intend to extend the idea of refinement from Event-B to problem frames in this paper.

4 Example: A Traffic Light Controller

We are going to demonstrate the approach presented here by creating the model of a traffic light system that allows pedestrians to cross a street. We already introduced this example in [JHLJ10]. The system consists of two traffic lights for pedestrians (one on each side of the street), two

corresponding traffic lights for the cars, and push buttons for the pedestrians to request a green light for crossing the street.

We consider this example useful, because it is simple enough to understand, but complex enough to be interesting. Further, the example concerns state (which we model formally) as well as real-time (which we specify informally), allowing us to demonstrate the mixing of formal and informal modelling elements. In the following, we only present the interesting aspects of the example.

4.1 Requirements Specification

Following our approach, we would apply a specification approach of choice, in this case Problem Frames. This may lead to a the problem diagram shown in Figure 2. The Problem Frames diagram is incomplete. For instance, information regarding the temporal properties of the system are missing. This is by design, as the problem diagram only depicts the contextual aspects of the model and their relationships in the form of shared phenomena. The textual representation is still the central repository for all information regarding the system. This leads to a new natural language specification, shown in Table 1. In the table, the phenomena are highlighted. The vocabulary is managed in a separate glossary (Table 2).

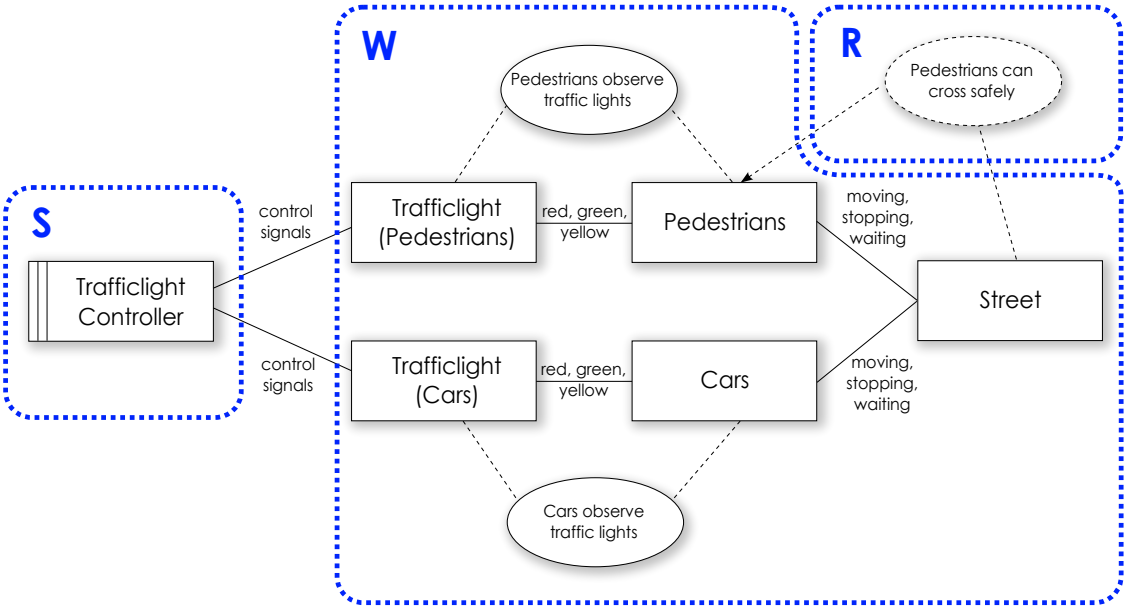


Figure 2: A simplified Problem Frames diagram for the traffic light problem

Note that it can be useful to introduce an informal notion of refinement already in the textual description of the system to structure it. We see that in the description of the traffic light states, that are sometimes referred to abstract as the abstract *stop* and *go*, and sometimes as the concrete colours *red*, *yellow* and *green*. We can take advantage of this in the modelling phase by establishing abstract properties that are simple and easy to trace. The refinement concept of Event-B allows us to introduce the concrete colours later on, while preserving the original properties (assuming correct data refinement), as demonstrated in Section 4.3.

The specification in Table 1 is already more precise than the original requirements, while still comprehensible by the stakeholders. We already identified items as *R*, *W* and *S*. This makes it easier to reason about the model. It also allows us to identify the proper role for validating or justifying each artefact: Stakeholders are concerned with *R*, domain experts with *W* and designers with *S*.

R-2.1	*Pedestrians* can cross safely. They are crossing when they are not *waiting*.
W-2.1	*Pedestrians* observe the traffic lights (tl_{peds}). This means that they may move (*moving*) when the traffic lights allows them to *go*. Upon indicating *stop*, they finish moving (*stopping*) and then wait (*waiting*).

W-2.2	*Cars* observe the traffic lights (tl_{cars}). This means that they may move (*moving*) when the traffic lights allows them to *go*. Upon indicating *stop*, they finish moving (*stopping*) and then wait (*waiting*).
W-2.3	*stopping* of *Pedestrians* takes time.
W-2.4	*stopping* of *Cars* takes time.
S-2.1	The traffic lights for pedestrians (tl_{peds}) and cars (tl_{cars}) never indicate *go* at the same time.
S-2.2	tl_{cars} must wait for a certain time ($delay_{cars}$) before switching to *go* after tl_{peds} turned to *stop*.
S-2.3	$delay_{peds}$ is 3 seconds ($\pm$ 100ms).
S-2.4	tl_{peds} must wait for a certain time ($delay_{peds}$) before switching to *go* after tl_{cars} turned to *stop*.
S-2.5	$delay_{cars}$ is 3 seconds ($\pm$ 100ms).

Table 1: Requirement, Domain Assumptions and Specification of a Traffic Light System (partial)

Pedestrians (e_h)	are modelled as *moving*, *stopping* or *waiting*.
Cars (e_h)	are modelled as *moving*, *stopping* or *waiting*.
tl_{peds} (s_v)	Traffic lights for pedestrians, modelled as *go* and *stop*.
tl_{cars} (s_v)	Traffic lights for cars, modelled as *go* and *stop*.
$delay_{peds}$ (e_v)	is modelled as an event that delays for 3 seconds after tl_{peds} turns from *go* to *stop*.
$delay_{cars}$ (e_v)	is modelled as an event that delays for 3 seconds after tl_{cars} turns from *go* to *stop*.
go (s_v)	is the state of a traffic light where only the *green* lamp is on.
stop (s_v)	are all states of a traffic light that are not *go*.

Table 2: The Glossary (partial)

4.2 System Modelling

We decided to use the Event-B formalism (Section 3.1), making it easier to model some aspects of the model and more tricky to model others. In particular, it is easy to express safety properties like R-2.1, more difficult to express state transition properties like S-2.2, and almost impossible to express real-time properties like S-2.3.

Following the incremental approach described in Section 2, we start with the safety requirement R-2.1, for which a state-based formalism like Event-B is well-suited.

$$Pedestrians \neq waiting \Rightarrow Cars = waiting \tag{12}$$

Not all properties can be modelled as easily as R-2.1. For instance, the behaviour of pedestrians (W-2.1) cannot be represented by an invariant. Instead, we can model it according to the approach described in Section 3.3 by representing it as a before-after predicate of an event. The property W-2.1 doesn't have the proper granularity for this approach, so we rewrite it to specify each transition separately. This rewrite is part of the incremental specification process, and the result must be validated with the domain experts.

W-2.1 (a) *Pedestrians* that are *moving* can only change their state to *stopping*.

W-2.1 (b) *Pedestrians* that are *stopping* can only change their state to *waiting*.

W-2.1 (c) *Pedestrians* that are *waiting* can only change their state to *moving*.

W-2.1 (d) *Pedestrians* may only change to *moving* if tl_{peds} indicates *go*.

W-2.1 (e) If tl_{peds} indicates *stop*, then *Pedestrians* must change to *stopping* if they are *moving* and change to *waiting* if they are *stopping*.

Rewritten like this, it can be modelled in Event-B as follows:

Event *peds_moving_to_stopping* $\widehat{=}$
when
$\mathtt{W-2.1a}$: $Pedestrians = moving$
then
$\mathtt{W-2.1a}$: $Pedestrians := stopping$

Event *peds_stopping_to_waiting* $\widehat{=}$
when
$\mathtt{W-2.1b}$: $Pedestrians = stopping$
then
$\mathtt{W-2.1b}$: $Pedestrians := waiting$

Event *peds_waiting_to_moving* $\widehat{=}$
when
$\mathtt{W-2.1c}$: $Pedestrians = waiting$
$\mathtt{W-2.1d}$: $tl\_peds = go$
then
$\mathtt{W-2.1c}$: $Pedestrians := moving$

Note how we could establish a clear traceability according to (3). The exception is W-2.1e, which is difficult to model in Event-B. Event-B allows us to enforce that something does *not* happen (via a guard), but difficult to guarantee that something does happen (implying that all events except one are disabled). The missing traceability to W-2.1e reminds us that this property must be justified outside this formal model. This could be done by reasoning, testing, or with a different formalism like temporal logic.

This justification may be invalidated if the source or target of the traceability relationship changes. Thus, it has to be verified after each such change. A tool may support this by invalidating that relationship if either of the elements involved changes.

The reader may have noticed that the above represents a state machine. It could be useful to develop an approach specific to state machines.

4.3 Data Refinement

In Section 3.2, we described how consistency is maintained across refinement levels. We will demonstrate this concept by showing how the traffic light states *stop* and *go* are transformed via data refinement into *red*, *yellow* and *green*.

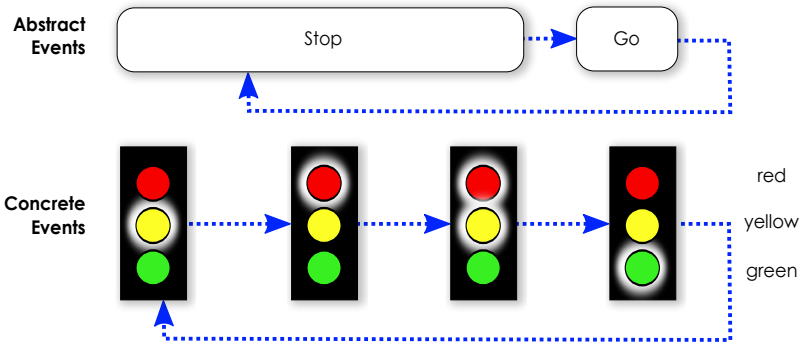


Figure 3: Data Refinement of the Traffic Light States

Data refinement allows us to state abstract properties in a concise way, while the implementation details are addressed later. This allows us to reason about some fundamental properties. Consider S-2.1 as an example of such a property. By arguing simply about *stop* and *go*, the safety property can be stated in a very concise way. The detail on how *stop* and *go* are realised (through colours), can be provided later. Carrying the notion of refinement to the requirements allows us to write more concise requirements: In this case, we can separate the safety requirement from the actual representation of traffic light states, which is also a requirement, but a different one.

There are other situations where this approach can be exploited: For product lines, some abstract properties could be realised in different concrete implementations. In this example, *stop* and *go* could be signalled with a barrier, as found in railroad crossings. A carefully crafted abstraction would therefore support the automated verification of different concrete implementations.

We can model S-2.1 formally as follows:

$$\neg(tl_{peds} = go \wedge tl_{cars} = go) \tag{13}$$

The definition of *stop* and *go* in terms of colours was already given in Table 2, leading to the following gluing invariant that can be introduced in a new refinement:

$$tl_{peds} = go \Leftrightarrow colors_{peds} = \{green\} \tag{14}$$

Introducing (14) into the model results in non-discharged proof obligations, as the newly introduced gluing invariant will be violated without any further modifications. The abstract events that control the traffic light's *stop* and *go* states must also be refined into concrete events that cycle through the corresponding colour states, as shown in Figure 3.

The refinement will take on a similar form as the Event-B shown in Section 4.2, where each state transition corresponds to one event. The proof obligations will ensure that the safety requirement (13) is not violated once they are all discharged, assuming that the gluing invariant (14) is modelled correctly. Discharging all proof obligations will require additional guards.

4.4 Adding Requirements with Refinement

Another application of refinement is the gradual inclusion of formal requirements into subsequent refinements, as hinted at in (9). In the traffic light example, this can be demonstrated by adding a push button for the pedestrians, allowing them to request crossing the street.

Table 3 shows the structured requirements and their formal representations:

R-2.2	*Pedestrians* can *request* to cross any time.
S-2.6	Upon switching of *tl_peds* from *go* to *stop*, the *request* is reset.
S-2.7	*Pedestrians* must not wait longer than 60 seconds for permission to cross after issuing the *request*.

Table 3: Requirement and Specification for allowing Pedestrians to Request Crossing the Street

These two properties can be incorporated into the model in a separate refinement with a new event and the extension of an existing event with a straightforward traceability, as shown in the following:

Event *request_crossing* $\widehat{=}$
  when
    R − 2.2 : $request := TRUE$
  end
Event *set_tl_peds_go* $\widehat{=}$
extends *set_tl_peds_go*
  when
    S − 2.6 : $request := FALSE$

The requirement S-2.7 cannot be modelled formally as stated. This informal artefact simply has to be verified outside the formal model. We could break down S-2.7 further to model some aspects formally (e.g. by introducing a "tick" interval). Our approach could handle this, but we omitted this for brevity.

, but there are techniques that In this case we decided to not model S-2.7 formally, something that our approach can handle without difficulty.

5 Tool Support

A tool supporting this approach would have to provide a mechanism to mark informal artefacts to be marked as "justified", and a place to write this justification down. Further, all *Ri* would have to be marked for re-justification, as soon as any *W* or *S* changes.

We developed a platform for requirements engineering called ProR[2] [JG11]. While the tool can be used stand-alone, we designed it with the goal to ease the integration of natural language requirements and formal models.

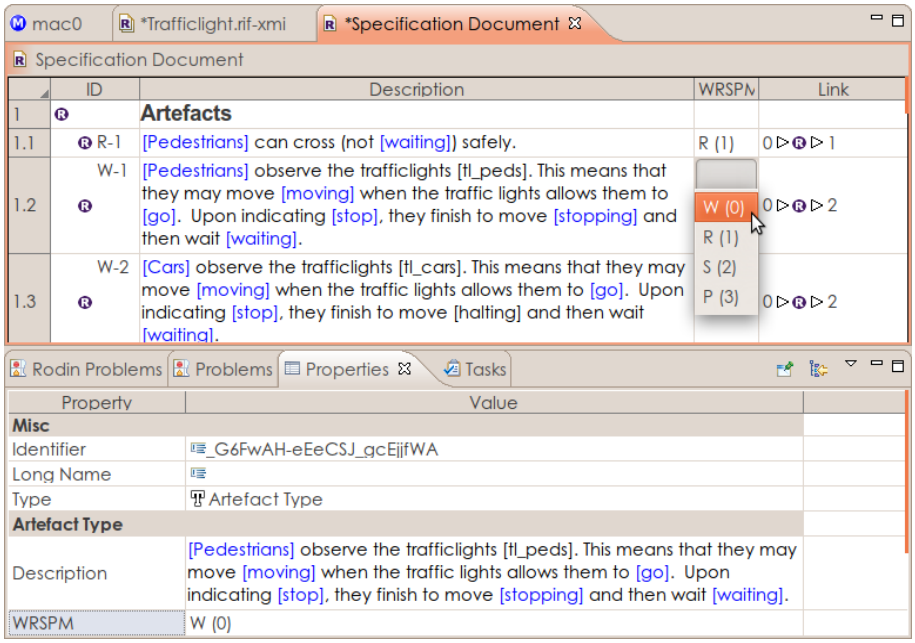


Figure 4: Integration of WRSPM-structured artefacts and formal Event-B elements.

ProR is based on the Requirements Interchange Format RIF/ReqIF [OMG11]. RIF was created in 2004 by the "Herstellerinitiative Software", a body of the German automotive industry that oversees vendor-independent collaboration. In 2010, the Object Management Group (OMG) took over the standardisation process and released a new version of the standard under the name ReqIF. Our tool environment is currently based on RIF 1.2, support for ReqIF 1.0 is planned.

ProR is part of the Requirements Modeling Framework, which is an official Eclipse Project.

ProR can be installed directly into Rodin. A tight integration can be achieved with plugins that access both the Rodin and ProR data structures.

We created a plugin that allows us to manage the vocabulary of the natural language requirements as Event-B models. Via this plugin, ProR supports highlighting of formal model elements directly in the requirements text. Annotated traces can be used to record information regarding relationships. For instance, this mechanism can be used to record the justification argument between a textual requirement and a formal model element.

Formal Event-B elements have a corresponding proxy object in the RIF model that is automatically synchronised with the Event-B model. The integration is currently manual via drag and drop. The proxy object can be extended with additional attributes to store arbitrary information.

[2] http://www.pror.org

The plugin is built using the Eclipse EMF technology [3]. This allows us to "hook" code into the models to perform various tasks. Depending on the specification approach used, we could provide validators to ensure consistency according to the approach taken.

The application of the tool is shown in Figure 4, where the elements from the formal model are highlighted in the requirement text. We also see how a classification of elements can be performed, in this example following WRSPM. The desired artefact type is selected from a drop-down directly in the editor.

The *Properties View* in the lower pane shows additional information regarding the selected element.

The right column shows the number of incoming and outgoing links, providing a quick summary of each element's traceability. These links can be unveiled, as shown in Figure 5. Rows with a triangle represent an annotated trace. In this example, an informal justification has been provided.

For links, the rightmost column contains the link target. Selecting it shows the target's properties in the Property View. In the screenshot we see that the link target is the event *stopping_peds*. As it is selected, the Property View shows its attributes, including the event itself. This is a reference to the model, not a copy of the event.

The tool is currently in a prototypical state and is actively developed. Specifically, it currently support the manual creation of links and colour highlighting. We envision a tool that identifies unaccounted requirements and model elements, and that invalidates traces when related model elements change, as well as change impact analysis.

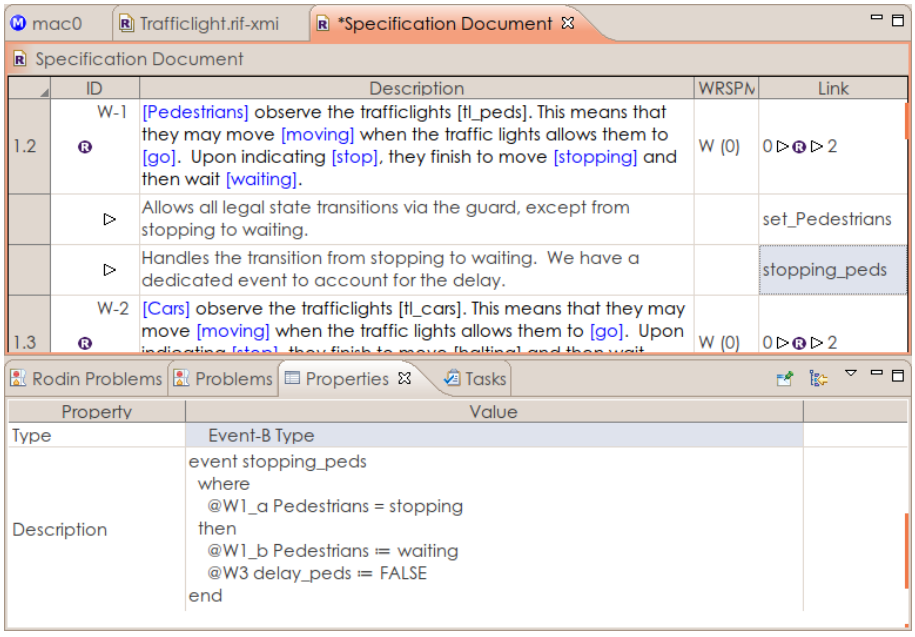


Figure 5: The unveiled traces of an element. As the link target is selected, the link target's properties are shown in the Property View (the lower pane)

6 Related Work

The issue of traceability has been analysed in depth by Gotel et. al. [GF94]. Our research falls into the area of post-requirements specification traceability.

Abrial [Abr06] recognises the problem of the transition from informal user requirements to a formal specification. He suggests to construct a formal model for the user requirements, but acknowledges that such a model would still require informal requirements to get started. He covers this approach in [Abr10].

The WRSPM reference model [GJGZ00] was attractive because it allowed us to discuss the specification in general terms, while still being meaningful in the context of a specific approach

[3] Eclipse Modelling Framework, http://www.eclipse.org/emf/

like Problem Frames [Jac01] or the functional-documentation model [PM95].

There have been successful attempts in applying Problem Frames and Event-B together. In [LGG+10], the authors show how these are being applied to an industrial case study. In contrast to our approach, only requirements that were actually modelled formally were included in the specification in the first place.

There are approaches spanning from requirements to formal model, a well-known one being KAOS [DDML97]. But rather than allowing informal elements that are omitted from the formal model, it provides so-called "soft-goals" that are broken down into requirements that can still be modelled formally.

Reveal [Pra03] is an engineering method based on Michael Jackson's "World and the Machine" model. There are a lot of similarities to our approach, including the acknowledgement of requirements that are not part of the formal model. However, Reveal is more of a process description of the overall requirements engineering process. Therefore it could be quite attractive to apply the Reveal process with the approach described here.

Last, [WAC10] describes a much more comprehensive case study where a number of the concepts described in this paper can be found.

7 Conclusion

In this paper, we presented an approach for incrementally building a formal model from structured informal requirements. Our approach supports partial formal modelling and provides traceability for both formal and informal specification elements. This approach allows us to take advantage of the formal model regarding automated verification, while providing a systematic (albeit manual) approach to validation of the remaining specification elements.

We demonstrate our ideas on a specification and model of a traffic light system. While this is arguably a teaching example, it contains examples of specification elements that are challenging in formal modelling and demonstrates how these can be addressed.

We believe that tool support is a crucial element for such an approach to work and presented an integration of the ProR platform for requirements engineering and the Rodin platform for Event-B modelling to support our approach.

Future Work. We will continue investigating different specification methods. While we find WRSPM useful, it is a reference framework that is not intended to be applied as is. We have experimented with Problem Frames, which are useful but does not match well with our approach to refinement (based on Event-B).

We will explore the suitability of Event-B for modelling bigger specifications with our approach, if possible real-world examples.

Last, we will continue our work on tool support.

Acknowledgements. The work in this paper is partly funded by Deploy[4]. Deploy is a European Commission Information and Communication Technologies FP7 project.

Bibliography

[ABH+10] J.-R. Abrial, M. J. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, L. Voisin. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *STTT* 12(6):447–466, 2010.

[Abr06] J. Abrial. Formal methods in industry: achievements, problems, future. In *Proceedings of the 28th international conference on Software engineering*. Pp. 761–768. 2006.

[Abr10] J. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, June 2010.

[4] http://www.deploy-project.eu

[BS03] E. Börger, R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[DDML97] R. Darimont, E. Delor, P. Massonet, A. van Lamsweerde. GRAIL/KAOS: An Environment for Goal-driven Requirements Engineering. In *Proc. of the 19th int. conf. on Software engineering*. Pp. 612–613. ACM, 1997.

[GF94] O. Gotel, A. Finkelstein. An Analysis of the Requirements Traceability Problem. In *Proc. of the First Int. Conf. on Requirements Engineering*. Pp. 94–101. 1994.

[GJGZ00] C. A. Gunter, M. Jackson, E. L. Gunter, P. Zave. A Reference Model for Requirements and Specifications. *IEEE Software* 17:37–43, 2000.

[Heh93] E. C. R. Hehner. *A Practical Theory of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[HJ98] C. A. R. Hoare, H. Jifeng. *Unifying Theories of Programming*. Prentice Hall, 1998.

[HL09] S. Hallerstede, M. Leuschel. How to Explain Mistakes. In Gibbons and Oliveira (eds.), *TFM*. Lecture Notes in Computer Science 5846, pp. 105–124. Springer, 2009.

[Jac01] M. Jackson. *Problem frames: analysing and structuring software development problems*. Addison-Wesley/ACM Press, 2001.

[JG11] M. Jastram, A. Graf. Requirements, Traceability and DSLs in Eclipse with the Requirements Interchange Format (RIF/ReqIF). In *Tagungsband des Dagstuhl-Workshop MBEES*. fortiss GmbH, München, 2011.

[JHLJ10] M. Jastram, S. Hallerstede, M. Leuschel, A. G. R. Jr. An Approach of Requirements Tracing in Formal Refinement. In *VSTTE*. Springer, 2010.

[Jon90] C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1990.

[KJ10] E. Kang, D. Jackson. Dependability arguments with trusted bases. In *Requirements Engineering Conference (RE), 2010 18th IEEE International*. P. 262–271. 2010.

[Lam02] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[LB08] M. Leuschel, M. Butler. ProB : an automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer* 10(2):185–203, 2008.

[LGG+10] F. Loesch, R. Gmehlich, K. Grau, C. Jones, M. Mazzara. Report on Pilot Deployment in Automotive Sector. Technical report D7, DEPLOY Project, 2010.

[OMG11] OMG. Requirements Interchange Format (ReqIF) 1.0.1. 2011. http://www.omg.org/spec/ReqIF/

[PM95] D. L. Parnas, J. Madey. Functional documents for computer systems. *Science of Computer programming* 25(1):41–61, 1995.

[Pra03] Praxis. Reveal – A Keystone of Modern Systems Engineering. Technical report, 2003.

[WAC10] J. Woodcock, E. G. Aydal, R. Chapman. The Tokeneer Experiments. *Reflections on the Work of CAR Hoare*, p. 405–430, 2010.

[Wie03] K. Wiegers. *Software Requirements: Practical Techniques for Gathering and Managing Requirements throughout the Product Development Cycle*. Microsoft Press, Redmond Wash., 2nd ed. edition, 2003.

Specification and refinement of discrete timing properties in Event-B

Mohammad Reza Sarshogh[1], **Michael Butler**[2]

[1] mrs2g09@ecs.soton.ac.uk, http://www.ecs.soton.ac.uk/people/mrs2g09
University of Southampton, Southampton, UK

[2] mjb@ecs.soton.ac.uk, http://users.ecs.soton.ac.uk/mjb/
University of Southampton, Southampton, UK

Abstract: Event-B is a formal language for systems modeling, based on set theory and predicate logic. It has the advantage of mechanized proof, and it is possible to model a system in several levels of abstraction by using refinement. Discrete timing properties are important in many critical systems. However, modeling of timing properties is not directly supported in Event-B. In this paper we identify three main categories of discrete timing properties for trigger-response pattern, *deadline*, *delay* and *expiry*. We introduce language constructs for each of these timing properties that augment the Event-B language. We describe how these constructs can be mapped to standard Event-B constructs. To ease the process of using the timing constructs in a refinement-based development, we introduce patterns for refining the timing constructs that allow timing properties on abstract models to be replaced by timing properties on refined models. The language constructs and refinement patterns are illustrated through some generic examples. Event-B refinement allows atomic events at the abstract level to be broken down into sub-steps at the refined level. The goal of our refinement patterns is to provide an easy way to represent and correctly refine timing constraints on abstract atomic events with more elaborate timing constraints on the refined events. This paper presents an initial set of patterns.

Keywords: Real-time System, Event-B, Event, Deadline, Delay, Expiry, Refinement Patterns

1 Introduction

In Event-B [Abr10], systems are modeled formally by a collection of events (i.e. guarded actions) that act on abstract variables. The aim in this work is to introduce an approach to formally model the timing properties for the trigger-response pattern in control systems. This pattern is common and useful in specification of control systems. It is natural to talk about these kinds of systems in term of possible events of the system. For example in the trigger-response pattern, trigger and response are both events of the control system.

One of the main advantages of Event-B method is its support for stepwise modeling by refinement. The other strength of this method is the mechanized proof obligation generator and the prover which make the verification process, efficient and productive. These advantages of Event-B, make it a suitable approach for formal modeling of critical systems.

An Event-B model has two main parts, context and machine. The context specifies the static

part of the system and the machine models the dynamic part. In the machine, system behavior and its properties can be modeled by using states variables, invariants and events. Variables represent the current state of the system. Invariants specify the global specifications of the state variables and system behaviors. Finally, events represent the transition of the system from a state to another. Events are guarded atomic actions where guards specify the state of the machine where the event can occur in, and actions indicate how the that event modifies the state variables. By refining a machine it is possible to introduce new state variables and events, strengthen the guards of the abstract events or introduce new actions on new state variables. Standard refinement techniques are used to verify the refinement between models at different abstraction levels.

Event-B lacks explicit support for expressing and verifying timing properties. Modeling time-critical systems, using Event-B has been investigated in several studies. What distinguish our work, is categorizing timing constraints in three groups, introducing a systematic way of encoding each of them in an Event-B model, introducing patterns for refining timing constraints and proving satisfaction of abstract timing constraint by their concrete ones. In this way, the consistency of the system timing properties in the system specification can be proved by using refinement feature of the language.

2 Timing Properties Categories

In order to formalize the process of adding time properties to an Event-B model, it has been decided to categorize the mostly used time related specifications in time-critical system descriptions. Hence, several time-critical system specifications like a car gear-controlling system, a message passing algorithm in a network, a water tank level controller, etc., had been studied to extract their timing properties. The next step was to categorize them in several groups according to the nature of their restriction. The result was three groups of timing properties; *Deadline*, *Delay* and *Expiry*. These three will be explained in more details in the following. As mentioned before, these timing properties are essentially trigger-response patterns, and trigger and response are naturally modeled as events. As a result, all the definitions in this work are event based, where A is the trigger event and B is the response event.

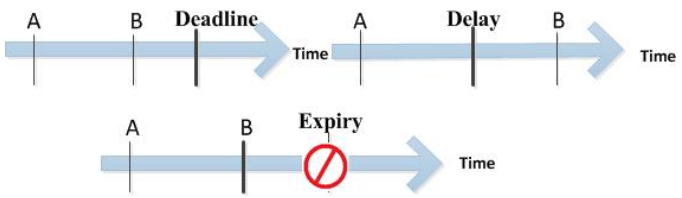


Figure 1: Time Boundary Diagrams

Imagine a system with two events, A and B where first event A has to happen to make event B possible to occur. The three types of timing boundaries which may be declared between event A and event B are as follow:

- **Deadline:** Event B must occur within time D of event A occurring,

- **Delay:** Event B cannot occur within time W of event A occurring,
- **Expiry:** Event B cannot occur after time E of event A occurring.

Based on the definition of these three restrictions, the deadline forces an event to happen before a specific time, delay prevents an event from happening before a specific time and expiry prevents an event from happening after a specific time. Accordingly, by having a deadline between two events, it is guaranteed that by the deadline the deadline event has already occurred, by having delay it is guaranteed that there will be a minimum gap between occurrence of two events, and by having expiry it is guaranteed that if the restricted event has happened it was before a specific time. In order to have a better understanding of these constraints, Figure 1 illustrates how these boundaries restrict events.

In this section delay, deadline and expiry have been introduced informally. In the following we explain how they are formalized.

3 Modeling Timing Properties In Event-B

In order to explicitly represent timing properties we extend the Event-B syntax with constructs for deadlines, delays and expiries. These timing properties place a discrete timing constraint between trigger events and response events. A typical pattern is a trigger followed by one of a choice of responses thus our timing constructs specify a constraint between a trigger event A and a set of response events Bx. The syntax for each of these constructs is as follows:

- **Deadline**(A, {B1,..,Bn}, t),
- **Delay**(A, {B1,..,Bn}, t),
- **Expiry**(A, {B1,..,Bn}, t).

The property **Deadline**(A, {B1,..,Bn}, t) means that one of the response events Bx must occur within the time t of trigger event A occurring. In the case of delay, if any of the events in the response set happens it has to happen after its declared delay. Finally in the case of expiry, if any of the events in the expiry set happens it has to happen before the specified expiry time.

Now a specification consists of an Event-B machine consisting of variables, invariants and events, together with a list of timing properties using the above syntax. Having the annotations standardizes the process of specifying discrete timing properties in Event-B models and allows us to define patterns for refining timing properties as we show in Section 4.

We give a semantic to our timing constructs by translating them into Event-B variables, invariants, guards and actions that are added to the machine to which the timing properties belong. The effect of additions to the Event-B machine will be to add clock increment event and constrain further the order between events. In particular they constrain the order between trigger, response and clock increment events. For example, the additional Event-B elements that a deadline property give rise to will prevent more than t clock increments occurring in between a trigger event and a corresponding response event.

We define rules for encoding each of the three timing constructs in Event-B in turn. In each case we assume there is already a partial order between the trigger event and the corresponding

response events, that is, we assume that the response events are only enabled after the corresponding trigger event has occurred. This ordering assumption is encoded using boolean flags as shown in Figure 2(a). As shown in Figure 2(a), event A sets the boolean variable A as one of its actions, so when variable A has the value of $TRUE$, it shows event A has happened. Also, in event Bx the flag of event A will be checked to see if event A has already happened. Other than checking the flag and setting the flag, in their guard, $Xgrds$ represents the other possible guards of the event and in the action section $Xacts$ represent the other possible actions of the event.

Note we do not assume that the trigger and response events will occur only once. Typically the trigger and response events will be part of an iterative loop and the ordering flags will be reset at the end of each iteration of the loop by an appropriate event.

3.1 Modeling Delay

In this section we explain how delay is encoded in an Event-B model. As mentioned before, in order to have discrete time in Event-B a natural number variable is declared to represent the current time in the machine and an event is added to model the progress of time.

In order to explain how delay is encoded in Event-B, we will go through the process, for a generic trigger event A and some generic alternative response events $B1 \ldots Bn$.

EVENT $A \widehat{=}$
  WHERE
    $A = FALSE$
    $Agrds$
  THEN
    $A := TRUE$
    $Aacts$
  END
EVENT $Bx \widehat{=}$
  WHERE
    $A = TRUE$
    $Bx = FALSE$
    $Bxgrds$
  THEN
    $Bx := TRUE$
    $Bxacts$
  END

(a) Event A and Bx without delay

EVENT $A \widehat{=}$
  WHERE
    $A = FALSE$
    $Agrds$
  THEN
    $A := TRUE$
    $tA := time$
    $Aacts$
  END

EVENT $Bx \widehat{=}$
  WHERE
    $A = TRUE$
    $Bx = FALSE$
    $time \geq tA + t$
    $Bxgrds$
  THEN
    $Bx := TRUE$
    $Bxacts$
  END
EVENT $Tick\_Tock \widehat{=}$
  THEN
    $time := time + 1$
  END

(b) Event A and Bx with delay

Figure 2: Events A and Bx in 2(a) along with the Delay property will implicitly define the model in 2(b)

There are two steps in order to add a delay constraint which is defined as follow to an Event-B model:

$$Delay(A, \{B1, .., Bn\}, t). \tag{1}$$

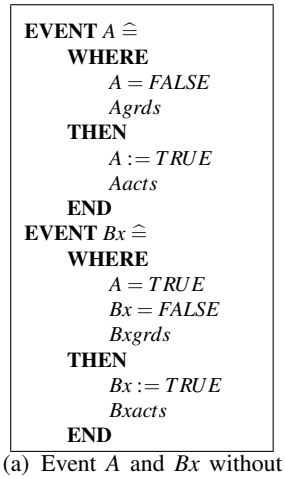
First the occurrence time of the trigger event is recorded in a variable (tA). Then in the event which should be delayed (event A), a guard is needed which forces the event to be eligible to occur after the stated delay period has been passed from the occurrence of the trigger event. In Figure 2 a general pattern of delayed trigger-response and the event which progress the time

(Tick_Tock), in an Event-B model, has been shown. As explained in this section, it is possible to add a delay to a standard Event-B model.

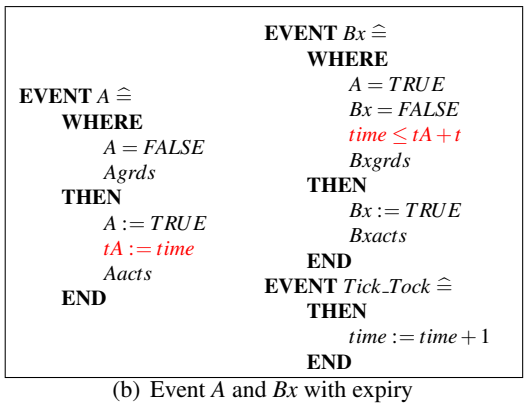
3.2 Modeling Expiry

Modeling expiry is similar to the delay. Again the first step is to record the occurrence time of the trigger event and the next step is to guard the restricted event according to the recorded time and the specified expiry period. Suppose, we want to force timing property 2 to the trigger-response pattern which is shown in Figure 3(a), how the model should be changed to contain the timing property is shown in Figure 3(a).

$$Expiry(A, \{B1, .., Bn\}, t) \tag{2}$$



(a) Event A and Bx without expiry



(b) Event A and Bx with expiry

Figure 3: Events A and Bx in 3(a) along with the expiry property will implicitly define the model in 3(b)

As shown in Figure 3, in order to have expiry for an event, an action is needed to record the occurrence time in the trigger event (event A), and a guard in the restricted event to prevent it from happening if the expiry period has been passed.

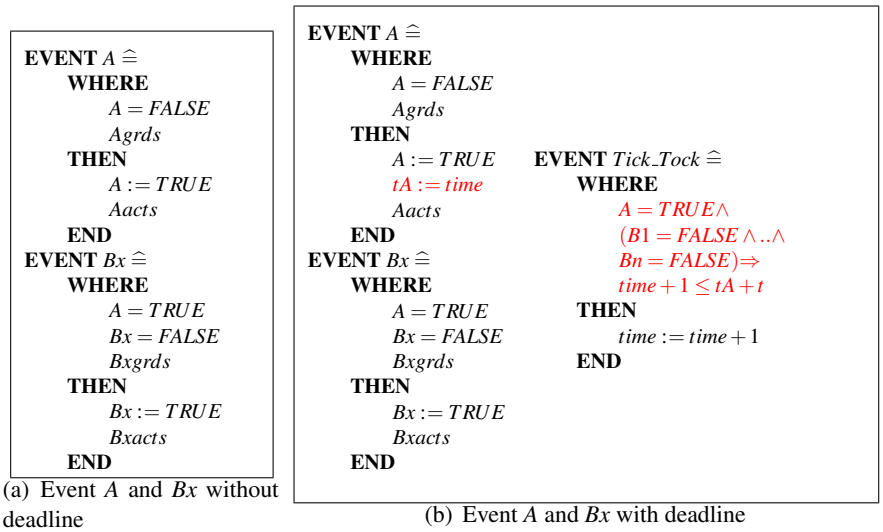
3.3 Modeling Deadline

In order to encode expiry and delay, just the trigger and the response events are involved. But, this is not the case for modeling deadline. In order to model a deadline the *Tick_Tock* event is involved as well, because if the trigger event has happened, we want to force the response event to occur, before passing the deadline. Guardin the *Tick_Tock* event is a possible way to enforce one of the events $B1$ to Bn to occur before the deadline passes. As it will be explained in Section 6 guarding the clock in order to model deadline has been used in several timed specifications theories and tools.

Suppose, a deadline has been declared by our timing annotation as follow:

$$Deadline(A, \{B1, .., Bn\}, t). \tag{3}$$

In order to model this restriction in an Event-B model, first the occurrence time of event A should be recorded by adding a new action. Then a guard on the $Tick\_Tock$ event is needed, to enforce the deadline. In Figure 4 how deadline 3 can be added to a standard Event-B model is shown in detail.



(a) Event A and Bx without deadline

(b) Event A and Bx with deadline

Figure 4: Events A and Bx in 4(a) along with the deadline property will implicitly define the model in 4(b)

Multiple deadline constraints may be added to a model. In this case, a deadline guard similar to what has been shown in Figure 4(b) should be added to the $Tick\_Tock$ event for each deadline constraint.

It is possible to cause a deadlock by declaring a longer delay between two events than an existing deadline between them. There are two approaches to detect this kind of deadlock, either by running a model checker (e.g. ProB) and then check the uncovered events or by declaring an invariant which implies if a deadline guard is not true (current time is equal to the deadline and none of the restricted event has yet occurred) then one of the restricted events should be eligible to occur. As a result, if there is a deadlock, the invariant will not be proved for the $Tick\_Tock$ event.

4 Some Patterns to Refine Deadline, Delay and Expiry and Their Uses

In this section, some patterns of refining the introduced types of timing boundaries will be explained and their uses in order to synchronize different events will be shown by explaining some

general examples. In this section Event Refinement Diagrams [But09] are used to present the order between events of a refinement and also their relations with their abstract events. As a result the Event Refinement Diagram notation will be explained briefly in the next section.

4.1 Refinement and Event Refinement Diagram

Usually, a real world system has a complex specification with a lot of details. If we want to model all the details of a system specification in a single stage, the complexity and the size of the model can cause a lot of difficulties. One solution is to model systems, step by step by using refinement. The system specification should be broken to different levels of abstraction. Then, the first step will be the modeling of the most abstract specification of the system. Then by each refinement more details of the system specification will be added to the model. By this approach, the model will be a more explicit representation of the target system by each refinement.

In Event-B refinement process, it is possible to introduce new events which do not exist in the abstract machine. Other events extend abstract events or refine them. Those events which do not exist in the abstraction, refine *skip*. They model the pre-steps or post-steps of abstract events which are not visible in the abstraction in order to reduce the complexity. Although they do not refine any abstract event, they are related to abstract events.

In order to simplify tracking the relations between abstract and concrete events, refinement diagrams have been introduced by Butler in [But09]. In a refinement diagram there is a tree structure in which the abstract event is positioned as the root of the tree, and its concrete events or events which are new but model the pre/post-steps of the abstract events are represented as leaves. The other characteristic of this notation is that the concrete events which exist in the abstract machine and refine abstract events, are connected to their corresponding abstract event by solid lines and the new events which model the pre/post-steps of abstract events are connected by to their related abstract events by dash lines. Figure 5 is an event refinement diagram, illustrating

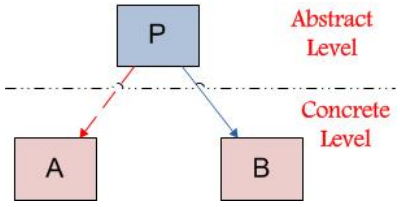


Figure 5: Refinement Diagram Example

that abstract event P is refined by a combination of concrete event A followed by concrete event B. Event A is a pre-step of event B that refines skip, while event B refines event P.

By this introduction to the Refinement Diagram, how the elicited timing properties can be refined, will be explained in the following.

4.2 Refining a Deadline to Sequential Sub-Deadlines

Consider an abstract model of a system where there is a deadline between event A and event B. As shown in Figure 7, event B can only occur if event A has already happened. The deadline

properties for this level of abstraction, is shown in Figure 7(a). In the next refinement event B will be broken to two steps, as shown in Figure 6. By breaking event B to $B1$ followed by $B2$, its related deadline needs to be broken too. Also the other important issue is that, the abstract event has been refined by the second step, because the accomplishment of the second step is equivalent to accomplishment of abstract event(B). So the first step should refine *skip*.

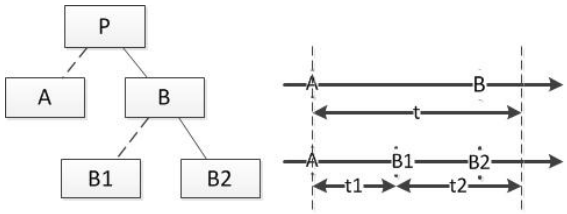


Figure 6: Refining an abstract deadline to two sub-deadlines

$Deadline(A,\{B\},t)$.

EVENT $A \widehat{=}$
WHERE
$A = FALSE$
$Agrds$
THEN
$A := TRUE$
$Aacts$
END

EVENT $B \widehat{=}$
WHERE
$A = TRUE$
$B = FALSE$
$Bgrds$
THEN
$B := TRUE$
$Bacts$
END

(a) Event A and B

$Deadline(A,\{B1\},t1)$
$Deadline(B1,\{B2\},t2)$
Where
$t1 + t2 \leq t$

EVENT $B1 \widehat{=}$
WHERE
$A = TRUE$
$B1 = FALSE$
$B1grds$
THEN
$B1 := TRUE$
$B1acts$
END

EVENT $B2$ *refines* $B \widehat{=}$
WHERE
$B1 = TRUE$
$B2 = FALSE$
$B2grds$
THEN
$B2 := TRUE$
$B2acts$
END

(b) Event $B1$ and $B2$

Figure 7: Events A and B in abstract Machine in 7(a) and events $B1$ and $B2$ in the concrete machine in 7(b)

Now, in order to respond to the trigger, two steps have to be accomplished where each of them has its own deadline. In the concrete level, the trigger event of deadline constraint for event $B1$ is event A and the trigger event for the deadline of event $B2$ is event $B1$. Hence, the abstract deadline should be broken as shown in Figure 7(b) where the sum of new deadlines does not violate the abstract deadline.

The relation between the concrete states and the abstract ones is expressed by a *gluing invariant* [ABH+10] in Event-B, in order to verify the refinement. Two kinds of gluing invariants are needed in order to prove that the concrete deadlines satisfy their abstraction. The first type is required to clarify the relation between the order of the abstract and concrete events which are involved in the deadline. The other type is needed to specify the relation between the new deadlines in the concrete machine and the abstract deadline. In the explained pattern these invariants should be as follow:

- The relation between abstract events and its refining events ($B2$ and B are the boolean

variables which act as the occurrence flag of events $B2$ and B):

$$B2 = B, \tag{4}$$

- The order between concrete events:

$$B1 = TRUE \Rightarrow A = TRUE, \tag{5}$$

- The relation between the abstract deadline trigger time and its concrete one (tA is an integer variable which records the occurrence time of event A and $tB1$ does the same thing for event $B1$):

$$B1 = TRUE \Rightarrow tB1 \leq tA + t1, \tag{6}$$

$$A = TRUE \wedge time > tA + t1 \Rightarrow B1 = TRUE. \tag{7}$$

Invariant (4) specifies that the occurrence of event $B2$ is equivalent to the occurrence of event B. Invariant (5) specifies that event $B1$ must occur after event A. Invariant (6) shows the relation between occurrence time of even $B1$ and the trigger time of abstract deadline and Invariant (7) specifies the deadline for occurrence of event $B1$ which is the trigger for occurrence of event $B2$. Invariant (7) is required in order to prove Invariant (6) for event $B1$, because it specifies that $B1$ must occur before $tA + t1$.

It should be mentioned that the abstract deadline can be broken into more than two sub-deadlines either by successive refinement steps or by refining the abstract event with more than two sub sequential events in one refinement step.

4.3 Refining An Abstract Deadline to Alternative Sub-deadlines

Often, when a process has to finish by a specific time there is a recovery scenario which will guarantee that by the deadline either the desired response or some recovery response will be achieved. So by the deadline either the normal or the recovery scenario has been accomplished. For example, consider, instead of refining event B in the example of Section 4.2, by two sequential sub steps, it has been refined by breaking it into two alternative events, $B1$ and $B2$. So, after occurrence of event A either event $B1$ or event $B2$ should happen. How event B and the abstract timing property are refined is shown Figure 8.

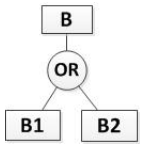
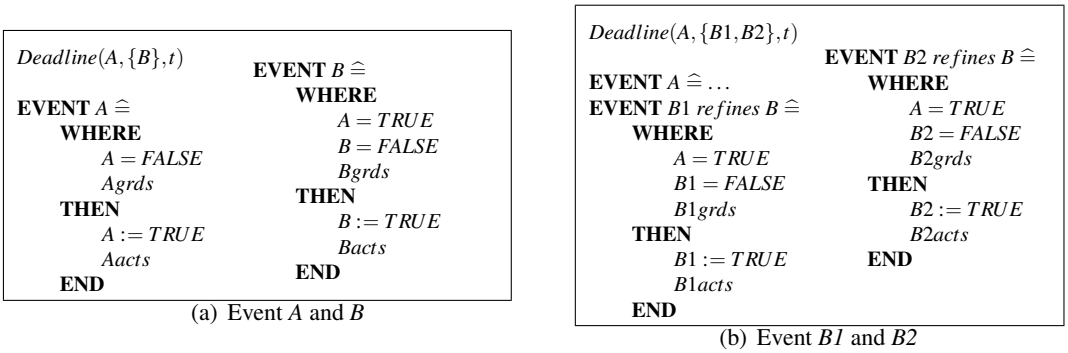


Figure 8: Refining an event to two possible events

As shown in Figure 9, in the refinement event B has been broken to event $B1$ (normal response) and event $B2$ (recovery response) and both of them refine the abstract event. As a result the deadline will be refined as shown in Figure 9(b). As explained in Section 3.3 by declaring a

deadline which has more than one member in its deadline set, we specify the behavior where, after occurrence of event A, before passing the deadline time, either process will be accomplished by occurrence of event $B1$ or event $B2$.



(a) Event A and B

(b) Event $B1$ and $B2$

Figure 9: Events A and B in abstract Machine in 9(a) and events $B1$ and $B2$ in the concrete machine in 9(b)

In this case the only kind of invariant which is required is the one which connected the concrete events occurrences to their abstract one. In the above example the required invariants will be as follow:

$$B2 = TRUE \vee B1 = TRUE \Leftrightarrow B = TRUE. \tag{8}$$

Based on invariant 8 occurrence of event B is equivalet to the occurrence of event $B1$ or event $B2$.

4.4 Refining Alternative Sub-Deadlines by Sequential Sub-Deadlines and Expiries

We now present a pattern for refining an abstract deadline by some alternative deadlines and then refine these by sequential deadlines.

In order to explain this pattern, the example of Section 4.3 will be continued. So, in the current state, we have a trigger event A and two alternative responses, event $B1$ and $B2$. The deadlines of each level of abstraction, are shown in Figure 9.

In the next refinement, each of the events $B1$ and $B2$ will be refined to two sequential steps and their deadline will be refined to two sequential deadlines, same as the pattern shown in Section 4.2 (event $B1$ will be broken to events $B1\_1$ and $B1\_2$ and event $B2$ will be broken to events $B2\_1$ and $B2\_2$). In this system, the first response case is desirable (modeled by event $B1$), but if its first step (modeled by event $B1\_1$) has not been accomplished by $t4$, the second response case (modeled by event $B2$) will be activated and its first step (modeled by event $B2\_1$) has to happen before the specified deadline ($t1$). As a result by the first deadline in the concrete machine, either the first response case has been activated or the second one (by occurrence of their first steps). For the next step, event $B1\_1$ triggers event $B1\_2$, and event $B2\_1$ triggers event $B2\_2$ as shown in Figures 10 and 11. The other specification of this system is that the deadline between the first ($B1\_1$) and the second ($B1\_2$) steps of the first response case ($B1$) is greater than

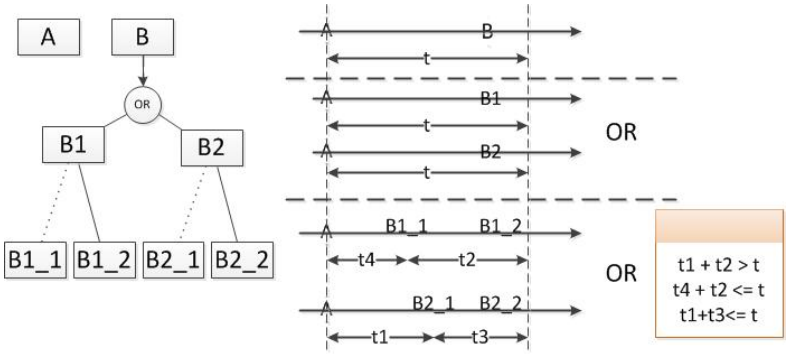


Figure 10: Refining each of the possible scenarios to two steps

$Deadline(A,\{B1\_1,B2\_1\},t1)$,
$Expiry(A,B1\_1,t4)$
$Deadline(B1\_1,\{B1\_2\},t2)$,
$Deadline(B2\_1,\{B2\_2\},t3)$
where
$t1+t2>t \wedge t4+t2 \leq t$
$t1+t3 \leq t$

EVENT $B1\_1 \widehat{=}$
WHERE
$A = TRUE$
$B1\_1 = FALSE$
$B1\_1grds$
THEN
$B1\_1 := TRUE$
$B1\_1acts$
END

EVENT $B1\_2$ *refines* $B1$ $\widehat{=}$
WHERE
$A = TRUE$
$B1\_1 = TRUE$
$B1\_2 = FALSE$
$B1\_2grds$
THEN
$B1\_2 := TRUE$
$B1\_2acts$
END

(a) Events $B1\_1$ and $B1\_2$

EVENT $B2\_1 \widehat{=}$
WHERE
$A = TRUE$
$B2\_1 = FALSE$
$B2\_1grds$
THEN
$B2\_1 := TRUE$
$B2\_1acts$
END
EVENT $B2\_2$ *refines* $B2 \widehat{=}$
WHERE
$A = TRUE$
$B2\_1 = TRUE$
$B2\_2 = FALSE$
$B2\_2grds$
THEN
$B2\_2 := TRUE$
$B2\_2acts$
END

(b) Events $B2\_1$ and $B2\_2$

Figure 11: Events $B1\_1$ and $B1\_2$ in 11(a) and events $B2\_1$ and $B2\_2$ in 11(b)

the equivalent deadline for the second alternative response case ($B2$). So according to the system specification the concrete deadlines should be as shown in Figure 11.

By the concrete deadlines, the abstract deadline will not be satisfied for the first response case ($t1+t2>t$). This problem is caused by the nature of deadline constraint. In the deadline we just guarantee that by passing the deadline time, at least one of the events of the deadline set has already happened.

As mentioned above, event $B1\_1$ has an expiry constraint too. So after a specific time, it cannot happen anymore and the only possible response will be the second response case. According to the system specification, event $B1\_1$ can just happen before $t4$ time units since event A occurrence. Hence, by enforcing this constraint by declaring an expiry as shown in Figure 11(a), the concrete timing properties will satisfy their abstract ones. It has been guaranteed in the model that if event $B1\_1$ happens, at most $t4$ time units have been passed from event A occurrence. From that time, event $B1\_2$ has $t2$ time units to happen. As a result, the abstract deadline is

satisfied. This pattern shows how combination of deadline and expiry can be useful in modeling and refining the timing properties of a time-critical system.

One question that can be raised here is why we do not use two disjunctive deadlines instead of a deadline and an expiry for this case. To explain it, we will apply this approach on a similar pattern. Suppose we want to encode these timing properties by two disjunctive deadlines, and we were to allow those to be declared as follow:

$$Deadline(A,\{B1\},t1) \vee Deadline(A,\{B2\},t2) \quad \textbf{Where } t2 > t1. \tag{9}$$

To encode this in an Event-B model, the guard on *Tick_Tock* event could be as follow:

$$\begin{aligned} A = TRUE \wedge B1 = FALSE \Rightarrow time + 1 \leq tA + t1 \qquad \vee \\ A = TRUE \wedge B2 = FALSE \Rightarrow time + 1 \leq tA + t2, \end{aligned} \tag{10}$$

Since $t2 > t1$ is easy to show that Guard (11) is equivalent to the guard we would use for the following deadline specification:

Deadline(A, { B1, B2 }, t2).

Intuitively, this is because guard (11) can be satisfied if the $B1$ event occurs after $t1$ time unit since occurrence of event A. As a result by having two disjunctive deadlines, the expiry constraint on event $B1$ will not be enforced.

In this section some approaches have been introduced in order to refine our three groups of timing properties. These patterns do not contain all the possible cases of refining timing properties and we are still working on other possible refinement patterns. In the following section how this research can be improved will be discussed briefly.

5 Future Work

We would like to develop a Plug-in for the Rodin tool-set in order to add time to a standard Event-B model automatically, based on the timing constraints declared in the form of deadline, delay and expiry which are expressed by the introduced annotations.

One of the features which has been added to Rodin [SPHB10], is decomposition. By using this feature, it is possible to break a machine in an Event-B model, to several independent machines where each machine represents one of the sub-components of the system. In this way it is possible to refine each sub-component independently. As a result, the complexity of the model will be decreased. Based on that, the other possible area to improve our pattern to add timing properties to an Event-B model, is to investigate the effect of decomposition on timing specification. It will strengthen the approach by eliciting the possible issues and challenges in decomposing a timed Event-B machine. What can be studied more about the decomposition are how the universal clock should be handled after the decomposition, how the time passing event should be decomposed, or how time related guards and actions will be separated between different machines in a decomposition.

Also, we would like to investigate the possibility of generalizing the introduced timing constraints in order to decrease the redundancy during the timing properties specification process in future.

6 Related Work

Many studies have been dedicated to formalizing and verifying timing properties of real-time systems. Delay, deadline and expiry can be seen in many of those works, sometimes with different names. In real-time calculus TCCS of Wang [Yi90] there is a delay construct $\varepsilon(d) \cdot P$, which enforce the model to wait for d time units and then behave as process P and time cannot proceed if d time-units passed and process P has not started yet. Same mechanism has been used in Timed Modal Specification of Cerans et al [CGL97] to model maximal progress assumption where there is a must modality which enforces the maximum delay to the model. Delay in TCCS and maximal progress in Timed Modal Specification present the same constraint as deadline in our work. Also, what is called a loose delay in Timed Modal Specification forces the same behavior as a delay does in our work. Besides, Urgent Event in Evans and Schneider work [ES00] has been encoded by preventing the time proceeding, if an urgent event is eligible to occur. This behavior of urgent event is the same as deadline events when current time is equal to the deadline and none of the deadline events have occurred yet. In Timed CSP [Sch99] time-out presents the same constraint as expiry does in our work and a delay in Timed CSP causes a similar behavior to what can be enforced by combining introduced delay and deadline in our work.

Modeling time-critical systems by using Event-B has been investigated in several studies. Butler et al. in [BF02] explained how it is possible to model discrete time in B (which is the root of Event-B), by having a natural number variable which represents the current time and an operation which increases it. In that study a deadline has been modeled by preventing the progress of time if the current time is equal to the deadline. This work does not investigate different kinds of timing constraint and timing constraint refinement have not been investigated. Cansell and Rehm in [CMR07] have modeled a message passing algorithm in Event-B by using similar principle, having a natural number variable, represents the current time, and an event which forwards the time, guarded by a set of activation times. Again in here, other kinds of timing constraints have not been mentioned, but more importantly, it is not possible to refine a timing constraint to several sub-timing constraints by this approach. Because, in order to do that, some new values should be added to the activation set in the refinement which is not possible without declaring a new activation set. The problem will be specifying the relation between the new activation set and its abstract one. Bryans et al. in [BFRR10] has introduced an approach to keep track of timing boundaries between different events in a model by adding them to a set and guarding events by them. In their study, deadline cannot be modeled. Similar to the previous approach, refining the timing constraint will be an issue because of tracking the timing constraints by a set for this approach too.

7 Conclusion

According to the gear controller case study[LPY01] which has been done in Even-B, and some other experiences, it seems that these three kinds of timing constraints can be used to model most of the timing properties of a time-critical system. In our case study we managed to first model the system without time, then declare the required timing constraints by using introduced annotations. In the end, according to the declared timing constraints, we added time to the

model. All the refinements' proof obligations which have been generated for relation between the concrete timing constraints and their abstractions have been discharged. If we manage to develop a plug-in which can add the required guards, actions, invariants and *Tick_Tock* event in order to add time to an Event-B model, based on declared timing properties, the process of modeling time-critical systems will be the same as modeling a non-time-critical system by using Event-B.

There are some similarities between our approach and the existing approaches to model time-critical systems. How we encoded deadline, delay and expiry is similar to the approach that timed automata [Alu99] verifiers use, like guarding state transitions (system events) or forcing timing properties to the global clock of the model. In timed automata, it is possible to check the temporal properties [Eme95] of a system. In this approach, the temporal properties can be checked by using refinements where a temporal property is modeled by an abstract invariant and by refinement how it will be gained by detailed behavior of the system will be modeled and verified. For example, in the abstraction we say, a gear-change request should be responded by an error message or a successful change, then by several refinements how system will manage to satisfy it will be modeled and verified.

Our approach is based on modeling discrete timing properties of reactive systems according to their events and through several levels of refinement. But it was not an isolated work, and we tried to develop an approach to add time to an Event-B model by learning from the existing works.

Acknowledgements: This work is partly supported by the EU research project ICT 214158 DEPLOY (Industrial deployment of system engineering methods providing high dependability and productivity) www.deploy-project.eu.

Bibliography

[ABH+10] J.-R. Abrial, M. J. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT* 12(6):447–466, 2010.

[ABHV06] J.-R. Abrial, M. J. Butler, S. Hallerstede, L. Voisin. An Open Extensible Tool Environment for Event-B. In *ICFEM*. Pp. 588–605. 2006.

[Abr10] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[Alu99] R. Alur. Timed Automata. *Theoretical Computer Science* 126:183–235, 1999.

[BF02] M. Butler, J. Falampin. An Approach to Modelling and Refining Timing Properties in B. In *Refinement of Critical Systems (RCS)*. January 2002.

[BFRR10] J. W. Bryans, J. S. Fitzgerald, A. Romanovsky, A. Roth. Patterns for Modelling Time and Consistency in Business Information Systems. In Calinescu et al. (eds.), *15th IEEE International inproceedings on Engineering of Complex Computer Systems,*

ICECCS 2010, Oxford, United Kingdom, 22-26 March 2010. Pp. 105–114. IEEE Computer Society, 2010.

[But09] M. Butler. Decomposition Structures for Event-B. In *Integrated Formal Methods IFM2009, Springer, LNCS 5423*. Volume LNCS(5423). Springer, February 2009.

[CGL97] K. Cerans, J. C. Godskesen, K. G. Larsen. Timed Modal Specification – Theory and Tools. In *IN PROC. OF THE 5TH INT. CONF. ON COMPUTER AIDED VERIFICATION, VOLUME 697 OF LECTURE NOTES IN COMPUTER SCIENCE (LNCS*. Pp. 253–267. Springer–Verlag, 1997.

[CMR07] D. Cansell, D. Méry, J. Rehm. Time Constraint Patterns for Event B Development. In Julliand (ed.), *B 2007: Formal Specification and Development in B 7th International Conference of B Users, January 17-19, 2007*. Lecture Notes in Computer Science 4355, pp. 140–154. Springer-Verlag, Besançon France, 2007. ISSN : 0302-9743 (Print) ; 1611-3349 (Online) ; ISBN : 978-3-540-68760-3.

[DHQ+08] J. S. Dong, P. Hao, S. Qin, J. Sun, W. Yi. Timed Automata Patterns. *IEEE Trans. Softw. Eng.* 34(6):844–859, 2008.

[Eme95] E. A. Emerson. Temporal and modal logic. In *HANDBOOK OF THEORETICAL COMPUTER SCIENCE*. Pp. 995–1072. Elsevier, 1995.

[ES00] N. Evans, S. Schneider. Analysing Time Dependent Security Properties in CSP Using PVS. In *ESORICS*. Pp. 222–237. 2000.

[HLM+08] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, A. Skou. Testing Real-Time Systems Using UPPAAL. 2008.

[Jac83] M. Jackson. *Michael Jackson System Development*. Englewood Cliffs, N.J. : Prentice/Hall., New York, NY, USA, 1983.

[Kop97] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[LPY01] M. Lindahl, P. Pettersson, W. Yi. Formal design and analysis of a gear controller. *STTT* 3(3):353–368, 2001.

[Sch99] S. Schneider. *Concurrent and Real Time Systems: The CSP Approach*. John Wiley Sons, Inc., New York, NY, USA, 1999.

[SPHB10] R. Silva, C. Pascal, T. S. Hoang, M. Butler. Decomposition Tool for Event-B. In *Workshop on Tool Building in Formal Methods - ABZ Conference*. January 2010.

[Yi90] W. Yi. Real-Time Behaviour of Asynchronous Agents. In *CONCUR*. Pp. 502–520. 1990.

Combining Model Checking and Discrete-Event Supervisor Synthesis

Nicolas Chausse[1], Helen Xu[1], Juergen Dingel[1] and Karen Rudie[2]

[1] chausse, helen, dingel@cs.queensu.ca, School of Computing, Queen's University, Canada

[2] karen.rudie@queensu.ca, Dept. of Elec. and Comp. Eng., Queen's University, Canada

Abstract:

We present an approach to facilitate the design of provably correct concurrent systems by recasting recent work that uses discrete-event supervisor synthesis to automatically generate concurrency control code in Promela and combine it with model checking in Spin. This approach consists of the possibly repeated execution of three steps: manual preparation, automatic synthesis, and semi-automatic analysis. Given a concurrent Promela program C devoid of any concurrency control and an informal specification E_{in}, the preparation step is assumed to yield a formal specification E of the allowed system behaviours and two versions of C: C_e to identify the specification-relevant events in C and enable supervisor synthesis, and $C_{e,a}$ to introduce "checkable redundancy" and used during the analysis step to locate bugs in: the specification formalization E, the event markup in C_e, or the implementation of the synthesis. The result is supervised Promela code C_{sup} that is more likely to be correct with respect to E *and* E_{in}. The approach is illustrated with an example. A prototype tool implementing the approach is described.

Keywords: Concurrency control, formal verification, control theory, discrete-event systems, controller and supervisor synthesis.

1 Introduction

The poor integration between computer science and electrical engineering in academia has been observed before. In [HS07], Henzinger and Sifakis blame the "wall" between these two disciplines for keeping the "potential of embedded systems" at bay. Indeed, the potential for fruitful interaction between them seems large. Consider, for instance, Discrete-Event Systems (DES) control theory, a branch of control theory which is concerned with the *Supervisory Control Problem* (SCP), i.e., the automatic synthesis of a supervisor (controller) S that restricts the execution of an unrestricted discrete-event system G (called "plant") to enforce some specification E. DES theory originated in the 1980s [RW87, RW89] and offers a large body of research on the SCP which, for instance, considers different formalisms to represent S, G and E including finite state automata (FSA), Petri nets, and the mu-calculus [CL08, ZS05]. Recent work has shown how results and tools from DES theory can be used to alleviate the challenges of concurrent programming. In [WLK+09, WCL+10], automatically generated supervisors are used to guarantee deadlock-free execution of multi-threaded code, based on a structural analysis of a Petri-net representation of the plant. In [DDR08], standard DES based on FSAs is employed to generate

supervisors that enforce deadlock-freedom and safety properties (also expressed as FSAs) on Java programs with static concurrency. In [ADR09], this work is extended to dynamic concurrency which then requires the use of Petri nets.

We extend this line of work and suggest the integration of DES theory with model checking by combining the constructive and generative aspects of DES theory with the analysis and bug detection capabilities of Spin. We aim to facilitate the development of provably correct concurrent systems by increasing the degree of automation. This paper makes the following contributions: (1) The work in [DDR08] is recast in Promela. Given an unrestricted system C expressed in Promela and a specification E expressed as a FSA, a supervised system C_{sup} is automatically generated and is guaranteed to satisfy E and deadlock-freedom. Moreover, the supervisor component in C_{sup} is provably minimally restrictive (maximally permissive), i.e., any behaviour in C but not in C_{sup} will violate E or deadlock-freedom. (2) Despite the theoretical guarantees, bugs can still creep in not only in the various synthesis steps' implementation, but also in the inputs to the synthesis steps, all of which are, at least partially, manually created. We show how model checking can be used to debug them. (3) We describe a prototype tool using Spin and show how Spin's support for shared-memory and message-passing concurrency can be leveraged to generate supervisors supporting the two concurrency paradigms and to optimize the analysis of the combined system. A detailed example illustrates the approach and the tool's utilization.

This paper is structured as follows: Related work is reviewed in Section 2 and relevant background on DES theory is given in Section 3. Section 4 describes our approach and Section 5 illustrates it with an example. Section 6 describes our prototype tools and Section 7 concludes.

2 Related Work

Automatically generating parts of concurrent systems from specifications has been an active research topic. We focus here on approaches that combine synthesis and formal analysis via model checking. While the use of DES in software development and execution has been suggested before [RW90, RW92a, Laf88, TMH97, WKL07], generating control code for concurrent software has received particular interest recently. The work of two authors of this paper on using DES for generating concurrency control code has already been mentioned [DDR08] where the JPF model checker was used to validate the generated supervisor code, but not the manually created inputs. Moreover, despite recent advances in software model checking, model-level analyses are still more likely to be tractable rather than at code-level. Independently, Wang *et al.* have used DES to obtain supervisors that guarantee deadlock-freedom [WLK+09, WCL+10] where concurrent programs are represented as Petri nets and deadlock freedom is characterized by the absence of reachable empty siphons. Our work in this paper (and [DDR08]) is based on FSAs and supports general safety properties rather than just deadlock-freedom. Also, no support for analysis of the generated artifacts is mentioned in [WLK+09, WCL+10]. Timed DES is based on timed automata; recently, UPPAAL-TIGA has been used for an industrial case study involving climate control systems [BCD+07] where the synthesis and analysis capabilities of UPPAAL-TIGA have been combined with Simulink and Real-TimeWorkshop to provide a complete tool chain for synthesis, simulation, analysis and automatic generation of production code. The work in [GPT06] uses symbolic model checking for supervisor synthesis from specifications given

in CTL specifications and a plant description given in NuSMV. The work in [ZS05] introduces DES theory based on the mu-calculus and thus generalizes Ramadge and Wonham's standard DES theory. However, no tool supporting the generalization appears to be available.

There exists additional work that does not make explicit use of DES theory. For instance, some work is aimed at facilitating software architecture component composition (e.g., [TI08, BBC05]). In [TI08], Tivoli and Inverardi generate coordinators which enforce a given global coordination policy [TI08] where components are assumed to adhere to a coordinator-based architectural style and message sequence charts are used for behavioural interface specification. Correctness and maximal permissiveness (called completeness) are proved and the work has been integrated with CHARMY, a tool for architectural analysis. Despite many differences in technical details and terminology, the approach is similar to supervisor synthesis[1]. In the context of concurrent programming, the approach presented by Deng *et al.* explicitly shares our interest in supporting the combined use of synthesis and verification [DDHM02]. It generates synchronization statements for concurrent Java code from invariant specifications and the new code can be fed into the Bandera model checker for analysis. Some related work appears in the literature as environment (assumption) generation. For instance, in [GPB05], the LTSA tool is used to determine the weakest assumptions that the concurrent environment E of a component C has to satisfy such that the composition of C and E satisfies some specification B where E, C, and B are given as FSAs. LSTA also supports model checking. Synthesis has also been used to achieve fault-tolerance. In [AAE04], a method is presented for the synthesis of fault-tolerant concurrent programs from specifications expressed in the temporal logic CTL. However, no implementation allowing the integration with CTL model checkers such as nuSMV is mentioned. Finally, in [IST07] and [IS08], CSP||B is used to control machines or processes via control "annotations" which may represent states, next operations or control flow. A synthesis process is used to: verify the annotations against the machine, manually produce a "Controller" and verify it against the annotations, and finally refine if needed.

We conclude that while the integrated use of synthesis and formal verification has been suggested before, our work differs from each of the existing approaches in at least one of the following two aspects: it uses Spin, one of the most popular and powerful model checkers available; it explicitly uses DES theory and thus allows the large body of existing results and tools to be leveraged. Interestingly, the recent interest in autonomic and adaptive software has produced proposals to design software directly informed by control theory [MPS08, Dah10]. However, so far, controller synthesis does not appear to be part of this research agenda. In [Dah10], validation and verification of autonomic and adaptive systems are singled out as particularly important research topics.

[1] In [TI08, p. 206], it is claimed that supervisor synthesis based on DES requires explicit specification of the deadlocking behaviours; this, however, is not the case.

3 Background

3.1 DES Theory

In DES theory, systems are modelled by FSAs called *plants*. Transitions represent events that are either *controllable* (can be enabled or disabled at will) or *uncontrollable* (may happen arbitrarily). In a *non-blocking* model, all states are *reachable* (from the initial state) and *co-reachable* (lead to a final state) which implies the absence of deadlocks and livelocks. A specification describing a plant's desired behaviour can be modelled using specification FSAs and is called the *specification*, or *legal language*. A specification E *is controllable* with respect to plant G if for any series s of events in G and legal in E (s is in E's prefix closure), there is no uncontrollable event σ that can then happen in G and that is illegal in E ($s\sigma$ is not in E's prefix closure).

Given a specification E and plant G, where E is not necessarily controllable with respect to G, we want to get the least restrictive sub-specification (or largest sub-language) $K \subseteq E$ such that K is controllable with respect to G. If there is no such nonempty subset of E then $K = \emptyset$. If E is controllable with respect to G, then $K = E$. We call a *recognizer* S for K the *supervisor* or the *supremal controllable sub-language* of E with respect to G, denoted $sup\underline{C}(G,E)$ [CL08]. The supervisor is also modelled with an FSA and will *control* G by *enabling* and *disabling* G's controllable events. When a plant G is controlled by a supervisor S, the resulting behaviour is given by the intersection of the language accepted by G and the language accepted by S and is captured by a FSA denoted as S/G.

Composing Specifications and Processes: The plant G and the specification E may consist of several parallel processes G_i and sub-specifications E_j, respectively. We assume that the sub-specifications share all events (i.e., use the same set of events), which means that each node in a sub-specification has a self-loop labelled with all the events that do not directly belong to any sub-specification but belong to the processes. Processes, however, may not share all events. We will combine processes and sub-specifications using an operation that forces the FSAs to synchronize on shared (common) events, while allowing independent interleavings of the non-shared events. We will call this operation *synchronous product*[2].

Complexity and Tool Support: The supervisor $sup\underline{C}(G,E)$ can be computed in time $O(n^2m^2e)$ where n and m are, respectively, the number of states in G and E and e is the total number of events in G and E (Section 3.5.3 of [CL08]). The time complexity of the synchronous product operation is $O(mn)$ where n is the number of sub-FSAs provided and m the maximum number of states in all these sub-FSAs. Several DES tools supporting supervisor synthesis are available including IDES [IDE], TCT [TCT], and DESUMA [DES].

3.2 DES Theory for Generation of Concurrency Control Code

As described in Section 2, previous work has already observed that DES theory can be used directly to control the execution of software with respect to certain specifications [DDR08, WLK+09]. The area of application here has been concurrent programming where the supervisor manages concurrent processes such that deadlock-freedom and the safety properties expressed as FSAs are enforced — the generated supervisor inheriting the strong theoretical guarantees

[2] Note that if two FSAs share all events, the synchronous product reduces to language intersection.

offered by DES theory. The key idea is to view the concurrent system as the plant G and to interpret concurrency- or specification-relevant operations in the code as controllable events. To obtain the closed loop system S/G, the event markup in G is replaced by an interaction with the supervisor in which a request by a process in G to execute an operation is only granted by the supervisor if its execution cannot possibly lead to a deadlock or specification violation. The approach requires the (manual or automated) identification of relevant events in the code and then the transformation of the code and the specification into a format supported by current DES tools. For instance, in [WCL+10] concurrent C code is automatically converted into a Petri net by extracting and combining the control flow graph of each of the threads and modelling execution via token flow. In [DDR08], a similar technique is used to convert Java threads into FSAs which are then combined using the synchronous product operation.

4 Combining Supervisor Synthesis and Spin Analysis

A graphical overview of our approach to integrate supervisor synthesis and analysis is given in Figure 1, which shows the flow of artifacts (solid arrows) between possibly nested activities (boxes). Stick figures indicate activities requiring user interaction and the dashed arrow shows control flow.

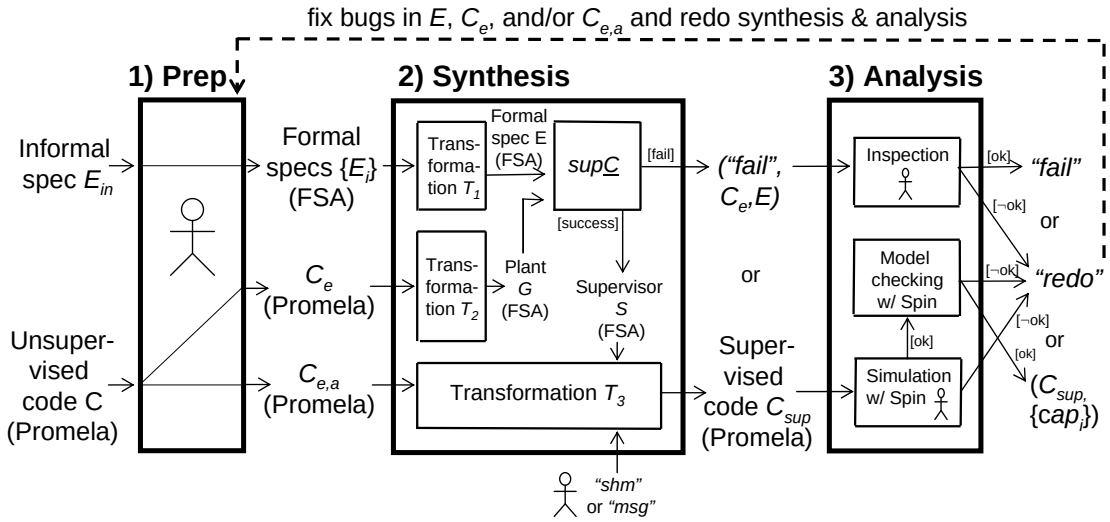


Figure 1: Overview of Approach to Integrate Synthesis and Spin Analysis

This approach recasts the preparation and synthesis steps for concurrency control code generation proposed in [DDR08] using Promela (instead of Java) as the implementation language. Moreover, an additional artifact ($C_{e,a}$) is introduced and the synthesis is followed by an analysis step in which manual inspection, user-guided simulation, and model checking are used to identify bugs in any of the artifacts created during the manual preparation step. If bugs are found, the preparation and the synthesis are redone. We describe each step in more detail.

1) Preparation: The informal specification E_{in} is assumed to express a safety property identify-

ing permissible sequences of events such as precedence constraints, mutual exclusion constraints or capacity constraints. The unsupervised code C is a concurrent Promela program devoid of any concurrency control. The user then (1) translates E_{in} into a collection $\{E_i\}$ of FSAs, (2) marks up specification-relevant events in C to create C_e, and (3) adds assertions and possibly auxiliary variables to C_e to obtain $C_{e,a}$. The transitions in E should distinguish between controllable and uncontrollable events. The assertions in $C_{e,a}$ capture (aspects of) the informal specification E_{in} and offer "checkable redundancy", which will be used in the analysis step to validate E against E_{in}. For instance, a capacity constraint in E_{in} may be checked by an assertion containing a counter variable.

2) Synthesis: Consists of the $sup\underline{C}$ operation, sandwiched between three transformations: T_1 and T_2 to prepare the inputs and T_3 to process the output:

a) The formal specifications E_i are combined into a single one by computing their synchronous product E (transformation T_1 in Figure 1).
b) The unsupervised code with event markup C_e is translated into plant FSA G (transformation T_2). Similar to [DDR08, WCL+10], G is obtained using compiler technology to extract the control-flow graph of every process in C_e and to build FSA-representations. These FSAs are combined by computing their synchronous product.
c) An off-the-shelf DES tool is used to perform the $sup\underline{C}$-operation on E and G.
d) If $sup\underline{C}(G,E) = \emptyset$, the operation fails. Otherwise, the generated supervisor S is automatically implemented in Promela and integrated in $C_{e,a}$ to obtain the supervised code C_{sup} (transformation T_3). Transformation T_3 allows the generation of code that implements the supervision using shared-memory (input *"shm"* in Figure 1) or message-passing (*"msg"*).

3) Analysis: The analysis process is described in Figure 2. If the $sup\underline{C}$-operation fails (line 3),

```
1  input: ('fail',E,Ce) or Csup
2  output: 'fail', 'redo', or Csup
3  if input=='fail' then                                  % SupCon operation failed
4     check that Ce and E are correct wrt C and Ein;      % Manual inspection
5     if bug found then                                   % E and/or event markup in Ce wrong
6        output 'redo' and stop;                          % Fix bug and redo synthesis
7     else output 'fail' and stop                         % Ein may be unenforceable on C; done
8  else
9     simulate Csup in Spin;                              % Does Csup behave as expected? (semi-automatic step)
10    if Csup has unexpected behaviour then
11       output 'redo' and stop;                          % Fix bug and redo synthesis
12    else
13       modelcheck Csup in Spin;                         % Do assertions hold?
14       if violation found then                          % E or assertions in Ce,a must be wrong wrt Ein
15          output 'redo' and stop;                       % Fix bug and redo synthesis
16       else
17          use Spin to determine minimal channel capacities {capi};
18          output (Csup,{capi}) and stop.                % Done
```

Figure 2: Pseudocode for Analysis Step in Figure 1 (indentation indicates nesting)

it may be because C_e or E are incorrect. For instance, event markup in C_e may be misplaced or missing; E may have incorrect transitions or may erroneously mark a controllable event as uncontrollable. If manual inspection uncovers such an issue (line 4), the preparation and the

synthesis are redone. Otherwise, C_e and E are assumed to be correct (w.r.t. E_{in} and C) and the process ends in a fail (because E is unenforceable on C) (line 7). If the $sup\underline{C}$-operation is successful (i.e., $sup\underline{C}(G,E) \neq \emptyset$), the supervised code C_{sup} is simulated by the user (line 9); if unexpected behaviour is encountered, the preparation and the synthesis are redone; otherwise, C_{sup} is model checked (line 13). Assertion violations indicate that either E or the assertions are incorrect and a new iteration is initiated (line 15). If no violations are found, Spin is used to determine the smallest channel capacities $\{cap_i\}$ necessary to implement C_{sup} and the supervised code C_{sup} is output with $\{cap_i\}$.

4.1 Theoretical Guarantees

Strong guarantees can be given for the result of the $sup\underline{C}$ operation at the heart of our approach. The combination of G and S satisfies E and is deadlock-free. Moreover, S is guaranteed to be maximally permissive. Unfortunately, these strong guarantees do not carry over to the artifacts produced from $sup\underline{C}(G,E)$ using our approach. For instance, if our approach stops with output "fail", it is possible that a supervisor for C and E_{in} exists, because the manual inspection overlooked that, e.g., E does not correctly capture E_{in}. In addition, if the approach stops with output C_{sup}, it is still possible that C_{sup} violates E_{in}, because, e.g., the added assertions are not suitable to detect that E actually does not capture E_{in} correctly. The manual steps involved make this situation unavoidable. Moreover, since E_{in} is given only informally, it is difficult to establish theoretical guarantees with respect to E_{in}. Nonetheless, our experience suggests that the approach is still useful. During our case studies it repeatedly helped us identify inputs with unexpected, non-seeded bugs to the synthesis step. A few of these cases will be illustrated in the next section.

Also, in our experiments, we routinely found that the shared-variable implementation of the supervised code had substantially fewer states than the message-passing implementation. This suggested that the generation of the message-passing version, if necessary at all, be postponed until the very end of the prepare-synthesize-analyse cycle.

5 Working Example: Transfer-Line

We have applied our approach on several examples and used the IDES DES tool [IDE] to compute the synchronous product and $sup\underline{C}$ operations. Our working example was taken from [Won11]. A widget processing transfer-line (shown in Figure 3) consists of two production machines $M1$ and $M2$ and one test unit TU. The three machines form a production line and are connected via two widget buffers $B1$ and $B2$. $M1$ may be requested to start production of one widget at a time and deliver it to $B1$ in an unpreventable way after a arbitrary time. Similarly, $M2$ may be requested to pick-up one widget from $B1$ and then deliver it to $B2$. Finally, TU can pick up one widget from $B2$, test it and then either uncontrollably return it to $B1$ on failure or deliver it away.

Figure 4 lists the corresponding unsupervised Promela code. Code doing actual work is abstracted out with comments and the widget test in TU is replaced by a non-deterministic choice.

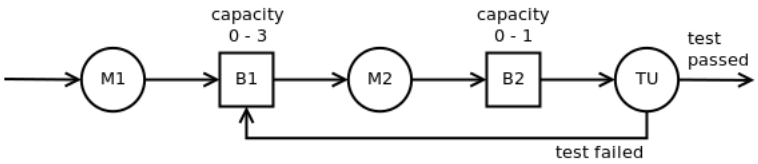


Figure 3: Transfer-Line Example

```
active proctype M1() {
  do :: true ->
    // Idle
    // Create new widget
    // Deliver widget to B1
  od; }
active proctype M2() {
  do :: true ->
    // Idle
    // Pick up widget from B1
    // Process widget
    // Deliver widget to B2
  od; }
```

```
active proctype TU() {
  do :: true ->
    // Idle
    // Pick up widget from B2
    // Test widget
    if
    :: true -> // Passed: deliver away
    :: true -> // Failed: return to B1
    fi;
  od; }
```

Figure 4: Unsupervised Promela Code C

5.1 Step 1: Preparation

Addition of Event Markup and Assertions: Since the event names chosen for the event markup in C_e will also be used for the construction of $\{E_i\}$, we start by identifying the relevant events in C and assertions suitable for checking aspects of E_{in}. The resulting code $C_{e,a}$ is shown in Figure 5. C_e is like $C_{e,a}$ except that the assertions are removed. Three controllable events (*M1MakeWidget*, *M2PickUpWidget*, and *TUPickUpWidget*) and six uncontrollable events (*M1WidgetDelivered*, *M2WidgetPickedUp*, *M2WidgetDelivered*, *TUWidgetPickedUp*, *TUWidgetPassed*, and *TUWidgetFailed*) have been identified. Event *M1MakeWidget* indicates that $M1$ is ready to produce a new widget, similarly for *M2PickUpWidget* with $M2$ from $B1$ as well as for *TUPickUpWidget* with TU from $B2$. Completed widget deliveries are signalled using *M1WidgetDelivered* and *M2WidgetDelivered* and TU signals a failed widget returned to $B1$ with *TUWidgetFailed*.

Assertions warrant the capacity constraints via auxiliary variables (`B1` and `B2`) that store the number of widgets in each buffer and model widget deliveries and pick-ups. Although not essential, the action of picking up widgets was made non-instantaneous to admit more concurrency.

Formal Specifications E_{B1} and E_{B2}: Two specifications are produced capturing how the number of elements in each of the buffer changes in response to certain events (Figure 6). Plain arrows represent uncontrollable events.

5.2 Step 2: Synthesis

Build E (Transformation T_1): The synchronous product of E_{B1} and E_{B2} was generated and contains 8 states and 58 transitions. It is not shown here due to space limitations.

Generate Plant G (Transformation T_2): Plant FSAs (Figure 7) were automatically generated from the control flow graphs of the processes in $C_{e,a}$ using standard parsing technology. Dashed

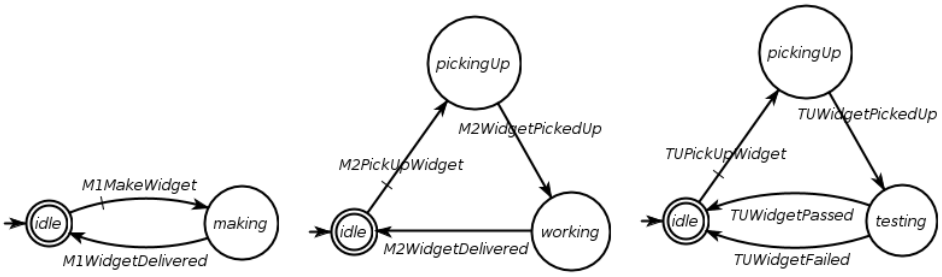
```
short B1 = 0, B2 = 0;
active proctype M1() {
  do :: true ->
    // Idle
    // relevant controllable event: M1MakeWidget
    // Create new widget
    atomic{assert(B1 < 3); B1++;}     // Deliver widget to B1
    // relevant uncontrollable event: M1WidgetDelivered
  od; }
active proctype M2() {
  do :: true ->
    // Idle
    // relevant controllable event: M2PickUpWidget
    atomic{assert(B1 > 0); B1--;      // Pick up widget from B1
    // relevant uncontrollable event: M2WidgetPickedUp
    // Process widget
    atomic{assert(B2 < 1); B2++;}     // Deliver widget to B2
    // relevant uncontrollable event: M2WidgetDelivered
  od; }
active proctype TU() {
  do :: true ->
    // Idle
    // relevant controllable event: TUPickUpWidget
    atomic{assert(B2 > 0); B2--;}     // Pick up widget from B2
    // relevant uncontrollable event: TUWidgetPickedUp
    // Test widget
    if :: true -> // Passed: deliver away
                  // relevant uncontrollable event: TUWidgetPassed
       :: true -> // Failed: return widget to B1
                  atomic{assert(B1 < 3); B1++;}
                  // relevant uncontrollable event: TUWidgetFailed
    fi; od; }
```

Figure 5: Unsupervised Code $C_{e,a}$ with Event Markup and Assertions

(a) E_{B1} (b) E_{B2}

Figure 6: Formal Specifications E_{B1} and E_{B2} (self-loops with events $M1MakeWidget$, $M2PickUpWidget$, $TUPickUpWidget$ and $TUWidgetPassed$ at each node omitted)

arrows represent controllable events. The synchronous product of $M1$, $M2$ and TU was then generated and contains 18 states and 60 transitions. It is not shown here due to space limitations.
Generate Supervisor S: The supervisor for plant G and specification E was generated with *supC*. It contains 41 states and 94 transitions. Due to space limitations it is not shown here.
Generate Supervised Code C_{sup} (Transformation T_3): We created a conversion script that implements FSAs generated by the DES tool used, and inserts concurrency control code in the original Promela code for each relevant event markup. Our script generates two distinct solu-



(a) FSA for $M1$ (b) FSA for $M2$ (c) FSA for TU

Figure 7: FSAs for $M1$, $M2$ and TU

tions: one that implements the communication between the processes and the supervisor using shared variables and another one that uses message passing.

Shared Variable Solution: For each controllable event e, a global boolean variable `_e` indicates whether e is currently enabled. Communicating the occurrence of an event to the supervisor is achieved using global variable `_Event`. When `_Event` = -1, all the events currently enabled are allowed to occur. One such event e_n (with $n \in \mathbb{N}$) is selected non-deterministically (in Spin) and its corresponding process signals its triggering by setting `_Event` to n. The supervisor indicates that it has noted and processed the occurrence of event e_n by resetting `_Event` back to -1.

During transformation T_3, for both controllable and uncontrollable events, every occurrence in the Promela source code of

```
// relevant (un)controllable event: Eventn
```

is replaced by

```
// relevant (un)controllable event: Eventn
atomic{((_Event < 0) && _Eventn) -> _Event = n;}
```

Figure 8 shows the abridged generated supervisor. The first `if` statement enables and disables all events according to the current state of the supervisor FSA. Once an event is triggered by one of the processes via global variable `_Event`, the second `if` statement realizes the corresponding transition. Note that processes can possibly block at uncontrollable events. This may be counter-intuitive, but it is required to ensure that the supervisor can process all event occurrences. However, the process will never block for long as DES guarantees that the supervisor will enable all uncontrollable events that can possibly occur after a controllable one, and therefore that it will (eventually) process any uncontrollable event to occur after a controllable one.

Message Passing Solution: Two channels are used to connect the processes with the supervisor. Channel `_EventReady` is used by processes to signal the readiness of controllable events and to indicate the occurrence of uncontrollable events. Channel `_EventGo` is used by the supervisor to trigger a controllable event (selected non-deterministically in Spin if more than one is ready).

During transformation T_3, every occurrence in the Promela source code of

```
// relevant (un)controllable event: Eventn
```

```
short _Event = 0;                          // Global mutexes
bool _Event1 = false, _Event2 = false, _Event3 = false, ...;
active proctype _Supervisor() {            // Supervisor process
   atomic { short state = 0;               // Current (and firstly initial) state
      do                                   // Main loop
         :: if                             // Enable and disable all events
               :: (0 == state) -> _Event1 = true; _Event2 = false; ...;
               :: (1 == state) -> _Event1 = false; _Event2 = true; ...;
               :: (2 == state) -> _Event1 = true; _Event2 = true; ...;
               ...                         // More cases here
            fi;
          -> _Event = -1; _Event > -1;     // Wait for an event from one of the processes
            if                             // Transition to next state
               :: ((0 == state) && (1 == _Event)) -> state = 1;
               :: ((0 == state) && (2 == _Event)) -> state = 2;
               :: ((1 == state) && (1 == _Event)) -> state = 3;
               ...                         // More cases here
            fi; od; } }
```

Figure 8: Generated Supervisor Using Shared Variables

is replaced for controllable events by

```
// relevant controllable event: Eventn
atomic{assert(nfull(_EventReady)); _EventReady ! n; _EventGo ?? n;}
```

and for uncontrollable events by

```
// relevant uncontrollable event: Eventn
atomic{assert(nfull(_EventReady)); _EventReady ! n;}
```

Figure 9 shows the abridged generated supervisor. Both channels are initially set to maximum capacity as deadlock-freedom may be lost if either channel overflows. To detect this, every send to either channel is prefixed with an "`assert(nfull())`". Both minimal capacities are determined through repeated analyses with decreasing capacities. Each event e received on `_EventReady` causes array position `eventReady[e]` to be incremented so to in effect wait on all events concurrently for a relevant event r. If event r is controllable, then r is sent back on `_EventGo` to allow the corresponding process blocked on "`_EventGo ?? r`" to proceed. The second `if` statement realizes the FSA transitions. Contrary to the shared variable solution, no process ever blocks on any uncontrollable event.

5.3 Step 3: Analysis

The analysis is used to find bugs in the formal specifications ($\{E_i\}$), the event markup (C_e), or the implementation of the transformations T_2 or T_3[3]. Simulation allowed us to locate a bug in the creation of the FSAs for the Promela processes in transformation T_2. The FSAs for $M2$ and TU did not have *M2WidgetPickedUp* and *TUWidgetPickedUp* transitions, respectively. This omission allowed $M1$ to put a fourth widget into $B1$ causing it to overflow. Verification allowed us to locate an event markup that was incorrectly placed. More precisely, event *M1WidgetDelivered* was accidentally put before `B1++` which allowed $M2$ to attempt to pick up a widget from an empty $B1$ causing the assertion `B1 > 0` in $M2$ to be violated.

[3] Since transformation T_1 just takes the synchronous product of the specifications and is assumed to be implemented using a DES tool, it is substantially simpler and is unlikely to contain bugs.

```
chan _EventReady = [255] of { byte };          // Global channels
chan _EventGo = [255] of { byte };
active proctype _Supervisor() {                // Supervisor process
   atomic { byte eventReady[10], event;        // Event buffer and variable
      do                                       // Main loop
         :: if // Find an event relevant to current state else buffer next event
              :: (0 == state) -> do
                 :: (eventReady[1] > 0) -> assert(nfull(_EventGo)); _EventGo ! 1;
                                           event = 1; break;    // Controllable
                 :: (eventReady[2] > 0) -> event = 2; break;    // Uncontrollable
                 :: else -> _EventReady ? event; eventReady[event]++;    od;
              :: (1 == state) -> do
                 :: (eventReady[3] > 0) -> event = 3; break;    // Uncontrollable
                 :: else -> _EventReady ? event; eventReady[event]++;    od;
              ...                              // More cases here
            fi;
          -> eventReady[event]--;
            if                                 // Transition to next state
               :: ((0 == state) && (1 == event)) -> state = 1;
               :: ((0 == state) && (2 == event)) -> state = 2;
               ...                             // More cases here
            fi; od; } }
```

Figure 9: Supervisor Using Message Passing

5.4 Performance Results

We also applied our method to the Dining Philosophers problem and the Cigarette Smokers Problem [Pat71]. We obtained the verification results listed in Table 1, with `ispin.tcl` and Spin Version 6.0.1[4]. We verified our three examples both with shared variables and message passing. In all cases, the following options were selected: invalid endstates and assertion violations safety checks, depth-first search, exhaustive storage mode, no compression or reduction. We also determined the minimum channel capacities. Note that for our examples, message passing requires at least 12 times more states and transitions than shared variables.

Program	Depth Reached	Stored States	Transitions	Atomic Steps	Minimum Channel Capacity Ready, Go	Number of Processes	Time to Compute $supC$ in IDES
Transfer-line							4 sec.
Shared Variables	718	1240	3207	2552	N/A	4	
Message Passing	3887	18868	47209	327715	7, 2	4	
Philosophers							1 sec.
Shared Variables	6022	10464	46033	21632	N/A	6	
Message Passing	9999	157827	580416	1326625	7, 2	6	
Smokers							1 sec.
Shared Variables	194	608	1849	904	N/A	5	
Message Passing	1996	10461	27543	82703	5, 1	5	

Table 1: Verification Results for the Three Examples

[4] A 64 bit AMD Dual Core 2.4GHz CPU with 1.5GB of DDR2 RAM was used.

6 Implementation

All our FSAs were drawn and created using a DES tool called IDES [IDE] developed by the Discrete-Event Control Systems Lab at Queen's University. Synchronous products and $sup\underline{C}$ were computed with IDES which saves its FSA files in a text XML format. Our prototype script for implementing transformation T_2 was written in Ruby and can parse most of Promela except for the `goto` statement and the newly introduced `for` statement. It takes as input a Promela text source file (C_e) and generates plant FSAs readable by IDES. Our script for doing transformation T_3 was also written in Ruby and uses the REXML XML processor. It takes as input a Promela source file (containing $C_{e,a}$), an FSA XML text file generated by IDES (containing E) and generates the supervised code (C_{sup}).

7 Conclusion

We have presented an approach which integrates DES supervisor synthesis and model checking to help facilitate the development of provably correct concurrent code. The approach recasts the process described in [DDR08] using Promela and it uses Spin for validation of the synthesis itself and the inputs to this process. We have described a prototype implementing the approach which supports shared memory and message passing concurrency and have shown how this choice can be used to optimize the verification of the generated Promela code. We have illustrated the approach with an example and provided some performance results.

Future work: There are many interesting avenues for future research. An immediate one is investigating the use of modular [WR88] and decentralized DES theory [RW92b]. Modular DES theory leverages the structure of the system and the specification to combat the explosion of the state space during the synthesis, while decentralized DES allows decentralized control by synthesizing a collection of supervisors. Ultimately, DES theory is concerned with the prevention of undesirable sequences of events. As such, it should also be applicable to other problems in software engineering. Adaptor synthesis (as in, e.g., [BBC05]) and protocol synthesis for web services (as in, e.g., [BIPT09]) are just two examples.

Finally, the development of a tool that seamlessly integrates DES theory as described here and model checking would be interesting not only for research but also for educational purposes and it would, in our opinion, represent a useful first step towards combining concepts from computer science and electrical engineering curricula as advocated in [HS07].

Bibliography

[AAE04] P. C. Attie, A. Arora, E. Emerson. Synthesis of fault-tolerant concurrent programs. *ACM Trans. Program. Lang. Syst., 26(1):125–185* 26(1):125–185, 2004.

[ADR09] A. Auer, J. Dingel, K. Rudie. Concurrency Control Generation for Dynamic Threads. In *47th Annual Allerton Conf. on Communication, Control, and Computing*. 2009.

[BBC05] A. Bracciali, A. Brogi, C. Canal. A formal approach to component adaptation. *Journal of Systems and Software* 74(1):45–54, 2005.

[BCD+07] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. Larsen, D. Lime. UPPAAL-TIGA: Time for Playing Games! (Tool Paper). In *CAV07*. 2007.

[BIPT09] A. Bertolino, P. Inverardi, P. Pelliccione, M. Tivoli. Automatic synthesis of behavior protocols for composable web-services. In *ESEC/SIGSOFT FSE'09*. 2009.

[CL08] C. Cassandras, S. Lafortune. *Introduction to Discrete Event Systems*. Springer, 2 edition, 2008.

[Dah10] W. Dahm. US Air Force Chief Scientist Report on Technology Horizons: A Vision for Air Force Science & Technology During 2010-2030. Technical report, US Air Force, AF/ST-TR-10-01, 2010.

[DDHM02] X. Deng, M. B. Dwyer, J. Hatcliff, M. Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *ICSE '02*. 2002.

[DDR08] C. Dragert, J. Dingel, K. Rudie. Generation of Concurrency Control Code using Discrete-Event Systems Theory. In *FSE 16*. 2008.

[DES] DESUMA Software. Univ. Michigan and Mount Allison Univ. Avail. at www.eecs.umich.edu/umdes/toolboxes.html, last accessed March 2011.

[GPB05] D. Giannakopoulou, C. S. Pasareanu, H. Barringer. Component verification with automatically generated assumptions. *Automated Software Engin.* 12(3):297–320, 2005.

[GPT06] A. Gromyko, M. Pistore, P. Traverso. A tool for controller synthesis via symbolic model checking. In *WODES'06*. IEEE, 2006.

[HS07] T. Henzinger, J. Sifakis. The Discipline of Embedded Systems Design. *IEEE Computer*, Oct. 2007.

[IDE] IDES: The integrated discrete-event systems tool. Queens Univ. Avail. at www.ece.queensu.ca/hpages/labs/discrete/software.html, last accessed March 2011.

[IS08] W. Ifill, S. S. A step towards refining and translating B control annotations to Handel-C. In *Concurrency and Computation: Practice and Experience*. 2008.

[IST07] W. Ifill, S. Schneider, H. Treharne. Augmenting B with Control Annotations. In *LNCS*. 2007.

[Laf88] S. Lafortune. Modeling and analysis of transaction execution in database systems. *IEEE Transactions on Automatic Control* 33:439–447, 1988.

[MPS08] H. Mueller, M. Pezze, M. Shaw. Visibility of Control in Adaptive Systems. In *SEAMS 2008*. 2008.

[Pat71] S. Patil. Limitations and capabilities of Dijkstra's semaphore primitives for coordination among processes. Technical report, MIT, 1971.

[RW87] P. J. Ramadge, W. M. Wonham. Supervisory control of a class of discrete-event processes. *SIAM Journal of Control and Optimization* 25(1):206–230, 1987.

[RW89] P. J. Ramadge, W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE* 77(1):206–230, 1989.

[RW90] K. Rudie, W. M. Wonham. Supervisory Control of Communicating Processes. In *Protocol Specification, Testing, and Verification*. Elsevier, 1990.

[RW92a] K. Rudie, W. M. Wonham. Protocol verification using discrete-event systems. In *31st IEEE Conference on Decision and Control*. 1992.

[RW92b] K. Rudie, W. M. Wonham. Think globally, act locally: Decentralized supervisory control. *IEEE Transactions on Automatic Control* 37(11):1692–1708, 1992.

[TCT] TCT tool. Univ. of Toronto. Avail. at www.control.toronto.edu/DES, last accessed March 2011.

[TI08] M. Tivoli, P. Inverardi. Failure-free coordinators synthesis for component-based architectures. *Science of Computer Programming* 71(3):181–212, 2008.

[TMH97] J. Thistle, R. P. Malhamé, H. Hoang. Feature interaction modelling, detection and resolution: A supervisory control approach. In *Feature Interactions in Telecommunications and Distributed Systems IV*. 1997.

[WCL+10] Y. Wang, H. Cho, H. Liao, A. Nazeem, T. Kelly, S. Lafortune, S. Mahlke, S. Reveliotis. Supervisory Control of Software Execution for Failure Avoidance: Experience from the Gadara Project. In *WODES'10*. 2010.

[WKL07] Y. Wang, T. Kelly, S. Lafortune. Discrete control for for safe execution of it automation workflows. In *EuroSys'07*. 2007.

[WLK+09] Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, S. Mahlke. The theory of deadlock avoidance via discrete control. In *POPL'09*. 2009.

[Won11] W. M. Wonham. Supervisory Control of Discrete-Event Systems. 2011. Avail. at www.control.utoronto.ca/~wonham, v. 2010.07.01, last accessed April 4, 2011.

[WR88] W. M. Wonham, P. J. Ramadge. Modular supervisory control of discrete-event systems. *Mathematics of Control, Signals, and Systems* 1:13–30, 1988.

[ZS05] R. Ziller, K. Schneider. Combining supervisor synthesis and model checking. *ACM Transactions on Embedded Computing Systems* 4(2):221–362, 2005.

The Belgian Electronic Identity Card: a Verification Case Study

Pieter Philippaerts, Frédéric Vogels, Jan Smans[†], Bart Jacobs, Frank Piessens

IBBT-DistriNet, Dept. of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200a, B3001 Leuven, Belgium

Abstract: In the field of annotation-based source code level program verification for Java-like languages, separation-logic based verifiers offer a promising alternative to classic JML based verifiers such as ESC/Java2, the Mobius tool or Spec#. Researchers have demonstrated the advantages of separation logic based verification by showing that it is feasible to verify very challenging (though very small) sample code, such as design patterns, or highly concurrent code. However, there is little experience in using this new breed of verifiers on real code. In this paper we report on our experience of verifying several thousands of lines of Java Card code using VeriFast, one of the state-of-the-art separation logic based verifiers. We quantify annotation overhead, verification performance, and impact on code quality (number of bugs found). Finally, our experiments suggest a number of potential improvements to the VeriFast tool.

Keywords: verification, VeriFast, separation logic, Java Card

1 Introduction

Software verification is finally reaching a point where it is possible to verify relatively complex applications written in popular programming languages. Even though it is still often a significant effort to annotate applications in order to help them get verified automatically, the benefits outweigh the cost for a number of software markets. In particular, software with a very high cost of failure (for example, airplane controllers) or software for systems that are difficult to update after deployment (for example, smart cards) are perfect candidates for software verification.

VeriFast [JSP10] is a verifier for single-threaded and multithreaded C and Java programs annotated with separation logic specifications. The approach enables programmers to ascertain the absence of invalid memory accesses, including null pointer dereferences and out-of-bounds array accesses, as well as compliance with programmer-specified method preconditions and postconditions.

This paper will assess the applicability of verification of Java Card applets using the VeriFast approach. Two non-trivial applets are annotated and an analysis of the verification effort and results is made. Section 2 introduces the VeriFast tool and gives a short introduction to Java Card technology. Section 3 describes the applets that were used in this case study and gives a short overview of some of the solutions we used to annotate certain features. Section 4 evaluates the results of the case study and Section 5 summarizes the future work. Finally, Section 6 concludes the paper.

[†] Jan Smans is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO)

2 Background

The results in this paper build on two technologies that will be presented in this section. Section 2.1 presents a short overview of the verification technology used in this case study, whereas Section 2.2 introduces the features of the Java Card platform relevant to the applets being verified.

2.1 VeriFast

VeriFast [JSP10] is a verifier for Java and C programs annotated with separation logic [ORY01] specifications. The tool modularly checks via symbolic execution [BCO05] that each method in the program satisfies its specification. If VeriFast deems a Java program to be correct, then that program does contain neither null dereferences, array indexing errors, API usage violations nor data races. Moreover, all user-specified assertions are guaranteed to hold.

At the heart of the separation logic lies the concept of permissions. In particular, a method can only access a field if it has permission to do so. For example, consider the class `Interval` shown below. `o.low |-> v` denotes (1) the permission to access (read and write) the field `low` of the `Interval` object `o` and (2) that the current value of that field is `v`.

Listing 1: VeriFast annotations in Java code

```
1  /*@ predicate interval(Interval i, int l, int h) =
2    i.low |-> l &*& i.high |-> h &*& l <= h;
3  @*/
4
5  public class Interval {
6    int low, high;
7
8    void shift(int amount)
9      //@ requires interval(this, ?low, ?high);
10     //@ ensures interval(this, low + amount, high + amount);
11   {
12     //@ open interval(this, low, high);
13     this.low += amount;
14     this.high += amount;
15     //@ close interval(this, low + amount, high + amount);
16   }
17 }
```

To distinguish full (read and write) from read-only access, permissions are qualified with fractions [Boy03] between 0 (exclusive) and 1 (inclusive), where `1` corresponds to full access and any other fraction represents read-only access. For example, `[f]o.low |-> v` denotes read-only access if `f` is less than `1` and full access if `f` equals `1`. We typically omit explicitly writing `[1]` for full permissions. Permissions can be split and merged as required during the proof. For example, two read-only permissions `[1/2]o.low |-> v` and `[1/2]o.low |-> v` can be combined into a single full permission `[1]o.low |-> v` and vice versa. In the context of

Java Card, we rely on fractional permissions to check that fields are not assigned to outside of transactions.

To abstract over the set of permissions required by a method, permissions can be grouped and hidden via predicates. For example, consider the predicate interval shown above. This predicate groups the permissions to access `low` and `high`, and additionally states that the value of `low` must be less or equal to the value of `high`. Just like basic permissions, predicates can be split and merged as required during the proof.

Each method in the program has a corresponding method contract consisting of a pre- and postcondition that respectively describe the permissions required and returned by the method. More specifically, the permissions described by the precondition conceptually transfer from the caller to the callee on entry to the method, and vice versa for the postcondition when the method returns. For example, consider the method `shift` in the class `Interval`. `shift`'s precondition states that the method may only be called if **`this`** points to a valid interval (where the meaning of "valid interval" is determined by the predicate `interval`). The precondition imposes no restriction on the interval's bounds, but binds the lower bound to the variable `low` (indicated by the question mark) and the upper bound to `high`. The postcondition ensures that **`this`** is still a valid interval, and its bounds have been shifted by `amount` with respect to the method pre-state. Note that our verification tool requires all annotations to be written inside special comments (`/*@ ... @*/`) which are ignored by the Java compiler but recognized by our verifier.

VeriFast does not automatically fold and unfold predicate definitions. Instead, folding and unfolding must be done explicitly by developers via ghost commands (unless the predicate is marked precise). For example, the open statement in the body of shift unfolds the definition of the predicate `interval`, and similarly the close statement folds the definition. Verification of the code snippet shown above fails if any of the ghost statements is removed.

In addition to static predicates (placed outside of a class), VeriFast also supports instance predicates (placed inside of a class). Just like instance methods, instance predicates are dynamically bound. That is, the variable **`this`** is an implicit argument to each instance predicate, and its dynamic type determines the exact meaning of the predicate. For example, the interface `Vehicle` shown below defines the instance predicate `valid`. The meaning of valid depends on the subclass at hand. For example, `o.valid()` denotes the permission to read the field `maxspeed` if the dynamic type of `o` is `Car`. In the context of Java Card, we use instance predicates to describe consistency conditions for applets (i.e. the invariant that must be preserved by each transaction).

Listing 2: Instance predicates.

```
1 interface Vehicle {
2   //@ predicate valid();
3 }
4 class Car implements Vehicle {
5   int maxspeed;
6
7   //@ predicate valid() = [1/2]this.maxspeed |-> _;
8 }
```

The extended static checker for Java (Esc/Java) [FRL+02] is another program verifier that has been used to verify Java Card programs [MP10, CH]. However, Esc/Java is unsound [LNS00, Appendix C.0]. This means that Esc/Java can fail to detect certain bugs. For example, the extended static checker reasons incorrectly about object invariants in the presence of reentrant calls. Unlike Esc/Java, the VeriFast methodology has been proven to be sound [JP11].

Gomes *et al.* [GMD05] have investigated using the B method to generate correct Java Card implementations from abstract models via refinement. Contrary to the B method, VeriFast does not start from an abstract model, but instead reasons directly about the applet's source code. The advantage of our approach is that we can retroactively prove correctness of existing implementations.

VeriFast performs *modular* verification. This means that the verifier analyzes each method in isolation using only method contracts (not the callees' implementations) to reason about method calls. An advantage of modular reasoning is that verification scales (verification times remain low) and that deep properties can be specified and checked. A disadvantage however is that the developer must write annotations at method boundaries. To avoid having to write annotations, Huisman *et al.* [HGSC04] have used model checking to find bugs in Java Card applications. Unlike VeriFast, Huisman *et al.* do not aim to prove the absence of all errors, but only of certain undesired applet interactions.

Mostowski [Mos07] has written a specification for the Java Card API in dynamic logic. In addition, he has used this specification to verify a number of applets using the Key verifier. A recurring problem encountered during these case studies was bad prover performance. For example, Mostowski states that "*it is not uncommon for the prover to run over an hour to finish the proof of one method*". Contrary to [Mos07], we use separation logic to specify the Java Card API. While separation logic has proven to be a powerful specification formalism for reasoning about complex (but small) examples such as design patterns and highly concurrent code, there is only limited experience in applying separation logic to larger, realistic programs. This paper reports on our experience in applying separation logic to verify realistic Java Card code. An explicit goal of VeriFast is to keep verification times low. For example, the time needed to verify full functional correctness of a single method is typically under one second.

2.2 Java Card

The Java Card platform [Ora11] was initially launched by Sun in 1996 and aimed to simplify the development of smart card applications. Until then, smart card code was largely written in C, which is difficult to write in the first place, and also has distinct disadvantages in terms of security and reliability.

The Java Card platform allowed developers to write smart card applets in a subset of the Java language that targets a specifically optimized Java framework for smart cards. The older (and most popular) platform, now called Java Card Classic Edition, does not support floating point operations, strings, multi-threading, garbage collection, stack inspection, multidimensional arrays, reflection, etc. The newest Java Card 3.0 Connected Edition supports more features but is still lacking compared to the full Java language and framework.

Java Card is now the dominant platform for smart cards, with applications for GSM, 3G, finance, PKI, e-commerce and e-government. Due to the absence of (potential) competitors and

the improvements of the latest incarnation of the Java Card platform, it can be expected that this will remain the case for the near future.

2.2.1 Applets

The entry point of each Java Card applet is a class that extends the built-in, abstract class `javacard.framework.Applet`. This class defines a number of methods that are called by the Java Card runtime to interact with the applet. In particular, the class `Applet` defines an abstract method `process` that must be overridden by the subclass. The implementation of `process` forms the core of the applet. More specifically, `process` takes an apdu (i.e. a wrapper around a byte array) as input, processes it, and possibly returns an updated apdu to the runtime. Typically, the apdu contains both information on the action that should be performed by the applet and data associated with that action.

A subclass of `javacard.framework.Applet` is a valid applet only if it declares a static method called `install`. The goal of this method is to create a new applet instance and to register this instance with the runtime. The class `MyApplet` shows the prototypical structure of a Java Card applet.

Listing 3: The prototypical structure of an applet.

```
1  class MyApplet extends Applet {
2    public static void install(byte[] arr, short offset, byte length) {
3      MyApplet applet = new MyApplet();
4      // initialize the applet
5      applet.register();
6    }
7
8    public void process(APDU apdu) {
9      // process the apdu
10   }
11 }
```

2.2.2 Transactions

Java Card applets use two types of memory to store data and intermediate results. Fields and objects are stored in persistent EEPROM memory, whereas the stack (and hence local variables) are stored in volatile RAM memory. In addition, the applet can also choose to allocate arrays in RAM memory, because this type of memory is faster and is harder for attackers to read. This complicates things because the smart card may lose power at any time during the computation, which results in the RAM memory being wiped, whereas the EEPROM memory retains the intermediate results.

To preserve consistency of the data stored in persistent memory, Java Card supports transactions. More specifically, the platform defines three methods to interact with the transaction mechanism: `beginTransaction`, `commitTransaction`, and `abortTransaction`. When `beginTransaction` is called, all changes to persistent memory are made conditionally. Only

when a call to `commitTransaction` is executed, the changes to the persistent memory are committed atomically. If `abortTransaction` is called instead, or if the card suddenly loses power before calling `commitTransaction`, the persistent memory is restored to its original state (before the call to `beginTransaction`). Note that the transaction mechanism does not impact values stored in RAM. Incorrect use of the transaction API, for example calling `beginTransaction` while a transaction is already in progress, results in an exception.

2.2.3 Java Card and VeriFast

VeriFast was originally developed for C and Java programs, but has been modified to also support Java Card applications. The Java language used for Java Card applications is a precise subset of the full Java language, thus adding Java Card support to VeriFast was easy.

Java Card does, however, use a very different class library optimized for smart cards. VeriFast needs to know for every function in the library the pre- and postconditions in order to reason about code. These specifications are placed in a separate file that defines all the classes and methods in the Java Card framework. The specifications are based on the descriptions of these methods in the official Java Card documentation. The actual implementation of these library functions is not checked.

Building the specification file is an incremental process. VeriFast only needs pre- and postconditions for the methods that are actually used by the applications you want to verify. Hence, only a subset of the full Java Card class library has been annotated in the specification file. It is critical that the specifications of library functions is correct; errors in their annotations could lead to errors in the verification process. Therefore, extreme diligence is used when adding new function definitions to the specification.

3 Case Study

Software verification is still a very time-consuming process. Existing or new source code must be annotated in order to express assumptions and invariants, and to let the verifier reason about the code. Minimizing these required annotations is an active field of research where a lot of work remains to be done. For current verification technologies the overhead of annotating code is far from negligible, so it is not (yet) economically profitable to try to annotate and verify every piece of code. Large, non-critical code bases are examples where the effort probably is not worth the hassle.

However, smart card applications have a number of properties that *do* make them ideal candidates for software verification. First of all, they are typically small, in the order of a few thousand lines of code. Secondly, they are critical, in the sense that they usually offer some kind of security service. And last but not least, it is extremely difficult to update the code once it has been deployed. If a serious bug is discovered in the code, it might be necessary to recall all the deployed smart cards and issue new ones, which would be a commercial disaster.

This paper reports on the verification of two Java Card applets: one large open source applet that implements a clone of the Belgian Electronic Identity Card, and another smaller commercial applet.

The remainder of this section focusses on the larger, open source applet. Unfortunately, due to contractual constraints we are not allowed to discuss the details of the commercial applet.

3.1 The Belgian Electronic Identity Card

The Belgian Electronic Identity Card (eID) was introduced in 2003 as a replacement for the existing non-electronic identity card. Its purpose is to enable e-government and e-business scenarios where strong authentication is necessary. The card has the size of a standard credit card and features an embedded chip. In addition to containing a machine readable version of the information printed on the card, the chip also contains the address of the owner and two RSA key pairs with the corresponding X509 certificates. One key pair is used for authentication, whereas the other key pair can be used to generate legally binding electronic signatures.

The card is implemented on top of the Java Card platform (Classic Edition) and implements the smart card commands as defined in the ISO7816 standard. Unfortunately, the actual code that runs on the eID cards is not publicly available. For our case study, we used an open source, cloned version of the eID applet that implements the same functionality as the real eID card[1]. It is aimed at developers who wish to interact with eID cards as an easy to use and customizable testing platform.

The eID implementation consists of one large class called `EidCard` and a few other small helper classes. The `EidCard` class inherits from the `Applet` class and encapsulates about 80% of the entire code base. It is a complex class with over 1,300 lines of code and no less than 38 fields.

3.2 Specification of Transaction

Java Card offers transactions to preserve consistency of the data stored in persistent memory. However, what does it mean for an applet to be consistent? In VeriFast, developers can explicitly write down what fields are part of the persistent state together with the desired consistency conditions. More specifically, the class `Applet` defines an instance predicate called `valid`. Each subclass must override this predicate. The implementation of the predicate given in the subclass defines the consistency conditions for the applet at hand. For example, consider the applet class `ExampleApplet` shown below. The predicate `valid` indicates that both the fields `arr` and `i`, and the array pointed to by `arr` are part of the persistent state (line 6). Moreover, the predicate imposes the consistency condition that `i` is a valid index in `arr` (line 7).

While reading fields is possible at any time, updates to persistent memory should be made inside of a transaction. The permission system used by VeriFast is the key to enforcing this property. More specifically, at the start of the `process` method, no transaction is in progress. As shown in Listing 5, the precondition of `process` contains $1/2$ of the `valid` predicate. This means that the method can read but not update fields included in `valid` (as the method only has one half of the permissions included in `valid`). The predicate `current_applet` is simply a token describing the currently active applet.

To update the fields of the applet, the method should somehow gain additional permissions (namely the other half of the `valid` predicate). These additional permissions can be acquired

[1] The code can be downloaded from http://code.google.com/p/eid-quick-key-toolset/

Listing 4: The contract of the `process` method, using fractional permissions.

```
1  class ExampleApplet extends Applet {
2    int i;
3    int[] arr;
4    /*@
5    predicate valid() =
6      this.arr |-> ?arr &*& this.i |-> ?i &*&
7      array_slice(arr, 0, ?len, _) &*&
8      0 <= i &*& i < len;
9    @*/
10 }
```

Listing 5: The contract of the `process` method, using fractional permissions.

```
1  public void process(...)
2     //@ requires current_applet(this) &*& [1/2]valid() &*& ...;
3     //@ ensures current_applet(this) &*& [1/2]valid() &*& ...;
4  {
5     ...
6  }
```

by calling `beginTransaction`. In particular, the postcondition of `beginTransaction` shown in Figure 6 gives $1/2$ of the `valid` predicate. The `process` method can then merge `[1/2]valid()` (gained from the precondition of `process`) and `[1/2]valid()` (gained from the postcondition of `beginTransaction`) into `[1]valid()`. The full permission to `valid` gives the applet the right to modify the applet's fields for the duration of the transaction. When calling `commitTransaction`, half of the permissions included in the `valid()` predicate return to the system again. Note that it is impossible to call `endTransaction` if the applet is in an invalid state (according to the conditions described by `valid`), as the precondition of `commitTransaction` requires the consistency conditions to hold.

Listing 6: The declaration of the `beginTransaction` and `commitTransaction` methods

```
1  public static void beginTransaction();
2     //@ requires current_applet(?a) &*& ...;
3     //@ ensures current_applet(a) &*& [1/2]a.valid() &*& ...;
4
5  public static void commitTransaction();
6     //@ requires current_applet(?a) &*& a.valid() &*& ...;
7     //@ ensures current_applet(a) &*& [1/2]a.valid() &*& ...;
```

3.3 Inheritance

The ISO7816 standard specifies a mechanism to access files that are stored on a smart card. Three types of files are defined:

1. **Master files** represent the root of the file system. Each smart card contains at most one master file.
2. **Elementary files** contain actual data.
3. **Dedicated files** behave like directories. They can contain other dedicated or elementary files.

To represent this structure, the eID implementation uses helper classes that form a class hierarchy. The root of the hierarchy is the abstract `File` class. This class has two subclasses: `DedicatedFile` and `ElementaryFile`. And finally, the `MasterFile` class inherits from `DedicatedFile`.

When a class is defined in the source code, it can be annotated with a predicate that represents an instance of that class. These predicates can then be used elsewhere to represent a fully initialized instance of that class. Listing 7 shows how a `File` predicate can be defined for the corresponding `File` class. The class consists of two fields, which are also represented in the predicate. The predicate can also contain other information about the class such as invariants.

Listing 7: A first definition of the *File* class and predicate.

```
1  public abstract class File {
2     /*@ predicate File(short theFileID, boolean activeState) =
3         this.fileID |-> theFileID &*&
4         this.active |-> activeState; @*/
5
6     private short fileID;
7     protected boolean active;
8
9     ...
10 }
```

The `ElementaryFile` class redefines the `File` predicate as shown in lines 2-4 of Listing 8. A `File` predicate that is associated with an `ElementaryFile` class is defined as an `ElementaryFile` predicate where three of the five parameters are undefined.

The definition of the `ElementaryFile` predicate (lines 5-13) consists of a link to the `File` predicate defined in Listing 7 and some extra fields and information that are specific to elementary files.

When an object is cast from the `File` to the `ElementaryFile` class (or vice versa), the corresponding predicate on the symbolic heap must be changed as well. We 'annotated' this by adding the methods that are defined in Listing 9 to the `ElementaryFile` class and calling these methods when required. Obviously, this solution is far from elegant because it requires adding calls to stub functions in the code of the applet. The most recent version of VeriFast

Listing 8: A first definition of the *ElementaryFile* class and predicate.

```
1  public final class ElementaryFile extends File {
2     /*@ predicate File(short theFileID, boolean activeState) =
3           ElementaryFile(theFileID, ?dedFile, ?dta,
4             activeState, ?sz); @*/
5     /*@ predicate ElementaryFile(short fileID,
6             DedicatedFile parentFile, byte[] data,
7             boolean activeState, short size) =
8           this.File(File.class)(fileID, activeState) &*&
9           this.parentFile |-> parentFile &*&
10          this.data |-> data &*& data != null &*&
11          this.size |-> size &*&
12          array_slice(data, 0, data.length, _) &*&
13          size >= 0 &*& size <= data.length; @*/
14
15    private DedicatedFile parentFile;
16    private byte[] data;
17    private short size;
18
19    ...
20 }
```

supports annotating this behavior as lemma functions, removing the requirement of modifying the applet's code.

One problem that occurs with the methods presented in Listing 9 is that information is lost when an `ElementaryFile` is cast to a `File` and then back again to an `ElementaryFile`. This loss of information happens in the `castFileToElementary` method where three parameters are left undefined.

There are some instances in the eID applet where this loss of information was problematic. The solution was to extend the `File` and `ElementaryFile` predicates to contain an extra parameter that can store any information. The result can be seen in Listing 10. Line 3 shows the definition of this extra parameter. In the case of the `File` class, no extra information is kept and the parameter is defined to be empty (denoted as 'unit' on line 5). Similarly, line 22 defines the parameter to be empty for the `ElementaryFile` predicate, because all state information that can be stored in the predicate is fully defined by the other parameters.

Line 14 shows the case where the predicate needs the extra parameter to store additional information about the object. In this case, the `info` parameter stores a quad of extra information that can be used to correctly initialize the embedded `ElementaryFile` predicate without losing information.

4 Evaluation

The main goal of this case study was to see how practical it is to use VeriFast to annotate a Java Card applet that is more than a toy project. It gives us an idea of how much the annotation

Listing 9: Functions to cast predicates.

```
1  public void castFileToElementary()
2     //@ requires [?f]File(?fid, ?state);
3     //@ ensures [f]ElementaryFile(fid, _, _, state, _);
4  {
5     //@ open [f]File(fid, state);
6  }
7
8  public void castElementaryToFile()
9     //@ requires [?f]ElementaryFile(?fid, ?dedFile, ?dta, ?state, ?sz);
10    //@ ensures [f]File(fid, state);
11 {
12    //@ close [f]File(fid, state);
13 }
```

overhead is, where we can improve the tool, and whether we can actually find bugs in existing code using this approach.

4.1 Annotation Overhead

The more information the developer gives in the annotations about how the applet should behave, the more VeriFast can prove about it. It is up to the developer to choose whether he wants to use VeriFast as a tool to only detect certain kinds of errors (unexpected exceptions and incorrect use of the API), or whether he wants to prove full functional correctness of the applet. Both modi operandi are supported by the tool. For the open source applet, we used the annotations to prove that the applet does not contain transaction errors, performs no out of bounds operations on buffers, does not perform invalid casts, and never dereferences null pointers. The annotations in the commercial applet supported the same guarantees as for the open source applet, as well as full functional correctness of the applet.

The eID applet and helper classes consist of 1,573 lines of Java Card code. In order to verify the project, we added 602 lines of VeriFast annotations (or about one line of annotations for every three lines of code). The majority of these annotations were **`requires`**/**`ensures`** pairs (88 pairs, one for each method) and **`open`** and **`close`** statements (99 and 112 instances respectively). Remarkably, only 8 predicates are defined throughout the entire code base, reflecting the design decision of the authors of the applet to write most of it as one huge class file.

The commercial applet consists of 348 lines of Java Card code, which we annotated with 218 lines of VeriFast annotations. There were 13 **`requires`**/**`ensures`** pairs, 25 **`open`** statements and 29 **`close`** statements. The applet required a higher density of annotations (about one line of annotations for every one and a half lines of code) because of the full functional correctness verification.

Another type of annotation overhead is the time it took to actually write the annotations. The verification of the eID applet was performed by a senior software engineer without prior experience with the VeriFast tool, but with regular opportunities to consult VeriFast expert users during the verification effort. We did not keep detailed effort logs, but a rough estimate of the

Listing 10: A more complete definition of the *File* and *ElementaryFile* predicates that supports downcasting.

```
1  public abstract class File {
2     /*@ predicate File(short theFileID, boolean activeState,
3             any info) =
4           this.fileID |-> theFileID &*&
5           this.active |-> activeState &*& info == unit; @*/
6
7     ...
8  }
9
10 public final class ElementaryFile extends File {
11    /*@ predicate File(short theFileID, boolean activeState,
12            quad<DedicatedFile, byte[], short, any> info) =
13          ElementaryFile(theFileID, ?dedFile, ?dta, activeState,
14          ?sz, ?ifo) &*& info == quad(dedFile, dta, sz, ifo); @*/
15    /*@ predicate ElementaryFile(short fileID,
16            DedicatedFile parentFile, byte[] data, boolean activeState,
17            short size, any info) =
18          this.File(File.class)(fileID, activeState, _) &*&
19          this.parentFile |-> parentFile &*&
20          this.data |-> data &*& data != null &*& this.size |-> size
21          &*& array_slice(data, 0, data.length, _) &*&
22          size >= 0 &*& size <= data.length &*& info == unit; @*/
23
24    ...
25 }
```

effort that was required is 20 man-days. This includes time spent learning the VeriFast tool and the Java Card API specifications. The commercial applet was annotated by a VeriFast specialist and took about 5 man-days, excluding the time it took to add some new required features to the tool.

4.2 Bugs and Other Problems

Because the eID applet in our case study is aimed at developers, the authors did not spend a lot of time worrying about card tearing. This is demonstrated by the fact that they did not use the Java Card transaction system at all. Using VeriFast, we found 25 locations where a card tear could cause the persistent memory to enter an inconsistent state.

Three locations were found where a null pointer dereference could occur. An additional three variable casting problems were found, where a variable holding a reference to the selected file (of type `File`) was cast to an `ElementaryFile` instance. These bugs could be triggered by sending messages with invalid file identifiers to the smart card. Seven potential out of bounds operations were also found in the code. These bugs could be triggered by sending illegal messages to the smart card.

The commercial applet had already been verified using another verification technology, so it was not very surprising that no functional problems were found in the code. However, VeriFast *did* identify four locations in the code where transactions were not used properly. Transactional safety is a property that the other tool did not verify.

4.3 VeriFast Strengths

Compared to other program verifiers that target Java Card [Mos07, FRL+02], VeriFast has two advantages: speed and soundness. That is, VeriFast usually reports in only a couple of seconds (usually less) whether the applet is correct or whether it contains a potential bug. Secondly, if VeriFast deems a program to be correct, then that program is guaranteed to be free from unexpected exceptions, API usage and assertion violations.

A feature that proved to be crucial in understanding failed verification attempts is VeriFast's symbolic debugger. As shown in Figure 1, the symbolic debugger can be used to diagnose verification errors by inspecting the symbolic states encountered on the path to the error. For example, if the tool reports an array indexing error, one can look at the symbolic states to find out why the index is incorrect. This stands in stark contrast to most verification condition generation-based tools that simply report an error, but do not provide any help to understand the cause of the error.

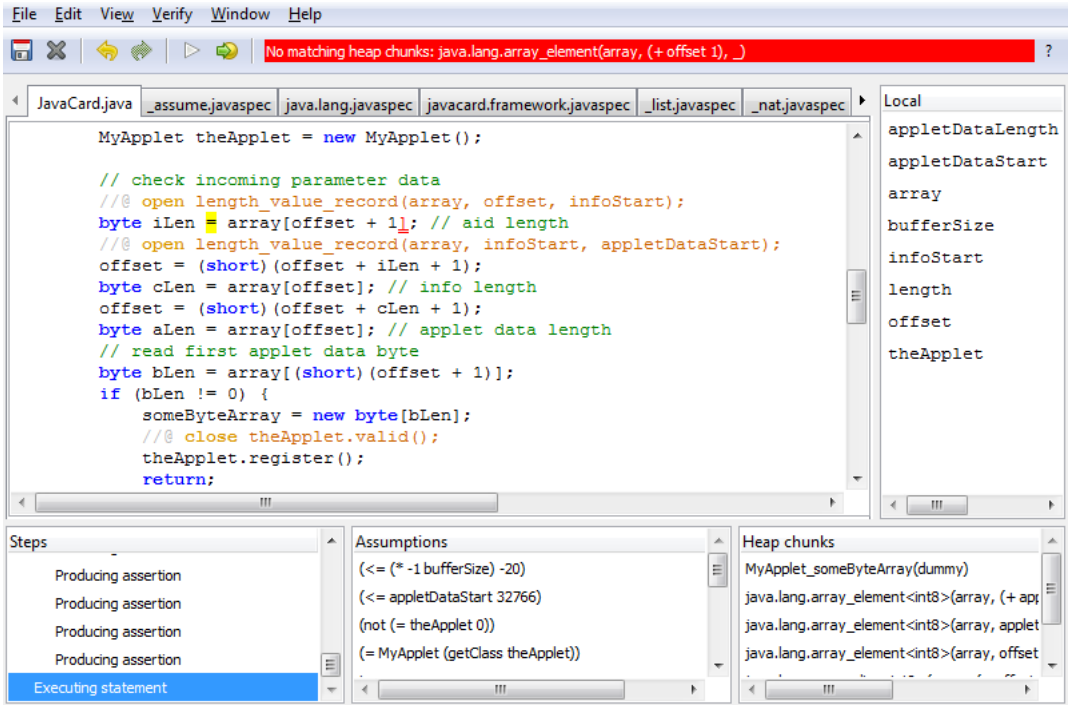


Figure 1: The symbolic debugger of VeriFast

5 Future Work

This case study has led to a number of useful insights and showed us some of the rougher edges of the tool that need to be polished some more. Most of the issues were small and were either bugs in the tool (for instance, Java parsing errors) or functionality that was easy to implement but hadn't been done yet due to time constraints.

An important, missing feature that would greatly reduce the annotation overhead (and hence reduce the cost of verification) is automatic inference of open and close statements and of lemma applications. For example, the eID applet contains 99 open and close statements. While VeriFast already infers some ghost statements, we believe one of the most important steps to improve the verification experience is extending this inference mechanism.

Currently, only a subset of the Java Card API is supported by VeriFast. For example, we do not support multi-applet applications that communicate via the shareable interface mechanism yet. We intend to support these additional features and write specifications for all library functions in the Java Card API.

6 Conclusion

This paper reported on a case study for the VeriFast program verifier. Two non-trivial Java Card applets were annotated and verified for correctness with respect to certain common programming errors. In particular, the verification proved that the applet does not contain transaction errors, performs no out of bounds operations on buffers, does not perform invalid casts, and never dereferences null pointers. In addition, one of the applets was verified for full functional correctness as well.

The results of the case study are encouraging: with an annotation overhead of about one line of annotations per three lines of code we found a total of 13 bugs in the eID applet, and 25 locations where transactions were not properly used.

This case study has been invaluable for us to improve the tool. A number of bugs were fixed and small additions were made in order to support the verification of the applets. A longer term plan has also been established to further add improvements and optimizations to the tool. In particular, the automatic generation of `open` and `close` statements will become an important part of the future work, as well as language and technology-specific extensions to the tool.

Acknowledgements: This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, by IWT, by the Research Fund K.U.Leuven, and by the EU FP7 project SecureChange.

Bibliography

[BCO05] J. Berdine, C. Calcagno, P. O'Hearn. Symbolic Execution with Separation Logic. In *Asian Symposium on Programming Languages and Systems (APLAS)*. 2005.

[Boy03] J. Boyland. Checking Interference with Fractional Permissions. In *Static Analysis: 10th International Symposium*. 2003.

[CH] N. Cataño, M. Huisman. Formal specification of Gemplus' electronic purse case study using ESC/Java. In *Formal Methods Europe*. Pp. 272–289.

[FRL+02] C. Flanagan, K. Rustan, M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata. Extended Static Checking for Java. In *Programming Language Design and Implementation (PLDI)*. 2002.

[GMD05] B. E. G. Gomes, A. M. Moreira, D. Déharbe. Developing Java Card Applications with B. In *Brazilian Symposium on Formal Methods (SBMF)*. 2005.

[HGSC04] M. Huisman, D. Gurov, C. Sprenger, G. Chugunov. Checking Absence of Illicit Applet Interactions: A Case Study. In *Formal Aspects of Software Engineering*. 2004.

[JP11] B. Jacobs, F. Piessens. Expressive modular fine-grained concurrency specification. *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2011)* 46(1):271–282, 2011.

[JSP10] B. Jacobs, J. Smans, F. Piessens. A Quick Tour of the VeriFast Program Verifier. In *Asian Symposium on Programming Languages and Systems (APLAS)*. 2010.

[LNS00] K. R. M. Leino, G. Nelson, J. B. Saxe. ESC/Java User's Manual. Technical report, Compaq Systems Research Center, 2000.

[Mos07] W. Mostowski. Fully Verified Java Card API Reference Implementation. In *International Verification Workshop (VERIFY)*. 2007.

[MP10] W. Mostowski, E. Poll. Midlet Navigation Graphs in JML. In *Brazilian Symposium on Formal Methods (SBMF)*. 2010.

[Ora11] Oracle. Java Card Technology. 2011.
http://www.oracle.com/technetwork/java/javacard/overview/

[ORY01] P. O'Hearn, J. Reynolds, H. Yang. Local Reasoning About Programs that Alter Data Structures. In *International Workshop on Computer Science Logic (CSL)*. 2001.

A Visualization Framework for the Modeling and Formal Analysis of a Computer Based Interlocking System

Qiuzi Lu, Tianhua Xu, Tao Tang, Haifeng Wang, Yan Cao and Gengqin Chen*

10120302,thxu,ttang,hfwang,09120343,07274032@bjtu.edu.cn
State Key Laboratory of Rail Traffic Control and Safety
Beijing Jiaotong University
Beijing 100044, P.R.China

Abstract: A functional specification for a Computer Based Interlocking System (CBI) plays a vital role in signalling design and installation processes. Many attempts have been made to verify interlocking tables by model checkers in the area of train signaling systems. Unfortunately, the complexity and volume of the verification results tends to make them hard for users to understand. In order to tackle this problem, this paper introduces a generic visualization framework that provides developers with visual interpretation of the analysis results based on Domain Specific Language for Computer Based Interlocking Systems (DSL-CBI). Within this framework, a playback mechanism modifies the railway station that depicts the problem scenario. It displays the execution path that has led to a model checking violation. We also discuss the advantages of the framework and the significant contribution in developing CBI based on the proposed toolset.

Keywords: Computer Based Interlocking System (CBI); interlocking table; railway station; counterexample visualization

1 Introduction

Due to its important role in providing safe conditions for train movements, the Computer Based Interlocking System (CBI) is considered to be a safety critical system. The interlocking table of a given station defines all the train routes and the concrete safety rules associated with these. So the interlocking table is considered to be the foundation during the development of an interlocking system. Mechanization of the verification of interlocking tables from the railway station model can be an efficient approach to guarantee the reliability of overall interlocking systems [1].

Recently, there have been several efforts to translate interlocking tables from the railway station model to formal specification languages to be analyzed for adherence to behavioral properties by model checkers, such as Spin and NuSMV. They are intended to support the verification of interlocking tables. That is, does the CBI satisfy the Fail-safety criterion in railway signaling systems. A notable feature of model checkers is that if a system model does violate a property, a

* The authors wish to acknowledge the support of NSFC No.60736047, the State Key Laboratory of Rail Traffic Control and Safety of Beijing Jiaotong University within the frame of the project (No. RCS2008ZZ005) and Scientific Research Foundation for Scholars (Beijing Jiaotong University, Grant No. 2011JBM160).

counterexample depicting the violation is returned [2]. However, when we want to make use of the results of the counterexample, we are faced with two challenges. First, the counterexample file is usually verbose and non-intuitive we have to spend a lot of time deciphering it. Second, it is difficult to manually link the cause of the error with the original interlocking table, so we can't quickly locate the path in the original diagram. This paper describes a counterexample visualization framework that interprets the analysis output from model checkers in terms of the original railway station. It consists of two parts: the trace interpreter and the visualization tool. Based on numerous trace output files generated from each model checker, firstly, we constructed a parser, and then developed a translator for each model checker to be supported by this framework; the parser extracts useful information for a given trace file, then the translator for each formal analysis tool that traverses the information to generate a generic XML representation containing only the railway station model elements, such as conflicting segment numbers. Secondly, we developed a visualization tool that processes the XML representation of the counterexamples to mark on the railway station model. In summary, if we input a railway station model and the trace file for a counterexample generated from a model checker for an error detected in the interlocking table which corresponds to the given railway station, the visualization framework can illustrate the violation trace in terms of the railway station.

The visualization framework has been developed to provide a critical piece of a larger project supporting a roundtrip-engineering approach to the construction of a railway station for modeling and analyzing the CBI requirements. Specifically, we have previously developed a toolset based on Domain Specific Language for Computer Based Interlocking Systems (DSL-CBI) [3], in which both a route information search algorithm and a symbolic model checker are integrated. The route information search algorithm can be used to generate interlocking tables by inputting the XML file of the railway station designed by DSL-CBI, whereas the symbolic model checker is used to guarantee the correctness of the generated interlocking tables. Finally, a counterexample visualization framework is tailored to understand the analysis result easily. DSL-CBI is developed by the formal verification research group of Rail Traffic Control and Safety State Key Laboratory in Beijing Jiaotong University. The model designed by DSL-CBI is stored in XML format.

The structure of this paper is as follows: Section 2 briefly describes the background of the supporting elements of the roundtrip-engineering process. Section 3 gives the architecture for the visualization framework and describes the visualization capabilities. Section 4 presents a case study involving the NuSMV model checker results. Finally, Section 5 concludes the paper and presents some discussion of future work.

2 Background

Fig. 1 shows the activity diagram depicting the analysis process for CBI. The counterexample visualization framework is illustrated in the shaded area. Moreover, we describe the process of creating a railway station model designed by DSL-CBI, automatic generation of the interlocking table of the railway station, and verification of the generated interlocking table via model checking.

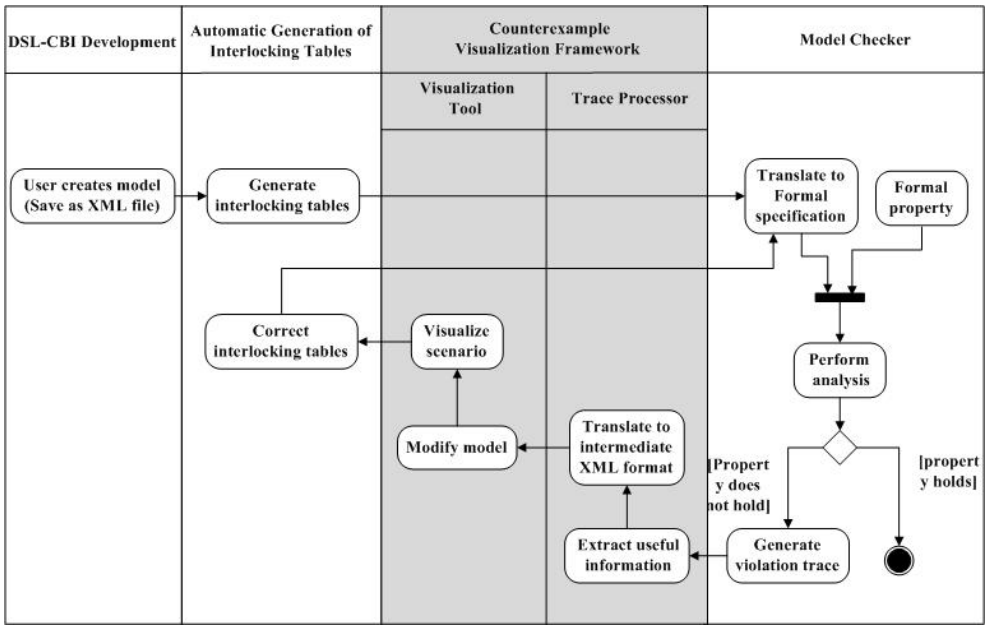


Figure 1: Activity diagram depicting the whole process

2.1 Step 1: Creating a railway station model

In the first step, the developer creates a railway station model designed by DSL-CBI.This process comprises three steps: 1. DSL-CBI description; 2. Define a modeling language; 3. Create a DSL-CBI editor.

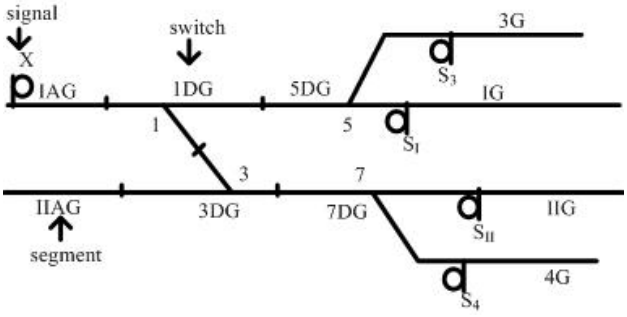


Figure 2: Example of railway station

Fig. 2 shows a simplified model of a railway station, which contains segment, signal, switch, etc. Interlocking systems specify the necessary conditions between physical objects of the railway station. All entities have two states (e.g., segment: occupied or clear; signal: proceed or stop; switch: normal or reverse). In other words, the function of interlocking systems is preparing a route for a train. When a train is entering the station, some actions must be taken. For example, the segment must be cleared and the switch must be set in the right position. After

these, the entry signal of the route can be lit. In these conditions, the route is ready so the train can proceed along the route. Simultaneously, these key safety requirements are recorded in the interlocking table.

2.2 Step 2: Automatic generation of the interlocking table

For the railway station created in step 1, we provide an algorithm for automatic generation of the interlocking table from the station model. For a given station, all the train routes and the concrete safety rules associated with these are defined in the interlocking table of the station. Table 1 depicts a sample of an interlocking table. If a train is moving according to the Route1, starting with entry signal X (yellow(U) indicates proceed), ending with exit signal SI and passing Segments IAG, 1DG, 5DG, IG, it will collide with the other train which started moving from signal D5 or SI. Therefore, Route1 and Route2 are conflicting routes. Notice that Route1 and Route3 are not conflicting routes. They can't be used at the same time because the switch 1/3 (1/3 means switch 1/3 is required to be in the normal position; (1/3) means it is required to be in reverse) is not required in the same state.

Table 1: Sample of Interlocking Table

R-Number	R-Type	Entry Signal	color	Exit Signal	Switch	Conflict Signal	Segment
1	TrainRoute	X	U	SI	1/3 5	SI	IAG,1DG,5DG,IG
2	TrainRoute	SI	U	X	1/3 5	X	IAG,1DG,5DG,IG
3	TrainRoute	X	UU	SII	(1/3) 7	SII	IAG,1DG,3DG,7DG,IIG

2.3 Step 3: Verification of the generated interlocking table via model checking

In order to check whether the interlocking table which generated automatically satisfies the safety properties, the developer translates the interlocking table into a FSM model that defines the behavior of the physical objects of the railway station, based on generic interlocking table semantics, which define the meaning of the interlocking table. Suppose that all entities can be clearly moved to the required position. A FSM model is given as a sequence of states S_i; it translates from a given initial state S_0 by executing transition $\stackrel{\sigma_1}{}$. Each transition is specified by a group of transition rules. Specifically, the train will move from one segment to another (S_i represents Segment i) if one transition is satisfied[3].

$$S_0 \stackrel{\sigma_1}{\longrightarrow} S_1 \stackrel{\sigma_2}{\longrightarrow} \ldots \stackrel{\sigma_n}{\longrightarrow} S_n \ldots$$

It then puts this model together with a model that captures assumptions on how trains can move through a layout. This final model is exhaustively checked against the safety properties, using a model checking algorithm. In this paper, the model-checking tool of choice is NuSMV. If the model checker finds a violation of the safety property, it reports an output file to the developer. An output of most model checkers is a counterexample: an execution trace illustrates exactly how a specification was violated. In most analysis environments, this output is a list of the model variables and their values at each step in the execution trace [4].

3 Counterexample visualization framework

Verification of the generated interlocking table via model checkers has been done in the previous development. Once we obtain a NuSMV code we can try to model-check it and report the result to the developer. If there is no violation, then the specification has been formally proven to be valid for the model. Otherwise, a counterexample is produced. This counterexample contains the sequence of steps performed by the system that led to the violation. Comparing the conflicting routes the model checker detects the conflicting routes described in the interlocking table, there may be the wrong generated interlocking table to lead to the non-correspondence between them.

```
/* description */ "(AG (~(t1.position=t2.position))) "
/* time */ 1309419696
/* state 1 */{
...
#18 "interlocking_revised_over.smv"
\t1 .\position   = \s17 ,
...
#20 "interlocking_revised_over.smv"
\t2 .\position   = \s13 ,
...
/* state 2 */{
...
#400 "interlocking_revised_over.smv"
\t1 .\position   = \s15 ,
#362 "interlocking_revised_over.smv"
\t1 .\st_Train   = \S17 ,
...
#374 "interlocking_revised_over.smv"
\t1 .\before_Train   = \s17 ,
...
#400 "interlocking_revised_over.smv"
\t2 .\position   = \s15 ,
#362 "interlocking_revised_over.smv"
\t2 .\st_Train   = \S13
}
```

Figure 3: The corresponding violation trace generated by NuSMV

Fig. 3 is an excerpt of the corresponding violation trace generated by NuSMV. The violation trace has been used to repeat the verification process, through comparing results by changing some steps of the process. Hence the essential information should be offered : the safety property that has been verified, the moving traces of the trains based on the interlocking table, the NuSMV program codes and the internal NuSMV information. This case of two trains collision violation trace is just an example for analysis. Of course, more general properties and safety properties can be verified in this way.

However, many users need to comprehend the structure and behavior of interlocking tables that they may not be acquainted with. How to understand and use the error descriptions from

the analysis results to revise the original interlocking table is a tough challenge. The counterexample visualization framework, shown in the shaded region of the activity diagram in Fig. 2, is explicitly designed to help analysts interpret counterexamples. It supports visual interpretation of the analysis results generated by model checkers in terms of the original railway station.

In this subsection, we develop a playback mechanism to display the execution path that has led to a model checking violation. It consists of two components to depict this scenario: the trace interpreter and the visualization tool. From the trace files, the counterexample visualization framework extracts useful information such as the train's id, the train's position, etc. According to the information, the interpreter specifies the analysis results in an intermediate XML representation. In this way, a flexible, useful and uniform data interchange format is built for CASE tools, which is a key issue to make it much easier to develop visualization tools.

The visualization tool takes the XML intermediate representation and the original railway station model as inputs and produces the modified railway station. Here we describe the trace interpreter and visualization tool in more detail.

3.1 Violation trace interpreter

The main safety property of a railway interlocking system is that two trains will never collide. Now we take this safety requirement for an example. Firstly, we analyse the contents of an interlocking table which includes the safe train movement conditions, and we manually check for any conflicting settings in the table. Then we translate the interlocking table into formal specification languages and use NuSMV to analyze the formalized interlocking table for adherence to the safety requirement. If there is a violation, an execution trace is produced. It includes the detailed information of a trains moving trace. Our goal is to identify the collision with the pass path within the trace file and to specify this behavior in an intermediate XML format. Fig. 4 shows that the trace file is translated into an intermediate XML format.

To achieve these ends, we constructed a parser which must be constructed for each syntactically unique trace file format and developed a translator to generate a generic XML representation.

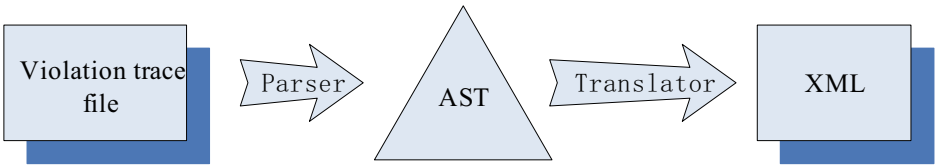


Figure 4: Violation trace interpreter overview

3.1.1 The trace parser

At this stage, we have gotten the trace file in the previous development. An excerpt of the corresponding violation trace generated by NuSMV is shown in Fig. 3. The first phase of the violation trace interpreter is trace parsing. Note that different formats of trace files will be generated by different input options. However, each parser is reusable. In this section, we construct a parser

for violation trace files created by NuSMV. The goal of the parser is to identify the collision with the pass path and detect that the code in the trace file is either syntactically ill formed or semantically unsound.

As Fig. 5 shows, there are three components in the trace parser: 1. The extractor extracts information which describes conflict specified by the trace file then outputs the simplified code. 2. The lexical analyzer reads the stream of characters making up the simplified code and produces tokens. 3. The syntax analyzer uses the tokens to constructs an AST (abstract syntax tree) representation of the dynamic behavior. The syntax analyzer interacts with the lexical analyzer. It invokes the lexical analyzer when it needs a token during execution.

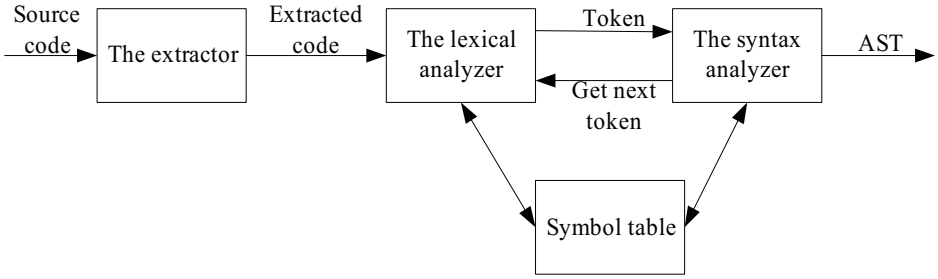


Figure 5: The structure of a parser

A. The extractor We present a search algorithm for extracting information about the movement of the trains. Violation traces generated by NuSMV provide four different sources of information: the safety properties that have been verified, the trains moving trace according to the interlocking table, the NuSMV program code and internal NuSMV information.

The extractor extracts the information corresponding to dynamic behaviors of interest. Some examples are given as follows: The safety properties that have been verified.

```
/* description */ "(AG (~(t1.position=t2.position))) "
```

This safety property can be described as two trains (t1 and t2) can not occupy the same segment at the station. It means no conflict is allowed.

The violable trace works step by step.

```
/* state 1 */
```

It shows the trace at the first step.

A segment has been visited:

```
\t1 .\before_Train  = \S17
```

This statement specifies that the train t1 occupied s17 earlier.

A segment is visited:

```
\t1 .\position  = \S15
```

This statement specifies that the train t1 occupies s15 now.

So does the train t2.

The extractor preserves the train movement information. If t1 and t2 occupy the same segment, it means that two conflicting routes are set at the same time. Documenting the analysis results according to the NuSMV, the parser can figure out the conflicting segment easily.

Algorithm

```
1:  Read the first state
2:  if the TrainID ==t1 then
3:    CurrentRoute: =RouteStack.firstTrain
4:  else
5:    CurrentRoute: =RouteStack.secondTrain
6:    SaveTrainID (TrainID)
7:  endif
8:  <RouteSet ID>: =SearchTrainPosition(CurrentRoute)
9:  if RouteSet is not empty then
10:   RouteStack.push(CurrentRoute)
11:     for(Route i:∈RouteSet)
12:       if Route i in RouteStack then
13:         continue
14:       end if
15:     end for
16:  end if
17:  if <RouteSet 1>(i)==< RouteSet 2>(j)
18:   Save ConflictingRoute=<RouteSet 1>(i)
19:  end if
20:  Read the next state
```

Figure 6: Algorithm of route information search

The specific search algorithm for extracting information which describes conflict specified by the trace file is illustrated in Fig. 6.

It is obvious that the extracted code should be broken up into constituent pieces and impose a grammatical structure on them. So we also need a lexical analyzer and a syntax analyzer.

B. The lexical analyzer At first, the lexical analyzer reads the characters of the source code and then the characters will be grouped into meaningful sequences. That is called lexeme. At the same time, a token for each lexeme will be produced by the parser. The token consists of two

parts: token-name and attribute-value.

For example, suppose a source program contains the assignment statement

```
\t1 .\position  = \s15
```

The representation of the assignment statement after lexical analysis as the sequence of tokens

```
< trainid, 1> <.> <positon, 2> <=> <segid, 3>
```

Blanks and meaningless symbols separating the lexemes would be discarded by the lexical analyzer.

C. The syntax analyzer Here by the syntax analyzer, the token-name produced by the above lexicer is used to create a tree-like intermediate representation. The tree depicts the grammatical structure of the token stream and can then be translated to an AST.

The grammar used in NuSMV is probably not comprehensive, since the violation trace file not only contains the information from the NuSMV program code. Thus, defining the best grammar is not quite effortless. By analyzing a good deal of violation trace files, we define the code syntax rule by using Extended Backus Naur Form (EBNF).

In EBNF the grammar can be written as:

```
assignment statement : : = trainid "." affair * : : = var ;
trainid : : = letter {letterOnum}* num ;
affair : : = trainstateOexternalstate;
var : : = segidOboolen;
trainstate : : = positionObeforeposition
...
```

3.1.2 The translator

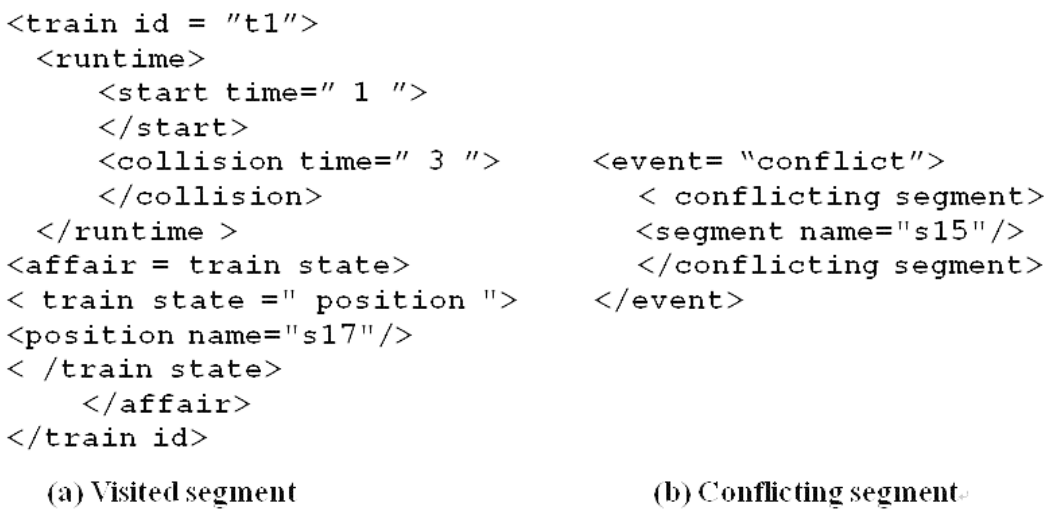
The translator traverses the AST and creates an intermediate XML representation of the dynamic behavior. We have defined a set of XML tags and attributes. A well-defined intermediate XML representation is used to store this scenario.

The intermediate XML representation provides programmers with a way to easily obtain information about violation traces in a self-descriptive manner, which is useful to develop a visualization tool. Furthermore, the intermediate XML representation can also be viewed as a DSL, because they can be good languages to describe the trains moving. It contains information of the abstract syntax tree and the static semantics.

Fig. 7(a) shows a sample XML element, which specifies that in the first state t1 occupies segment6. Fig. 7(b) specifies the segment where the two trains collide. The intermediate XML is automatically generated from the translator, and then used as input for visualization tools.

3.2 Visualization tool

Because the railway station established by DSL-CBI is externalized into XML, it is possible to mark the conflicting segment by manipulating the XML file. We need to read an existing


<train id = "t1">
<runtime>
<start time=" 1 ">
</start>
<collision time=" 3 ">
</collision>
</runtime >
<affair = train state>
< train state =" position ">
<position name="s17"/>
< /train state>
</affair>
</train id>
(a) Visited segment
<event= "conflict">
< conflicting segment>

</conflicting segment>
</event>
(b) Conflicting segment

Figure 7: Sample of XML elements

XML file which includes all information about the railway station, modify it and write it back. Modifying can be pretty complex depending on trains actions performed on the counterexample.

The visualization tool we have developed has the following features: 1. By analyzing the counterexample, we obtain the XML file and the property which needs to be verified. 2. Putting all the properties into the corresponding places in the options screen, we can chose running step by step, so it is possible to observe the whole driving process of the trains one by one. 3. Another option is that you can press the run-all button and it will execute the whole driving process by all trains.

Reading the scenario from the XML file and presenting the counterexample through the railway station, this processing program helps users to understand the cause of a property violation clearly. Any special ways to highlight the conflicting routes can be used to reveal the fault of the interlocking table. Specifically, the modified railway station depicts that a track is visited (increased the display font size of the segments, colored it in blue or green and make it italic) and that a conflicting segment exists (increased the display font size of the segments and colored them in red). In this way, the user is alleviated from the burden of deciphering the frequently cryptic and verbose trace output.

A case study about the whole toolset is presented in Section 4.

4 Case study

This section describes a case study we performed to validate our visualization framework. Firstly, the DSL-CBI is developed to describe railway stations. Secondly, an algorithm is proposed to automatically generate interlocking tables. Thirdly, conflicting routes in the generated interlocking tables are checked by NuSMV. Lastly, the conflicting routes of the railway station are vividly expressed in the railway station through the counterexample framework. So that we can

compare the conflicting routes the model checker detects the conflicting routes described in the interlocking table. Fig. 8 shows an excerpt of the whole process.

The railway station designed by DSL-CBI is depicted in Fig. 8(a). Fig. 8(b) presents the generated interlocking table according to the railway station. Fig. 8(c) shows the verification of the generated interlocking table via NuSMV, Fig. 8(d) describes the result of verification by outputting a counterexample in tabular form. A violation trace in terms of the railway station is illustrated as Fig. 8(e).

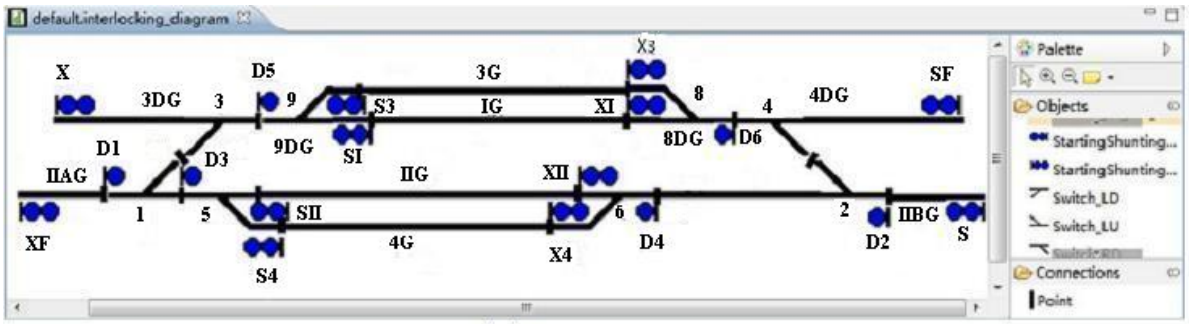
According to Fig. 8(c) and Fig. 8(d), trains t1 and t2 starting from two states (3DG and 4DG), will collide at IG. It means that all the train-routes (the train runs ignoring shunting signals such as D5, if not, it will be called shunting-route) starting from the signal X and the signal SF are conflicting routes. The result is in line with the information in the generated interlocking table (Fig. 8(b)). Finally, a screen shot of the modified railway station depicting the violation trace is shown in Fig. 8(e), where the key thing to note is the different colors and size of the segment's name. By choosing the train ID in the drop-down list, the track of the train's movement can be displayed clearly. When t1 is selected, the segments it has passed will be marked in blue. Similarly, when t2 is selected, the segments it has passed will be marked in green. Finally, we can see that 'IG' is the conflicting segment, which is pointed out in red and framed.

With the railway station and the corresponding violation trace as input, the counterexample visualization framework processed the violation trace and visualized the counterexample in terms of the original railway station. It is easy to find that the result of model checking meets the requirement (safety property) of the generated interlocking table. The counterexample framework makes consistency more visible through the modified railway station.

5 Conclusions

In this paper, we have presented a generic visualization framework. It provides a critical link in a roundtrip-engineering process for modeling and analyzing interlocking systems. Using the visualization framework, the user is alleviated from the burden of deciphering the frequently cryptic and verbose trace output, which is often denoted in an analysis tool-specific language, including references to line numbers of the specification, internal process numbers, temporary variable names, etc. Moreover, the users have the option of either train running through the complete counterexample, where color and size changes are used to depict the train movement according to the interlocking table. So they can understand the case of the error well, even if the error exists in a hidden place. Specifically, the visualization framework completes the roundtrip modeling and analysis process for CBI. In order to validate our work, we have applied our roundtrip-engineering process to the analysis of several different railway stations.

Other tools [5] [6] visualize analysis results from model checkers in terms of UML. To the best of our knowledge, none of these tools is concerned with the generation and visualization of violation traces for the railway station. In contrast to work suggested by others [7] [8] [9], our work focuses on the results of a model checker. Our visualization is explicitly designed to help analysts interpret counterexamples. However, such a visualization may be useful for debugging models during development. Without the counterexample visualization framework, we would be forced to understand the syntax and semantics of the trace output and determine the



(a)

R_Number	EntrySignal	ExitSignal	Switch	ConflictSignal	Segment
1	X	SF	1/3 9 8 2/4	SI D5 D6 SF	3DG,9DG,IG 8DG 4DG
2	XF	S3	(1/3) (9)	D1 D5 S3	IIAG 1DG 3DG 9DG 3G
3	X	SI	1/3 9	D5 SI SF	3DG 9DG IG
4	X	S3	1/3 (9)	D5 S3 SF	3DG 9DG 3G
5	SF	X	1/3 9 8 2/4	XI D5 D6 X	3DG,9DG,IG 8DG 4DG

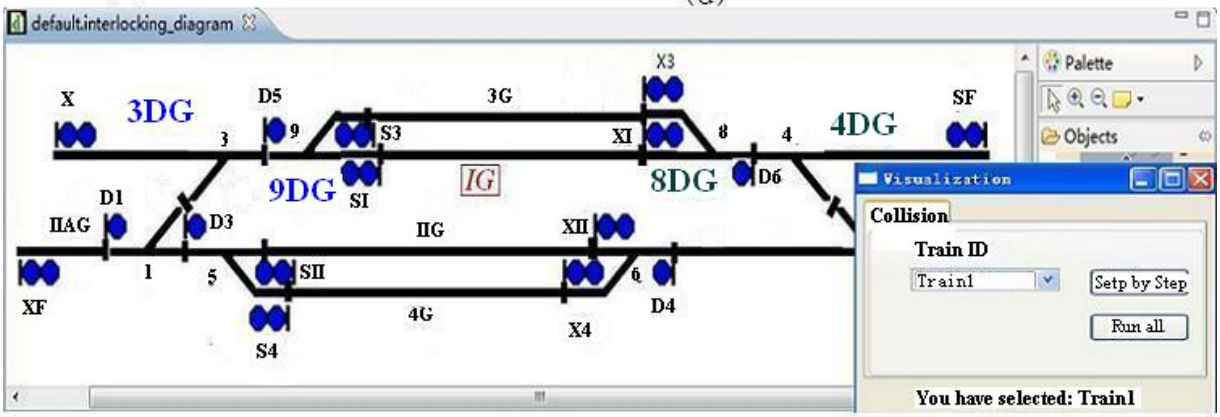
(b)

```
   t1 : Train(4DG,8DG,
3DG,9DG);

   t2 : Train(4DG,8DG,
 4DG,8DG);

 --** SYSTEM PROPERTIE
S    (CTL FORMULAS)*
 SPEC AG!(t1.position
 =t2.position)
```

(c)

t1.P8S4	0	0	0	t2.P8S4	0	0	0
t1.P9	0	0	0	t2.P9	0	0	0
t1.P9S3	0	0	0	t2.P9S3	0	0	0
t1.before_Train	3DG	9DG	IG	t2.before_Train	4DG	4DG	8DG
t1.in_S5	0	0	0	t2.in_S5	0	0	0
t1.in_S6	0	1	0	t2.in_S6	0	0	0
t1.in_S9	0	0	0	t2.in_S9	0	0	0
t1.position	9DG	IG	IG	t2.position	4DG	8DG	IG

(d)



(e)

Figure 8: Whole process of the case study

relationship between the output and the railway station. Even if the developer is very familiar with the interlocking table, it will take a lot of time to locate the cause of the error in the railway station.

As future work, we plan to present a step-by-step animation of the counterexample through the railway station, which will make the violation trace more vivid.Furthermore, a more seamless integration should be explored between tools we have developed before to further improve our system.

Bibliography

[1] A. Mirabadi, M. B. Yazdi.: Automatic Generation and Verification of Railway Interlocking Control Tables Using FSM and NuSMV. Transport Problems, pp103-110 (2009)

[2] Heather Goldsby, Betty Cheng, Sascha Konrad, Stephane Kamdoum.: Enabling a roundtrip engineering process for the modeling and analysis of embedded systems. Computer Science, pp707-721 (2006)

[3] Yan Cao, Qiuzi Lu, Tianhua Xu, Tao Tang, Haifeng Wang, Yongcheng Xu.: Integrating DSL-CBI and NuSMV for modeling and verifiying Interlocking Systems. The Fifth IEEE International Conference on Secure Software Integration and Reliability Improvement (2011)

[4] Matthew L. Bolton, Ellen J. Bass.: Using Task Analytic Models to Visualize Model Checker Counterexamples. Systems Man and Cybernetics (SMC), 2010 IEEE International Conference (2010)

[5] Lilius, J., Paltor, I.P.: vUML: A tool for verifying UML models. In: Proc. of the14th IEEE Int. Conf. on Automated Software Engineering, Washington, DC (1999)

[6] Mikk, E., Lakhnech, Y., Siegel, M., Holzmann, G.J.: Implementing state charts in promela/spin. In: WIFT '98: Proc. of the Second IEEE Workshop on Industrial Strength Formal Specifcation Techniques (1998) 90

[7] Winter, K., Johnston, W., Robinson, P., Strooper, P., van den Berg, L., .: Modeling Large Railway Interlockings and Model Checking Small Ones. Proceedings of the 26th Australasian Computer Science Conference 35 (2003) 309-316

[8] Winter, K., et al.: Tool Support for Checking Railway Interlocking Designs. Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software (2006) 101-107

[9] Huber, M.: Towards an industrially applicable model checker for railway signaling data. University of York. Master's thesis(2001)

A Survey on Event-B Decomposition

Thai Son Hoang[1] and Alexei Iliasov[2] and Renato A Silva[3] and Wei Wei[4]

[1] Department of Computer Science
ETH-Zürich
Email: htson@inf.ethz.ch
and
[2] School of Computer Science
Newcastle University, UK
Email: Alexei.Iliasov@ncl.ac.uk
and
[3] School of Electronics and Computer Science
University of Southampton, UK
Email: ras07r@ecs.soton.ac.uk
and
[4] SAP Research Darmstadt
SAP AG
Email: wei01.wei@sap.com

Abstract: Model decomposition is a powerful tool to scale the design of large and complex systems. It enables developers to separate components development from the concerns of their integration and orchestration. Event-B is a refinement-based formal method, equipped with three decomposition styles that come with solid semantic foundations and strong tool support. This paper intends to give some useful insights and modelling guidelines for using these decomposition styles, illustrated by an actual development of a master data updating system.

Keywords: Decomposition, Event-B, Modeling, Guidelines, Formal Methods

1 Introduction

Modern software systems are becoming more complex everyday. This trend is going to continue as industry witnesses unprecedented explosive demands for social and business connectivity. This poses a huge challenge to system modelling for rigorous quality assurance, while scalability becomes a great issue. Hence model structuring is paramount in mastering complexity of large systems. Although there has never been a lack of theoretical research in this area, few modelling platforms provide a sufficiently good level of tool support to evaluate structuring techniques and their impact on industrial application of formal modelling. Moreover, there are no clear guidelines for model designers which often impedes the application of formal verification to large-scale designs.

Event-B [3] has established itself as a popular formal system development with broad industrial adoptions. Event-B was designed as a minimalistic formalism to form the core of the Rodin platform [4]. A great range of features have been provided as extensions to the modelling platform, covering almost all aspects of model-driven engineering such as requirement, simulation, visualization, testing, formal verification, and code generation. The extension architecture

allows a modeller to enable only those features that are relevant for the domain of modelled problem. This keeps the core features of the platform as simple as possible while not sacrificing any functionalities.

The basic Event-B language supports model structuring by nothing more than the use of events and refinement inside one model. However, it is highly impractical to construct a large scale formal design as one monolithic model, which results in numerous problems with legibility, maintainability, team work, reuse, and so on. Notably, it also affects proof structuring: autonomous provers are suffocated by a large number of hypotheses and thus anything that makes the context of a proof smaller is extremely beneficial.

Model decomposition comes as a great potential to solve the above problem. Three different decomposition styles exist for Event-B, all providing a guarantee of refinement monotonicity: the model after decomposition is a correct refinement of the one before decomposition, provided all obligated proofs are given. This allows a decomposed part of the model to be treated as an independent artefact so that the modeller can concentrate on this part and does not have to worry about the other parts. The tool supports for these techniques, realized as Rodin plugins, not only provide assistance on how to decompose a model, but also generate explicit constraints and relations between the decomposed model and the resulting sub-components.

Technically, decomposition is a special form of refinement by which a single abstract model is refined by several concrete models and their aggregation. Like refinement, a properly planned decomposition step results in very low proof cost. It requires, however, a good degree of foresight. First, a model is difficult to decompose if the model does not possess a structure that can be easily divided and mapped to independent components to be produced by decomposition. Second, any careless design decision before decomposition may be difficult to correct afterward, which often leads to complete rework on the development of each individual local component. Therefore, a modeller can serve a better job when some guidelines are available so that the modeller knows which decomposition style to choose from, and which design decisions need to be taken care of at each stage.

In this paper we provide some insights and propose modelling guidelines for applying model decomposition, drawn from our personal experiences and illustrated by a case study by an industrial user of Event-B. These guidelines are not supposed to give a one-for-all solution for all decomposition attempts. We introduce stages in decomposition and point out which design problems (that can be easily overlooked) need to be addressed in each step.

Outline. Sec. 2 briefly introduces Event-B and decomposition in general. Sec. 3 gives our modelling guidelines. Different decomposition styles are detailed in Sec. 4 – Sec. 6 by showing their applications on the modelling of a master data updating system. We compare the different approaches in Sec. 7. Finally, we draw some conclusions in Sec. 8.

2 Background

2.1 Event-B

Event-B is a formal modelling method for developing *correct-by-construction* hardware and software systems. An Event-B model is a state transition system where the state corresponds to a set of *variables* v and transitions are represented by a collection of *events* evt. The most general

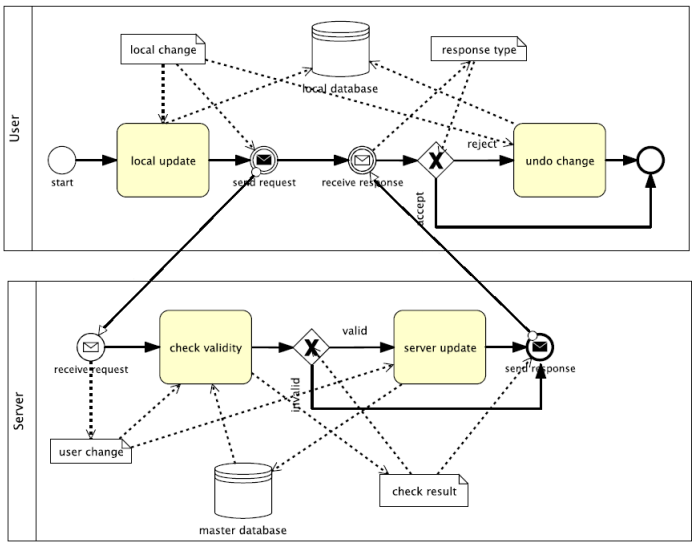


Figure 1: The Process Model of Update Master Data

form of an event is: evt $\widehat{=}$ **any** t **where** $G(t,v)$ **then** $A(t,v)$ **end** , where t is a set of parameters, $G(t,v)$ is the enabling condition (called guard) and $A(t,v)$ is an action changing the value of v. An action comprises several assignments executing in parallel. Each assignment can have one of the following forms: $x := E(t,v)$, $x :\in S(t,v)$ or $x :| P(t,v,x')$, where x are some variables in v. The first form assigns value of expression $E(t,v)$ to x. The second assignment form non-deterministically assigns to x some element of set $S(t,v)$. The third assignment form non-deterministically assign to x some after value x' satisfying the *before-after predicate* $P(t,v,x')$. In the first and last assignment forms, x can be a vector of variables. The last assignment form is also the most general one: other assignment forms can be equivalently represented using before-after predicates. Essential to Event-B is the formulation of *invariants* $I(v)$: safety conditions to be preserved at all times.

To facilitate the construction of large-scale models, Event-B advocates the use of *refinement*: the process of gradually adding details to a model. An Event-B development is a sequence of models linked by refinement relations. It is said that a concrete model refines an abstract one. Abstract variables v are linked to concrete variables w by a *gluing invariant* $J(v,w)$. Any behaviour of the concrete model must be *simulated* by some behaviour of the abstract model, with respect to the gluing invariant $J(v,w)$.

Rodin [4] is an industrial-strength toolset supporting Event-B. Rodin provides an integrated modelling environment with a range of editors, modelling assistants, automatic generator of verification conditions and a set of automated provers tasked to discharge verification conditions.

Example – A Master Data Updating System Fig. 1 shows a master data updating system that we use as the case study of this paper. The system consists of a User process and a Server process keeping some master data in sync. When User proposes a data change, it first updates its local copy, and then sends a request message to Server and waits for the answer. Upon receiving

the request, Server checks the validity of the proposed change, and updates the master copy only when the change is deemed valid. Then, Server sends to User a response containing either an approval or a rejection. User has to roll back the change if a rejection is received. We are interested in the global property that the *local and master databases are always identical before and after each data update procedure*.

The Event-B model in Fig. 2 serves as the most abstract view of the above system. The initial model is designed to have as few variables and events as sufficient to express the above mentioned property (see invariant **inv0_3**). The model contains variables *udb* and *sdb* denoting User and Server's database respectively. The Boolean variable *is* denotes if the global system is *in synch*. There are three events, namely u_update, s_update and u_final. When the system is in synch, u_update changes the local database, which invalidates the insynch status. While the system is out of synch, s_update may occur to update the server database (either to be the same as *udb* or unchanged). Finally, u_final occurs to put the system back in synch by making the local database to be identical as the server database. Note that s_update may be skipped when the system is out of synch. Although this cannot happen in the real system, we permit it in the abstract model to simplify proofs, which does not affect the satisfaction of the global property in consideration. This spurious behavior will be removed by refinement and decomposition in the further developments.

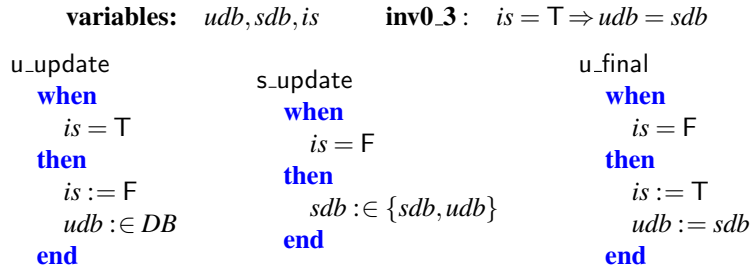


Figure 2: The top abstract model of Update Master Data

2.2 Decomposition

The *top-down* style of development used in Event-B allows the introduction of new events and data-refinement of variables during refinement steps. A consequence of this development style is an increasing complexity of the refinement process when dealing with many events and state variables. *Decomposition* addresses such difficulty by providing a mechanism for splitting a large model into several sub-models (that can be further developed independently). Several decomposition techniques have been proposed by extending the existing Event-B notation. This paper is concerned with three existing approaches: *shared-variable* [2], *shared-event* [7] and *modularisation* [11], all of which are supported by Rodin plug-ins [13, 1]. These decomposition techniques differ in that different model elements are shared among sub-components. For shared-variable decomposition, a part of state information (*variables*) is shared among sub-components. Further refinements then concentrate on how each component processes shared state information. For

shared-event decomposition, a set of *events* are synchronised and shared by sub-components. Hence, it is important to take care of the inputs/outputs of these synchronised events. Modularisation defines a set of *interfaces* that are shared and accessed by different components. Interfaces provide callable operations and promises that these operations can deliver. The implementation of an operation should guarantee that the promises are fulfilled for any given circumstance.

Shared-variable decomposition is similar to rely/guarantee approach from Jones [12]: internal/external events is essentially an encoding of rely/guarantee conditions. It also corresponds to *concurrent action systems* [5] where a solution for the interleaving semantics is proposed.

Shared-event decomposition allows separation of aspects by using synchronisation and communication based on Butler's work [6] combining Action System and CSP [10]. CSP value passing channels correspond to events that communicate via shared parameters.

The separation of procedure declaration from implementation have a long history both in computer programming and modelling. Modularisation closely relates to the treatment of procedures in Hoare logic [9, 8]: procedure calls are used as a metaphor to benefit from refinement monotonicity. Consequently independent model aspects can be considered separately although this should not be confused with modelling a procedure call as a construct of a programming language.

3 Guidelines

The primary challenge of applying decomposition is to ensure that the structure of the original model fits the requirements of the chosen decomposition style, leading to helpful sub-models that can be developed separately with a tangible advantage in terms of proof efforts and overall model scale. As with any *top-down* approach for system development using refinement, the more abstract models are initially, the more useful the decomposition step will be. Here we do not focus on directly justifying the use of a particular decomposition style. Instead we focus on *how to proceed* when decomposing using one of the suggested decomposition styles.

We define a general top-down guideline for the three decomposition techniques based on the following common template.

Stage 1 To model the system abstractly, expressing all the relevant global system properties.

Stage 2 To refine the abstract model to fit the structure expected by a given decomposition technique.

Stage 3 To apply decomposition.

Stage 4 To develop the resulting sub-models independently.

Following this guideline, global properties are captured early in the model and guaranteed to hold in the final models by combining refinement and decomposition. The development of each decomposed part is done independently of the others. Consequently, we can have different implementations for a decomposed model that is guaranteed to work with any implementation of other decomposed models.

In the subsequent sections, we elaborate on the application of different decomposition techniques using our proposed modelling guideline.

4 Shared-Variable Decomposition

Consider Fig. 3 where machine M has four events, e1 to e4, and three variables, *v1* to *v3*. The solid lines connect variables used by events. In Fig. 3, M is *shared-variable* decomposed and events are partitioned into sub-components: e1 and e2 are allocated to machine M1; e3 and e4 are allocated to machine M2. Consequently, *v1* belongs to M1 and *v3* belongs to M2 (*private variables*). Variable *v2* is *shared* between M1 and M2. Furthermore, additional *external events* are required to simulate how shared variables are handled in the other sub-component (e3_e is added to M1 and e2_e to M2). Assuming that e2 has the general form

$$\mathsf{e2} \;\widehat{=}\; \textbf{any}\ t\ \textbf{where}\ G(t,v1,v2)\ \textbf{then}\ v1,v2\ :|\ P(t,v1,v2,v1',v2')\ \textbf{end}\ ,$$

the corresponding external event e2_e can be generated as follows.

$$\mathsf{e2\_e} \;\widehat{=}\; \textbf{any}\ t,v1\ \textbf{where}\ G(t,v1,v2)\ \textbf{then}\ v2\ :|\ \exists v1'\cdot P(t,v1,v2,v1',v2')\ \textbf{end}\ .$$

Intuitively, e2_e is a projection of e2 onto the state without variable $v1$.

There exist certain constraints about shared-variable decomposition during the development of the resulting sub-components: they can be refined independently but shared variables and external events *must be present and cannot be refined*. More information on shared-variable decomposition in Event-B can be found in [2].

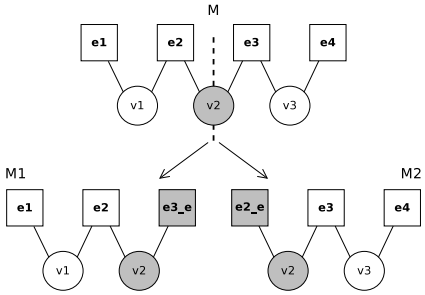


Figure 3: Shared-variable decomposition

4.1 Master Data Updating System

We describe in detail how to develop the example of update master data using shared-variable decomposition. The model described in Section 2 represents our abstract model of **Stage 1**. We continue with the subsequent stages of our modelling guideline.

Stage 2. Shared Channels Between Components. In this preparation stage, we introduce the channels (the shared elements) acting in between User and Server. The channels are modelled by two variables *creq* and *cres*, corresponding to the set of messages going through the request and response channels respectively.

Memo SV1 In this preparation stage, variables going to be shared are introduced.

Moreover, since in **Stage 4** the shared variables and external events can be *neither removed nor refined*, the shared elements introduced in this preparation stage must be *concrete*.

> **Memo SV2** The shared variables must be concrete.

Furthermore, we are going to split the variables and events into two groups, corresponding to each site, preparing for the later decomposition step. An important design constraint here is that User's events can only reference the variables belonging to User and the shared channels, but not the variables of Server. The same for Server's events.

> **Memo SV3** Events belonging to each sub-component only reference its own variables and shared variables.

As a result, variable *is* must be refined away. We replace *is* by *uis*, the local in-synch flag, with a gluing invariant $uis = is$, i.e. the global in synch is consistent with the User's view. A separated in-synch flag *sis* is introduced for Server. Moreover, in order to separate User and Server completely, we introduce new variables for keeping some information belong to each site. For User, variable *udb_old* is added in order to keep the old value of User's database for undoing later if necessary. For Server, variable *sc* keeps the user's change to the database on the server site for updating Server's database if the change is valid.

Despite of the details that we have to introduce in order to clearly separate the future sub-components, we aim to keep the model at this stage fairly abstract. It should contain only necessary information for maintaining the global properties and specifying the shared elements between future sub-components. Other information, e.g. control/data flows within each sub-component can/should be abstracted away.

> **Memo SV4** Unnecessary details irrelevant to decomposition should be abstracted from the model in **Stage 2**.

For example, we assume for the moment that the update of the database and sending the request message from User happens simultaneously. This is represented by event u_update_and_req, a refinement of the abstract event u_update.

u_update_and_req **refines** u_update
any ch **where**
$uis = \mathsf{T} \wedge ch \in CH$
then
$uis, udb, udb\_old, creq := \mathsf{F}, upd(udb \mapsto ch), udb, \{ch\}$
end

In u_update_and_req, the local database *udb* is updated to be $upd(udb \mapsto ch)$, the new value obtained by applying changes *ch*; the old local database is saved in *udb_old*; and the actual change is send as a request to the server via channel *creq*. Note that u_update_and_req satisfies our **Memo SV3**, i.e. reference only variables belonging to User and the shared channel *creq*.

Other events in this model include: s_receive_req for Server to receive some request; s_accept_res for Server to update its database and send a positive response; s_reject_res for Server to send a negative response without updating its database; u_receive_res_acc and u_receive_res_rej for User to receive some (positive/negative) response and act accordingly.

An important advantage during the model design in this stage is the use of the abstract model from **Stage 1**. Consistency enforced by refinement guides our design in **Stage 2**, i.e. constraints on the shared variables will be *derived* from the need to maintain the global properties introduced in **Stage 1** (typically in terms of invariants).

In our example, the following invariants are *discovered* during the process of discharging proof obligations such as guard strengthening and invariant preservation of the model. They relate the content of the channels and the internal status of User and Server. Invariants **inv1_7** and **inv1_8** relate Server's database *sdb* with the User's database (current *udb* or old *udb_old*) depending on the content of the channel *cres*. Invariants **inv1_9** and **inv1_10** state that User is out of synch if there are some request or response messages.

$$\begin{array}{ll} \textbf{inv1\_7}: & cres = \{\mathsf{T}\} \Rightarrow sdb = udb \\ \textbf{inv1\_8}: & cres = \{\mathsf{F}\} \Rightarrow sdb = udb\_old \\ \textbf{inv1\_9}: & creq \neq \varnothing \Rightarrow uis = \mathsf{F} \\ \textbf{inv1\_10}: & cres \neq \varnothing \Rightarrow uis = \mathsf{F} \end{array}$$

Stage 3. Decomposition Summary. This stage is semi-automatic: we provide the tool with input on how the events are partitioned into different future sub-models. Intuitively, we separate our events into two groups, corresponding to User and Server accordingly. The variables distribution amongst these model are calculated according to the information about events distribution. The summary of our decomposed models is as follows.

	User	Server
Internal events	u_update_and_req u_receive_res_acc u_receive_res_rej	s_receive_req s_accept_res s_reject_res
Private variables	udb, udb_old, uis	sdb, sc, sis
Shared variables	$creq, cres$	$creq, cres$

Stage 4. Developments of Sub-models. We present a summary of the additional refinement steps for each model. Most invariants in the sub-models are technical and related to the sequentialisation of the actions, reflecting the process flows in Figure 1.

User The control flow is introduced via means of a program counter *upc* to capture the actual sequential steps inside the User process. Other internal variables of User are introduced accordingly, i.e. User's change *uch* and User's stored response type *ures*.

Server Similarly, the control flow of Server is introduced via means of a program counter *spc*. Internal variables of Server, such as the check result *scr*, are introduced.

5 Shared-Event Decomposition

In Fig. 4, M is *shared-event* decomposed into two parts: M1 and M2. Variables are partitioned into the sub-components: *v1* is placed in M1 and *v2*, *v3* are placed in M2. Unlike the shared variable approach, no variable sharing is allowed. Events using variables allocated to different sub-components (e2 shares *v1* and *v2*) must be *split*.

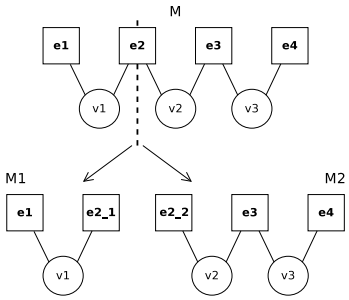


Figure 4: Shared-event decomposition

Assuming that event e2 has the following form

$$
\begin{array}{l}
\mathsf{e2} \\
\quad \textbf{any} \quad t \quad \textbf{where} \\
\qquad G_1(t, v1) \\
\qquad G_2(t, v2) \\
\quad \textbf{then} \\
\qquad v1 \ :| \ P_1(t, v1, v1') \\
\qquad v2 \ :| \ P_2(t, v2, v2') \\
\quad \textbf{end}
\end{array} \quad ,
$$

event e2_1 is defined as a partial version of e2 only referring to variable *v1*, i.e.

$$\mathsf{e2\_1} \ \widehat{=} \ \textbf{any}\ t, v1\ \textbf{where}\ G(t, v1)\ \textbf{then}\ v1 \ :| \ P_1(t, v1, v1')\ \textbf{end}\ .$$

Event e2_2 is defined similarly but only refers to *v2*.

> **Memo SE1** The abstract model of shared-event decomposition is such that each event updating non-private variables may be syntactically split into two events.

5.1 Master Data Updating System

Using the same initial model in Fig. 2, we describe the following stages in the application of a shared event decomposition. The system is designed to be decomposed into components User and Server *synchronously* communicating by value passing messages.

Stage 2. The Value Passing Protocol. The goal of this stage is to have a model where the state variables are *partitioned* amongst the future sub-models. Typically, this stage involves refinement of events to introduce the shared elements in the form of *events' parameters*. Similar to the shared-variable style, a good abstract model is pursued where only necessary information related to the global properties and the shared elements are specified.

> **Memo SE2** Irrelevant details should be abstracted away from the model before decomposition.

In this refinement, we prepare the decomposition by introducing synchronous channels and respective value passing protocol. The content of the protocol is represented by shared parameters (in the resulting sub-events). At this stage, the communication is abstract and occurs in a single event.

Memo SE3 Shared elements are introduced by means of event parameters.

The global flag *is* is replaced by *uis* and *sis* for User and Server sync respectively. The gluing invariants between *uis*, *sis* and *is* are given by *inv*1_1, *inv*1_2 and *inv*1_3: *uis* always matches *is*; while a request is being processed, *sis* matches *is*; otherwise, the server is synchronised ($sis = \mathsf{T}$).

variables:	$udb, sdb, uis,$	**inv1_1**	$uis = is$
	u_ch, u_rq	**inv1_2**	$u\_rq = PRC \Rightarrow sis = is$
	sis, s_st, s_ch	**inv1_3**	$u\_rq \neq PRC \Rightarrow sis = \mathsf{T}$

Some control variables are added: *u_ch* corresponds to the User change; *u_rq* holds the request state on the User side where *PRC* corresponds to the processing state; *s_st* corresponds to the server state and *s_ch* holds the change from the Server's viewpoint. Refined event u_update models a modification that is stored in *u_ch* before being sent to the server by the new event rq. Event rq simultaneously sends the request from User and receives it in the server. Then the request is stored in *s_ch* and User ($u\_rq := PRC$) and Server ($s\_st := VAL\_RQ$) states are updated . The server is considered out of sync once receives a request ($sis := \mathsf{F}$).

u_update **refines** u_update
  any *ch* **where**
    $ch \in CH$
    $uis = \mathsf{T}$
    $u\_rq = IDLE$
  then
    $uis, u\_ch := \mathsf{F}, ch$
  end

rq
  any *msg* **where**
    $uis = \mathsf{F} \wedge msg = u\_ch \wedge u\_rq = IDLE$
    $s\_st = S\_IDLE$
  then
    $u\_rq := PRC$
    $s\_ch, s\_st, sis := msg, VAL\_RQ, \mathsf{F}$
  end

Note that event rq has been designed so that it can be syntactically split into parts concerning only with variables of the User or Server (**Memo SE1**). The request validation can be deferred until the decomposition because it is irrelevant to the considered global property. The server is updated in the refined event s_update for a valid request. Even when the request is deemed invalid, a response is sent back by the new event rsp. This event syncs in the Server and updates the User's request. If the request is valid, *u_ch* is applied locally; if the request is invalid, *udb* remains the same. In either case, *udb* is back in sync with the server.

Several gluing invariants are discovered as a result of the generated proof obligations. **inv1_4** state that while *u_rq* is processed, User/Server changes match; if *u_rq* is deemed invalid, *sdb*/*udb* are identical (**inv1_5**); a valid request results in *sdb* matching with *udb* updated with *u_ch* (**inv1_6**).

inv1_4 :	$uis = \mathsf{F} \wedge u\_rq = PRC \Rightarrow s\_ch = u\_ch$
inv1_5 :	$u\_rq = INVLD \Rightarrow udb = sdb$
inv1_6 :	$u\_rq = VLD \Rightarrow upd(udb \mapsto u\_ch) = sdb$

Our model is synchronous since the messages exchanged by User and Server are sent and received simultaneously. Alternatively, we could also model asynchronous communication by introducing a *buffer* between *udb* and *sdb* suggesting a three way decomposition.

Stage 3. Decomposition Summary. Sub-models User and Server result from the allocation of the original variables according to their use. The decomposition is summarised in the following table:

	User	Server
Variables	udb, u_ch, uis, u_rq	sdb, s_st, sis, s_ch
Events	u_update, u_final, rq, rsp	s_update, rq, rsp

Stage 4. Developments of Sub-models. The decomposition allows the separation of sending/receiving a request by defining the request as parameter *msg* shared by User and Server (similarly applied to the server's response). The resulting sub-models can be refined independently:

User A program counter is added defining the local states (update *udb*, send request, receive response, commit/discard change). Two events refined the two possible outcomes: *l_commit* for valid modifications updating *udb* and *l_discard* to discard the modification. An additional refinement could add a request queue removing the waiting between the server reply and the next modification.

Server The server is refined by modelling the request validation with a new event s_val.

6 Modularisation

In modularisation, *interfaces* are defined for sub-components such as the interface I in Fig. 5, which contains interface variable *iv* and operations o1 and o2. Operations are specified by pairs of pre/post-conditions. An interface is separated from its implementation IM that provides concrete behavior for each of the interface operations. An abstract machine of the integrated system is modeled in M, which is refined by M1 where sub-component behavior is replaced with respective operation calls.

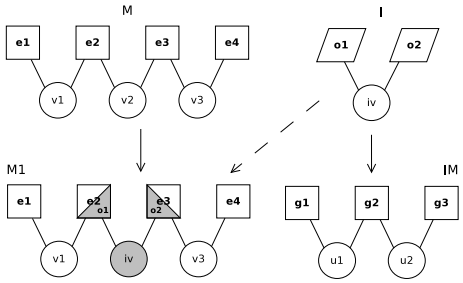


Figure 5: Decomposition via modularisation

In M, logical subunits must be identified so that they can be easily mapped to operation calls. Respective variables in M are mapped to interface variables, and actions can be replaced by interface operations. An interface must be carefully designed such that a minimal level of details of the sub-component behavior is exposed.

Memo M1 A module interface should be as general (weak) as possible because of re-usability.

Following the above principle, we should expect that sometimes an interface is too weak in that the post-conditions of its operations are not strong enough to establish the soundness of the caller machine.

Memo M2 In case of a too weak interface, we should strengthen operation post-conditions using the undischarged proof obligations as guidance.

Machine M1 can be further refined. However, operation calls to o1 and o2 must stay intact.

Memo M3 Operation calls must be preserved in the chain of refinements except for parameter refinement.

Unlike the two other decomposition approaches, modularisation offers a greater degree of flexibility on how to construct a decomposed specification because of lower coherence among sub-components. One consequence is that the modularisation approach applies to both *top-down* and *bottom-up* designs, and even blurs the boundary. For top-down development, the guideline in Sec. 3 can be used in modularisation approaches as well. In bottom-up development, sub-component interface may already exist before a global integration scenario is designed. This is useful for many industrial use cases in which service integrations and customizations are formally analyzed.

6.1 Master Data Updating System

We take a different approach here that does not follow the guideline in Sec. 3. Even though we will explain the approach in a top-down fashion, it resembles certain aspects of a bottom-up development in that the design of sub-component interfaces is relatively independent of the global integration, because the interfaces are standard communication interfaces that produce and consume messages.

We start with an abstract machine that only specifies message flows and does not state the global property. Interfaces are defined for the two processes and implemented by separate machines by adding local control and data flow information. A sufficient amount of implementation details, such as local variables and properties, is carefully chosen and exposed in their interfaces to enable the verification of the global property. Finally, the top abstract machine is refined by adding details of operation calls and message buffers. The global property is then verified on a final refinement.

Stage 1. Specifying Abstract Message Flows. The top abstract machine uses flags to indicate whether a certain message has been sent or received. For example, if the flag *req_snt* is ⊤ then a request message has been sent. Several invariants describing the order of message events are added for property verification later. As an example, **inv6** specifies that a response message has to be sent before it can be received. Message events are abstract at this level and simply set the flags accordingly.

variables: $req\_snt, req\_rcv, res\_snt, res\_rcv$
inv6 : $res\_snt = \mathsf{F} \Rightarrow res\_rcv = \mathsf{F}$
send_req $\widehat{=}$ **when** $req\_snt = \mathsf{F} \wedge res\_rcv = \mathsf{F}$ **then** $req\_snt = \mathsf{T}$ **end**

Stage 2. Process Interfaces. The interface of each process defines a set of message operations. These operations do not consider how messages are transported, but merely specify the types of messages that they provide or expect. For example, interface User provides messages to be sent (get_request), and take incoming messages fed to them for local consumption (put_response). In particular, get_request produces a message equivalent to the local change (ch) proposed by the user.

get_request
pre
$req\_snt = \mathsf{F} \wedge res\_rcv = \mathsf{F}$
return
msg
post
$msg' = ch$
$req\_snt' = \mathsf{T}$
end

put_response
any msg
pre
$msg \in \mathrm{BOOL} \wedge req\_snt = \mathsf{T}$
$res\_rcv = \mathsf{F}$
post
$ures' = msg$
$res\_rcv' = \mathsf{T}$
end

Stage 3. Process Implementations. Concrete details of sub-components are added in implementations, such as events that describe how control states and local variables are updated. Message-related flags are no longer present, thus we provide a link between those flags and local control states (u_cs) as gluing invariants like the one below among others.

inv18 $req\_snt = \mathsf{F} \Leftrightarrow u\_cs \in \{start, upd, req\}$

An interface implementation is essentially an Event-B refinement step. We need to prove that the postcondition of any interface operation must be fulfilled by the corresponding events that implement the operation. We also need to prove relative deadlock freedom that, whenever an interface operation is enabled, some of its implementing events must be executable.

Stage 4. Final Global Machine. The top-level machine is refined at this level to contain operation calls and actual message exchanging behavior. Each process has a buffer to store incoming messages. When a message needs to be sent, the corresponding interface operation of the sender process is called to retrieve the outgoing message, which is then added to the corresponding buffer. When a message is to be received, the message is taken out of the buffer and passed to the receiver process by calling the respective interface operation. In the following code, the prefix `user_` is used in interface variables and operations of the `user` module.

send_req $\widehat{=}$ **when** $user\_req\_snt = \mathsf{F} \wedge user\_res\_rcv = \mathsf{F}$ **then** $buf\_s = buf\_s \cup \{\text{user\_get\_request}\}$ **end**

Stage 5. Property Verification. Unlike the other approaches, the global property is encoded and proved at the final global machine. The proof is based on the symbolic values of the local and master databases delivered as operation post-conditions in the process interfaces.

7 Discussions

All approaches decompose global machines into two components, one per process[1]. Shared-variable and shared-event approaches start with similar abstractions, specifying a minimal set of events reflecting how the local and remote databases are updated while preserving the global property of interest. A series of refinements are introduced with appropriated chosen gluing invariants and proofs of deadlock freedom and convergence. The two approaches are different in that the shared-event version implements a synchronous message passing model. The choice is mostly motivated by the fact that synchronised message passing events can be shared by two processes. However, the system could be modelled in an asynchronous communication by introducing a *buffer* sub-component. These two approaches ensure that the global properties are preserved before decomposition. Afterwards each individual sub-component focus on their specific properties. A system involving a shared object is favoured by a shared variable decomposition where the shared object can be accessed by all the sub-components. On the other hand, communicating system parts can be shared event decomposed possibly introducing a middleware to allow an asynchronous approach.

In contrast to the other approaches, the modularisation version formulates and proves the global property at the final stage, after the module interfaces are designed. This is possible because the global machines and modules are loosely coupled, only linked by interfaces. An advantage that immediately comes to mind is flexibility: changes to existing components and inclusion of new components do not necessarily affect unchanged modules nor their proofs. However, the largest challenge of modularisation is the design of appropriate interfaces as they play a crucial role in linking multiple worlds while preserving the global property. During the design of our model, we go through iterations of "trial and error" to find the appropriate amount of information to be exposed in interfaces. As we learned from our experiences, a good practice is to start with an initial interface containing a minimal amount of information and weak possible post-conditions for operations. When these are insufficient to prove the global property, we can gradually bring more information to the interface and strengthen post-conditions.

8 Conclusion

We have presented modelling guidelines for three decomposition techniques of Event-B, illustrated by a case study. Due to limited space, it was impossible to tell the complete story of the case study development. We emphasise on the several design decisions that were taken during the development of the examples, in order to successfully apply decomposition, resulting in helpful sub-models for further independent elaboration. Together, the guidelines and these design decisions act as a document of our experience in applying decomposition techniques.

Decomposition is a powerful technique to cope with the complexity of system development but applying decomposition comes at a price: the cost of planning a development strategy that fits a certain decomposition style. It turns out that mastering the technicalities related to establishing decomposition correctness and even an extensive tool support are still not enough to systematically apply decomposition. Decomposition is rarely successful without prior planning

[1] The models are available online at http://eprints.ecs.soton.ac.uk/22164/

and such development planning is not yet supported by an established methodology.

One fascinating topic for future research is to see how different decomposition techniques complement each other in the same development. The diversity in decomposition approaches may be exploited to make decomposition more flexible and simpler for an end user. We are also planning to formalise the criteria of model decomposability and, if successful, mechanise them in a tool. Such a tool, in principle, could give an immediate answer on which decomposition technique, if any, would succeed for a given model.

Acknowledgment. This work has been supported by the EC FP7 Integrated Project Deploy, the EPSRC grant TrAmS (EP/E035329/1) and Fundação Ciência e Tecnologia (FCT-Portugal).

Bibliography

[1] Modularisation plug-in for Event-B. http://wiki.event-b.org/index.php/Modularisation_Plug-in.

[2] J.-R. Abrial. Event model decomposition. Technical Report 626, ETH Zurich, May 2009.

[3] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[4] J.-R. Abrial, M. J. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010.

[5] R-J Back. Refinement Calculus II: Parallel and reactive programs. In *Proc. of REX 89*, volume 430 of *LNCS*, pages 67–93. Springer, 1989.

[6] M. Butler. Stepwise refinement of communicating systems. *Sci. of Comp. Prog.*, 27(2):139–173, September 1996.

[7] M. Butler. Decomposition structures for Event-B. In *IFM*, volume 5423 of *LNCS*, pages 20–38. Springer, 2009.

[8] D. Gries. *The Science of Programming*. Springer, 1987.

[9] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In *LNM*, 188, pages 102–116. Springer, 1971.

[10] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[11] A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, D. Ilic, and T. Latvala. Supporting reuse in Event B development: Modularisation approach. In *ASM*, volume 5977 of *LNCS*, pages 174–188. Springer, 2010.

[12] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[13] R. Silva, C. Pascal, T. S. Hoang, and M. Butler. Decomposition tool for Event-B. *Software: Practice and Experience*, 41(2):199–208, February 2011.

Partial-Order Reduction for Verifying PLUSCAL-2 Algorithms

Sabina Akhtar and Stephan Merz

LORIA – INRIA Nancy and Nancy University, Nancy, France

Abstract:

The PLUSCAL-2 language allows algorithm designers to formulate concurrent and distributed algorithms in a form that is similar to pseudo-code. However, the language has a precise semantics, and the PLUSCAL-2 compiler generates a TLA$^+$ model that can be verified using the TLC model checker. We adapt Holzmann's partial-order reduction technique to PLUSCAL-2 algorithms in order to alleviate the state explosion problem. In particular, we exploit the structure of PLUSCAL-2 algorithms for computing predicates that ensure that the TLA$^+$ actions corresponding to two PLUSCAL-2 statements (or groups of statements) are independent. We have extended the TLC model checker to take into account these predicates and reduce the number of states that must be explored. We validate our approach by showing its efficiency for several examples drawn from the literature.

Keywords: Model checking, partial-order reduction, verification, distributed algorithms

1 Introduction

Model checking [CGP99] is a popular verification technique for concurrent and distributed algorithms. It provides tools for deciding automatically whether properties (typically expressed in temporal logic) are verified for finite instances of systems or algorithms, described in a formal modeling language. Its main limitation is the well-known state explosion problem, which can be mitigated by verifying algorithms at a high level of abstraction. For example, Lamport's specification language TLA$^+$ [Lam02], which is supported by the model checker TLC [YML99], is intended for high-level models of distributed and concurrent algorithms. Although TLA$^+$ is very expressive and can concisely describe complicated algorithms, algorithm designers are often reluctant to adopt formal modeling languages, and prefer to express their ideas in pseudo code or similar informal notation. Lamport therefore introduced the PLUSCAL algorithm language [Lam06]. While retaining the high level of abstraction of TLA$^+$ expressions, it provides familiar constructs of imperative programming languages for describing algorithms, such as processes, assignments, and control flow.

The PLUSCAL compiler generates a TLA$^+$ specification, which is then verified using TLC. PLUSCAL is a high-level and powerful modeling language for algorithms, featuring mathematical abstractions, non-determinism, and user-specified grain of atomicity; it emphasizes the analysis, not the efficient execution of algorithms. Aiming at a simple translation to TLA$^+$, PLUSCAL also imposes some significant limitations. In particular, it does not support process hierarchies, which are often natural for distributed algorithms, where physically separate nodes communicate by message passing, but local threads access shared memory. Moreover, PLUSCAL does not

enforce variable scoping, and it requires the user to state properties in TLA$^+$ rather than at the PLUSCAL level. In previous work, we proposed a new version PLUSCAL-2 [AMQ10] that aims at overcoming these limitations.

We encoded several distributed algorithms in PLUSCAL-2, compiled them to TLA$^+$ models, and used the TLC model checker to verify the algorithms. However, the well-known state space explosion problem limits the sizes of instances that can be verified effectively. Since TLA$^+$ and PLUSCAL-2 are mainly intended for verifying asynchronous distributed algorithms, it became clear that we should turn to partial-order reduction methods, which are known to provide effective reduction techniques in this context.

Partial-order reduction relies on actions being recognized as independent. It is substantially easier to compute dependencies between PLUSCAL-2 statements than inferring them from the resulting "flat" TLA$^+$ models. We therefore decided to extend the compiler by a static analyzer that produces the necessary information and feeds it as an additional input to the model checker, helping it to reduce the state explosion problem.

The remainder of the paper is organized as follows: we discuss the background and motivations for our work in section 2. In section 3 we define independence predicates for PLUSCAL-2 algorithms. In Section 4 we present our extensions to the TLC model checker. Some experiments and their results appear in section 5. Related work is discussed in section 6, and section 7 concludes the paper.

2 Background

2.1 PLUSCAL-2

Lamport [Lam06] introduced the PlusCal algorithm language, which is intended to make formal verification techniques easily accessible to algorithm designers. PlusCal has the flavor of pseudocode but also has a precise semantics, and the compiler generates a TLA$^+$ model from a PlusCal algorithm, which can then be verified using TLC [YML99], the TLA$^+$ model checker.

We designed a variant of PlusCal, called PLUSCAL-2 [AMQ10], with the objectives of (1) allowing for hierarchical process structures, (2) enforcing the scopes of locally declared variables, and (3) making the language self-contained by including fairness annotations and correctness properties, which in PlusCal need to be manually added in the generated TLA$^+$ model. For the purposes of the present paper, item (2) is the most important one because it helps us determine under which conditions two PLUSCAL-2 statements are independent. As a running example, Figs. 4 and 5 in the appendix show an encoding of an algorithm due to Dolev, Klawe, and Rodeh [DKR82] ("the DKR algorithm") for electing a leader in a unidirectional ring.

The PLUSCAL-2 compiler first converts algorithms into an intermediate language, which is based on atomic blocks of guarded commands [Dij75]. We will explain in Sect. 3 how we generate independence predicates from this intermediate representation.

2.2 Reduction Techniques

The state explosion problem is well known to be the most serious limitation for the application of model checking techniques. It refers to the fact that the state space generated by a transition

# Processes	Time(seconds)	Total States	Distinct States
4	0.2	205	95
8	2.3	31289	7121
10	22.9	352426	63986
12	349.2	3811181	575747

Table 1: Model checking results for the leader election algorithm.

system usually grows exponentially in the number of processes. For example, Table 1 shows the numbers of (total and distinct) states generated when model checking the DKR algorithm using TLC, the TLA$^+$ model checker, after translating the PLUSCAL-2 model to TLA$^+$. Many different techniques have been proposed to mitigate state explosion. Symbolic state space representation aims at compact data structures that are less sensitive to the number of actual states, and state space hashing reduces the memory footprint by storing only hash codes instead of actual states. Symmetry reduction identifies states that are equivalent modulo an equivalence relation, helping to reduce the number of states that need to be explored.

In distributed algorithms, which are the focus of PLUSCAL-2 and TLA$^+$, the main potential for reducing state spaces comes from the fact that many actions executed by different processes commute, i.e. the same global configuration is obtained when performing these actions in either order. Whereas the standard interleaving model of concurrency distinguishes two executions that differ in the order in which two independent transitions are performed, a semantics based on partially ordered executions would identify them. Partial-order reduction techniques [GW94, Val90, FG05, HP94] aim at identifying independent transitions and avoiding the construction of equivalent runs. We have adapted the notion of independence for transitions given by Holzmann and Peled [HP94].

A statement a is characterized by the set $Cond(a)$ of states where a is enabled and, for any state $s \in Cond(a)$, the set $Act(a,s)$ of states that can be reached by executing a in s. Two statements a and b are independent at state s if the following conditions hold:

- $s \in Cond(a)$, i.e., statement a is enabled at s,
- $s \in Cond(b)$, i.e., statement b is enabled at s,
- $Act(a,s) \subseteq Cond(b)$, i.e., the execution of a cannot disable b,
- $Act(b,s) \subseteq Cond(a)$, i.e., the execution of b cannot disable a, and
- $\bigcup_{s' \in Act(a,s)} Act(b,s') = \bigcup_{s' \in Act(b,s)} Act(a,s')$, i.e., the same sets of states can be reached by executing a and b in either order.

In practice, it is usually too costly to compute independence of statements precisely, and one settles for an underapproximation: any two transitions that cannot be shown to be independent are considered as dependent. The following section describes how we compute an approximate independence relation for (blocks of) PLUSCAL-2 statements. Section 4 explains how we have adapted the partial-order technique proposed by Holzmann and Peled [HP94] to PLUSCAL-2 and implemented it in TLC.

3 Computing independence predicates for PLUSCAL-2

We now explain how we compute predicates that ensure that two blocks of PLUSCAL-2 statements are independent at a given state. Our definition is simplified by the fact that PLUSCAL-2 obeys variable scoping and that the compiler outputs an intermediate format that is based on three kinds of blocks.

3.1 Intermediate representation of PLUSCAL-2 algorithms

The PLUSCAL-2 compiler first produces an intermediate representation of the given algorithm, which consists of labeled blocks of loop-free guarded commands that will be executed atomically. Each block is then translated to a TLA$^+$ action, and sequencing between blocks is ensured by adding explicit control variables. More precisely, blocks of the intermediate language are given by the following grammar, where brackets denote optional parts:

$$
\begin{array}{rcl}
\text{block} & ::= & \text{assignment } [\,;\text{block}\,] \\
& | & \exists\, id \in expr : \text{block} \\
& | & \textbf{branch} \\
& & \quad C_1 \ \textbf{then}\ \text{block} \\
& & \quad \vdots \\
& & \textbf{or}\ \ C_n \ \textbf{then}\ \text{block} \\
& & \textbf{end}
\end{array}
$$

Left-hand sides of assignments can be simple variables, array components as in

net[out] := Append(net[out], [type $\mapsto$ “one”, number $\mapsto$ mynumber])

or record components. The existential quantification statement executes its corresponding block for some value of *id* from the set that is produced by evaluating the expression *expr*. The **branch** statement is executable only if some guard (state predicate) C_i is true at the current state.

3.2 Inductive definition of independence predicates

In general, two blocks A and B are independent if they modify different parts of the state space, and if neither reads a variable that may be modified by the other. We will make this intuition more precise by inductively defining a predicate $P_{indep}(A,B)$ that guarantees that blocks A and B are independent at any state satisfying the predicate and where both blocks are enabled. In the definition of $P_{indep}(A,B)$, we make use of an auxiliary predicate $P_{unch}(A,E)$ for a block A and an expression E, which ensures that the value of E is unaffected by the execution of A.

Before we present the formal definition, consider the example where both blocks A and B correspond to the assignment at line 19 of the DKR algorithm, executed by two different processes p and q. The representation of this assignment in the intermediate format is

_net[_Node_data[self].out] :=
Append(net[_Node_data[self].out], [type $\mapsto$ “one”, number $\mapsto$ _Node_data[self].mynumber])

where the index self into array _Node_data denotes the identities of the processes executing the statement. Because out is a local variable, the compiler has transformed the reference to it into the access to the field out of record _Node_data[self], which holds the local variables of process self of process type Node. We might consider the two assignments to be dependent since both update the same global variable net, and compute the independence predicate FALSE. However, they will actually be independent provided the values of the local variables out of processes p and q are different, and we may therefore generate the independence predicate

$$\_\text{Node\_data}[p].\text{out} \neq \_\text{Node\_data}[q].\text{out}.$$

The same predicate is computed as the "unchanged predicate" $P_{unch}(A,E)$, where A is the block corresponding to the single assignment above (executed by process p) and E is the expression Head(net[out]).type = "one" occurring in the code of process q.

Predicate for two assignments. In the definition of the independence predicate for two assignments, we may assume that they are both enabled, and therefore execution of one may not disable the other one. We rely on the set of locations read and updated by two assignments in order to compute their independence predicate. Locations can be scalar variables or array or record components, and we represent them as

$$\text{loc} ::= \text{base} \mid \langle \text{base}, \langle \text{path}, \text{loc}^* \rangle \rangle$$

where *base* is a variable name, *path* is a sequence of expressions e_1, e_2, ... e_n that resolves to an index in an array or record, and loc^* is the set of locations accessed in the *path* expression. For two assignment blocks A and B, we first compute the locations loc_A and loc_B updated by A and B, as well as the sets rd_A and rd_B of locations read by these blocks. For example, if A is

$$a[i] := q[j+4, j+k]$$

then we obtain

$$\begin{aligned} loc_A &= \langle a, \langle [i], i \rangle \rangle \\ rd_A &= \{ \langle q, \langle [j+4][j+k], j, k \rangle \rangle, i \} \end{aligned}$$

where the location i in rd_A comes from the read access to i on the left-hand side of the assignment.

Now, to ensure that the two assignments are independent, we must guarantee that one assignment statement does not update a location that is accessed by the other assignment statement:

- the location written by assignment A should not be read by assignment B,
- similarly, the location written by assignment B should not be read by assignment A, and
- they should not write to the same locations.

We therefore compare pairs of locations $\langle l_1, l_2 \rangle$ and compute predicates P^{l_1,l_2} that ensure that l_1 and l_2 do not denote the same location:

- If the base variables for l_1 and l_2 are different then the locations cannot be the same and $P^{l_1,l_2} = \text{TRUE}$.
- Otherwise, we compute P^{l_1,l_2} by comparing the paths for the two locations.
  - If both the paths have the same number of expressions $e_i^1, \ldots, e_i^n$ (for $i = 1,2$) then

    $$P^{l_1,l_2} \triangleq \bigvee_{j=1}^{n} e_1^j \neq e_2^j.$$

    In particular, $P^{l_1,l_2} = \text{FALSE}$ if $n = 0$.
  - Otherwise, we define $P^{l_1,l_2} \triangleq \text{FALSE}$.

Finally, the independence predicate $P_{indep}(A,B)$ is the conjunction of all predicates P^{l_1,l_2} for all locations that must be checked to be different:

$$P_{indep}(A,B) \triangleq \Big(\bigwedge_{l \in rd_B} P^{loc_A,l}\Big) \wedge \Big(\bigwedge_{l \in rd_A} P^{l,loc_B}\Big) \wedge P^{loc_A,loc_B}$$

We now define the predicate $P_{unch}(A,E)$ that ensures that the assignment A leaves the expression E unchanged. Clearly, this is the case if the location modified by the assignment A is not read by the expression E, and given the location loc_A updated by A and the set rd_E of locations read by expression E, we set

$$P_{unch}(A,E) \triangleq \bigwedge_{l_E \in rd_E} P^{loc_A,l_E}.$$

As we mentioned in Section 2.2, we compute a sound approximation of independence predicates, and our computation could be refined. For example, the independence predicate for two array updates

$$v[a] := e \qquad \text{and} \qquad v[b] := e'$$

includes the conjunct $a \neq b$. In fact, the two assignments can be independent if $a = b$ but also $e = e'$ and each assignment leaves the right-hand side of the other assignment unchanged. Since the right-hand sides of assignments are often complex expressions whereas the left-hand sides are usually simple, we chose not to implement this improvement in order to keep independence predicates small.

Other refinements depend on the concrete operations that appear in the right-hand sides. For example, FIFO channels are represented in TLA$^+$ (and PLUSCAL-2) by sequences, where message sending corresponds to appending at the end of the sequence, and message reception to removing the first element of the sequence by the Tail operation. If both actions are actually enabled, the sequence must be non-empty, and the two actions indeed commute, as illustrated in Fig. 1. Since these operations occur frequently in the algorithms we consider, we implement this optimization.

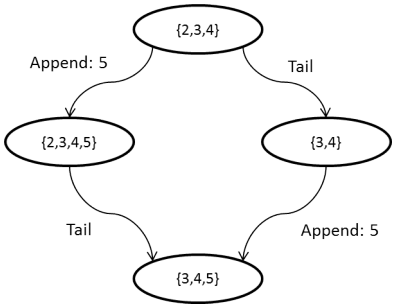


Figure 1: Append and Tail commute over non-empty sequences.

Independence predicates for sequential composition. Independence predicates for complex blocks are computed recursively. In particular, assume that block A is of the form

$$lhs := e;\ A'$$

and that we have already computed the independence predicates P_1 for the leading assignment and B and P_2 for the blocks A' and B. The overall independence predicate, $P_{indep}(A,B)$, is the conjunction $P_1 \wedge P_2'$ where P_2' is obtained by replacing the base variable of lhs by the value of that variable after assignment. Considering again the assignment of the running example, we obtain

$$\begin{array}{lll} P_2' \triangleq & \text{LET} & \_\text{net} \triangleq [\text{net EXCEPT } ![\_\text{Node}[p].\text{out}] \\ & & \qquad = \text{Append}(@, [\text{type} \mapsto \text{“one”}, \text{number} \mapsto \text{mynumber}])] \\ & \text{IN} & P_2[\_\text{net}/\text{net}]. \end{array}$$

The predicate $P_{unch}(A,E)$ is defined in an analogous way.

Independence predicates for branches. Now assume that block A is of the form

branch
  C_1 **then** A_1
or C_2 **then** A_2
end

(the generalization to a branch block with n arms will be obvious). The overall independence predicate for a branch must ensure the following conditions:

1. whenever C_1 holds (and therefore A_1 may be executed), A_1 and B are independent,
2. the symmetric condition for A_2 and B, and
3. executing B cannot disable any execution of A that would have been possible in the original source state, or enable an execution that would have been impossible.

Assume that we have already computed independence predicates $P_{indep}(A_1, B)$ and $P_{indep}(A_2, B)$ that ensure independence of A_1 and A_2 with B. These predicates will be used for ensuring conditions (1) and (2). For condition (3) to hold, we require that the conditions C_1 and C_2 are unaffected by any execution of B. We therefore obtain the overall independence predicate

$$\begin{array}{l} \wedge\ C_1 \Rightarrow P_{indep}(A_1, B) \\ \wedge\ C_2 \Rightarrow P_{indep}(A_2, B) \\ \wedge\ P_{unch}(B, C_1) \wedge P_{unch}(B, C_2) \end{array}$$

The unchanged predicate $P_{unch}(A, E)$ for the block A and any expression E is given by the conjunction

$$\begin{array}{l} \wedge\ C_1 \Rightarrow P_{unch}(A_1, E) \\ \wedge\ C_2 \Rightarrow P_{unch}(A_2, E) \end{array}$$

Independence predicates for existential quantification. To compute the independence predicate $P_{indep}(A, B)$ for an existentially quantified block

$$A \ \equiv\ \exists\, id \in expr : A_1$$

where A_1 is the sub-block that is to be executed for some value of identifier *id* in the set obtained by evaluating *expr*, we again assume that we already have computed the independence predicate $P_{indep}(A_1, B)$. The predicate $P_{indep}(A, B)$ must ensure the following conditions:

- A_1 (for any value of *id*) should be independent of block B and
- execution of B should leave unchanged the value of expression *expr*.

These two conditions suggest the definition of $P_{indep}(A, B)$ as

$$\begin{array}{l} \wedge\ P_{unch}(B, expr) \\ \wedge\ \forall id \in expr : P_{indep}(A_1, B) \end{array}$$

Similarly, the unchanged predicate $P_{unch}(A, E)$ can be defined as

$$\forall id \in expr : P_{unch}(A_1, E).$$

Generating a matrix of independence predicates. When computing the independence predicates P_{ij} for pairs of atomic blocks A_i and B_j, we perform some elementary simplification, and in particular propagation of constants TRUE and FALSE. We then define a TLA$^+$ operator that represents the matrix of independence predicates, and that will be passed to the model checker. The operator takes four parameters, which correspond to the names of the blocks and the process identifiers p and q, and is defined as

$$\begin{array}{l} IndepMatrix(p, q, A, B) \ \triangleq \\ \quad \text{CASE} \\ \qquad\quad A = name_1 \wedge B = name_1 \ \rightarrow\ P_{11} \\ \qquad \Box \quad A = name_1 \wedge B = name_2 \ \rightarrow\ P_{12} \\ \qquad\quad \vdots \\ \qquad \Box \quad A = name_n \wedge B = name_n \ \rightarrow\ P_{nn} \end{array}$$

where $name_1, \ldots, name_n$ are the names of the TLA$^+$ actions that are generated for the atomic blocks of the PLUSCAL-2 algorithm in intermediate representation. (In the actual implementation, we only give those entries of the matrix for which P_{ij} is different from FALSE, and add a catch-all clause that returns FALSE for all other inputs.)

4 Implementation of Partial-order Reduction in TLC

In order to benefit from the independence information computed for all actions, we had to implement partial-order reduction in the TLC model checker. TLC builds the state graph of a TLA$^+$ model in breadth-first order, using a FIFO queue. Once the state graph has been computed, any liveness properties are verified. TLC starts by enqueueing all the initial states, then launches several threads which repeatedly execute the following algorithm:

- Remove a state from the FIFO queue. If the state has not yet been explored, generate all its successor states and add them to the state graph.
- For each successor state, check if it satisfies all the invariant properties and add it to the end of the FIFO queue.
- If some successor does not satisfy some invariant property, report an error, abort model checking, and print the corresponding counter example.

We extend TLC by an implementation of the partial-order reduction technique first proposed by Holzmann and Peled in [HP94]. They explain the technique for a depth-first search algorithm whereas in our implementation we adapted it to breadth-first search. The other modification concerns the concept of processes, which is fundamental in [HP94]. Processes are not present in TLA$^+$, but a parameter *self* representing the process executing the statement is introduced by the PLUSCAL-2 compiler.

The pseudo-code of the partial-order reduction algorithm that we implemented is as follows:

```
1  main() {
2    curstate = Pop new state from FIFO queue
3    order enabled actions for curstate
4    for each action a in enabled actions of curstate {
5      NotInStack = true
6      AtLeastOneSuccessor = false
7      for each succstate in successors of action a {
8        if succstate in not already seen {
9          add succstate to the list of already seen states
10         add succstate to the FIFO queue
11       }
12       else {
13         NotInStack = false
14       }
15       AtLeastOneSuccessor = true
```

```
16      }
17      if AtLeastOneSuccessor and NotInStack {
18        break from loop over enabled actions
19  } } }
```

This algorithm picks a state from the FIFO queue of the TLC model checker and performs a critical step at line 3, to reorganize the list of actions. The processes are reordered on the basis of a "safety" criterion: an action is safe if it is independent of all other currently enabled actions and if it is non-observable by the formulas under verification. An action is non-observable if it does not modify any variable that appears in the formula. The algorithm in Fig. 2 explains how actions are ordered, and it is here where the independence predicates produced previously are evaluated at the current state.

```
1  orderActions(curstate) {
2    for each action a in enabled actions of curstate {
3      isIndependent = action a independent of all other enabled actions
4      isNotObservable = check observability for each successor state of a at curstate
5      if isIndependent and isNotObservable {
6        mark action a as safe
7      }
8      else {
9        mark action a as unsafe
10   } }
11   reorganize the list of actions as safe:unsafe:disabled
12 }
```

Figure 2: Algorithm for ordering actions using safety principle.

The actions are reordered in such a way that the actions which satisfy the safety properties are placed at the head of the list, then the actions which are unsafe and finally, the actions that are disabled. Enabledness of actions is determined by computing the successors of all actions at the current state; the results of this computation is reused when actually producing the successor states, although not all of them have to be stored for further verification if the reduction is successful.

5 Experimental Results

We evaluated our reduction techniques over two algorithms taken from the distribution of the Spin model checker [Hol04]. The two algorithms are the DKR leader election algorithm and a concurrent sorting algorithm. Because the verification results are similar for the two algorithms, we only discuss those for the DKR algorithm.

The Leader election algorithm that we implemented in PLUSCAL-2 for generating TLA$^+$ specifications along with the independence relation is the algorithm for electing a leader in a unidirectional ring due to Dolev, Klawe, and Rodeh [DKR82], shown in Figs. 4 and 5.

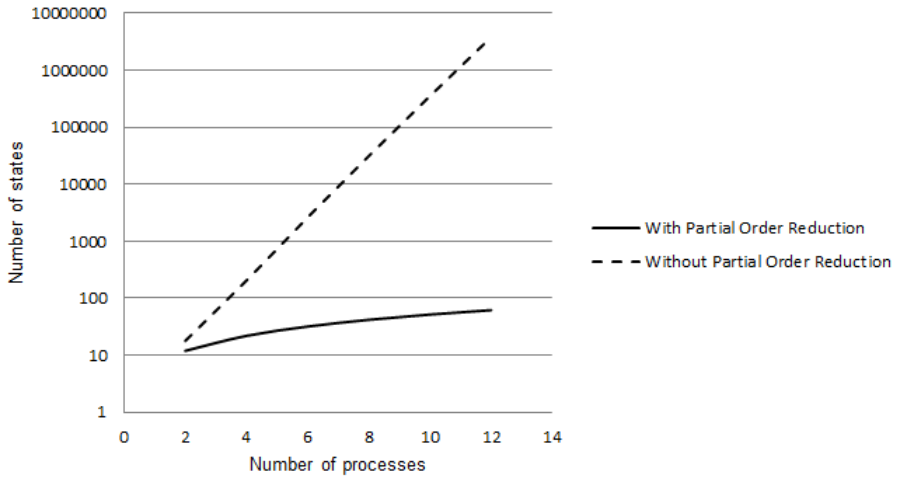


Figure 3: Results for the DKR algorithm with and without partial-order reduction in TLC.

The PLUSCAL-2 compiler generates two actions for this algorithm, which correspond to the blocks of actions labeled start (initialization) and forever (one pass through the loop). It should be noted that the second action is of much coarser granularity than that of a typical transition in Spin, which makes computing useful independence predicates more challenging. Nevertheless, our algorithm computed non-trivial predicates for these actions. Figure 3 shows the numbers of states generated for different numbers of processes with and without partial-order reduction.

As the number of processes increases, the number of states generated by TLC increases exponentially, resulting in state space explosion. This makes it impractical to verify larger instances of this algorithm. In contrast, the number of states increases only linearly when partial-order reduction is used. The running time of TLC is reduced accordingly, varying between 0.09 seconds for 4 processes to 0.17 seconds for 12 processes (on a standard laptop running Windows) compared to 0.2 and 349 seconds, respectively, without partial-order reduction. In particular, the overhead of evaluating the independence predicates during model checking is negligible compared to the gains due to state space reduction. These results clearly show that our method is effective for this algorithm.

6 Related Work

Our work on partial-order reduction for PLUSCAL-2 is closely related to that for the Spin model checker [Hol04], as our technique is based on the static partial-order reduction algorithm proposed by Holzmann and Peled and implemented in Spin. It computes the independence predicates before the actual verification. Compared to other static techniques for partial-order reduction [GW94, Val90], the technique of [HP94] tends to have lower runtime overhead; it also applies to the verification of safety and liveness properties.

Dynamic partial-order reduction is another variant of partial-order reduction techniques. Flanagan and Godefroid [FG05] proposed a dynamic partial-order reduction technique for model

checking software. They dynamically compute the redundant parts of the state graph to avoid the unnecessary exploration. Their algorithm also adapts to dynamic changes in the structure of the network, such as creation of new processes and threads, and new memory allocations. Its implementation is less complicated as it does not require static analysis of the algorithm and is particularly suited for verifying software whose source need not be available. On the other hand, it may imply higher run-time overhead because independence of actions is determined dynamically. We believe that the technique that we have implemented, which is based on conditional independence of actions, is a better compromise for higher-level models of algorithms, which we target in PLUSCAL-2.

The original technique of [FG05] is intended for stateless model checking. In [YCGK08], another dynamic partial-order reduction approach has been introduced that is compatible with stateful model checking. They propose a scheme for storing abstract local states using their identities along with the difference between the two successive local states. It becomes inexpensive as compared to capturing the entire state. Both the techniques of [FG05] and [YCGK08] are targeted towards examining programs rather than models and do not rely on static analysis of specifications.

7 Conclusions

In this paper, we have reported on our implementation of partial-order reduction for the verification of algorithms written in the PLUSCAL-2 language. Based on the approach proposed by Holzmann and Peled [HP94], which is known to support the verification of safety and liveness properties, it relies on generating predicates that ensure the independence of (groups of) statements. We have also extended the TLA$^+$ model checker TLC so that it takes advantage of this information to avoid redundant explorations of equivalent paths. We have shown that the technique performs well over two standard examples. The main difference with respect to the implementation in Spin is that we work within a language that provides a much more expressive expression language. A minor difference is our adaptation of the partial-order reduction to a breadth-first search algorithm. We believe that there are many possibilities for identifying "domain-specific" independence predicates, beyond what we have so far implemented for operations on FIFO channels. Also, the TLA$^+$ actions that we generate from PLUSCAL-2 algorithms are typically of much coarser granularity than in standard Promela models. A priori, this reduces the effectiveness of partial-order reduction, and complicates the computation of useful independence predicates. We have been pleasantly surprised by how well the method worked for the two examples we presented, but more validation is necessary over existing PLUSCAL-2 algorithms.

Acknowledgements: We wish to thank the anonymous referees for their helpful comments.

Bibliography

[AMQ10] S. Akhtar, S. Merz, M. Quinson. A High-Level Language for Modeling Algorithms and Their Properties. In Davies et al. (eds.), *13th Brazilian Symp. on Formal Meth-*

ods (SBMF 2010). Lecture Notes in Computer Science 6527, pp. 49–63. Springer, Natal, Brazil, 2010.

[CGP99] E. M. Clarke, O. Grumberg, D. Peled. *Model Checking*. MIT Press, Cambridge, Mass., 1999.

[Dij75] E. W. Dijkstra. Guarded commands, non-determinacy and formal derivation of programs. *Communications of the ACM* 18(8):453–457, 1975.

[DKR82] D. Dolev, M. M. Klawe, M. Rodeh. An O(n log n) Unidirectional Distributed Algorithm for Extrema Finding in a Circle. *J. Algorithms* 3(3):245–260, 1982.

[FG05] C. Flanagan, P. Godefroid. Dynamic partial-order reduction for model checking software. In Palsberg and Abadi (eds.), *32nd ACM Symp. Principles of Programming Languages (POPL 2005)*. Pp. 110–121. ACM, Long Beach, CA, U.S.A., 2005.

[GW94] P. Godefroid, P. Wolper. A Partial Approach to Model Checking. *Information and Computation* 110(2):305–326, 1994.

[Hol04] G. J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[HP94] G. Holzmann, D. Peled. An Improvement in Formal Verification. In *IFIP WG 6.1 Conference on Formal Description Techniques*. Pp. 197–214. Chapman & Hall, Bern, Switzerland, 1994.

[Lam02] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[Lam06] L. Lamport. Checking a Multithreaded Algorithm with +CAL. In Dolev (ed.), *20th Intl. Symp. Distributed Computing (DISC 2006)*. Lecture Notes in Computer Science 4167, pp. 151–163. Stockholm, Sweden, 2006.

[Val90] A. Valmari. A stubborn attack on state explosion. In *2nd International Workshop on Computer Aided Verification*. Lecture Notes in Computer Science 531, pp. 156–165. Springer Verlag, Rutgers, June 1990.

[YCGK08] Y. Yang, X. Chen, G. Gopalakrishnan, R. M. Kirby. Efficient Stateful Dynamic Partial Order Reduction. In *SPIN*. Pp. 288–305. 2008.

[YML99] Y. Yu, P. Manolios, L. Lamport. Model checking TLA+ Specifications. In Pierre and Kropf (eds.), *Correct Hardware Design and Verification Methods (CHARME'99)*. LNCS 1703, pp. 54–66. Springer, Bad Herrenalb, Germany, 1999.

A Example: Leader election algorithm

```
1  algorithm Leader
2  extends Naturals                    (* standard modules *)
3  constants
4     N,                               (* Number of processes *)
5     I                                (* node given the smallest number *)
6
7  variable
8        net = [p ∈ 0..(N-1) ↦ ⟨⟩],    (* the network represented as a queue *)
9        nr_leaders = 0                (* number of leaders elected *)
10
11 process Node[N]
12    variables
13       active = TRUE, know_winner = FALSE,
14       mynumber = (N+I-self)%(N+1), neighbourR = 0,
15       maximum = (N+I-self)%(N+1), in = self-1, out = self%N,
16       msg = ⟨⟩
17    begin
18    start:
19       net[out] := Append(net[out], [type ↦ "one", number ↦ mynumber]);
20    forever:
21       loop
22         if Len(net[in]) > 0 then
23           msg := Head(net[in]);
24           if msg.type = "one" then
25             if active then
26               if msg.number # maximum then
27                 net[out] := Append(net[out], [type ↦ "two", number ↦ msg.number]);
28                 neighbourR := msg.number;
29               else
30                 know_winner := TRUE;
```

Figure 4: Leader election algorithm.

```
1                net[out] := Append(net[out], [type ↦ “winner”, number ↦ msg.number]);
2              end if
3            else
4              net[out] := Append(net[out], [type ↦ “one”, number ↦ msg.number]);
5            end if;
6          else if msg.type = “two” then
7            if active then
8              if (neighbourR > msg.number) ∧ (neighbourR > maximum) then
9                maximum := neighbourR;
10               net[out] := Append(net[out], [type ↦ “one”, number ↦ neighbourR]);
11             else
12               active := FALSE;
13             end if
14           else
15             net[out] := Append(net[out], [type ↦ “two”, number ↦ msg.number]);
16           end if;
17         else if msg.type = “winner” then
18           if msg.number = mynumber then
19             nr_leaders := nr_leaders + 1;
20           end if;
21           if ˜know_winner then
22             net[out] := Append(net[out], [type ↦ “winner”, number ↦ msg.number]);
23           end if;
24         end if;
25         net[in] := Tail(net[in]);
26       end if;
27     end loop;
28   end process
29 end algorithm
30
31 (* Invariant for model checking *)
32 invariant nr_leaders <= 1
33
34 (* Finite instance for model checking *)
35 constants N = 3, I = 1
```

Figure 5: Leader election algorithm (part 2).

Structuring Functional Requirements of Control Systems to Facilitate Refinement-based Formalisation

Sanaz Yeganefard[1] **and Michael Butler**[2]

[1] sy2g08@ecs.soton.ac.uk
[2] mjb@ecs.soton.ac.uk
Electronics and Computer Science
University of Southampton, UK, SO17 1BJ

Abstract: Good requirements structure can greatly facilitate the construction of formal models of systems. This paper describes an approach to requirements structuring for control systems that aims to facilitate refinement-based formalisation. In addition to the well-known monitored and controlled phenomena used to analyse control systems, we also identify commanded phenomenon reflecting the special role that an operator plays in system control. These system phenomena guide the structure of the requirements analysis and documentation as well as the structure of the formal models.

We model systems using the Event-B formalism, making use of refinement to support layering of requirements. The structuring provided by the system phenomena and by the refinement layers supports clear traceability and validation between requirements and formal models. As a worked example, we structured the requirements of an automotive lane departure warning system using this approach. We found missing requirements through this process and we evolved the requirement document through domain experts' feedback and formal modelling.

Keywords: structuring requirement, requirement engineering, validation, formal verification, lane departure warning system

1 Introduction

Control systems are usually complex as they continually interact with and react to the evolving environment. Because of the complexity of these systems, constructing and structuring their functional requirement documents (RD) can be a time consuming process. In addition, their RD may not be clear and complete for developers of the system. However, since these systems are usually used in life critical situations it is essential to have a comprehensive RD to help with the improvement of safety and reliability of the system.

Formal methods are mathematical based techniques used for specification and development of systems as well as verifying their properties [Win90]. Modelling using formal methods is known to improve system understanding and thus help to find missing and ambiguous requirements. However, one difficulty of using formal modelling is formalising an informal RD.

In this paper we propose an approach to construct the RD of control systems incrementally to help with understanding the system requirements and to facilitate the process of for-

mal modelling. This approach consists of three stages and is based on monitored, commanded and controlled (MCC) phenomena introduced in [But09] as an extension of Parnas' 4-variable model [PM95].

In the first stage an RD is constructed and structured incrementally through iterations, as our understanding of the system improves (i.e. by considering the requirements in more depth). The second stage involves modelling the RD in a step-wise manner by using refinement. Here, requirements are layered and each layer is modelled in one refinement level. The third stage of this approach deals with any identified missing and ambiguous requirements by revising the RD and the model.

This approach also provides the means for validating a model against its RD in order to ensure that the model is an accurate representation of the system's requirements. This validation also facilitates the traceability between a model and its RD.

As a worked example, we structured the requirements of an automotive lane departure warning system (LDWS) using the proposed approach. Requirements of this system are evolved in three phases. In the first phase we produce and structure the RD of the LDWS based on information in the *public domain*. In the second phase, the generated RD is discussed with *domain experts*. In the third phase, the RD of LDWS is *formally modelled* using Event-B formal language. Also, as will be discussed any changes in requirements, i.e. identified missing and ambiguous requirements, are applied to both the RD and the Event-B model.

This paper is organised as follows: in Section 2 MCC phenomena are discussed. An overview of the proposed approach is given in Section 3. Section 4 introduces the lane departure warning system (LDWS) briefly. The RD of the LDWS and its formal model are represented in sections 5, 6 and 7. The validation of the model against some of its requirements is shown in Section 8. Section 9 and 10 discuss the related and future work. In Section 11 some of the advantages of the proposed approach are represented.

2 Guidelines for Modelling Control Systems

The guidelines outlined in [But09] can be used for formal modelling of control systems. The formal models consist of variables and guarded actions (events) and control systems consist of plants, controllers and in some cases operators who can send commands to the controller, shown in Figure 1[1].

The modelling steps suggested in this guidelines are based on the four-variable model of Parnas [PM95]. Variables shared between a plant and a controller, labelled as 'A' in Figure 1, are known as environment variables and are categorised into *monitored variables* whose values are determined by the plant and *controlled variables* whose values are set by the controller. There are also *environment events* and *control events* which update/modify monitored and controlled variables respectively. The other two variable categories of the four-variable model are *input* and *output*. In [But09] it is suggested that these are not used in abstract formal model; instead [But09] provides patterns for introducing them as refinements.

If a system involves operators, according to [But09] in addition to the phenomena introduced in the four-variable model, phenomena shared between controller and the operator can be iden-

[1] The diagram uses Jackson's Problem Frame notation [Jac01].

tified. These phenomena, labelled as 'B' in Figure 1, are represented by *command events* which are the commands given by an operator and *commanded variables* whose values are determined by command events and can affect the way other events behave.

In [YBR10] a cruise control system is modelled following MCC guidelines. In addition [YBR10] shows that modelling based on the MCC guideline helps to have a more structured process of modelling and refinement for a control system.

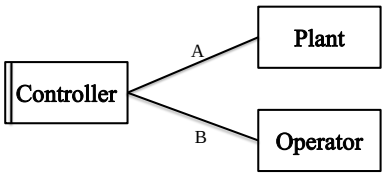


Figure 1: A control system.

3 Overview of the Proposed Approach

Modelling guidelines represented in [But09] requires the modeller to identify the MCC phenomena of a system before the commencement of the modelling process. This inspired us to propose an approach for structuring RD and modelling using MCC phenomena. This approach comprises of three stages, which are shown in Figure 2.

Notice that we use the term *phenomena* when we deal with (informal) requirements of a system and the term *variable* when we model the system formally. Also as our focus in this paper is not on requirement elicitation methods, it is assumed that the textual RD of the control system exists.

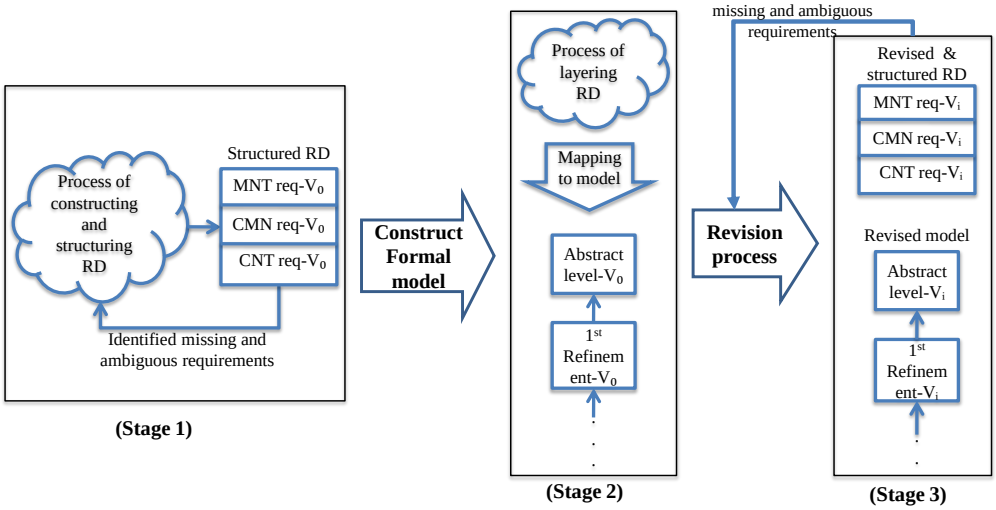


Figure 2: Overview of the three stages. Stage 1 - Structuring RD; Stage 2 - Layering structured RD and designing the initial formal model; Stage 3 - Revising the RD and the model.

3.1 Stage 1: Structuring Requirement Document

The following are the steps suggested for structuring the existing textual RD into monitored, commanded and controlled sections:

1. *Identify the system's MCC phenomena* based on its textual RD. An example is the monitored phenomenon *speed* in LDWS.
2. *Organise RD* into three monitored (MNR), commanded (CMN) and controlled (CNT) sections, each representing requirements of the corresponding phenomenon. If the requirement refers to
   - only one phenomenon, it will be moved to the relevant MCC section.
   - more than one phenomenon, but of the same type (e.g. they all are monitored phenomena), the requirement will be added to the corresponding section.
   - more than one phenomenon, but of different type, then it is designer's judgment which section is mostly appropriate.
3. *Add unique ID labels.* Every ID starts with the section that the requirement belongs to (i.e. MNR, CMN, CNT), followed by a unique number for that requirement.
4. *Revise RD* to accommodate any identified missing or ambiguous behaviour of the system. The revision step involves going back to Step 1 to identify any phenomena that the new requirement represents and then adding the requirement to the appropriate section.

The last step helps one to seek the most obvious MCC phenomena in the initial structuring of the RD and improve it incrementally through iterations. This iteration is shown in Figure 2-Stage 1. Section 5 represents structuring the RD of an LDWS using these steps.

3.2 Stage 2: Layering Requirements and Designing the Initial Models

In order to deal with the complexity of a control system, our aim is to use refinement to introduce system requirements in a step-wise manner. However, deciding on how to layer requirements and what to model in each levels is usually difficult.

We propose to overcome this problem by modelling one **feature** and the minimum number of requirements essential for this feature to be meaningful in one level of refinement. A feature is usually one of the MCC phenomenon of the system. However, sometimes a phenomenon is interrelated to other phenomena and thus they should be modelled simultaneously. Examples of features for the LDWS are phenomena *warning* and *status.*

We also suggest to focus on the **main role or behaviour** of the system, which usually corresponds to a controlled phenomenon, in the most abstract level. If the system has more than one role, it is the modeller's judgment to choose the most important role to be initially modelled. This means the abstract model will focus on the role of the control system, while the rest of the requirements will be elaborated into the model through refinement levels. For instance, the main behaviour of an LDWS is to issue *warnings* and this is modelled in the abstract level. After that in the first refinement the phenomenon *status* is introduced. Section 6 describes this stage in more details through the LDWS example.

3.3 Stage 3: Revision of RD and Formal Model

Modelling a system formally can result in finding missing and ambiguous requirements of the system. We suggest to handle these requirements similarly to the revision step in Stage 1, where phenomena of new requirements are identified and based on them requirements are added to the corresponding MCC sections of the structured RD. However, in addition to revising the RD, it is necessary to update the formal model. This is because the RD and the model should be kept consistent to help with the process of validation and traceability.

If a new requirement is related to any of the previously modelled phenomena (modelled in Stage 2), this requirement can be modelled in the same refinement level as its phenomenon. For instance if the new requirement gives further information about the main behaviour of the system, which is modelled in abstract level, we update this level. However, if the newly identified requirement has no effects on any levels of the current formal model, for instance if the requirement introduces a new phenomenon, it can be introduced to the model in a new refinement level.

As shown in Figure 2, Stage 3 can be iterated meaning that as long as new missing or ambiguous requirements are identified, the RD and the model of the system should be revised. This stage is explained further in Section 7 using the example of LDWS.

4 An Overview of LDWS

LDWS is a driver assistance system which receives camera observations of the lane and uses this information to warn the driver of a lane departure, when the car is travelling above a certain speed. One way to detect the car departing the lane is by estimating the car's current position in the lane using lane detection algorithm on camera's observation [RME00].

In order to warn the driver before the vehicle crosses the lane, a virtual lane width which is inside the lane boundaries is assumed. This virtual lane, called the earliest warning lines (EWL), is determined by the LDWS based on the speed of the car. When the vehicle is within the earliest warning lines, the LDWS does not issue any warnings. This area is called the "no-warning zone", and the area pass EWL is the "warning zone" [Fed05], shown in Figure 3.

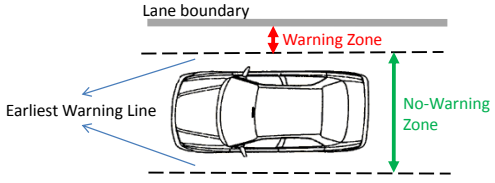


Figure 3: Warning and no-warning zone for an LDWS.

5 Stage 1: Constructing and Structuring RD of LDWS

In this section we discuss some of the requirements of the LDWS which are structured according to the first stage of the proposed approach, Section 3.1. The RD which is produced based on the information available in the public domain [Fed05, RME00, PMGB05] is outlined in Section 5.1.

This RD then is modified in Section 5.2 according to the feedback received from the domain experts.

5.1 First Version of Requirement Document

To structure the RD we first examined the public sources to identify the MCC phenomena. The identified **monitored phenomena** are *position* of the car relative to the centre of the lane, *lane width* and current car *speed* which also determines the EWL. The identified **commanded** and **controlled phenomena** are respectively *status* of the LDWS which is set through a switch button and *warning*. Requirements related to these phenomena are organised into MNT, CMN and CNT sections using appropriate identifiers for every requirement as shown in Table 1.

Table 1: First version of the structured RD of the LDWS.

Req ID	Requirement Description
MNR1	LDWS should detect the earliest warning line (EWL) and vehicle position relative to visible lane boundaries based on the lane width and car width.
MNR2	LDWS should track lane boundaries where lane markings are clearly visible in daylight (sunny/cloudy), night times and twilight (sunrise/sunset) lighting conditions.
MNR3	The width of the "warning zone" depends on the speed of the car. The higher the speed of the car the closer the earliest warning line (EWL) to the centre of the lane.
CMN1	LDWS can be switched on and off by the driver through a single button.
CNT1	LDWS should issue a warning when the vehicle has left the no warning zone (has crossed the EWL), and is entering the warning zone.
CNT2	When LDWS is on it starts its role provided that the car speed is greater or equal to a certain speed.

5.2 Second Version of Requirement Document

Discussing the first version of the LDWS's RD with domain experts from GM India Science Lab resulted in identifying some missing requirements. One of these requirements was that the driver can change the EWL by setting the offset they wish to have from the lane boundaries.

We retook the structuring steps and as the result the commanded phenomenon *offset* was identified. Thus, the three requirements related to offset, shown in Table 2, are added to CMN section. Notice that we refer to requirement MNR3 in CMN4 to show that these requirements are related.

6 Stage 2: Layering Requirements and First Model of LDWS

The initial formal model of LDWS was produced based on the second RD. We have used Event-B formal language which also provides supports for refinement. The process of modelling starts with identifying requirements necessary for constructing the abstract level. After building the abstract model, the remainder of the RD is introduced in refinement levels.

Table 2: Second RD - Requirements are added based on experts' feedback.

Req ID	Requirement Description
CMN2	The driver can set the offset they want to have from either side of the lane boundaries through two buttons which are responsible for increasing and decreasing the distance.
CMN3	The offset is always within a certain positive range.
CMN4	In addition to speed (MNR3) the width of the "warning zone" or earliest warning line depends on the offset from the lane boundaries. The greater the determined offset the closer the EWL to the centre of the lane.

6.1 Event-B and its Tool

For the purpose of this paper we only focus on two elements of an Event-B [Abr10] model. Firstly, *variables* and secondly *events*. An event consists of two elements, *guards* which are predicates defined for describing the conditions need to hold for event occurrence, and *actions* which determine the changes of state variables. An event becomes *enabled* if its guards hold. One of the advantages of Event-B is its open source tool, known as Rodin, which provides automatic proof and a wide range of plug-ins [ABH+10].

6.2 Requirements for Modelling the Abstract Level

As mentioned in Section 3.2, Stage 2 involves layering and modelling requirements based on the features of the system. In this stage, each feature is modelled in one level of refinement. Also, the main behaviour of the system (i.e. the main controlled phenomenon) is modelled in the most abstract level. Examining the structured RD of the LDWS shows that the requirement **CNT1** represents the main role of this system; "issuing warnings when the car has left the no warning zone". Thus, the controlled phenomenon *warning* is to be modelled at the abstract level.

After this we identify any interrelated phenomena, usually monitored and commanded requirements (MNR, CMN), which are vital for modelling *warning*. Thus, requirements related to crossing EWL should also be modelled at this level. These are firstly, the requirement **MNR1**, since the controller should receive the monitored phenomena *car position* and *lane width* in order to decide whether or not the EWL is crossed. Secondly, **MNR3** and **CMN4**, since the position of the EWL depends on the monitored phenomenon *speed* and the commanded phenomenon *offset*.

The next step is to identify requirements which describe the limitations and restrictions of the identified phenomena, or requirements which show how these phenomena affects/restricts the system. Based on this we need to model firstly **CNT2**, which describes the restriction imposed on *warning* by the system. The second is **CMN3**, as it gives details about restrictions of *offset*.

Finally we identify any requirements which represent state changes of the identified phenomena. This results in identifying **CMN2**, since it represents how *offset* can be modified. Notice that in Event-B requirements which are identified as restrictions of phenomena are usually modelled as guards for events or as types of the phenomena. Also, requirements which define state changes are usually modelled as actions of events.

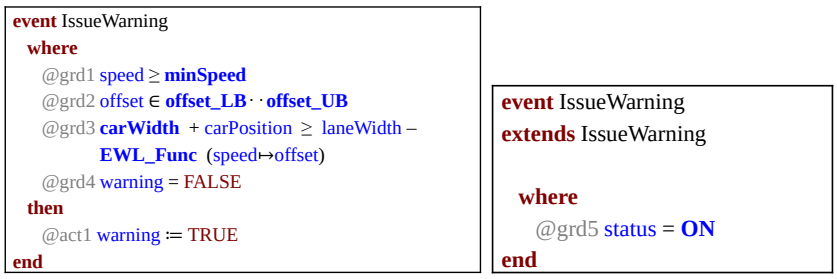
6.3 Modelling Abstract Level

In the previous section, requirements and phenomena for modelling the abstract level were identified. In this section we start modelling the abstract level by representing each of these phenomena as a system variable. Also according to [But09], the corresponding *environment*, *command* and *control* events for every variable is defined.

At this level, the monitored variables are *speed*, *laneWidth*, and *carPosition* (the difference between the car centre and the lane centre). The LDWS also needs to know the width of the car, modelled as the constant *carWidth*, in order to detect the value of *carPosition*. Environment events which are responsible for modifying monitored variables are defined to simply set the monitored variables non-deterministically through a parameter (MNR1 and MNR3).

The commanded variable modelled at this level is *offset* (CMN2). Since the value of *offset* is within a specific range (CMN3), two constants called *offset_LB* and *offset_UB* are defined to represent the lower and upper bounds of this variable. The command events which modify the value of *offset* are *IncreaseOffset* and *DecreaseOffset*. Also, the controlled variable *warning* is defined as a Boolean variable. The control event *IssueWarning*, shown in Figure 4a, is defined to set *warning* to TRUE when the car crosses the EWL (CNT1).

To model crossing EWL, we firstly define a function named *EWL_Func* which returns the distance of the EWL from the lane boundaries based on the car *speed* and the *offset* (MNR3 and CMN4). This function is defined as $EWL\_Func \in \mathbb{N} \times offset\_LB..offset\_UB \rightarrow \mathbb{N}$, meaning that for every possible tuple of *speed* (of type $\mathbb{N}$) and *offset* (within the range $offset\_LB..offset\_UB$) there is a value for the *EWL_Func*. We assume that the return value of this function, which represents the position of EWL, are provided. Based on this function *grd3* in Figure 4a is defined to model that event *IssueWarning* will be enabled when the car passes the EWL. In addition, CNT2 is modelled by defining a constant *minSpeed* and adding *grd1* in Figure 4a.



(a) Abstract level (b) Refinement level

Figure 4: Control event *IssueWarning*.

6.4 First Refinement

In this level of refinement we focus on the feature *status* which is a commanded phenomenon (CMN1). The process of identifying requirements corresponding to this phenomenon is similar to the process mentioned in Section 6.2 for the controlled phenomenon *warning*.

Since no other phenomenon is interrelated to *status*, at this level of refinement only *status*

is introduced. Also, requirement CNT2 is added to the model at this level, as it represents a restriction imposed by the phenomenon *status* on the controller.

This phenomenon is modelled by defining the variable *status* and command events *SwitchOn* and *SwitchOff* which set *status* variable from OFF to ON and vice versa (CMN1). Also, CNT2 is modelled by adding *grd5* to the control event *IssueWarning* as shown in Figure 4b.

7 Stage 3: Revision of RD and Model of LDWS

Modelling requirements formally helped us to find some of the missing and ambiguous requirements. So, based on Section 3.3, in Stage 3 we revise the RD and the model of the LDWS.

7.1 Missing Requirements and Revision of the RD

Two of the identified missing requirements are discussed in this section. The first is that we realised requirements related to the situations where a car *has crossed* the actual lane boundary and is travelling on the boundary should be differentiated from when the car has crossed the EWL and therefore is *about to cross* the boundary. As shown in Figure 5, this is mainly because of limitations of the camera's field of view which can result in detection of only one boundary.

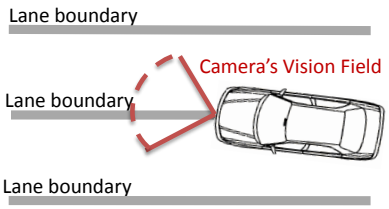


Figure 5: Limited field of vision for a camera when car is travelling on lane boundary.

We retake the structuring steps of Stage 1, Section 3.1, to add relevant requirements to the structured RD. Firstly, the LDWS should detect that the car has crossed the boundary. Thus, the requirement MNR4 is added to the RD. Secondly, the LDWS should issue warnings if crossing boundary is detected. This resulted in producing requirement CNT3, Table 3.

The other missing requirements are about the situations under which the LDWS should stop the warning process. Examining the LDWS showed that the process of issuing warnings should finish when the driver steers away from the lane boundaries and thus from the EWL. Here, the car is back within the lane and any issued warning should be stopped. Taking the structuring steps resulted in adding requirement CNT4 to the RD. Another situation where warning should stop is if the LDWS is switched off by the driver. This requirement is shown as CMN5 in Table 3.

7.2 Adjusting Abstract Model

To identify which of the newly found requirements of Table 3 need to be added to the abstract model, we retake the requirement selection process of Stage 2. As the phenomenon *warning* was modelled in the abstract level, we need to consider whether any of the new requirements are

Table 3: Third RD - Requirements are added based on Formal Modelling.

Req ID	Requirement Description
MNR4	LDWS should detect when the car has crossed the lane boundaries and is travelling on the boundary.
CMN5	If LDWS is issuing the warning and the driver switches the system off, the warning signal should be stopped.
CNT3	In addition to (CNT1), LDWS issues warning when it detects that the car has crossed the lane boundaries and is travelling on the boundary (MNR4).
CNT4	LDWS should stop the warning signal when the system is performing (CNT2) and the warning signal has been issued but the driver steers away from the boundaries and therefore the car remains within the no warning zone.

related to this phenomenon. The examination of the new RD shows that requirements **CNT3** and **CNT4** need to be introduced at this level. In addition, **MNR4** should be modelled here, because the monitored phenomenon *crossing lane* is required for modelling CNT3.

Therefore, the abstract model is modified by defining *crossingLane* as a Boolean monitored variable which is TRUE if the car has crossed the lane boundary. CNT3 shows that there are two cases where the LDWS should decide on issuing warnings. Firstly when the car is *about* to cross the lane boundaries because it has crossed the EWL (CNT1). Secondly, when the car *has* crossed the lane boundary (CNT3). These are modelled in the two control events, *IssueWarning_CloseToBound* and *IssueWarning_CrossingLane* respectively, Figure 6. The requirement CNT4 is modelled by introducing the new control event *FinishWarning*. At this level of abstraction this event has a guard as $warning = TRUE$ and the action $warning := FALSE$.

```
event
IssueWarning_CrossingLane
  where
    @grd1 speed ≥ minSpeed
    @grd2 warning = FALSE
    @grd3 crossingLane = TRUE
    @grd4  status = ON
  then
    @act1warning:=TRUE
end
```

```
event   IssueWarning_CloseToBound
  where
    @grd1  speed ≥ minSpeed
    @grd2  offset ∈ offset_LB · · offset_UB
    @grd3  carWidth + carPosition ≥
         laneWidth – EWL_Func (speed↦offset)
    @grd4  warning = FALSE
    @grd5  crossingLane = FALSE
    @grd6  status = ON
  then
    @act1warning:=TRUE
end
```

Figure 6: Control events *IssueWarning_CloseToBound* and *IssueWarning_CrossingLane*

7.3 Adjusting First level of refinement

As mentioned, requirements of the commanded phenomenon *status* were introduced in the first refinement level. Examining the new requirements of Table 3 show that CMN5 needs to be modelled at this level. This is done by defining two *SwitchOff* events. The event *SwitchOff1* sets

variables *status* to OFF and *warning* to FALSE, if it was previously TRUE, while *SwitchOff2* sets *status* to OFF and has the guard *warning* = *FALSE*.

8 Validation of the Model

Validation of the model against its RD can be done by adding a *validation column* to the right hand side of the structured requirement tables. Every requirement is then validated by adding the elements of the model, such as an event or a variable, which represent that requirement to its validation column. Also as quick reference, we refer to the level in which the requirement was modelled just after the ID of the requirement. Table 4 shows validation of some of the LDWS requirements against the model. For instance, requirement CNT1 is modelled in the abstract level using the following elements:

1. the controlled variable *warning*;
2. the control event *IssueWarning_CloseToBound*;
3. and showing that the car has entered the warning zone through the guard $carWidth + carPosition \geq laneWidth - EWL\_Func(speed \mapsto offset)$ in the control event *IssueWarning_CloseToBound*.

It is important to mention that the process of validation should take place at the end of every modelling step. This means as well as modelling RD, validation should be done incrementally. Thus, if a requirement is modified, the model and the validation column both should be updated to keep them consistent with the RD. Also, not always the entire RD is modelled, which means the validation column may only contain the reason for not modelling the requirement rather than the elements of the model. This is the case for the requirement MNR2, Table 4.

9 Related Work

In this section we look at some related works and compare them to our proposed approach.

9.1 Concretization and Formalization of Requirements

[FHP+05] has provided some guidelines for concretization and formalization of requirements of embedded systems. In formalization part, four steps have been suggested. *Identification* to produce an RD; *Normalization* to construct a glossary of terms that requirements use; *Structuring* to organise RD based on their "contents in a taxonomy", such as car speed; *Formalization* to formalise the structure, behaviour, interaction and data of the system.

In [FHP+05] grouping requirements is based on "different aspects", while we represent guidelines based on MCC phenomena for an engineer. In addition, our proposed approach allows one to revise the structured RD and construct a formal model incrementally. This means identified missing/ambiguous requirements can be addressed in the forthcoming iterations, while [FHP+05] does not tackle this issue. Also, we proposed a way for layering requirements which facilitates the use of refinement-based modelling, while this is not specified in [FHP+05].

Table 4: Validation of the requirements against the model.

Req ID	Requirement Description	Validation Column
MNR1 ABST	LDWS should detect EWL and vehicle position relative to visible lane boundaries based on the lane width and car width.	Monitored variable: *carPosition* & *laneWidth*; Constant: *carWidth*; Environment event: *UpdateCarPosition* & *UpdateLaneWidth*.
MNR2	LDWS should track lane boundaries where lane markings are clearly visible in daylight (sunny/cloudy), night times and twilight (sunrise/sunset) lighting conditions.	Not Considered, since we are not concerned with requirements related to the performance of the camera and image processing unit.
CMN1 Refine	LDWS can be switched on and off by the driver through a single button.	Set: STATUS= {ON,OFF}; Commended variable: *status*; Command event: *SwitchOn* & *SwitchOff*.
CMN2 ABST	The driver can set the offset they want to have from either side of the lane boundaries through two buttons which are responsible for increasing and decreasing the distance.	Commanded variable: *offset*; Command event: *DecreaseOffset* & *IncreaseOffset*.
CNT1 ABST	LDWS should issue a warning when the vehicle has left the no warning zone (has crossed the EWL), and is entering the warning zone.	Controlled variable: *warning*; Control event: *IssueWarning_CloseToBound*; Guard: $carWidth + carPosition \geq laneWidth - EWL\_Func(speed \mapsto offset)$.

9.2 HJJ

In the HJJ approach [HJJ03] the specification of control systems is initially based on the system view rather than the software view. In this respect our approach has a similarity to HJJ. In the HJJ approach the focus is to model the environment and requirements while also the properties that the control system relies on are captured as "rely condition".

In HJJ all requirements of a system are dealt with in one step of specification, while our approach uses refinement. Also, we differentiate between monitored, commanded and controlled phenomena which assists with structuring the RD and mapping it to a formal model.

9.3 Requirement Tracing based on WRSPM

[JHLR10] introduces an approach for tracing requirements to an Event-B formal model. This approach is based on WRSPM [GGJZ00] (World, Requirement, Specification, Program and Machine) which distinguishes between phenomena, system's state space, and artifacts which represent constraints. The method of [JHLR10] for traceability involves taking a requirement and identifying phenomena and artifacts of the environment and system for that requirement. The

identified phenomena and artifacts are then modelled and traceability information is provided.

Both our approach and [JHLR10] are concerned with formal modelling of informal requirements and providing traceability. While [JHLR10] is more focused on traceability between an RD and the formal model, the approach represented in this paper is more on structuring and formal modelling. Also, we provide some guiding steps for layering requirements for a refinement-based modelling, while this is not specified in [JHLR10].

9.4 SCR

Our approach shares features with the SCR (Software Cost Reduction) method [HJL96]. SCR is a formal method for specification of control systems with a tabular notation.

Like our approach, SCR is based on the four-variable model. In addition to these four variables, the SCR uses mode classes (the system states), conditions (predicates of system states), and events (represent changes in system variables and mode). The SCR method does not have commanded phenomena though these can be represented as monitored variables.

Our experience is that distinguishing monitored and commanded phenomenon facilitates requirements elicitation as they serve distinct roles. SCR is more a specification method, while in this paper we focus on structuring the RD as well as specification and traceability.

In SCR an engineer is required to identify the monitored, controlled, input and output variables of the system. However, we have a system-level view on the behaviour of the controller. Therefore, we focus on monitored, controlled and commanded variables in the more abstract models and introduce input and output variables in refinement levels.

10 Future Work and Limitations

This approach can be improved and developed further by experimenting its application in other case studies. Also, the current approach focuses on the functional RD and we are yet to deal with some of the challenges presented by non-functional requirements. In addition to improving the approach, part of our future work involves developing a more complete formal model of the LDWS. Examples of requirements which can be considered in the future work are timing and fault tolerance requirements. Our other future work involves evolution of the passive LDWS to an active lane centring system and examining the evolution of the requirement document in this case.

11 Conclusion

In this paper we discussed the evolution of the requirement document of the LDWS through domain expert's feedback and modelling using Event-B. The MCC modelling guidelines [But09] inspired us to structure the requirement document based on monitored, controlled and commanded phenomena. Also in this paper, some criteria for layering the requirements and mapping informal requirements to a formal model were provided. We followed the approach proposed in the paper to structure and model the RD of an LDWS. Some of the advantages provided by following the proposed approach are:

- Improving the requirement document by gathering and structuring the requirements incrementally in iterations.
- Facilitating the process of validation of the model against the requirement document and therefore helping with traceability between the model and requirements.
- Traceability enables us to maintain the requirement document and the model consistent in an easier and more manageable style.

The process of formal modelling of the LDWS also helped to identify missing requirements. We structured the newly identified requirements and revised the RD to accommodate them. These changes were also applied to the model. This shows that in order to achieve a more accurate requirement document and formal model the process of structuring requirements and modelling needs to be iterated.

We believe that the proposed approach can facilitate formal modelling of control systems and it can be used for modelling a structured RD using any refinement-based formal language. Furthermore, it is possible to use the MCC approach for organising requirement documents without modelling them formally.

Acknowledgment

This work is supported by GM India Science Lab and partly by the EU research project ICT 214158 DEPLOY (Industrial deployment of system engineering methods providing high dependability and productivity) www.deploy-project.eu.

We would like to thank Manoranjan Satpathy and S. Ramesh (GM India Science Lab) and also Abdolbaghi Rezazadeh and Reza Sarshogh for their helpful advice.

Bibliography

[ABH+10] J.-R. Abrial, M. Butler, S. Hallerstede, T. Hoang, F. Mehta, L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer (STTT)* 12:447–466, 2010.

[Abr10] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[But09] M. Butler. Chapter 8: Modelling Guidelines for Discrete Control Systems, Deploy Deliverable D15, D6.1 Advances in Methods Public Document. http://www.deploy-project.eu/pdf/D15-D6.1-Advances-in-Methodological-WPs..pdf, 2009. cited 2011 Jan.

[Fed05] Federal Motor Carrier Safety Administration. Concept of Operations and Voluntary Operational Requirements for Lane Departure Warning Systems (LDWS) On-board Commercial Motor Vehicles. http://www.fmcsa.dot.gov/facts-research/research-technology/report/lane-departure-warning-systems.htm, 2005. cited 2011 Jan.

[FHP+05] A. Fleischmann, J. Hartmann, C. Pfaller, M. Rappl, S. Rittmann, D. Wild. Concretization and Formalization of Requirements for Automotive Embedded Software Systems Development. In *The Tenth Australian Workshop on Requirements Engineering (AWRE)*. Pp. 60–65. 2005.

[GGJZ00] C. Gunter, E. Gunter, M. Jackson, P. Zave. A reference model for requirements and specifications. *Software, IEEE* 17(3):37 –43, 2000.

[HJJ03] I. Hayes, M. Jackson, C. Jones. Determining the specification of a control system from that of its environment. In *FME 2003: Formal Methods*. Lecture Notes in Computer Science 2805, pp. 154–169. Springer Verlag, 2003.

[HJL96] C. L. Heitmeyer, R. D. Jeffords, B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Trans. Softw. Eng. Methodol.* 5:231–261, July 1996.

[Jac01] M. Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[JHLR10] M. Jastram, S. Hallerstede, M. Leuschel, A. Russo. An Approach of Requirements Tracing in Formal Refinement. In *Verified Software: Theories, Tools, Experiments*. Lecture Notes in Computer Science 6217, pp. 97–111. Springer Berlin / Heidelberg, 2010.

[PM95] D. L. Parnas, J. Madey. Functional Documents for Computer Systems. *Sci. Comput. Program.* 25(1):41–61, 1995.

[PMGB05] A. Polychronopoulos, N. Mhler, S. Ghosh, A. Beutner. System Design of a Situation Adaptive Lane Keeping Support System, the SAFELANE System. In *Advanced Microsystems for Automotive Applications 2005*. Pp. 169–183. Springer Berlin Heidelberg, 2005.

[RME00] R. Risack, N. Mohler, W. Enkelmann. A video-based lane keeping assistant. In *Intelligent Vehicles Symposium, 2000. IV 2000. Proceedings of the IEEE*. Pp. 356 –361. 2000.

[Win90] J. M. Wing. A Specifier's Introduction to Formal Methods. *Computer* 23(9):8–24, 1990.

[YBR10] S. Yeganefard, M. Butler, A. Rezazadeh. Evaluation of a Guideline by Formal Modelling of Cruise Control System in Event-B. In *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215*. Pp. 182–191. April 2010.

A Symbolic Model Checking Approach to Verifying Satellite Onboard Software

Xiang Gan, Jori Dubrovin and Keijo Heljanko

Aalto University
School of Science
Department of Information and Computer Science
P.O. Box 15400, FI-00076 Aalto, Finland
xiang.gan, jori.dubrovin, keijo.heljanko@aalto.fi

Abstract: This paper discusses the use of symbolic model checking technology to verify the design of an embedded satellite software control system called attitude and orbit control system (AOCS). This system is mission-critical because it is responsible for maintaining the attitude of the satellite and for performing fault detection, isolation, and recovery decisions of the satellite. An executable AOCS implementation by Space Systems Finland has been provided to us in Ada source code form. In order to use symbolic model checking methods, the Ada implementation of the system was modeled at a quite detailed implementation level using the input language of the symbolic model checker NuSMV 2. We describe the modeling techniques and abstractions used to alleviate the state explosion problem due to handling of timers and the large number of system components controlled by AOCS. The specification of the required system behavior was also provided to us in a form of extended state machine diagrams with prioritized transitions. These diagrams have been translated to a set of temporal logic properties, allowing the piecewise checking of the system behavior one extended state machine transition at a time. We also report on the scalability of symbolic model checking tools for the case study at hand as well as discuss potential topics for future work.

Keywords: symbolic model checking, AOCS, NuSMV 2, verification, satellite software

1 Introduction

Model checking [CGP99, BK08] is a technology where a formal model of a system's behavior is checked against its formal requirements often expressed in temporal logic. One of the main approaches in model checking is symbolic model checking using binary decision diagrams (BDDs) [BCM+92] that is especially suitable for hardware designs. Symbolic model checking is also suitable for analyzing other systems with a high branching degree due to environment non-determinism. Bounded model checking (BMC) [BCCZ99] was invented to scale symbolic model checking to analyzing even larger systems, especially for finding bugs in hardware designs with a high number of state bits. The basic idea in bounded model checking is to look for counterexamples to the specified property that are shorter than a user specified maximum

length called the bound. With this length restriction the search for counterexamples of bounded length can be reduced into propositional satisfiability (SAT), and the search can be performed by efficient SAT solvers, see e.g., [ES03].

The linear temporal logic (LTL) (see e.g., [BK08]) is a widely used temporal logic in model checking. Bounded model checking for LTL was shown to be linearly encodable to propositional satisfiability (SAT) in [HJL05, BHJ+06]. The same papers also describe a complete BMC algorithm that is guaranteed to terminate either with a counterexample or by proving the property holds but quite often requiring very high bounds in the case the property holds. Both the incomplete and complete variants of this approach have been implemented in the NuSMV 2 model checker [CCG+02, HJL05, BHJ+06], and are used in the experiments of this paper.

This paper discusses the use of symbolic model checking technology to verify the design of an embedded satellite software control system called attitude and orbit control system (AOCS). Our approach is based on modeling an implementation given in Ada source code in the input language of the NuSMV 2 model checker [CCG+02].

The AOCS system is mission-critical because it is responsible for maintaining the attitude of the satellite, and for performing fault detection, isolation, and recovery decisions of the satellite. An executable AOCS implementation by Space Systems Finland has been provided to us in Ada source code form. In order to use symbolic model checking methods, the Ada implementation of the system was modeled at a quite detailed implementation level using the input language of the symbolic model checker NuSMV 2. We describe the modeling techniques and abstractions used to alleviate the state space explosion problem due to handling of timers and the large number of system components controlled by AOCS.

The specification of the required system behavior was also provided to us in a form of mode transition diagrams, which basically are extended state machines with prioritized transitions. These diagrams have been translated to a set of linear temporal logic (LTL) properties, allowing the piecewise checking of the system behavior one extended state machine transition at a time. We also report on the scalability of symbolic model checking tools for the case study at hand as well as discuss potential topics for future work.

We have also done earlier work on using model checking methods for verifying safety-critical systems in the nuclear safety area [BFV+09a, BFV+09b, VPB+08]. Also in that context the modeled systems are quite similar to the embedded mission-critical software considered here: the systems have a relatively high number of timers, as well as having to cope with a highly non-deterministic environment that includes a number of faulty components that have to be recovered from during the runtime of the system. In that domain, we have been using both NuSMV 2 as well as the Uppaal model checker [BDL+06]. Our experience is that NuSMV 2 is usually better performing for systems with a high branching degree (such as the AOCS system considered here), while Uppaal is much better performing for systems having a complex timing behavior. The main difference between the safety-critical and the mission-critical environments is that in the mission-critical systems, bug hunting methods (e.g., incomplete BMC based methods) can typically be seen as sufficient, while for safety-critical systems, the main focus is on complete system verification (complete model checking, e.g., BDD-based LTL model checking).

The same AOCS system has been used as a case study in the DEPLOY project, and modeling the AOCS system using refinement methodology in Event-B can be found in [ITL+10a, ITL+10b]. The concrete Event-B models described in these works can be found in [ILT10]. Our

model is very detailed and directly based on the Ada code implementation. Our motivation has been analyzing the correctness of the Ada implementation against its specifications, not to use refinement methodology to derive correct implementations. Thus our approach does not require significant changes in the software engineering methodology.

2 Attitude and Orbit Control System

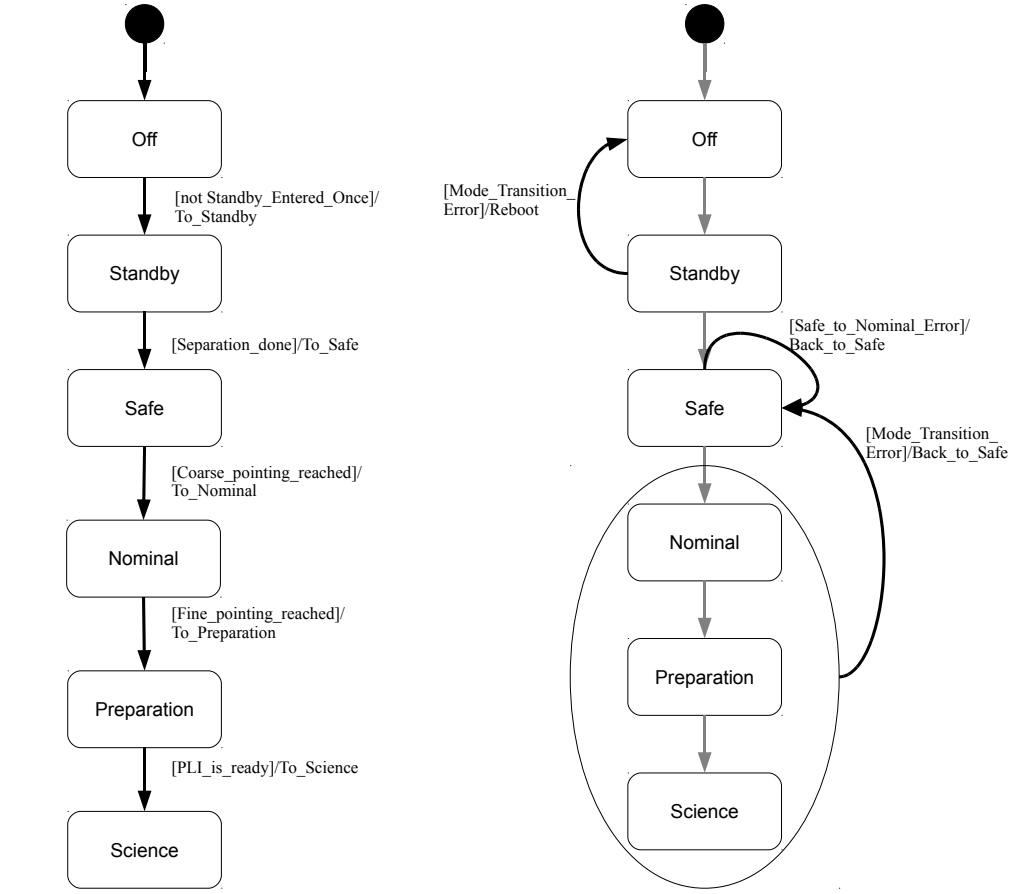
The Attitude and Orbit Control System (AOCS) [Var10] is a generic component of satellite onboard software. It is mainly used to determine and control the attitude of the spacecraft while it is in orbit. Since there is disturbance from environment, if left uncontrolled, the spacecraft will change its orientation. Because of this, its attitude needs to be monitored and adjusted continuously. The information from various sensors provides the necessary input for the AOCS computation. Based on this, the actuators are used to preserve or change the attitude or orbit of the spacecraft.

In AOCS, different software functionalities are realized by corresponding managers. There are four managers, AOCS manager, FDIR manager, Mode manager and Unit manager, which are executed in sequence to fulfill various functionalities. The AOCS manager has as its main responsibility to compute the attitude control algorithm. The FDIR manager (Fault Detection, Isolation and Recovery) is executed every time when new monitor data becomes available. There are three types of possible errors that are handled by the system: mode transition errors, attitude errors and unit errors. The Mode manager is in charge of mode transitions. In this AOCS implementation, there are six operational modes [Var10] listed below.

- **Off**. Normally, the spacecraft is in this mode after the system is booted.
- **Standby**. The system stays in this mode until separation from the launcher is completed.
- **Safe**. When the system is in this mode, it indicates that a stable attitude is obtained and the system endeavors to keep the coarse pointing control.
- **Nominal**. In this mode, the system is further trying to reach the fine pointing control.
- **Preparation**. The fine pointing control is reached and the unit Payload Instrument (PLI) is getting ready for the science mission.
- **Science**. The PLI is ready to carry out tasks. The overall goal is to stay in this mode as long as possible.

The normal mode transitions are shown in Figure 1(a) as a state diagram. In Figure 1(b), the bold arrows demonstrate the handling of mode transition errors in AOCS by the FDIR manager.

The Unit manager is used to manage unit resources, which encompasses avoiding conflicts in the usage of units and handling the units during their reconfiguration. There are in all seven different units: Earth Sensor (ES), Sun Sensor (SS), GPS, Star Tracker (STR), Reaction Wheel (RW), Thruster (THR) and Payload Instrument (PLI). The first four in this list are sensors, while RW and THR are actuators. The last unit, Payload Instrument, is a scientific measurement unit. In addition, there are two configurations, nominal and redundant, for each unit. Nominal unit



(a) Normal mode transitions. (b) Mode transition error handling.

Figure 1: Possible mode transitions in AOCS.

configuration is the start setup where the unit begins its operation. All of the units also have a redundant backup copy. The redundant unit is used when the nominal unit fails. Both the nominal and redundant units have two possible configurations, on or off, in the available operational modes.

3 Modeling and Verifying of AOCS

We modeled the previously described AOCS system in the input language of the NuSMV 2 model checker [CCG+02], starting from an Ada implementation. The system model is checked against a set of Linear Temporal Logic (LTL) properties that are generated from a specification given as extended state machine diagrams with prioritized transitions. Below, Section 3.1 describes the general modeling of the system, and Section 3.2 shows how the LTL properties are generated from the specification.

```
MODULE units
 VAR
   PLI : unit_t;

MODULE unit_t
 VAR
   br : branches;
   cubrid : {A_branch, B_branch};
   reconf_ongoing : boolean;
   orig_status : {free, locked};
   target_state : {none, pli_standby, pli_science};
   step : 0 .. 4;

MODULE branches
 VAR
   A_branch : branch_t;
   B_branch : branch_t;

MODULE branch_t
 VAR
   state : {none, pli_standby, pli_science};
   status : {free, locked};
   target_state : {none, pli_standby, pli_science};
   step : 0 .. 4;
   timer : {0, 3, 11, 16, 101};
   error : {none, timeout, loss_of_accuracy, invalid_data, commanding_failure};
   error_counter : boolean;
```

Figure 2: Modeling of the unit PLI.

3.1 Modeling the AOCS System

The current implementation of AOCS system is written in Ada, and the system is modeled in NuSMV 2 at a level of detail that closely follows the Ada source code. The main aspects of modeling are described below.

3.1.1 Modeling the Units

There are in all seven units in the system as described in Section 2. Incorporating so many units in a single model is likely to cause state explosion [Val96]. Our current solution to cope with this problem is to construct a concrete model of only one unit, the PLI, and to introduce an abstraction of the other units. Specifically, as only the *error* property of the other units is mentioned in the LTL formula to be checked, the abstraction omits the other aspects of the units. Thus, the basic strategy is to introduce one Boolean variable representing the *error* property of each unit that non-deterministically obtains its truth value at each time point modeling the fact that any subset of the other units can generate an error for the software to handle at any time. By these means, the state space of the model is controlled to a reasonable size.

Figure 2 shows the data structures for modeling the PLI unit in the NuSMV 2 language. The module **units** accommodates the concretely modeled units. As discussed above, the PLI is currently the only unit fully modeled, but new units could be added to the variable list in this module as needed. The module **unit_t** defines the basic properties of a unit. Here, the variable *orig_status*

```
Procedure Execute_Step is
begin
  case Trans.Target is
    when Standby =>
      Transitions.To_standbyMode;
    when ...
  end case;
end Execute_Step;


Procedure To_StandbyMode is
  begin
    case Trans.Step is
      when 0 =>
        Trans.Step := Trans.Step + 1;
      when ...
    end case;
end To_StandbyMode;
```

(a) Source code sample snippet.

```
Module main
VAR
  label : {execute_step, to_standbymode};
  trans.target : {off, standby};
  trans.step : 0 .. 1;

ASSIGN
  init(label) := execute_step;
  init(trans.target) := standby;
  init(trans.step) := 0;

  next(label) := case
    label = execute_step &
          trans.target = standby: to_standbymode;
    TRUE : label;
    esac;

  next(trans.step) := case
    label = to_standbymode & trans.step = 0 : 1;
    TRUE : trans.step;
    esac;
```

(b) NuSMV 2 modeling code sample.

Figure 3: Modeling of a code snippet in NuSMV 2.

records the status of the use of the unit. The unit can be either free or locked. The variable *step* is used to record the number of steps it has used to make transition to a new operational state since the transition is usually a multi-step process. One of the interesting properties is *br*, which specifies the two configurations, nominal and redundant, represented as A and B in the model of the unit. This property is further illustrated in module **branches**, which uses the module **branch_t** to fully define the properties of each configuration of the unit. The functions manipulating the properties of units are modeled in detail as described in Section 3.1.2 below.

3.1.2 Modeling Ada Code

The overall structure of the Ada code is an infinite loop in which the four managers, AOCS manager, FDIR manager, Mode manager and Unit manager, are executed in sequence. All the actual functions and procedures defined in the Ada source code implementation are modeled as state labels in the NuSMV 2 model. The modeling of source code generally mimics the implementation. At this point, the translation is not yet done automatically. The core of this manual modeling is to treat each function entry point in the implementation as a potential program counter value and encode each function invocation as a single time step of the model. A similar idea of constructing models from program code is presented in [BCG+09] for C programs.

Figure 3 presents an example on how the source code is manually mapped to a NuSMV 2 model. Figure 3(a) is a part of the source code excerpted from the implementation with some irrelevant code removed. The working procedure of this sample code is quite straightforward. In procedure **Execute_Step**, it checks whether the value of variable *Trans.Target* is Standby. If this condition holds, then it calls the procedure **To_StandbyMode**. The latter procedure first checks the value of the variable *Trans.Step*. If the value of this variable is 0, then it will be increased to 1. Figure 3(b) is the corresponding model in NuSMV 2. It first defines the variables used in

the procedures. Note that it has defined an additional variable *label* that does not appear in the source code. Generally, this variable is used as a program counter to determine which procedure or function is being executed at present. Thus, *label* has the two procedure names, **execute_step** and **to_standbymode**, as its possible values. Next, these variables are initialized with initial values as shown by the **init** clause. Lastly, the **next** state transitions are defined for variables *label* and *trans.step*. These transitions in the model are exactly as those implemented in the source code. Variable *label* is used as an example to show how variable values are updated. In the **case** clause, it is first checked whether the current values of *label* and *trans.target* are **execute_step** and **standby**, respectively. If this holds, the value of *label* is updated to **to_standbymode**. Otherwise, the value is unchanged. Note that the possible transitions of variable *trans.target* are not presented here for the ease of explanation. Note also that the Ada code has no recursion which makes modeling simpler, as there is no need to explicitly model the stack of the program.

3.1.3 Timer Abstraction

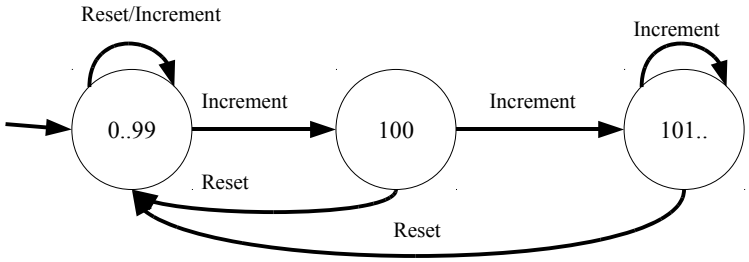
Besides abstracting entire units as described in Sect. 3.1.1, we apply a form of data abstraction. Most of the data structures in the implementation are modeled as they are, however, some of them are abstracted in the model. For instance, there are several timers in AOCS. The timers are mainly used to trigger certain events to occur and to record the timeouts of events in the Ada code. In the implementation, timers are defined as an integer type. If these were directly modeled as they are, then the modeling would become very expensive to analyze using NuSMV 2 due to state explosion. Therefore, we conduct an abstraction [CBKK94] of these timers that removes unnecessary details with non-deterministic choice in a sound way: the abstract model has more behaviors than the concrete one. If we are able to prove an LTL property for the abstract model, it will also hold in the concrete one.

As a simple example, suppose there is a timer initialized with value 0. Under normal conditions, this timer is incremented by 1 every time a periodic timer interrupt occurs. The timer is reset to 0 when a reset command is issued. Assume that increasing the timer to 100 triggers a special event. We create an abstraction of this timer as a state machine diagram shown in Figure 4(a). The value 100 of the timer is modeled as a separate state, while the values 0..99 and 101.. are collapsed into single abstract states, respectively. Especially note how non-determinism is used in state ”0..99” when the increment operation occurs to either stay in the same state or to go to the state ”100”. Such a timer abstraction can, of course, result in spurious counterexamples. However, in our case study, no spurious counterexamples are observed.

The NuSMV 2 model code for this timer is shown in Figure 4(b).

3.2 LTL Property Generation

The LTL properties that are used to check the constructed model are generated from the mode transition diagrams in the requirements document. In general, four state diagrams, normal mode transitions (Figure 1(a)), mode transition error handling (Figure 1(b)), attitude error handling and unit error handling, are used for the generation of LTL properties. The generation procedure of these LTL properties is automated from a tabular notation. The main steps are summarized below. First, the extended state machine diagrams with prioritized transitions are transformed



(a) Abstract states and transitions modeling the timer.

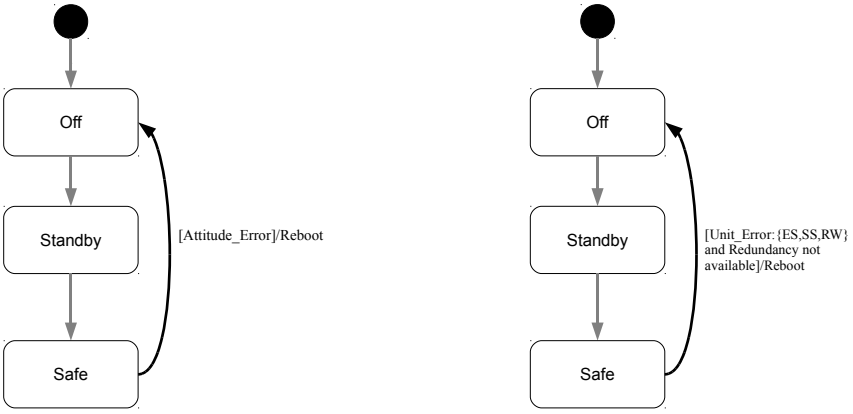
```
Module timer_xyz
VAR
  timer : {0, 100, 101};

ASSIGN
  init(timer) := 0;

  next(timer) := case
    reset_condition : 0;
    timer = 0 : {0, 100};
    timer = 100 : 101;
    TRUE : timer;
    esac;
```

(b) NuSMV 2 model code snippet for the abstract timer.

Figure 4: Abstraction of a timer.

manually to a tabular notation consisting of a priority list in which the possible mode transitions of each state are prioritized according to the specific mode. Next, a Python script is written which can read the priority list and generate the corresponding LTL properties from it.

Mode ”safe” is used below to demonstrate how its related LTL properties are automatically generated from the mode transition diagrams. First, there are three types of possible errors that can occur at a specific mode according to the previous description of the AOCS system. It might be possible that two or more errors are occurring at the same time. Thus, it is necessary to prioritize the possible errors as well as the normal mode transition so that the property can reflect the fact that the system is handling one type of mode transition at a time. According to the nature of the AOCS system, the priorities of possible state transitions are inferred as follows. The mode transition error will always have the highest priority. The priority of attitude error handling follows it. Unit error handling in turn follows the attitude error handling. Finally, normal mode transition has the lowest priority. In case of mode ”safe”, its state transition under the condition mode transition error can be found from Figure 1(b) while its related transition in the context of normal mode transition is depicted in Figure 1(a). Its mode transitions in case of attitude error and unit error are extracted from the related state diagrams and shown in Figure 5(a) and (b), respectively. Note that in Figure 5, the depicted mode transition diagrams are truncated in order



(a) Attitude error in Safe. (b) Unit error in Safe.

Figure 5: Attitude and unit errors handling in Safe mode.

Table 1: Priority list of mode transitions for Safe mode.

Mode	Priority	Guard	Target mode	Atomic
safe	1	safe_to_nominal_error	safe	yes
	2	attitude_error	off	yes
	3	unit_error_ES_SS_RW and redundancy_not_available	off	yes
	4	coarse_pointing_reached	nominal	no

to only highlight the modes involved in the transitions. The corresponding priority list of mode transitions can be constructed as shown in Table 1. In the column ”Priority” of Table 1, smaller integer indicates a higher priority of the specific mode transition.

The LTL properties related to a specific state can be generated automatically from the priority list composed in the former step. The core idea is to extract the information about current mode, guard and target mode from the list. All guards belonging to transitions of higher priorities than those of the current mode transition should be disallowed for a lower priority transition to be enabled. The following LTL formula is the template for the generation of all the properties.

$$\begin{aligned}&\mathbf{G}\big(\big((mode = CURRENT\_MODE) \wedge GUARDS\big) \rightarrow \\ &\quad \mathbf{X}\big((mode = CURRENT\_MODE)\,\mathbf{U}\,((mode = TARGET\_MODE) \vee \\ &\quad POSSIBLE\_HIGHER\_PRIORITY\_TRANSITION)\big)\big)\end{aligned} \tag{1}$$

In the above formula, *CURRENT_MODE*, *GUARDS* and *TARGET_MODE* are the variables that will be substituted by the concrete values extracted from the priority list. The variable *POSSIBLE_HIGHER_PRIORITY_TRANSITION* at the end of the template formula is needed for correct translation of mode transitions that are multi-step processes instead of atomic steps, as indicated in the column “Atomic” in Table 1. When we ran an earlier version of the experiments, we noticed an anomaly with some property that had an unexpected counterexample. We managed to trace this counterexample back to a mismatch of the levels of atomicity between the mode transition diagrams and the Ada code implementation of the system. Namely, in the Ada

implementation some of the state transitions are not atomic, and a lower priority mode transition needs to be aborted by the enabling of a higher priority transition "half-way through" a multi-step mode transition from one mode to the next. Such a mismatch between the specification and the implementation level of atomicity can possibly lead to subtle interpretations of the required behavior of the system, and our formal model checking was able to point such a case to us. The addition of *POSSIBLE_HIGHER_PRIORITY_TRANSITION* enables correct handling of aborted transitions.

As an example, the concrete LTL formulas generated from Table 1 are listed as follows.

$$\mathbf{G}\big(\big((mode = safe) \wedge (mode\_trans\_error \neq none)\big) \rightarrow \mathbf{X}\big((mode = safe)\,\mathbf{U}\,(mode = safe)\big)\big) \tag{2}$$

$$\mathbf{G}\big(\big((mode = safe) \wedge (mode\_trans\_error = none) \wedge (attitude\_error \neq none)\big) \rightarrow \mathbf{X}\big((mode = safe)\,\mathbf{U}\,(mode = off)\big)\big) \tag{3}$$

$$\mathbf{G}\big(\big((mode = safe) \wedge (mode\_trans\_error = none) \wedge (attitude\_error = none) \wedge unit\_error\big) \rightarrow \mathbf{X}\big((mode = safe)\,\mathbf{U}\,(mode = off)\big)\big) \tag{4}$$

$$\mathbf{G}\big(\big((mode = safe) \wedge (mode\_trans\_error = none) \wedge (attitude\_error = none) \wedge \neg unit\_error \wedge coarse\_pointing\_reached\big) \rightarrow \mathbf{X}\big((mode = safe)\,\mathbf{U}\,\big((mode = nominal) \vee (mode\_trans\_error \neq none) \vee (attitude\_error \neq none) \vee unit\_error\big)\big)\big) \tag{5}$$

Note that in formulas (3) and (4), the mode transition from **Safe** to **Off** is atomic according to the implementation. Thus, no additional higher priority transition needs to be added. By contrast, in formula (5), the mode transition from **Safe** to **Nominal** is a multi-step process in the implementation. The possible higher priority transitions, therefore, must be considered for the purpose of aborting handling.

It can be easily shown that these LTL formulas can be extended to related CTL formulas by simply adding the universal path quantifier **A**. Since CTL formulas are also checked in the following experiments, formula (2) is used as an example to show how CTL formula (6) can be obtained from the corresponding LTL formula.

$$\mathbf{AG}\big(\big((mode = safe) \wedge (mode\_trans\_error \neq none)\big) \rightarrow \mathbf{AXA}\big((mode = safe)\,\mathbf{U}\,(mode = safe)\big)\big) \tag{6}$$

4 Experimental Results

The experiment is carried out in a computing cluster environment with some background load. Each compute node in the cluster has two 6-core AMD Opteron 2345 CPUs and the memory

is 32 GB (about 2.5 GB per core). The cluster has in all 112 compute nodes. We use NuSMV 2.5.2 to model and verify the AOCS system. The code of the AOCS system is about 2200 lines of Ada while the NuSMV 2 model code is about 800 lines with only the unit PLI fully modeled. The model has roughly 80 variables and 25 state labels. The model currently has in total $2^{124.483}$ states, out of which 84.5 million states are reachable.

BDD-based LTL model checking is used in NuSMV 2 to check whether the generated LTL formulas hold or not. In addition, bounded model checking (BMC) is also used to find possible counterexamples. BMC is based on the reduction of the bounded model checking problem to a propositional satisfiability problem. NuSMV 2 internally invokes a propositional SAT solver to search for an assignment that satisfies the generated problem. In this experiment setup, NuSMV 2 is compiled to link to MiniSat [ES03], a high performance and open-source SAT solver. Specifically, the incremental BMC algorithm (`check_ltlspec_sbmc_inc`) [BHJ+06, ES03] is used in NuSMV 2 to check the generated LTL specifications. Since NuSMV's default BMC is incomplete, we also tried to supply the command line option *-c* that performs the completeness check [BHJ+06]. To obtain better performance results for BDD-based model checking, the value of the environment variable *partition_method* is configured as *Iwls95CP* instead of the default one. In general, this method is conjunctive partitioning with clusters generated and ordered according to the heuristic introduced in [RAB+95].

In this experiment, there are 28 LTL properties generated from the mode transition diagrams in the system according to the generation method previously described. In all the cases, the time bound set to check each LTL property is configured to be 30 minutes. The purpose is to make the checking time long enough so that it can go deeper into the state space and find possible counterexamples. CTL model checking is also carried out with the generated CTL properties as demonstrated in Subsection 3.2. In general, the CTL model checking can deliver almost the same amount of conclusive results as LTL model checking does. The used time, however, is about an order of magnitude slower than that of LTL model checking. For the BMC part, a very large integer is supplied as the bound, so that the check will only terminate when a timeout or memory out is reached, or when a counterexample is found. The experimental results are summarized in Table 2. The 28 properties are listed as P1 to P28 in the table.

In Table 2, the column "BDD_LTL" indicates the results using the NuSMV 2 BDD-based LTL model checking algorithm. **T** indicates that the property holds while **F** means that the property does not hold. The number after the forward slash is the time used by the checking and it is measured in seconds. T.O. means that the check exceeds the configured time bound. The column "BMC incomplete" represents the results using the incremental BMC algorithm while its neighbor "BMC complete" indicates the results using the same command but with command line option *-c* enabled so that it will perform the completeness check which also tries to prove the property holds. M.O. indicates that the check exceeds the memory limit. For P4', the numbers after the forward slash in these two columns indicate the time to find counterexamples. The number in the parenthesis is the step at which timeout, memory out is reached or a counterexample is found. For instance, the "BMC incomplete" column of property P1 states that the check is timed out at step 158. This means that there is no counterexample against P1 in 158 time steps. The table, as a whole, shows that the BMC without completeness check can reach larger bounds than its complete variant.

Table 2: Results of model checking LTL properties with partition method as Iwls95CP.

Property	BDD_LTL	BMC incomplete	BMC complete	Property	BDD_LTL	BMC incomplete	BMC complete
P1	**T**/101.59	T.O.(158)	M.O.(141)	P15	**T**/121.10	T.O.(434)	M.O.(141)
P2	**T**/100.99	T.O.(434)	M.O.(141)	P16	T.O.	T.O.(158)	T.O.(141)
P3	**T**/102.78	T.O.(158)	T.O.(113)	P17	T.O.	T.O.(119)	T.O.(113)
P4	**T**/79.82	T.O.(119)	T.O.(113)	P18	T.O	T.O.(119)	T.O.(113)
P4'	**F**/125.70	**F**(61)/5.23	**F**(61)/22.28	P19	T.O.	T.O.(119)	T.O.(113)
P5	**T**/99.76	T.O.(434)	M.O.(141)	P20	T.O.	T.O.(119)	T.O.(113)
P6	**T**/1133.09	T.O.(119)	T.O.(113)	P21	T.O.	T.O.(119)	T.O.(113)
P7	**T**/1434.05	T.O.(119)	T.O.(85)	P22	**T**/120.53	T.O.(434)	M.O.(141)
P8	T.O.	T.O.(119)	T.O.(85)	P23	T.O.	T.O.(158)	T.O.(141)
P9	**T**/105.49	T.O.(434)	T.O.(141)	P24	T.O.	T.O.(119)	T.O.(113)
P10	T.O.	T.O.(119)	T.O.(113)	P25	T.O.	T.O.(119)	T.O.(113)
P11	T.O.	T.O.(119)	T.O.(113)	P26	T.O.	T.O.(119)	T.O.(113)
P12	**T**/1205.76	T.O.(119)	T.O.(113)	P27	T.O.	T.O.(119)	T.O.(113)
P13	T.O.	T.O.(119)	T.O.(113)	P28	**T**/78.47	T.O.(198)	T.O.(141)
P14	T.O.	T.O.(119)	T.O.(113)				

Let us study in detail the property P4, which has the LTL representation

$$\begin{aligned}&\mathbf{G}\big(\big((mode = standby) \wedge (mode\_trans\_error = none) \wedge (attitude\_error = none) \wedge \\ &\quad separation\_done\big) \rightarrow \\ &\quad \mathbf{X}\big((mode = standby)\,\mathbf{U}\,((mode = safe) \vee (mode\_trans\_error \neq none) \vee \\ &\qquad\qquad (attitude\_error \neq none))\big)\big). \end{aligned} \tag{7}$$

The formula contains the condition *POSSIBLE_HIGHER_PRIORITY_TRANSITION*, as discussed in Section 3.2 above, to enable aborting the multi-step transition by a higher-priority transition. A previous version of the property, denoted by P4' in Table 2, has the LTL representation

$$\begin{aligned}&\mathbf{G}\big(\big((mode = standby) \wedge (mode\_trans\_error = none) \wedge (attitude\_error = none) \wedge \\ &\quad separation\_done\big) \rightarrow \\ &\quad \mathbf{X}\big((mode = standby)\,\mathbf{U}\,(mode = safe)\big)\big) \end{aligned} \tag{8}$$

and omits the possibility of aborting the transition. As seen from the table, using P4' results in a false negative model checking result.

5 Conclusions

The AOCS system is a typical instance of a mission-critical system with various mode transitions triggered by inputs from a highly non-deterministic environment, including recovering from components faults. We have described how a symbolic model checker input language can be used to model an implementation of the AOCS system given its implementation in Ada source code. One of the key methods employed has been abstraction which can control the state space of

the model to a reasonable size. We have described techniques and abstractions used to alleviate the state explosion problem due to handling of timers and the large number of system components controlled by AOCS. The LTL properties used to check the model are generated based on mode transition diagrams of the system and the generation procedure is automated from a tabular notation.

Even after the abstractions, the AOCS system is currently too large to be fully automatically verified using symbolic model checking methods. Instead of getting rid of the close one-to-one correspondence of the Ada source code and the corresponding NuSMV 2 model, and thus making the model state space more manageable, we have instead resorted to incomplete model checking methods, bounded model checking in particular. This leaves us with challenging problems for the symbolic model checker development work that we are also concurrently doing in other projects.

In the experiments, the BDD-based LTL model checking can deliver results to about half of all the checked properties while for the other properties the approach times out. The incremental BMC algorithm is mainly used for bug hunting but it does also show the non-existence of short counterexamples to the remaining properties. A timed out BDD-based LTL model checking run, however, does not provide any additional information of the property. Out of curiosity, we also ran the complete BMC algorithm, that tries to also prove properties correct, not only look for counterexamples. Similar to our previous experiences, the complete BMC algorithm is not able to prove properties correct for models of significant size.

There are many topics for further work. Firstly, the modeling from Ada source code to NuSMV 2 models is currently manual work. If this part was automated, a lot of optimizations that are easy to do automatically would become available. The Ada code is sequential, and as only the system mode changes are observed from the outside, many of the internal states could potentially be automatically removed by an optimizing “model compiler”. However, doing such optimizations manually is currently too time consuming and risky (it is easy to make modeling errors when “hand optimizing the model”).

On the BMC side, the parallel and distributed BMC engine of [WNH09] could be used to go deeper into the system state space by exploiting multiple multi-core computers running on a single BMC instance in parallel. While the properties we check are not strictly safety properties (some of the counterexamples could be infinite loops where e.g., the mode does not change at all), they still have some counterexamples that can be represented by finite paths (e.g., the mode entered next is a wrong one). For these latter “safety” counterexamples, the approach of [LHJ10] can be used, which is tailored to more efficiently finding the safety counterexamples of PSL (superset of LTL) properties using BDD-based engines.

Acknowledgements: We would like to thank Space Systems Finland for providing us with all the materials needed to complete this case study. This project has been financially supported by the RECOMP project funded by ARTEMIS-JU, Tekes - Finnish Funding Agency for Technology and Innovation, Conformiq Software, Space Systems Finland, and Academy of Finland (projects 128050 and 139402).

Bibliography

[BCCZ99] A. Biere, A. Cimatti, E. M. Clarke, Y. Zhu. Symbolic Model Checking without BDDs. In Cleaveland (ed.), *TACAS*. LNCS 1579, pp. 193–207. Springer, 1999.

[BCG+09] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, R. Sebastiani. Software model checking via large-block encoding. In *FMCAD'2009*. Pp. 25–32. 2009.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Inf. Comput.* 98(2):142–170, 1992.

[BDL+06] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, M. Hendriks. UPPAAL 4.0. In *QEST*. Pp. 125–126. IEEE Computer Society, 2006.

[BFV+09a] K. Björkman, J. Frits, J. Valkonen, J. Lahtinen, K. Heljanko, I. Niemelä, J. J. Hämäläinen. Verification of Safety Logic Designs by Model Checking. In *Proceedings of the Sixth American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies NPIC&HMIT 2009*. Knoxville, Tennessee, April 2009.

[BFV+09b] K. Björkmann, J. Frits, J. Valkonen, K. Heljanko, I. Niemelä. Model-Based Analysis of a Stepwise Shutdown Logic. VTT Working Papers 115, VTT Technical Research Centre of Finland, Espoo, 2009.
http://www.vtt.fi/inf/pdf/workingpapers/2009/W115.pdf

[BHJ+06] A. Biere, K. Heljanko, T. Junttila, T. Latvala, V. Schuppan. Linear Encodings of Bounded LTL Model Checking. *Logical Methods in Computer Science* 2(5:5), 2006.

[BK08] C. Baier, J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[CBKK94] P. J. Clarke, D. Babich, T. M. King, B. M. G. Kibria. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems* 16:1512–1542, 1994.

[CCG+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *CAV'2002*. LNCS 2404, pp. 359–364. Springer, 2002.

[CGP99] E. M. Clarke, O. Grumberg, D. A. Peled. *Model Checking*. The MIT Press, 1999.

[ES03] N. Eén, N. Sörensson. An Extensible SAT-solver. In Giunchiglia and Tacchella (eds.), *SAT'2003*. LNCS 2919, pp. 502–518. Springer, 2003.

[HJL05] K. Heljanko, T. Junttila, T. Latvala. Incremental and Complete Bounded Model Checking for Full PLTL. In Etessami and Rajamani (eds.), *CAV'2005*. Lecture Notes in Computer Science 3576, pp. 98–111. Springer, 2005.

[ILT10] A. Iliasov, L. Laibinis, E. Troubitsyna. An Event-B Model of the Attitude and Orbit Control System. http://deploy-eprints.ecs.soton.ac.uk/, 2010.

[ITL+10a] A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, D. Ilic, T. Latvala. Developing Mode-Rich Satellite Software by Refinement in Event-B. In Kowalewski and Roveri (eds.), *FMICS*. LNCS 6371, pp. 50–66. Springer, 2010.

[ITL+10b] A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, P. Väisänen, D. Ilic, T. Latvala. Verifying Mode Consistency for On-Board Satellite Software. In Schoitsch (ed.), *SAFECOMP*. LNCS 6351, pp. 126–141. Springer, 2010.

[LHJ10] T. Launiainen, K. Heljanko, T. Junttila. Efficient Model Checking of PSL Safety Properties. In *Proceedings of the 10th International Conference on Application of Concurrency to System Design (ACSD'2010)*. Pp. 95–104. Braga, Portugal, June 2010.

[RAB+95] R. K. Ranjan, A. Aziz, R. K. Brayton, B. Plessier, C. Pixley. Efficient BDD Algorithms for FSM Synthesis and Verification. In *IEEE/ACM Proceedings International Workshop on Logic Synthesis, Lake Tahoe (NV)*. 1995.

[Val96] A. Valmari. The State Explosion Problem. In Reisig and Rozenberg (eds.), *Petri Nets*. Lecture Notes in Computer Science 1491, pp. 429–528. Springer, 1996.

[Var10] K. Varpaaniemi. DEPLOY Work Package 3 Attitude and Orbit Control System Software Requirements Document (DEP-RP-SSF-R-005, issue 1.0). http://deploy-eprints.ecs.soton.ac.uk/, 2010. Documentation. DEPLOY project. Space Systems Finland Ltd.

[VPB+08] J. Valkonen, V. Petterson, K. Björkman, J.-E. Holmberg, M. Koskimies, K. Heljanko, I. Niemelä. Model-Based Analysis of an Arc Protection and an Emergency Cooling System – MODSAFE 2007 Working Report. VTT Working Papers 93, VTT Technical Research Centre of Finland, Espoo, Finland, Feb. 2008.
http://www.vtt.fi/inf/pdf/workingpapers/2008/W93.pdf

[WNH09] S. Wieringa, M. Niemenmaa, K. Heljanko. Tarmo: A framework for Parallelized Bounded Model Checking. In Brim and Pol (eds.), *Proceedings of the 8th International Workshop on Parallel and Distributed Methods in Verification (PDMC'09)*. Electronic Proceedings in Theoretical Computer Science (EPTCS) 14, pp. 62–76. 2009.
http://dx.doi.org/10.4204/EPTCS.14.5

Verifying Autonomous Systems

Michael Fisher*

Department of Computer Science, University of Liverpool, U.K.

The essential aspect of *autonomous systems* is that they must *decide for themselves* what to do and when to do it. How often they need to do this depends on the level of autonomy utilized.

Autonomous Systems are Useful. Modern business, industrial, and even domestic, systems increasingly rely on core autonomous software. But why? *Autonomy* is typically used where direct human control is (a) infeasible, or (b) expensive. In (a), such systems are required to work in distant, dangerous, or overly complex environments where humans cannot directly supervise activities; in (b), autonomy can often lead to cheaper and more efficient solutions.

Autonomous Systems are Critical. Autonomous software is able to *decide for itself* about the best course of action to take in any given situation. However, *how can we be sure the software will do what we would have done?* In spite of such concerns, autonomous systems are being deployed in both *safety critical* (e.g. aerospace, medical monitoring, industrial processes) and *business critical* (e.g. financial, security, privacy) areas. Examples include *robotic assistants* being developed for use in home healthcare, autonomous (air, road, space, underwater) vehicles, and pervasive environments where a complex monitoring, reporting and advisory system might act autonomously based upon multiple sensor inputs. For all these, we must know: *is it safe*; *is it secure*; and *will it always do what we wish it to*?

Autonomous Systems are Different. Crucially, we now need to assess not just *what* a system does, but *why* it chose to do it. In this, the "agent" concept has been found to be a very useful abstraction for describing autonomous behaviour in that it essentially captures *flexible autonomous action*, i.e. "the ability to make its own decisions and act independently from its environment". Yet the "rational agent" concept has now superseded this. A rational agent "should also have explicit *reasons* for making the choices it does, and should be able to explain these if necessary". Such agents are typically programmed and analyzed by describing their activities (e.g. 'actions'), motivations (e.g. 'goals'), information (e.g. 'knowledge'/'belief'), and how all these change over time. Thus, a rational agent must dynamically monitor and instigate new activity, assess, and possibly revise, the information it holds, generate new motivations or revise current ones, and also decide what to do, i.e. *deliberate* over motivations and actions. Such a model is, typically, easier to understand, program, and maintain, involves shorter code length, and is much more flexible than standard approaches [DFL+10a]. This has led to *hybrid* autonomous systems, combining rational agents for high-level autonomous decisions, with standard control systems for low-level activities, now being widely used in real applications [DFL+10b].

Autonomous Systems need *Formal Verification*. For autonomous software, we not only need to ask "*what will it do?*", but also "*why does it make this choice?*" and "*will its decisions change as it learns new behaviour or new behaviour emerges?*". The criticality and complexity of autonomous software calls for deep analysis which is provided through the *formal description of requirements using combinations of logics, and then formal verification*. Since we must verify not just *what* the software does, but *why* it chooses to do it, standard temporal specifications are extended with logics of *goals*, *beliefs*, etc. For example, in a robotic health-care scenario, we might require *"if a patient is in danger, then the medic robot believes that there is a probability of 95% that, within 2 minutes, a helper robot will* want *to assist the patient"*. With appropriate logical combinations we might formalize the above as the (complex) formula

$$in\_danger(patient) \Rightarrow B^{\geq 0.95}_{medic} \Diamond^{\leq 2} G_{helper}\, assist(patient)$$

* Support from EPSRC, through projects "Engineering Autonomous Space Software", "Verifying Interoperability Requirements in Pervasive Systems", and "Model Checking Agent Programming Languages". EMAIL: MFisher@liv.ac.uk

Given an *autonomous system* based on rational agent(s), together with *logical requirements*, we have several options for formal verification. For example, our AJPF model-checker, extends JPF [VHB+03], a Java model checker, and assesses logical agent specifications against the *actual* agent code executed. This code is provided in any agent programming language based on the AIL semantic toolkit [DFWB11].

Example: Formation-Flying Satellites. Increasingly, autonomous satellites will work together, via *formation-flying*, allowing them to carry out a range of tasks in a more flexible manner than a single, larger satellite. We have developed a hybrid agent architecture for such satellites [DFL+10b], have shown how this reduces the development complexity [DFL+10a], and how this interacts with adaption and learning in the underlying control systems [LVD+10]. Typical questions we address are: if an individual satellite senses that it is *"out of position"*, will it (*quickly?*) set a goal of getting back into position; if one of the satellites in the formation *fails*, either fully or partially, will the others be able to complete the mission; and will the satellites be able to remain in safe fuel bounds, yet still be available for interesting "sensing" opportunities?

Example: Autonomy in the Air. Formal Methods for Aerospace [BF09] is an increasingly important area. Formal verification can particularly play a role in the *certification* of unmanned air vehicles (UAVs). Since manned air vehicles are certified to fly, then what is the core *difference* between a UAV and a manned air vehicles? Clearly, one has a human pilot while the other has an "autonomous agent". So, why not show that, in all important aspects, the "agent" will behave just as a pilot would? In other words, that the agent *equivalent to* the pilot? Though this is *impossible* in general, for the "rules of the air" that the pilot should follow, we can use formal verification of the agent to establish this equivalence and even formalize and verify some aspects of "airmanship" [WFCJ11].

Example: Safe Human-Robot Interaction. When autonomous robots interact with humans, we must ask: *are they safe*; *will the robot understand what we want it to do*; and then *will the robot decide to do what we want?* We utilize the Brahms language for modelling the high-level interactions in human-robot teams [SC02] in which humans are treated (at an abstract level) as rational agents. We then aim to verify Brahms teamwork descriptions [BFS09] via use of their formal semantics [SSD+11].

Bibliography

[BF09] M. L. Bujorianu, M. Fisher (eds.). *Proceedings FM-09 Workshop on Formal Methods for Aerospace*. EPTCS 20. 2009.

[BFS09] R. H. Bordini, M. Fisher, M. Sierhuis. Formal Verification of Human-Robot Teamwork. In *Proc. 4th ACM/IEEE Int. Conf. on Human Robot Interaction (HRI)*. Pp. 267–268. ACM Press, 2009.

[DFL+10a] L. A. Dennis, M. Fisher, N. Lincoln, A. Lisitsa, S. M. Veres. Reducing Code Complexity in Hybrid Control Systems. In *Proc. 10th Int. Symp. on Artificial Intelligence, Robotics and Automation in Space (i-Sairas)*. 2010.

[DFL+10b] L. A. Dennis, M. Fisher, A. Lisitsa, N. Lincoln, S. M. Veres. Satellite Control Using Rational Agent Programming. *IEEE Intelligent Systems* 25(3):92–97, May/June 2010.

[DFWB11] L. A. Dennis, M. Fisher, M. Webster, R. H. Bordini. Model Checking Agent Programming Languages. *Automated Software Engineering*, pp. 1–59, 2011.

[LVD+10] N. Lincoln, S. M. Veres, L. A. Dennis, M. Fisher, A. Lisitsa. An Agent Based Framework for Adaptive Control and Decision Making of Autonomous Vehicles. In *Proc. IFAC Workshop on Adaptation and Learning in Control and Signal Processing (ALCOSP)*. 2010.

[SC02] M. Sierhuis, W. J. Clancey. Modeling and Simulating Work Practice: A Human-Centered Method for Work Systems Design. *IEEE Intelligent Systems* 17(5), 2002.

[SSD+11] R. Stocker, M. Sierhuis, L. A. Dennis, C. Dixon, M. Fisher. A Formal Semantics for Brahms. In *Proc. 12th Int. Workshop on Computational Logic in Multi-Agent Systems (CLIMA)*. 2011.

[VHB+03] W. Visser, K. Havelund, G. P. Brat, S. Park, F. Lerda. Model Checking Programs. *Automated Software Engineering* 10(2):203–232, 2003.

[WFCJ11] M. Webster, M. Fisher, N. Cameron, M. Jump. Model Checking and the Certification of Autonomous Unmanned Aircraft Systems. In *Proc. 30th Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP)*. 2011.

Towards an automatic formal model generation and verification derived from a graphical model

Lukas Ladenberger[1], **Aryldo G Russo Jr**[2]

[1] lukas.ladenberger@gmx.de [2] agrj@aes.com.br

[1] University of Duesseldorf [2] University of São Paulo

One of the most important safety critical systems is the signaling system which allows the train movement in a safe condition. In such systems, certain properties need to be exhaustively verified in order to guarantee a minimum confidence regarding the reliability of the system. *Boolean Equations* (BE) are used to express these properties [GVK95].

In this paper we propose a graphical tool prototype called VeRaSiS for generating and validating automatically the safety properties of train movement in a railway system. The tool has four main features: graphical simulation of track topologies, fully automatic BE generation, converting of BE to a Event-B model, and BE validation. Figure 1 gives an overview of the architecture and the work flow of the tool.

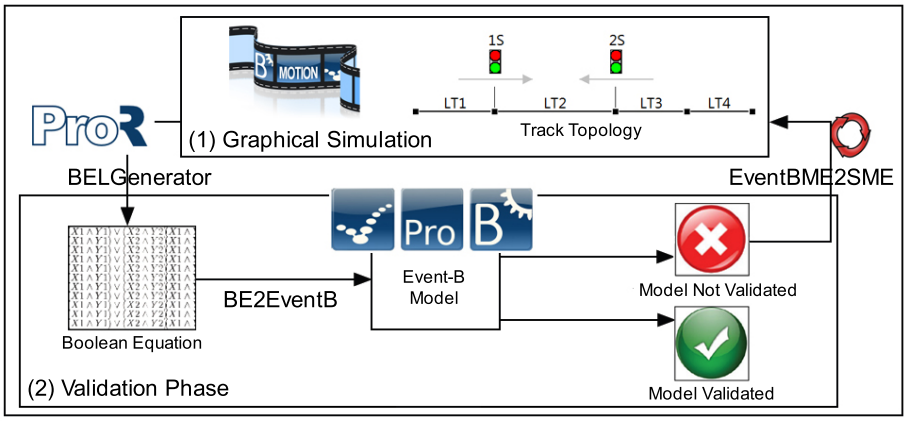


Figure 1: The VeRaSiS architecture and approach

For the *graphical simulation* of track topologies we narrowed our choices to (1) the development of a new tool from scratch, or (2) to extend an existing one. Since building a tool from scratch is a time consuming task, and several tools are already available to support graphical simulation, we decided to use an existing tool. Considering that we wanted to generate an Event-B model, some tools from the Event-B tool landscape were analyzed. There are, to our best knowledge, only two tools available that allow graphical simulation of Event-B models. Brama [Ser06] and BMotion Studio [LBL09] (BMS). In Brama, the modeler creates the graphical representation of the model with Flash. This tool required knowledge of Flash programming to be used. The second tool was BMS. Similar to Brama, BMS enables the developer of a Event-B model to set-up a domain specific visualization. However, BMS allows the model creation with static images and drag and drop, not requiring additional skills. For this reason and since the possibil-

ity to represent graphically a track topology already exists in BMS, we decided to use this tool. However, the approach of BMS is in conflict with the VeRaSiS work flow, where the formal model is generated from the graphical visualization. As a consequence, we had to extend BMS in order to create visualizations which are not based on an existing formal model.

The *validation* phase aims to generate and validate the BE extracted from the graphical simulation of the track topology. The tool used to translate the graphical representation to BE is called *BELGenerator*. To support the generation, a set of typical BE (or templates) are required to be created and managed to represent the properties of each component of a track topology. To manage this "library" we will use ProR [Jas10] which is a requirements management tool with the capability to trace between requirements and a Event-B model. The VeRaSiS tool will instantiate each necessary template with the information extracted from the graphical simulation. For instance, consider the track topology in figure 1 in the upper box. In order to move from the track block LT1 to LT4 the following BE must be validated:

$$LT1 - LT4 = S1(Green) * S2(Red) * LT2 * LT3 * LT4 \quad (1)$$

In other words, to move from LT1 to LT4, LT2, LT3 and LT4 have to be free, the signal S2 has to be red and the signal S1 has to be green.

Finally, the tool called *BE2EventB* converts automatically the created BE to a Event-B model which is then validated using ProB [LB08]. In the case that there are errors generated by ProB, or if ProB finds a problem, those messages are translated in a straightforward message with the *EventBME2SME*. The first version of this tool will manage just basic errors.

The tool is designed for industrial use, where we face the challenge that mainstream users are not familiar with formal modeling. The tool presents the specification in the user's domain, shielding them from the formalism.

In future work, we will demonstrate the VeRaSiS approach in a case study based on industrial specifications. As a consequence, our main goal is to develop a fully functional first version of the VeRaSiS tool in order to prove its usefulness. This includes in particular the creation of the BE templates in order to instantiate the BE for other properties of a track like the speed limit.

Bibliography

[GVK95] J. F. Groote, S. van Vlijmen, J. Koorn. The Safety Guaranteeing System at Station Hoorn-Kersenboogerd. In *Utrecht University*. Pp. 57–68. IEEE, 1995.

[Jas10] M. Jastram. ProR, an Open Source Platform for Requirements Engineering based on RIF. *SEISCONF*, 2010.

[LB08] M. Leuschel, M. Butler. ProB: An Automated Analysis Toolset for the B Method. *Journal Software Tools for Technology Transfer* 10(2):185–, 2008.

[LBL09] L. Ladenberger, J. Bendisposto, M. Leuschel. Visualising Event-B models with B-Motion Studio. In *Proceedings FMICS'2009*. LNCS 5825, pp. 202–204. Verlag, 2009.

[Ser06] T. Servat. BRAMA: A New Graphic Animation Tool for B Models. Springer Berlin / Heidelberg, 2006.

Designing synchronous to asynchronous model translations for interlocking systems verification

Yassin Chkouri[1], **José Esteves**[2], **Elie Soubiran**[2]

[1]yassin.chkouri@evosys.fr, [2]jose.esteves@evosys.fr, [3]elie.soubiran@evosys.fr
Evosys, France, http://www.evosys.fr

Abstract: In this article, we present a formal framework that enables the use of the Tina model checker to verify safety properties on synchronous interlocking systems that are concretely deployed on french stations. Our process is decomposed in two steps: first we translate the synchronous interlocking system in a semantically equivalent asynchronous Fiacre model, then we use the tool-chain frac-tina-selt to verify railway safety properties and computational properties.

Keywords: Model transformation, LTL model checking, interlocking system, synchronous and asynchronous model

EVOSYS lead during the last year a R&D project to evaluate the feasibility of formally verifying safety and computing properties of synchronous railway interlocking models. The main achievement of this project was the design of three translations from synchronous models written in Scade [6] or Lucid Synchrone [3] to semantically equivalent asynchronous models expressed in Fiacre [1]. The first translation is fashioned as a general translation from synchronous to asynchronous models and the two others are more optimized for our concrete case study. Considered systems are built from a set of nodes, containing one automaton, which are instantiated and evaluated by a main execution engine. An interesting property of these models is that instantiated nodes can be evaluated sequentially in any order without modifying the general behavior of the system. Our largest test case is still a small railway station, consisting in 244 automata, 300 commands and inputs, 375 outputs, 7 routes and 11 sub-routes.

Our first approach tends to be agnostic from the specific structure of interlocking systems and is based on the mapping of each equation and automaton of the synchronous model to a Fiacre process, and each node to a Fiacre component. These components compose in a parallel way all processes coming from theirs respective local equations and automata translation. In this approach, synchronous cycles are simulated through a dedicated rendez-vous which gather every Fiacre process.

Our second translation leverages the particular form of interlocking system written in Scade and allows us to build a more linear translation in terms of generated processes and channels. We translate each node in an equivalent Fiacre process, the inputs and the outputs of the nodes are carried through dedicated multiplexed channels. Since we can choose an arbitrary order for the evaluation of each instantiated node, we build a special finite-state-machine that sequentially stimulates each node process and that consequently simulates cycles.

Our last translation builds from the initial model a single process that simulates sequentially each node, and a component that declares variables and instantiates the process. Roughly speaking, everything happens as if we had inlined every node of the synchronous model in the execu-

tion engine, translated each automaton to an equivalent conditional statement[1] and then translated the result to a Fiacre model.

Experimental results, with the Tina toolbox [2], showed us the importance of the translation scheme to control the state space explosion. While the first translation is optimized to maximize the parallelism of computation, the third one is more adapted to model checking activities. Indeed, a new state is potentially generated each time a process is evaluated, in the first translation a process corresponds to an atomic Scade instruction while in the third a unique process corresponds to the whole interlocking system. We gain a 100 factor in term of size of the state space by using the third translation, it allowed us to verify a model with up to 5 routes randomly activated and random train traffic.

A similar work from ours is [5], the authors translate the interlocking system in an UML model and use an existing translation from UML to NuSMV. After a testing phase, they conclude that this approach is suitable for simulation but not for model checking. Another interesting work, that scales to medium sized interlocking systems is [4], the authors use a SAT solver to verify safety properties. Contrary to ours, their approach does not provide counter examples as traces and in our opinion this does not ease the analysis performed by railways engineers. We have studied a system, based on a synchronous product of automata, that mainly manipulate boolean variables. This is clearly not the perfect kind of models for explicit model checkers. However, with some optimizations, medium sized systems can be partially checked. While not being entirely satisfactory, this methodology allows us to quickly verify models that subsume large classes of unitary scenarios, whereas, a test based approach would have been time consuming. Our interest is now on symbolic model checking combined with some abstractions and slicing algorithms which are consistent with railway safety properties.

References

[1] B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, and F. Vernadat. Fiacre: an Intermediate Language for Model Verification in the Topcased Environment. In *ERTS 2008*, Toulouse France, 2008.

[2] B. Berthomieu and F. Vernadat. Time Petri Nets Analysis with TINA. In *QEST*, pages 123–124. IEEE Computer Society, 2006.

[3] J.-L. Colaço, B. Pagano, and M. Pouzet. A conservative extension of synchronous data-flow with state machines. In W. Wolf, editor, *EMSOFT*, pages 173–182. ACM, 2005.

[4] K. Kanso, F. Moller, and A. Setzer. Automated verification of signalling principles in railway interlocking systems. *Electronic Notes in Theoretical Computer Science*, 250:19 – 31, 2009.

[5] Y. Man Hon and M. Kollmann. Simulation and Verification of UML-based Railway Interlocking Designs. In S. Merz and T. Nipkow, editors, *Automatic Verification of Critical Systems*, pages 168–172, Nancy/France, Sept. 2006.

[6] E. Technologies. The scade reference manual, 2010.

[1] As it is done in the compilation scheme of the Lucid Synchrone language

Approximating idealised real-time specifications using time bands

Brijesh Dongol[1] **and Ian J. Hayes** [2*]

[1]brijesh@itee.uq.edu.au [2]Ian.Hayes@itee.uq.edu.au
School of Information Technology and Electrical Engineering,
The University of Queensland

Abstract: Timed specifications are often formalised at an absolute level of precision, which does not always reflect the real world that the specifications model, i.e., in the real world, inputs cannot be sampled with absolute precision and physical hardware cannot react instantaneously. As a result the developed specifications can often become unimplementable. In this paper, we consider the time bands model in which time may be structured into several layers of abstraction and relationships between bands may be formalised. This allows the ideal timed specifications to be approximated at the time band in which the variables are sampled. We consider implementation of the approximated specifications using teleo-reactive programs embedded with time bands.

Keywords: Time bands, Real-time systems, Teleo-reactive programs, Refinement, Cyber-physical systems

1 Introduction

There is an increasing prevalence of *cyber-physical* systems, where software agents are used to control physical systems. In safety-critical applications, one must ensure dependability of the overall system; however, formally reasoning about cyber-physical systems is complicated because we must inherently consider real-time properties, parallelism between an agent and its environment and hardware/software interactions. Furthermore, components tend to operate over multiple time granularities (e.g., days, milliseconds) and our reasoning must be able to incorporate these within a single formalism.

A more difficult task is the development of correct real-time implementations via stepwise refinement. Real-time logics tend to provide a specification that is too precise about its timing requirements and assumptions and the models that are developed often become unimplementable due to the mismatch between the idealised assumptions of the specification and imprecision of digital clocks, delays in processing and imprecision of physical hardware that are inevitable in the real world. However, development of implementations from idealised assumptions remains an attractive option because such developments only need to consider the interactions between components. We propose a compromise, where we develop implementations of idealised specifications that only require us to consider the functional aspects of the system. To facilitate implementation, we approximate the idealised specifications and consider the changes necessary to the ideal program so that it implements the approximated specification.

* This research is supported by Australian Research Council Discovery Grant DP0987452.

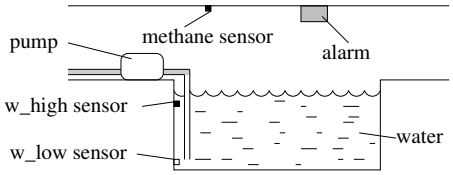


Figure 1: Mine pump example

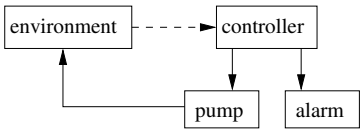


Figure 2: Controller diagram

To this end, we propose a method of refinement using the sampling logic of time bands [BH10], where timed specifications may be associated with notions such as *precision* and *accuracy*. The aim is to develop a refinement of an idealised specification (i.e., without considering real-world imprecisions). We then approximate the specification to the time bands in which the variables occur and consider the modifications that are necessary for the implementation to satisfy the approximated specification.

Throughout this paper, we consider the safety-critical system in Fig. 1 in which a pump is used to remove water from a mine shaft [BL91]. To prevent an explosion, the pump must not be operating if there is a critically high level of methane in the mine. As depicted in Fig. 2, the controller must sample sensor values from the environment (dashed arrow) and send signals to the pump and alarm to turn them on or off. In turn, the pump may change the water level, which affects the state of the environment.

We implement the controllers using teleo-reactive programming model, which is a high-level approach to developing real-time systems [Nil01, Hay08]. The teleo-reactive paradigm is radically different from frameworks such as action systems, timed automata and TLA$^+$ because actions are considered to be durative (as opposed to instantaneous). Teleo-reactive programs are particularly useful for implementing controllers for autonomous agents that must react robustly to their dynamically changing environments.

2 A real-time framework

We reason about a teleo-reactive program by reasoning about the contiguous time intervals over which it executes. We model time using the real numbers $\mathbb{R}$ and define:

$$\mathit{Interval} \mathrel{\widehat{=}} \left\{ \Delta \subseteq \mathbb{R} \,\middle|\, \Delta \neq \{\} \wedge \forall t, t' : \Delta \bullet \forall t'' : \mathbb{R} \bullet t < t'' < t' \Rightarrow t'' \in \Delta \right\}$$

Thus, if t and t' are in the interval Δ, then all real numbers between t and t' are also in Δ. Note that an interval may be open or closed at either end. We use $glb.S$ and $lub.S$ to refer to the *greatest lower* and *least upper bounds* of a set of numbers S, respectively, where we use '.' to represent function application. For intervals Δ, Δ', we define the *length* of Δ (denoted $\ell.\Delta$) and Δ *adjoins* Δ' (denoted $\Delta \propto \Delta'$) as follows:

$$\begin{array}{rcl} \ell.\Delta & \mathrel{\widehat{=}} & lub.\Delta - glb.\Delta \\ \Delta \propto \Delta' & \mathrel{\widehat{=}} & (lub.\Delta = glb.\Delta') \wedge (\Delta \cup \Delta' \in \mathit{Interval}) \wedge (\Delta \cap \Delta' = \{\}) \end{array}$$

The *partitions* of an interval $\Delta \in \mathit{Interval}$ is given by $\Pi.\Delta$, which is defined as follows:

$$\Pi.\Delta \mathrel{\widehat{=}} \{ z : \mathrm{seq}.\mathit{Interval} \mid (\Delta = \bigcup \mathrm{ran}.z) \wedge (\forall i : \mathrm{dom}.z - \{0\} \bullet z.(i-1) \propto z.i) \}$$

Given that variable names are taken from the set $V \subseteq Var$, a *state space* is given by $\Sigma_V \mathrel{\widehat{=}} V \to Val$, which is a total function mapping each variable in V to a value from Val. We leave out the subscript when the set V is clear from the context. A *state* is a member of Σ_V and a *predicate* over a type X is given by $\mathscr{P}X \mathrel{\widehat{=}} X \to \mathbb{B}$ (e.g., a *state predicate* is a member of $\mathscr{P}\Sigma_V$). The (real-time) trace of environment behaviours is given by $Stream_V \mathrel{\widehat{=}} \mathbb{R} \to \Sigma_V$, which is a total function from time to states representing the evolution of the state of the system over time. To define properties of programs running over a time interval Δ, we use an *interval stream predicate*, which has type $IntvPred_V \mathrel{\widehat{=}} Interval \to \mathscr{P}Stream_V$.

We use $\lim_{x \to a^+} f.x$ and $\lim_{x \to a^-} f.x$ to denote the limit of $f.x$ from the right and left, respectively. To ensure that limits are well defined, we assume all variables are piecewise continuous [GM01]. For an expression e, interval Δ and stream s, we define:

$$\overrightarrow{e}.\Delta.s \mathrel{\widehat{=}} \begin{cases} e.(s.(lub.\Delta)) & \text{if } lub.\Delta \in \Delta \\ \lim_{t \to lub.\Delta^-} e.(s.t) & \text{otherwise} \end{cases} \qquad \overleftarrow{e}.\Delta.s \mathrel{\widehat{=}} \begin{cases} e.(s.(glb.\Delta)) & \text{if } glb.\Delta \in \Delta \\ \lim_{t \to glb.\Delta^+} e.(s.t) & \text{otherwise} \end{cases}$$

Thus, $\overrightarrow{e}.\Delta.s$ and $\overleftarrow{e}.\Delta.s$ denote the value of e in the right and left limits of Δ, respectively. For a state predicate c and interval $\Delta \in Interval$ we define the *always* and *sometime* operators as follows.

$$\begin{aligned} (\circledast c).\Delta.s &\mathrel{\widehat{=}} \forall t{:}\Delta \bullet c.(s.t) \\ (\boxdot c).\Delta.s &\mathrel{\widehat{=}} \exists t{:}\Delta \bullet c.(s.t) \end{aligned}$$

Example 1 (Safety requirement for the mine pump.) *A required safety condition of our mine pump example is that in any state of the real-time stream (i.e., at the absolute level of precision), if the methane level, m, is above the critical level, C, then the pump must be stopped. Note that this means that the pump must have physically come to a stop, which we distinguish from the output control signal that turns the pump off.*

$$Safety \mathrel{\widehat{=}} \circledast(m \geq C \Rightarrow stopped) \tag{1}$$

For an interval predicate p and interval Δ, we say p holds in a *previous* interval if $(prev.p).\Delta$ holds, which is defined as follows:

$$(prev.p).\Delta \mathrel{\widehat{=}} \exists \Delta'{:}Interval \bullet \Delta' \propto \Delta \wedge p.\Delta'$$

We use $\circledast$ and *prev* to define *invariance* of a state predicate c and *stability* of a set of variables V as follows:

$$\begin{aligned} inv.c &\mathrel{\widehat{=}} (prev.\overrightarrow{c} \Rightarrow \circledast c) \\ st.V &\mathrel{\widehat{=}} \forall v{:}V \bullet \exists k{:}Val \bullet inv.(v = k) \end{aligned}$$

Hence, $(inv.c).\Delta$ holds iff $\circledast c.\Delta$ holds provided that c holds at the right limit of an interval that precedes Δ (i.e., c is *invariant* in Δ) and $st.V$ holds iff the value of each $v \in V$ does not change within Δ (i.e., V is *stable* in Δ).

We define the following operators to facilitate reasoning about interval predicates. The *chop* ';' operator allows one to split a given interval into two parts, where p_1 holds for the first interval

and p_2 holds for the second [ZH04]. The *weak chop* ':' states that either p_1 holds or the chop $p_1 ; p_2$ holds over the given interval.

$$(p_1 ; p_2).\Delta \;\widehat{=}\; \exists \Delta_1, \Delta_2 : Interval \bullet (\Delta = \Delta_1 \cup \Delta_2) \wedge (\Delta_1 \propto \Delta_2) \wedge p_1.\Delta_1 \wedge p_2.\Delta_2$$
$$(p_1 : p_2).\Delta \;\widehat{=}\; p_1.\Delta \vee (p_1 ; p_2).\Delta$$

Note that unlike the duration calculus [ZH04], we do not require that the intervals Δ_1 and Δ_2 in the definition of chop are closed.

We assume point-wise lifting of the boolean operators on stream and interval predicates in the normal manner, e.g., if p_1 and p_2 are interval predicates, Δ is an interval and s is a stream, we have $(p_1 \wedge p_2).\Delta.s = (p_1.\Delta.s \wedge p_2.\Delta.s)$. Furthermore, when reasoning about properties of programs, we would like to state that whenever a property p_1 holds over any interval Δ and stream s, a property p_2 also holds over Δ and s. Hence, we define universal implication over intervals and streams as follows. Operators '$\equiv$' and '$\Leftarrow$' are similar.

$$p_1 \Rrightarrow p_2 \;\widehat{=}\; \forall \Delta : Interval \bullet p_1.\Delta \Rrightarrow p_2.\Delta \qquad p_1.\Delta \Rrightarrow p_2.\Delta \;\widehat{=}\; \forall s : Stream \bullet p_1.\Delta.s \Rightarrow p_2.\Delta.s$$

3 Idealised teleo-reactive programs

Teleo-reactive programs are introduced by way of an example.

Example 2 (Teleo-reactive controller for the mine-pump.) *The teleo-reactive program in Fig. 3 implements a controller for the system in Fig. 2. The main program (*mine_pump*) consists of a sequence of two guarded actions. For such a sequence, the first alternative that has a true guard is executed continuously while that guard remains true and no earlier guard in the sequence becomes true. As soon as that guard becomes false or an earlier guard becomes true, the execution switches to the first true guard. For example, if program* mine_pump *is executing* pump_water*, it continues to do so while the methane level, m, is continuously less than the critical level, C, i.e., $m < C$. In doing so,* pump_water *may switch back and forth between its two alternatives, depending on the water level. However, as soon as the methane level reaches its critical level, the execution of* pump_water *is immediately terminated and control is passed to the first alternative of* mine_pump*. That is, within a hierarchical teleo-reactive program, the guards of the top-level program take precedence over an activity within a lower-level program.*

For any interval over which $m \geq C$ continuously holds, the first branch of mine_pump *executes. Execution of Alarm||Stop_Pump consists of the parallel execution of primitive actions Alarm and Stop_Pump. Note that the actions execute in a truly concurrent manner (as opposed to via an interleaving semantics).*

Definition 1 For a state predicate c, set of variables O and interval predicates r and p, the syntax of a *teleo-reactive program* is given by P where

$$\begin{array}{rcl} P & ::= & O : [\![p]\!] \mid \mathsf{seq}.GP \mid P \overrightarrow{\parallel} P \mid \mathsf{rely}\, r \bullet P \mid \mathsf{init}\, c \bullet P \\ GP & ::= & c \rightarrow P \end{array}$$

$$\mathsf{mine\_pump} \mathrel{\widehat{=}} \left\langle \begin{array}{rcl} m \geq C & \rightarrow & \mathit{Alarm} \| \mathit{Stop\_Pump} \ , \\ \mathit{true} & \rightarrow & \mathsf{pump\_water} \end{array} \right\rangle$$

$$\mathsf{pump\_water} \mathrel{\widehat{=}} \left\langle \begin{array}{rcl} \mathit{w\_high} \vee (\neg \mathit{w\_low} \wedge \neg \mathit{stopped}) & \rightarrow & \mathit{Run\_Pump} \ , \\ \mathit{true} & \rightarrow & \mathit{Stop\_Pump} \end{array} \right\rangle$$

Figure 3: Teleo-reactive controller for the mine pump

Here $O{:}[\![p]\!]$ denotes a primitive specification with outputs O that behaves as described by the interval predicate p. A guarded program $c \rightarrow P$ consists of a state predicate c that guards a teleo-reactive program. A program may either be a primitive specification, a sequence of guarded programs, the parallel composition of two programs, a program with a rely condition r or a program with an initialisation c. We follow the convention of using Z for a teleo-reactive program and S for a sequence of guarded programs. Sequences are written within brackets '$\langle$' and '$\rangle$' and '$\frown$' is used for sequence concatenation. We let $\mathit{vars}.p$ and $\mathit{vars}.c$ denote the set of all variables that occur free in interval predicate p and state predicate c, respectively. The set of variables of Z is given by $\mathit{vars}.\mathsf{Z}$, where

$$\begin{array}{rcl@{\qquad}rcl} \mathit{vars}.(O{:}[\![p]\!]) & \mathrel{\widehat{=}} & \mathit{vars}.p \cup O & \mathit{vars}.(\mathsf{Z}_1 \overrightarrow{\|} \mathsf{Z}_2) & \mathrel{\widehat{=}} & \mathit{vars}.\mathsf{Z}_1 \cup \mathit{vars}.\mathsf{Z}_2 \\ \mathit{vars}.\langle\rangle & \mathrel{\widehat{=}} & \{\} & \mathit{vars}.(\mathsf{rely}\, r \bullet \mathsf{Z}) & \mathrel{\widehat{=}} & \mathit{vars}.r \cup \mathit{vars}.\mathsf{Z} \\ \mathit{vars}.(\langle c \rightarrow \mathsf{Z}\rangle \frown S) & \mathrel{\widehat{=}} & \mathit{vars}.c \cup \mathit{vars}.\mathsf{Z} \cup \mathit{vars}.S & \mathit{vars}.(\mathsf{init}\, c \bullet \mathsf{Z}) & \mathrel{\widehat{=}} & \mathit{vars}.c \cup \mathit{vars}.\mathsf{Z} \end{array}$$

The functions $\mathit{in}, \mathit{out}{:}\, P \rightarrow \mathbb{P}\, \mathit{Var}$ return the input and output variables of a teleo-reactive program, respectively. We define:

$$\begin{array}{rcl@{\qquad}rcl} \mathit{out}.(O{:}[\![p]\!]) & \mathrel{\widehat{=}} & O & \mathit{out}.(\mathsf{Z}_1 \overrightarrow{\|} \mathsf{Z}_2) & \mathrel{\widehat{=}} & \mathit{out}.\mathsf{Z}_1 \cup \mathit{out}.\mathsf{Z}_2 \\ \mathit{out}.\langle\rangle & \mathrel{\widehat{=}} & \{\} & \mathit{out}.(\mathsf{rely}\, r \bullet \mathsf{Z}) & \mathrel{\widehat{=}} & \mathit{out}.\mathsf{Z} \\ \mathit{out}.(\langle c \rightarrow \mathsf{Z}\rangle \frown S) & \mathrel{\widehat{=}} & \mathit{out}.\mathsf{Z} \cup \mathit{out}.S & \mathit{out}.(\mathsf{init}\, c \bullet \mathsf{Z}) & \mathrel{\widehat{=}} & \mathit{out}.\mathsf{Z} \end{array}$$

The set of input variables of Z is given by $\mathit{in}.\mathsf{Z} \mathrel{\widehat{=}} \mathit{vars}.\mathsf{Z} \backslash \mathit{out}.\mathsf{Z}$.

Because the semantics of teleo-reactive programs is truly concurrent (as opposed to an interleaving semantics), two concurrent programs $\mathsf{Z}_1 \overrightarrow{\|} \mathsf{Z}_2$ may not modify the same variable, i.e., we require that $\mathit{out}.\mathsf{Z}_1 \cap \mathit{out}.\mathsf{Z}_2 = \{\}$. However, in program $\mathsf{Z}_1 \overrightarrow{\|} \mathsf{Z}_2$ the outputs of Z_1 may be used as inputs to Z_2, thus parallel composition is not necessarily commutative. We define *simple parallelism* $\mathsf{Z}_1 \| \mathsf{Z}_2$ as a special case of parallel composition in which the inputs of Z_1 and Z_2 are disjoint with the outputs of Z_2 and Z_1, respectively. Unlike $\mathsf{Z}_1 \overrightarrow{\|} \mathsf{Z}_2$, the programs under simple parallelism commute.

Teleo-reactive programs are often only required to execute correctly under certain environment assumptions; these assumptions may be formalised within a rely condition.

Definition 2 We say interval predicate r is a *rely condition* of teleo-reactive program Z iff $\mathit{vars}.r \cap \mathit{out}.\mathsf{Z} = \{\}$.

Definition 3 A set of variables V is an *output context* of a teleo-reactive program Z iff $V \supseteq \mathit{out}.\mathsf{Z}$ and $V \cap \mathit{in}.\mathsf{Z} = \{\}$.

Suppose O is a set of variables, p is an interval predicate, Z, Z_1 and Z_2 are teleo-reactive programs, and S and $T \mathrel{\widehat{=}} \langle c \to \mathsf{Z} \rangle \frown S$ are sequences of guarded programs and r is a rely condition of Z. If V is an output context of each of the programs below, we define:

$$beh_V.(O{:}[\![p]\!]) \mathrel{\widehat{=}} p \wedge st.(V \backslash O) \tag{2}$$

$$beh_V.\langle\rangle \mathrel{\widehat{=}} true \tag{3}$$

$$beh_V.T \mathrel{\widehat{=}} ((\circledast c \wedge beh_V.\mathsf{Z}) : (\overleftarrow{\neg c} \wedge beh_V.T)) \vee ((\circledast \neg c \wedge beh_V.S) : (\overleftarrow{c} \wedge beh_V.T)) \tag{4}$$

$$beh_V.(\mathsf{Z}_1 \overrightarrow{\parallel} \mathsf{Z}_2) \mathrel{\widehat{=}} beh_{V \backslash out.\mathsf{Z}_2}.\mathsf{Z}_1 \wedge beh_{V \backslash out.\mathsf{Z}_1}.\mathsf{Z}_2 \tag{5}$$

$$beh_V.(\mathsf{rely}\, r \bullet \mathsf{Z}) \mathrel{\widehat{=}} r \Rightarrow beh_V.\mathsf{Z} \tag{6}$$

$$beh_V.(\mathsf{init}\, c \bullet \mathsf{Z}) \mathrel{\widehat{=}} prev.\overrightarrow{c} \wedge beh_V.\mathsf{Z} \tag{7}$$

Figure 4: *beh* function

The behaviour of a teleo-reactive program Z in an possibly wider output context V is given by $beh_V.\mathsf{Z}$ as defined in Fig. 4. The behaviour of a specification $O{:}[\![p]\!]$ with respect to the set V is given by (2), where in addition to behaving as specified by p, the variables of V that are not in O are guaranteed to be stable. An empty sequence of programs (3), is chaotic and allows any behaviour. The behaviour of a non-empty sequence of guarded programs, (4), is defined recursively. There are two disjuncts corresponding to either c or $\neg c$ holding initially on the interval. If c holds initially, either $\circledast c \wedge beh_V.\mathsf{Z}$ holds for the whole interval or the interval may be split into an initial interval in which $\circledast c \wedge beh_V.\mathsf{Z}$ holds, followed by an interval in which $\neg c$ holds initially and $beh_V.T$ holds (recursively) for the second interval. The other disjunct is similar. Note that each chopped interval must be a maximal interval over which either $\circledast c$ or $\circledast \neg c$ holds. The behaviour of the parallel composition between Z_1 and Z_2 is given by (5) and is defined to be the conjunction of the two behaviours with the outputs of each branch removed from the output context of the other. The behaviour of a program Z with rely condition r is denoted $\mathsf{rely}\, r \bullet \mathsf{Z}$ and its behaviour is given by (6). Thus, the program executes as defined by $beh_V.\mathsf{Z}$ provided the rely condition r holds. The initialisation of Z specifies a condition that holds prior to execution of Z and hence, the behaviour of $\mathsf{init}\, c \bullet \mathsf{Z}$ is given by (7).

Definition 4 For teleo-reactive programs Z and Z', we say Z *is refined by* Z' (denoted $\mathsf{Z} \sqsubseteq \mathsf{Z}'$) iff $beh_V.\mathsf{Z}' \Rrightarrow beh_V.\mathsf{Z}$ for any V that is an output context of both Z and Z'.

Lemma 1 *If O, O' are a sets of variables and p, p' are interval predicates, then*

$$(O' \subseteq O) \wedge (st.(O \backslash O') \wedge p' \Rrightarrow p) \Rightarrow (O{:}[\![p]\!] \sqsubseteq O'{:}[\![p']\!])\,.$$

Definition 5 For an interval predicate p, we say p *splits* iff $p.\Delta \Rrightarrow \forall \delta{:}\Pi.\Delta \bullet \forall i{:}\mathrm{dom}.\delta \bullet p.(\delta.i)$, and p *joins* iff $\exists \delta{:}\Pi.\Delta \bullet \forall i{:}\mathrm{dom}.\delta \bullet p.(\delta.i) \Rrightarrow p.\Delta$.

For example, interval predicate $\ell < 3$ splits but does not join, $\boxdot c$ and $\ell \geq 3$ join but do not split and $\circledast c$ both splits and joins.

The next theorem presents a method for decomposing refinements that is specific to guarded sequences of actions. The theorem requires that the abstract guarantee predicate joins and the rely condition splits.

Theorem 1 *Suppose* $T \mathrel{\widehat{=}} \langle c \to \mathsf{Z} \rangle \frown S$ *is a teleo-reactive program. If* r *is a rely condition of* T *that splits and* g *is an interval predicate that joins, then* $\mathsf{rely}\, r \bullet O{:}[\![g]\!] \sqsubseteq T$ *holds provided that both* $\mathsf{rely}\, r \bullet O{:}[\![\circledast c \Rightarrow g]\!] \sqsubseteq \mathsf{Z}$ *and* $\mathsf{rely}\, r \bullet O{:}[\![\circledast \neg c \Rightarrow g]\!] \sqsubseteq S$ *hold.*

Although it is tempting to replace r in $\mathsf{rely}\, r \bullet O{:}[\![\circledast c \Rightarrow g]\!] \sqsubseteq \mathsf{Z}$ by $r \wedge \circledast c$, such a replacement is incorrect because c often refers to output variables. For example, the output variable *stopped* appears in first the guard of pump_water in Fig. 3.

Example 3 (Mine-pump in Fig. 3 satisfies *Safety*.) *Note that the program in Fig. 3 is idealised, i.e., we assume that the guards are continuously evaluated and that the pump reacts instantaneously. By defining* $MO \mathrel{\widehat{=}} out.\mathsf{mine\_pump}$ *and*

$$Stop\_Pump \quad \mathrel{\widehat{=}} \quad \{stopped\}{:}[\![\circledast stopped]\!] \tag{8}$$

it is straightforward to prove that $MO{:}[\![Safety]\!] \sqsubseteq \mathsf{mine\_pump}$ *holds, which proves that the mine pump program in Fig. 3 implements the safety requirement Safety.*

$MO{:}[\![Safety]\!] \sqsubseteq \mathsf{mine\_pump}$
$\Leftarrow$ *Theorem 1, Safety joins*
$MO{:}[\![\circledast(m \geq C) \Rightarrow Safety]\!] \sqsubseteq (Alarm \| Stop\_Pump) \wedge$
$MO{:}[\![\circledast(m < C) \Rightarrow Safety]\!] \sqsubseteq \mathsf{pump\_water}$
$\Leftarrow$ *first conjunct, LHS: Lemma 1, definition of Safety, RHS: definition of* $\|$
second conjunct: definition of Safety
$(MO{:}[\![\circledast stopped]\!] \sqsubseteq Stop\_Pump) \wedge$
$(MO{:}[\![true]\!] \sqsubseteq \mathsf{pump\_water})$
$\Leftarrow$ *first conjunct: (8) and Lemma 1, second conjunct: Lemma 1 and* $out.\mathsf{pump\_water} \subseteq MO$
true

The ideal execution of a teleo-reactive program would *continuously* evaluate its guards all the time. Of course, continuous evaluation is not feasible and it has to be approximated by repeated sampling and evaluation. One of the main issues addressed in this paper is handling the imprecision introduced by such implementations by making use of Burns' Time Band framework [BB06, BH10].

4 Teleo-reactive programs with sampling

4.1 Sampling

A reactive controller uses (discrete) *sampling events* to determine the state of its (continuous) environment. Although a sampling event is viewed as instantaneous in the time band of the controller, sampling events actually take time. Thus, sampling events are prone to *timing precision errors* (where there is a range of possible sampled values due to imprecise timing of when

the sample is taken) and *sampling anomalies*, i.e., sampling two or more sensors causes a non-existent state to be returned. Sampling is also prone to *sensor errors* (where the sensors have inaccuracies in measuring the environment), but such errors are not the focus of this paper.

We use a logic that assumes each environment variable in an expression is read exactly once during a sampling event and that this value is used for each occurrence of the variable in the evaluation of the expression. However, different variables may be read at different times within an interval, which makes it possible for a sampling event to return a state that does not actually exist [BH10]. Given a set of states $SS \subseteq \Sigma_V$, we define

$$values.SS \mathrel{\widehat{=}} \lambda\, v{:}V \bullet \{\sigma{:}SS \bullet \sigma.v\} \qquad apparent.SS \mathrel{\widehat{=}} \{\sigma{:}\Sigma \mid (\forall v{:}V \bullet \sigma.v \in values.SS)\}$$

where the notation $\{\sigma{:}SS \bullet \sigma.v\}$ is equivalent to $\{x \mid (\exists \sigma{:}SS \bullet x = \sigma.v)\}$. That is, $values.SS$ returns a state that maps each variable $v \in Var$ to the set of values that v may have in SS and $apparent.SS$ generates the set of all states in which each variable is one of its values in a state in SS, but different states could be used for different variables.

To reason about sampling anomalies, we define a function *states*, that returns the set of all states that occur within a real-time interval of a given stream, and a function *av* that returns the set of apparent states. If Δ is an interval and s is a stream, we define:

$$states.\Delta.s \mathrel{\widehat{=}} \{t{:}\Delta \bullet s.t\} \qquad av.\Delta.s \mathrel{\widehat{=}} apparent.(states.\Delta.s)$$

Using functions *states* and *av*, we formalise state predicates that are *definitely* true (denoted $\circledast$) and *possibly* true (denoted $\odot$) over a given interval $\Delta \in Interval$ and stream $s \in Stream$ as follows:

$$(\circledast c).\Delta.s \mathrel{\widehat{=}} \forall \sigma{:}av.\Delta.s \bullet c.\sigma \qquad (\odot c).\Delta.s \mathrel{\widehat{=}} \exists \sigma{:}av.\Delta.s \bullet c.\sigma$$

If $(\circledast c).\Delta.s$ holds, then c holds for each apparent state in the interval Δ and if $\odot c$ holds then c holds in some apparent state. Note that $\circledast c \Rrightarrow \boxplus c$ and $\boxdot c \Rrightarrow \odot c$. There are several relationships between $\circledast$ and $\odot$; we refer the interested reader to [BH10]. In this paper, we find the following lemma to be useful.

Lemma 2 *For a state predicate c and variable v, $st.(vars.c \setminus \{v\}) \Rrightarrow (\circledast c = \boxplus c) \wedge (\odot c = \boxdot c)$.*

4.2 Time bands

The set of all time bands is given by the primitive type *TimeBand*, which defines a unit of time, e.g., seconds, days, years. The precision of a time band is given by $\rho{:}TimeBand \to \mathbb{R}^{>0}$. We define a time band $b_1 + b_2$ to be a band such that $\rho.(b_1 + b_2) \mathrel{\widehat{=}} \rho.b_1 + \rho.b_2$. Given that $R[S]$ denotes the *relational image* of the set S through relation R, for a real-valued variable v, interval Δ and stream s, we define

$$diff.v.\Delta.s \quad \mathrel{\widehat{=}} \quad \mathsf{let}\ vs = (states.\Delta.s)[\{v\}]\ \mathsf{in}\ lub.vs - glb.vs$$

Thus, *diff* returns the difference between the maximum and minimum values of the given variable in the given interval and stream. To this end, we define the *accuracy* of a variable v in time band b using $accuracy.v.b$, which limits the maximum change to the variable within events of time band b. For any variable v and time band b, we implicitly assume the following rely condition:

$$\ell.\Delta \le \rho.b \quad \Rrightarrow \quad diff.v.\Delta \le accuracy.v.b \tag{9}$$

Lemma 3 *For a variable v, constant k, and time band b,*

$$\ell \leq \rho.b \wedge \boxdot(v < k - accuracy.v.b) \Rrightarrow \circledast(v < k)\,.$$

Proof. The proof is trivial by the assumption (9) on *accuracy*. □

4.3 Extended syntax and semantics

For a guarded sequence of actions T and a time band b, we use $T \dagger b$ to denote that the guards of T are repeatedly evaluated within the precision of time band b. We use functions $grd.(c \to \mathsf{Z}) \mathrel{\widehat{=}} c$ and $body.(c \to \mathsf{Z}) \mathrel{\widehat{=}} \mathsf{Z}$ to return the guard and body of the guarded program $c \to \mathsf{Z}$, respectively.

Definition 6 For any $i \in \mathrm{dom}.T$, the *effective guard* of $T.i$ (denoted $\mathit{eff}.(T.i)$) is given by $grd.(T.i) \wedge \bigwedge_{j:0..i-1} \neg grd.(T.j)$.

That is, the effective guard of $T.i$ is the actual guard of $T.i$ in conjunction with the negations of all guards that precede i in T.

Execution of program $T \dagger b$ consists of evaluation of all guards within intervals of size $\rho.b$ or less. Then, branch $T.j$ is executed over interval Δ if there is a partition δ of Δ such that the effective guard of $T.j$ possibly holds in each $\delta.i$, and furthermore, the body of $T.j$ executes as defined by the behaviour function over Δ. For a state predicate c and a time band b, we define shorthand:

$$(\!|c|\!)_b.\Delta \quad \mathrel{\widehat{=}} \quad \exists \delta{:}\Pi.\Delta \bullet \forall i{:}\mathrm{dom}.\delta \bullet (\ell \leq \rho.b \wedge \odot c).(\delta.i) \tag{10}$$

Lemma 4 *If c is a state predicate, b is a time band, p is an interval predicate that joins, s is a stream and* $\forall \Omega{:}\mathit{Interval} \bullet ((\ell \leq \rho.b) \wedge \odot c \Rightarrow p).\Omega.s$ *holds, then* $\forall \Delta{:}\mathit{Interval} \bullet ((\!|c|\!)_b \Rightarrow p).\Delta.s$.

Proof. For any interval Δ, we have the following calculation:

$$\begin{array}{ll}
 & (\!|c|\!)_b \Rightarrow p).\Delta.s \\
= & \quad \text{point-wise lifting, definition of } (\!|c|\!)_b \\
 & (\exists \delta{:}\Pi.\Delta \bullet \forall i{:}\mathrm{dom}.\delta \bullet ((\ell \leq \rho.b) \wedge \odot c).(\delta.i).s) \Rightarrow p.\Delta.s \\
= & \quad \text{logic} \\
 & \forall\, \delta{:}\Pi.\Delta \bullet \forall i{:}\mathrm{dom}.\delta \bullet ((\ell \leq \rho.b) \wedge \odot c).(\delta.i).s \Rightarrow p.\Delta.s \\
\Leftarrow & \quad p \text{ joins} \\
 & \forall \delta{:}\Pi.\Delta \bullet \forall i{:}\mathrm{dom}.\delta \bullet ((\ell \leq \rho.b) \wedge \odot c).(\delta.i).s \Rightarrow p.(\delta.i).s \\
\Leftarrow & \quad \text{logic} \\
 & \forall \Omega{:}\mathit{Interval} \bullet ((\ell \leq \rho.b) \wedge \odot c \Rightarrow p).\Omega.s
\end{array}$$

□

We prove the following lemma that relates guard evaluation over the precision of a time band to the approximated value of the variables.

Lemma 5 *For a variable v, constant k and time band b,* $(\!|v < k - accuracy.v.b|\!)_b \Rrightarrow \circledast(v < k)$.

$$\left\langle \begin{array}{rcl} \boxed{m \geq D} & \rightarrow & \mathit{Alarm} \| \mathit{Stop\_Pump}, \\ \mathit{true} & \rightarrow & \mathsf{pump\_water} \end{array} \right\rangle \boxed{\dagger M}$$

Figure 5: Top-level program with methane time band

Proof. For any interval Δ and stream s, we have

$$\begin{array}{ll} & (\!|v < k - \mathit{accuracy}.v.b|\!)_b \Rightarrow \circledast(v < k)).\Delta.s \\ \Leftarrow & \quad \text{Lemma 4, } \circledast c \text{ joins} \\ & \forall \Omega : \mathit{Interval} \bullet ((\ell \leq \rho.b) \wedge \odot(v < k - \mathit{accuracy}.v.b) \Rightarrow \circledast(v < k)).\Omega.s \\ \Leftarrow & \quad \text{Lemma 3 and Lemma 2} \\ & \mathit{true} \end{array}$$

□

If $j \in \mathrm{dom}.T$, we define the execution of guarded program $T.j$ within time band b as follows:

$$\mathit{exec}_V.(T.j).b.\Delta \mathrel{\widehat{=}} (\!|\mathit{eff}.(T.j)|\!)_b.\Delta \wedge \mathit{beh}_V.(\mathit{body}.(T.j)).\Delta$$

Thus, there must exist a partition of Δ, δ, such that for each index $i \in \mathrm{dom}.\delta$, the length of the interval $\delta.i$ is at most the precision, $\rho.b$, of time band b and the effective guard of $T.j$ possibly holds within $\delta.i$. Furthermore, the behaviour of $\mathit{body}.(T.j)$ holds within Δ. We say a teleo-reactive program T is *well-formed* iff $\mathit{last}.T = \mathit{true} \rightarrow \mathsf{Z}$ for some program Z.

Definition 7 For a well-formed teleo-reactive program $T \dagger b$, we define

$$\begin{array}{rcl} \mathit{beh}_V.(T \dagger b).\Delta & \mathrel{\widehat{=}} & \exists \delta : \Pi.\Delta \bullet \exists \mathit{act} : (\mathrm{dom}.\delta \rightarrow \mathrm{dom}.T) \bullet \forall i : \mathrm{dom}.\mathit{act} \bullet \\ & & \quad ((\mathit{exec}_V.(T.(\mathit{act}.i)).b).(\delta.i) \wedge ((i > 0) \Rightarrow \mathit{act}.i \neq \mathit{act}.(i-1)) \end{array}$$

Thus, we say $\mathit{beh}_V.(T \dagger b).\Delta$ holds iff there is a partition, δ, of Δ and a mapping, act, from the domain of δ to the domain of T such that for every $i \in \mathrm{dom}.\mathit{act}$, execution of $T.(\mathit{act}.i)$ holds in $\delta.i$ and furthermore, consecutive intervals of δ are mapped to different elements of T. Note that $\mathrm{dom}.\mathit{act} = \mathrm{dom}.\delta$. Definition 7 allows both Zeno and non-Zeno executions of T, however, we can only implement non-Zeno behaviour. This is not problematic because the definition does not require Zeno behaviour, i.e., it allows non-Zeno behaviour.

A version of the top-level mine_pump program from Fig. 3 that does not make idealised assumptions is given in Fig. 5, where the changes are identified within the boxes. In particular, we have introduced a time band $M \in \mathit{TimeBand}$ which represents the time band of the methane. Introduction of M within the program defines the minimum rate at which the methane is sampled. Because the program approximates the value of m using a sampling event, the guard $m \geq C$ has been replaced by $m \geq D$. We calculate the relationship between C and D that is necessary for proving *Safety* in Section 5.1.

5 Approximating specifications with time bands

We have developed a method of refining timed specifications and a formal semantics for both idealised and time-banded teleo-reactive programs. We are able to prove that the idealised teleo-

reactive programs implement the (ideal) specifications. However, as with robust automata, proving that the requirements that are specified at the absolute level of precision are implemented by the time-banded teleo-reactive programs is, in general, difficult.

5.1 Incorporating the methane time band

We have a decomposition theorem for time-banded teleo-reactive programs that is similar to Theorem 1 for idealised programs.

Theorem 2 *Suppose* $T \mathrel{\widehat{=}} (\langle c \to \mathsf{Z}\rangle \frown S) \dagger b$ *is a time-banded teleo-reactive program and* O *is a set of variables. If* r *is a rely condition of* T *that splits and* g *is an interval predicate that joins, then* $\mathsf{rely}\, r \bullet O{:}[\![g]\!] \sqsubseteq T$ *holds provided that both* $\mathsf{rely}\, r \bullet O{:}[\![(\!|c|\!)_b \Rightarrow g]\!] \sqsubseteq \mathsf{Z}$ *and* $\mathsf{rely}\, r \bullet O{:}[\![(\!|\neg c|\!)_b \Rightarrow g]\!] \sqsubseteq S$ *hold.*

Example 4 (Program in Fig. 5 satisfies *Safety*.) *Applying Theorem 2 to prove* $MO{:}[\![Safety]\!]$ *gives us the following proof obligations.*

$$MO{:}[\![(\!|m \geq D|\!)_M \Rightarrow Safety]\!] \sqsubseteq Alarm \| Stop\_Pump \tag{11}$$

$$MO{:}[\![(\!|m < D|\!)_M \Rightarrow Safety]\!] \sqsubseteq \mathsf{pump\_water} \tag{12}$$

We let $D \leq C - accuracy.m.M$ *and have the following calculations:*

$$\begin{array}{ll} & (\!|m \geq D|\!)_M \Rightarrow Safety \\ \Leftarrow & \quad logic \\ & \circledast stopped \end{array}$$

$$\begin{array}{ll} & (\!|m < D|\!)_M \Rightarrow Safety \\ \Leftarrow & \quad Lemma\ 5,\ D \leq C - accuracy.m.M \\ & \circledast(m < C) \Rightarrow Safety \\ \equiv & \quad antecedent\ of\ Safety\ is\ false \\ & true \end{array}$$

Hence, we have:

$$\begin{array}{ll} & (11) \\ \Leftarrow & \quad calculation\ above,\ definition\ of\ \| \\ & MO{:}[\![\circledast stopped]\!] \sqsubseteq Stop\_Pump \\ \Leftarrow & \quad (8)\ definition\ of\ Stop\_Pump \\ & true \end{array}$$

$$\begin{array}{ll} & (12) \\ \Leftarrow & \quad calculation\ above \\ & MO{:}[\![true]\!] \sqsubseteq \mathsf{pump\_water} \\ \Leftarrow & \quad out.\mathsf{pump\_water} \subseteq MO \\ & true \end{array}$$

We have considered the time taken to sample the methane into account and established a relationship between a threshold and true value of the methane to prove *Safety*. However, the program in Fig. 5 is not realistic because it assumes that the the pump is stopped instantaneously. In fact, the specification requires the pump to be stopped from the beginning of the interval over which the methane is sampled to be high, even before the first high sample is taken. In the next section, we describe how the program may be modified so that the safety condition holds for a pump that is guaranteed to stop in its own time band.

5.2 Incorporating the pump time band

We consider the program in Fig. 6, where the program begins executing after initialisation *stopped*, the guard for stopping the pump has been modified to $m \geq E$ and *Stop_Pump_P* is

used to stop the pump. Given that $P \in \mathit{TimeBand}$ is the time band of the pump, we define:

$$\mathit{Stop\_Pump\_P} \;\widehat{=}\; \{\mathit{stopped}\}\colon [\![\mathit{inv}.\mathit{stopped} \wedge ((\ell \leq \rho.P) : (\circledast \mathit{stopped}))]\!] \tag{13}$$

i.e., execution of *Stop_Pump_P* over any interval of length $\rho.P$ or greater (i.e., the precision of the pump) is guaranteed to stop the pump. Note that this specification does not limit the deceleration of the pump, i.e., there may be several possible implementations of this specification at higher precision time bands. Each implementation must only guarantee that $\overrightarrow{\mathit{stopped}}$ holds within an interval of the precision $\rho.P$. Note that the bands within the program in Fig. 6 serve slightly different purposes; band P restricts the precision of the pump events, while M restricts the rate at which the methane is sampled in the guard.

Lemma 6 *For a continuous variable, m, in time band M, and constant E,*

$$\mathit{prev}.(\!|m < E|\!)_M \Rrightarrow \overleftarrow{m \leq E + \mathit{accuracy}.m.M}$$

Proof. The proof makes use of the accuracy of m within time band M.

$$\begin{array}{ll} & \mathit{prev}.(\!|m < E|\!)_M \\ \Rrightarrow & \quad \text{Lemma 3} \\ & \mathit{prev}.(\circledast(m < E + \mathit{accuracy}.m.M)) \\ \Rrightarrow & \quad \text{continuity of } m \\ & \mathit{prev}.(\overrightarrow{m \leq E + \mathit{accuracy}.m.M}) \\ \Rrightarrow & \quad \text{continuity of } m \\ & \overleftarrow{m \leq E + \mathit{accuracy}.m.M} \end{array}$$

□

Lemma 7 *For a continuous variable m in time band P, and constant K,*

$$\overleftarrow{m \leq K} \wedge \ell < \rho.P \Rrightarrow \circledast(m < K + \mathit{accuracy}.m.P)\,.$$

Proof. Because m is continuous and no greater than K at the left limit of an interval that is of length bounded by $\rho.P$, m cannot increase by more than its accuracy in band P. □

We present a third decomposition theorem for proving refinements where the given program executes under some initialisation.

Theorem 3 *Suppose $T \mathrel{\widehat{=}} (\langle c \to \mathsf{Z}\rangle \frown S) \dagger b$ is a time-banded teleo-reactive program, O is a set of variables, r and g are interval predicates and d is an state predicate. If r is a rely condition of T that splits and g joins, then* $\mathsf{rely}\, r \bullet O\colon [\![g]\!] \sqsubseteq \mathsf{init}\, d \bullet T$ *holds provided:*

$$\mathsf{rely}\, r \bullet O\colon \left[\!\!\left[(\!|c|\!)_b \wedge \mathit{prev}.((\!|\neg c|\!)_b \vee \overrightarrow{d}) \Rightarrow g \right]\!\!\right] \;\sqsubseteq\; \mathsf{Z} \tag{14}$$

$$\mathsf{rely}\, r \bullet O\colon \left[\!\!\left[(\!|\neg c|\!)_b \wedge \mathit{prev}.((\!|c|\!)_b \vee \overrightarrow{d}) \Rightarrow g \right]\!\!\right] \;\sqsubseteq\; S \tag{15}$$

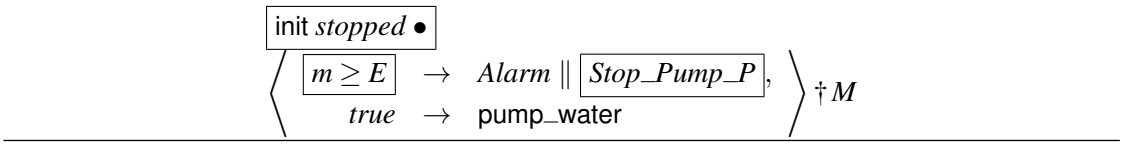


Figure 6: Top-level program with methane and pump time bands

Example 5 (Program in Fig. 6 satisfies *Safety*.) *Applying Theorem 3 to prove* $MO{:}[\![Safety]\!] \sqsubseteq$ init *stopped* • mine_pump *gives us:*

$$MO{:}\left[\!\!\left[(\!|m \geq E|\!)_M \wedge prev.((\!|m < E|\!)_M \vee \overrightarrow{stopped}) \Rightarrow Safety \right]\!\!\right] \quad \sqsubseteq \quad Alarm \| Stop\_Pump\_P \quad (16)$$

$$MO{:}\left[\!\!\left[(\!|m < E|\!)_M \wedge prev.((\!|m \geq E|\!)_M \vee \overrightarrow{stopped}) \Rightarrow Safety \right]\!\!\right] \quad \sqsubseteq \quad \text{pump\_water} \quad (17)$$

By assuming $E \leq C - accuracy.m.M$*, using Lemma 5, the proof of (17) is straightforward because the left-hand-side of* $\sqsubseteq$ *in (17) reduces to* $MO{:}[\![true]\!]$*. For (16), we strengthen the assumption to* $E < C - accuracy.m.(M+P)$ *and perform the following calculation:*

$$\begin{array}{ll}
 & (\!|m \geq E|\!)_M \wedge prev.((\!|m < E|\!)_M \vee \overrightarrow{stopped}) \Rightarrow Safety \\
\Leftarrow & \textit{weaken antecedent, definition of } prev \\
 & (prev.(\!|m < E|\!)_M \Rightarrow Safety) \wedge (prev.\overrightarrow{stopped} \Rightarrow Safety) \\
\Leftarrow & \textit{strengthen consequents} \\
 & (prev.(\!|m < E|\!)_M \Rightarrow ((\ell \leq \rho.P) : (\circledast stopped)) \wedge Safety) \wedge (prev.\overrightarrow{stopped} \Rightarrow \circledast stopped) \\
\Leftarrow & \textit{Safety joins, definition of } inv \\
 & (prev.(\!|m < E|\!)_M \Rightarrow ((\ell \leq \rho.P \wedge Safety) : (\circledast stopped \wedge Safety))) \wedge inv.stopped \\
\Leftarrow & \textit{Lemma 6; definition of } Safety \\
 & (\overleftarrow{m \leq E + accuracy.m.M} \Rightarrow ((\ell \leq \rho.P \wedge \boxplus(m < C)) : \circledast stopped)) \wedge inv.stopped \\
\Leftarrow & \textit{Lemma 7 and assumption } E < C - accuracy.m.(M+P) \\
 & (\overleftarrow{m \leq E + accuracy.m.M} \Rightarrow ((\ell \leq \rho.P) : \circledast stopped)) \wedge inv.stopped \\
\Leftarrow & \textit{logic} \\
 & ((\ell \leq \rho.P) : \circledast stopped) \wedge inv.stopped
\end{array}$$

Using Lemma 1 and the definition of $\|$, it is straightforward to verify that the specification $MO{:}[\![(\ell \leq \rho.P : \circledast stopped) \wedge inv.stopped]\!]$ is refined by $Alarm \| Stop\_Pump\_P$, which completes the proof of *Safety*.

6 Conclusions and related work

We have presented a model in which specifications defined over an absolute level of precision may be approximated to the time bands over which the input variables of the specification are sampled. This approximating process loosens the specification and the values of variables interpreted in a time band, say b, are taken to be the values of the variable within the precision of b. We have also described how the behaviour of output variables may be formalised over a

time band and presented methods for specifying actions using time band predicates. This allows one to prove properties of the different the time bands without limiting lower-level behaviour. Implementation of specifications is defined with respect to a refinement relation on interval predicates, which ensures that each real-time behaviour of the implementation is an behaviour of the abstract specification.

In the context of timed-automata, researchers have developed *robust timed automata* [GHJ97], which weakens the specification of the original automata to accept more traces. Implementation of robust automata is known to be problematic because the original specification is weakened. Algorithms for developing robust automata from idealised automata so that safety properties are preserved are currently impractical and preservation of general temporal logic properties is currently not possible [WDMR08].

Alur et al have considered *perturbed timed automata*, which focusses on clock errors (or perturbations) [ALM05]. However in their own words:

> Thus, checking equivalence of timed circuits composed of components with imperfect clocks, in terms of timed languages over inputs and outputs, remains an interesting open problem. [ALM05, pg84]

In the context of refinement, Boiten and Derrick have proposed "approximating refinements" [BD05], where metrics are used to develop implementations for situations in which refinement is not possible. The argument is that realistic implementations are limited by physical resources such as memory, which place restrictions on the ideal specifications. Our work differs from this in that we are concerned with approximating idealised timing specifications.

Henzinger presents a theory of timed refinement where sampling events are executed by a separate process [HQR99]. Moszkowski presents a method of abstracting between different time granularities for interval temporal logic, however the model uses a discrete framework of time [Mos95] as opposed to our continuous model. The formalisms above do not consider the possibility of sampling anomalies. Broy [Bro01] presents a refinement framework that formalises the relationships between different models of time. This includes abstraction techniques from dense to discrete models of time using sampling. However, the sampling theory is not well developed and the techniques only consider discretisation of dense streams.

Bibliography

[ALM05] R. Alur, S. La Torre, P. Madhusudan. Perturbed Timed Automata. In Morari and Thiele (eds.), *Hybrid Systems: Computation and Control*. LNCS 3414, pp. 70–85. Springer Berlin / Heidelberg, 2005.

[BB06] A. Burns, G. Baxter. Time bands in systems structure. In *Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective*. Pp. 74–88. Springer-Verlag, 2006.

[BD05] E. A. Boiten, J. Derrick. Formal Program Development with Approximations. In Treharne et al. (eds.), *ZB*. LNCS 3455, pp. 374–392. Springer, 2005.

[BH10] A. Burns, I. J. Hayes. A Timeband Framework for Modelling Real-Time Systems. *Real-Time Systems* 45(1):106–142, 2010.

[BL91] A. Burns, A. M. Lister. A Framework for Building Dependable Systems. *Comput. J.* 34(2):173–181, 1991.

[Bro01] M. Broy. Refinement of time. *Theor. Comput. Sci.* 253(1):3–26, 2001.

[GHJ97] V. Gupta, T. A. Henzinger, R. Jagadeesan. Robust Timed Automata. In Maler (ed.), *HART*. LNCS 1201, pp. 331–345. Springer, 1997.

[GM01] A. Gargantini, A. Morzenti. Automated deductive requirements analysis of critical systems. *ACM Trans. Softw. Eng. Methodol.* 10:255–307, July 2001.

[Hay08] I. J. Hayes. Towards reasoning about teleo-reactive programs for robust real-time systems. In *SERENE '08*. Pp. 87–94. ACM, New York, NY, USA, 2008.

[HQR99] T. A. Henzinger, S. Qadeer, S. K. Rajamani. Assume-Guarantee Refinement Between Different Time Scales. In *CAV '99*. Pp. 208–221. Springer-Verlag, 1999.

[Mos95] B. C. Moszkowski. Compositional reasoning about projected and infinite time. In *ICECCS*. Pp. 238–245. IEEE Computer Society, 1995.

[Nil01] N. J. Nilsson. Teleo-reactive programs and the triple-tower architecture. *Electronic Transactions on Artificial Intelligence* 5:99–110, 2001.

[WDMR08] M. Wulf, L. Doyen, N. Markey, J.-F. Raskin. Robust safety of timed automata. *Form. Methods Syst. Des.* 33:45–84, December 2008.

[ZH04] C. Zhou, M. R. Hansen. *Duration Calculus: A Formal Approach to Real-Time Systems*. EATCS: Monographs in Theoretical Computer Science. Springer, 2004.

Is my Formal Method Tool Ready for the Industry?

Christophe Ponsard, Jean-Christophe Deprez, Renaud De Landtsheer

{cp,jcd,rdl}@cetic.be, CETIC Research Centre, Belgium

Abstract: Using formal methods requires adequate tool support. Many formal tools emerge from academic prototypes and evolve towards Industry. This short paper summaries our on-going work under the auspices of DEPLOY project on providing answers to many practical questions frequently raised by Industry users regarding formal method tools notably performance, scalability, integration, user-friendliness, qualification/certification with respect to Industry standards.

Keywords: Formal Methods, Industry, Tooling, Certification

1 Introduction

The use of formal methods (FM) in the 21th century, especially in an industrial setting, cannot be considered without adequate tool support [BFLW09]. Quality aspects of FM tools are therefore a major factor influencing adoption. Potential industrial adopters frequently raise questions on this topic and it is not easy for them to find an answer given FM tools are a niche market requiring highly specialized skills.

This short paper presents some evidence taking the form of Frequently Asked Questions (FAQ) about tooling gathered during FM deployment experiments. Real experiments were carried out in the industry by the DEPLOY project (www.deploy-project.eu) and were compared with experience reported by others. In many cases, a comparative discussion is made between Open vs. Closed Source tools. A concise set of answers is presented here. More elaborated answers on this topic and other formal related themes can be found at: www.fm4industry.cetic.be.

2 Some FAQ about Formal Tools

Is there guarantee of long term Tool availability and support ? Industry projects may last tens of years from the development to the decommissioning of a system. It is therefore crucial for Industry to ensure proper support throughout the complete project lifetime including its retirement. Tools can be distributed under Open Source or Proprietary Licenses. Each model comes with its own risk to disappear (bankruptcy for proprietary code vs. community disappearance for Open Source). Given the niche market, securing the support is nontrivial task (e.g. escrow for proprietary code, direct community involvement or support for Open Source).

Is the Tool reliable? Closed source reliability is a mater of trust that can be provided by a certification scheme for example. Concerns have been raised about Open Source tools capability to achieve higher reliability [Cra99]. However the large number of industrial strength tools available nowadays tends to prove the contrary: e.g. PVS, nuSMV, and several others. Some reasons are related to the potential of massive peer review and at the design level, better defined

interfaces and careful designs required for a distributed development. Furthermore, extensive test suites are often available for such Open Source tools.

Is the Tool scalable? The ability to scale up depends on different factors. Tool-induced limitations may be due to the underlying formal technology, implementation problems (e.g. some bottleneck in a processing chain) or simply usability (e.g. limitation to manage large pieces of models). To assess scalability, references, feedback and reviews provide initial information that is useful to directly rule out inadequate tools for Industry. A second step is to challenge the tool on realistic case study in various Industry sectors as the way models are built can also impact the ability to scale up. Open Source tools might have higher risk of not scaling up, especially if they are still at the R&D stage. However, there are also highly scalable Open Source tools in the area of FM (e.g. SPIN and nuSMV model-checkers, ACL2 and Isabelle theorem provers).

Is the Tool usable? It is important that tools facilitate various tasks when building or modifying a model, carrying out validation and verification activities, working in team, etc. Commercial tools generally have better usability because special attention is devoted to this aspect whereas Open Source tools tend to focus more on the core functionality and efficiency, with sometimes only a command line interface.

Does the Tool integrate well in Industry tool chains? The ability to integrate into existing industrial tool chains is fundamental. This requires the existence of well-documented data format, availability of APIs/binaries on specific OS's/integration with popular tool platforms. This is an area where Open Source usually outperforms proprietary tools. Furthermore, Open Source often adopt open standard data format. On the other hand, heightened competition frequently pushes proprietary tools to keep internal data format hidden.

What is the impact of my Tool w.r.t. Certification? Using a formal tool in the design flow (i.e. at design time) might have an impact on the certification process, especially if the tool is generating production artifacts such as source code for systems requiring higher integrity levels. Evidence of correctness of the output produced by these tools has to be provided by various means: redundant implementation, extensive test coverage, and specific verification activities. As a supporting success story, the ProB tool used by Siemens and developed by the University of Düsseldorf is undergoing a qualification for the railways EN-50128 standard.

Acknowledgements: This work is funded by the European Commission under the EU project DEPLOY (project reference number 214158).

Bibliography

[BFLW09] J. C. Bicarregui, J. S. Fitzgerald, P. G. Larsen, J. C. Woodcock. Industrial Practice in Formal Methods: A Review. In *Proceedings of the 2nd World Congress on Formal Methods*. FM '09, pp. 810–813. Springer-Verlag, Berlin, Heidelberg, 2009.

[Cra99] D. Craigen. Formal Methods Adoption: What's Working, What's Not! In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*. Pp. 77–91. Springer-Verlag, London, UK, 1999.

Verifying FreeRTOS: from requirements to binary code

Jan Tobias Mühlberg[1] **and Leo Freitas**[2]

[1]Dept. of Computer Science, K.U.Leuven, Belgium, jantobias.muehlberg@cs.kuleuven.be
[2]School of Computing Science, Newcastle University, UK, leo.freitas@newcastle.ac.uk

Abstract: This paper reports on ongoing work towards verifying the FreeRTOS real-time operating system kernel. We discuss tools and techniques currently employed and outline future directions of research.

Keywords: Software verification, FreeRTOS, Z, SOCA, VeriFast

1 Introduction

FreeRTOS [Bar11] is an operating system (OS) kernel for embedded real-time applications. It has recently been proposed as case study in the context of the grand challenge on software verification [JOW06]. For this purpose, FreeRTOS is particularly interesting because it is open-source, reasonably small in size, yet relatively complex with respect to the functionality it provides. It features memory management, I/O-device control, tasks management and scheduling, communication and synchronisation directives, and real-time event handling. FreeRTOS has been ported to a range of computing platforms and compilers. The kernel comprises of roughly 3,000 lines of C code with a small fraction of assembly code.

The core of FreeRTOS is its scheduler. It implements different policies for scheduling tasks that share a single processing unit. Being a real-time OS, these policies are not aiming at fair scheduling, but at providing timely responses to events. The scheduler has been subject to recent verification effort [DGM09], where a specification of the task management is proposed.

In this paper we report on ongoing work on the verification of FreeRTOS for structural properties (*e.g.* pointer safety and arithmetic overflow) and liveness properties, ultimately aiming at functional correctness. This includes the reconstruction of a formal specification of FreeRTOS in *Z* [Lin10], bounded model checking of the actual implementation of FreeRTOS with the *SOCA-Verifier* [ML10], as well as annotating the source code with assertions in separation logic to apply the *VeriFast* [JSP10] software verifier.

2 Queues in Z

In [Lin10], an initial formal verification of FreeRTOS scheduler was performed. It included modelling and verification of key data structures using a theorem prover. We took this work forward, and abstracted its main components into simpler structures that were amenable for source-code and binary level verification. This involved refactoring and simplification of the Z model available, as well as proof for feasibility (*i.e.,* preconditions) and well-formedness. This was in preparation for using the SOCA verifier. Our aim is to establish abstract properties of the code within the theorem prover, and then use those as annotations for source-level verification.

The key data structure in the FreeRTOS scheduler is a *Queue* used for scheduling tasks. We modelled this queue (*e.g.,* 10 pages of Z) and proved properties of interest (*e.g.,* 7 theorems and

12 lemmas). For instance, sending items (*e.g.,* inter-task communication among queued tasks) requires knowledge of the task within the scheduler's queue, room for increasing messages from that task, *etc*. Similarly, for receiving items, queue must be known to the scheduler, the task to the queue, and the scheduled messages for that task within the queue must not be empty. These and other properties are specified as predicates proved against the scheduler's *Queue* model.

3 Applying SOCA & VeriFast

With the intention to verify pre- and post-conditions first specified in [DGM09] and improved in [Lin10] at the implementation level, the *SOCA-Verifier* [ML10] was applied to FreeRTOS binary. SOCA is particularly suited for analysing low-level OS components. It implements bounded symbolic execution of compiled and liked program executables. The tool features built-in checks for pointer safety and allows further properties to be specified. To analyse FreeRTOS, we extended the SOCA-Verifier so that programs compiled for ARM processors can be analysed. Using SOCA, we were able to analyse the scheduler functions modelled in [DGM09] with a statement coverage above 85%. The tool did not report any bugs related to pointer safety. Yet, we could not verify further properties with respect to the shape of data structures because these are hard to specify at the object-code level.

VeriFast [JSP10] performs rely-guarantee reasoning for programs written in C. It takes as input the sources of the program, which have to be annotated with method contracts in terms of separation logic and supports specifying and verifying deep data structure properties, such as the safe construction and usage of linked list. Currently, the scheduler and the implementations of data structures like lists and queues in FreeRTOS are being simplified, annotated and verified. In this process, we discover shortcomings in VeriFast that are being fixed to enable the verification of the unmodified source code.

4 Future Work

Beyond the completion of our ongoing research, we will investigate in joining our work on high-level specifications with that on applying implementation-level verification tools through refinement so that code and annotations can be generated from the specifications. A challenge for this will be in the development of a suitable model of pointers. Furthermore, we will research in techniques for specifying and verifying timing properties of FreeRTOS.

Acknowledgements: We thank Jim Woodcock for motivating FreeRTOS to us, and Piyawat Lamsam and Yuhui Lin for their initial input to our work.

References

[Bar11] R. Barry. FreeRTOS. 2011. http://www.freertos.org/.

[DGM09] D. Deharbe, S. Galvao, A. M. Moreira. Formalizing FreeRTOS: First Steps. In *SBMF '09*. LNCS 5902, pp. 101–117. Springer, 2009.

[JOW06] C. Jones, P. O'Hearn, J. Woodcock. Verified Software: A Grand Challenge. *Computer* 39(4):93–95, 2006.

[JSP10] B. Jacobs, J. Smans, F. Piessens. A quick tour of the VeriFast program verifier. In *APLAS 2010*. LNCS 6461, pp. 304–311. Springer, 2010.

[Lin10] Y. Lin. Formal analysis of FreeRTOS. Master's thesis, University of York, 2010.

[ML10] J. T. Mühlberg, G. Lüttgen. Symbolic Object Code Analysis. In *SPIN '10*. LNCS 6349, pp. 4–21. Springer, 2010.

A Simulator for Timed CSP

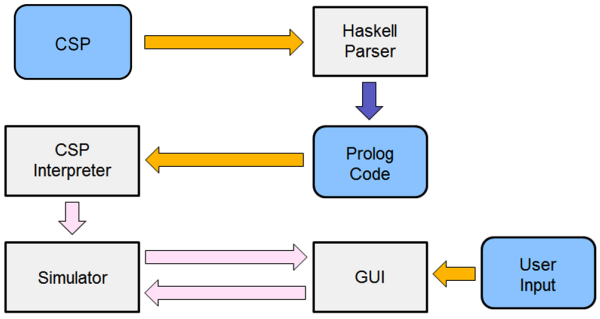
Marc Dragon, Andy Gimblett, Markus Roggenbach[†]

Swansea University, Wales, UK

Abstract: We present a simulator for Timed CSP based on the tool ProB.

Keywords: Timed CSP, ProB, Process Algebra, Real-Time, Simulation.

Time is an integral aspect of computer systems. It is essential for modelling a system's performance, but may also affect its safety or security. Timed CSP [Sch00] conservatively extends the process algebra CSP with timed primitives, where real numbers ≥ 0 model how time passes with reference to a single, conceptually global, clock. While there have been approaches for model checking Timed CSP ([Sch00, DHSZ06]), to the best of our knowledge we are the first to present a simulator for Timed CSP. Here, we restrict time to rational values only. Theoretically, this limits the expressibility of the language. Practically, this limitation turns out to be negligible (for instance all examples of Schneider's book [Sch00] can be dealt with in our simulator). The simulator is the outcome of an undergraduate project at Swansea University [Dra11].



Our Timed CSP simulator extends the open source tool ProB [Leu]. ProB's CSP simulator works as follows: The CSP specification is analyzed by a parser (written in Haskell) and translated to a representation in Prolog. A CSP Interpreter (in Prolog) stores the "firing rules" of CSP's operational semantics. The Simulator (also in Prolog) determines the actions available and the resultant states. A GUI (written in Tcl/Tk) allows the user to interact with the Simulator.

Timed CSP is closed under rational time [DNR11]. Consider, for example, the following firing rule ($\overset{t}{\rightsquigarrow}$ stands for a timed transition of duration t):

$$\frac{P \overset{d'}{\rightsquigarrow} P'}{(P \triangleright^{d} Q) \overset{d'}{\rightsquigarrow} (P' \triangleright^{d-d'} Q)} \quad [0 < d' \leq d]$$

Let $P \triangleright^{d} Q$ have rational times only (in particular, d is rational). Let d' be rational. Then $d - d'$ is rational and, by induction, P' has rational times only. Thus, $P' \triangleright^{d-d'} Q$ has rational times only.
Decision 1 *Our Timed CSP simulator deals with rational time only.*

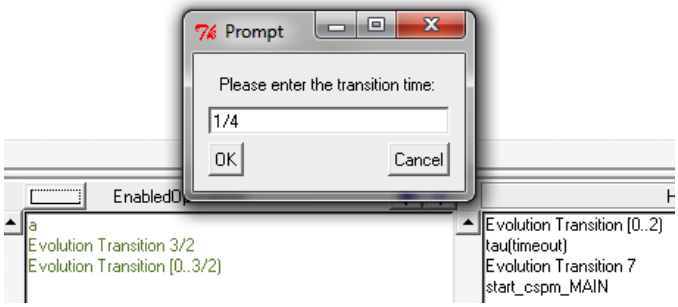
ProB also implements firing rules for those untimed CSP operators which usually are treated as syntactic sugar, e.g., the untimed timeout $P \triangleright Q = (P \,\Box\, Q) \sqcap Q$. We follow ProB's design:
Decision 2 *All untimed and timed operators have their own timed firing rules.*

[†] Acknowledging support by the SafeCap project, http://safecap.cs.ncl.ac.uk/index.php/Safecap_Project_Wiki.

To this end, in [DNR11] we extend Timed CSP's operational semantics as given in [Sch00]: (1) In a definitional way, as e.g. for the untimed timeout, (2) in a conservative way, as e.g. for the replicated alphabetized parallel or for the conditional (for which [Sch00] gave no firing rules).

The core of our simulator is a rational arithmetic built on top of Prolog's built-in proper integers. Simulating Timed CSP provides two challenges: (1) In order to calculate the largest time step possible for a Timed CSP process, one has to analyze the process recursively. Consider, for instance, the process $T = (P \rhd^e Q) \rhd^f R$ with $0 < e < f$ and untimed processes P, Q and R. In T, the process P is enabled within the time interval $[0, e)$. A time step of length e (and a τ-transition) leads to the new state $Q \rhd^{f-e} R$. Thus, the largest time step possible in T is e – see [DNR11] for details. (2) When the user chooses a timed transition of, say, d time units, the constant d needs to be propagated recursively along the term structure. Given, e.g., a time step $0 < d < e$ for the above term T, the resulting Timed CSP term is $(P \rhd^{e-d} Q) \rhd^{f-d} R$.

Currently, our simulator implements a slightly restricted sublanguage of Timed CSP: Processes can include only rational constants in timed operators; while most Timed CSP operators have been implemented, the operator $a@u \to P(u)$ (time of an action) is not supported yet.



The screen-shot shows a typical run of our simulator. Besides simulating examples given in [Sch00], we extensively use our tool within the SafeCap project in order to explore how the change of signalling rules affects railway capacity.

The ProB team has checked our implementation and intends to make it part of the next ProB distribution. This will require some minor changes to our code, mostly regarding syntax. It is future work to remedy the above mentioned, surmountable restrictions and to apply our tool within further application domains.

Acknowledgement We thank Erwin Catesbeiana (Jr.) for inspiring us to go the extra mile.

Bibliography

[DHSZ06] J.S. Dong, P. Hao, J. Sun, and X. Zhang. A Reasoning Method for Timed CSP Based on Constraint Solving. In LNCS 4260, Springer 2006.

[Dra11] M. Dragon. *A Timed* CSP *Simulator for Railway Systems*, BSc Dissertation, Swansea University, 2011.

[DNR11] M. Dragon, N. Nguyen, M. Roggenbach. *Theoretical foundations for simulating Timed CSP*, Technical report CSR 1-2011, Swansea University, 2011.

[Leu] M. Leuschel. The ProB Animator and Model Checker. Last accessed June 2011. http://www.stups.uni-duesseldorf.de/ProB/index.php5/Main_Page.

[Sch00] S. Schneider. *Concurrent and Real-time systems*. Wiley, 2000.

Assessing the Applicability of SVA in Analysing VHDL Models

James Sharp, Helen Treharne and Steve Schneider

University of Surrey

Abstract: Our work explores the potential of using Roscoe's Shared Variable Analysis (SVA) approach for modelling hardware systems. We demonstrate the functionality of SVA with an implementation of a Uniform Candy Distribution puzzle. This example illustrates how asserting eventual behavioural requirements on a system can be verified using SVA analysis. The modelling principles of the example resonates with the phasing of reading inputs, performing logical calculations and writing to outputs that are described in the behavioural semantics of VHDL. Thus, the example shows that SVA may be a suitable basis on which, if extended, would be applicable for analysis of hardware models. This is part of ongoing work which aims to look at methods for formally analysing VHDL models using model checking techniques and is sponsored by AWE plc.

Keywords: SVA, Formal Analysis, Hardware, VHDL

1 Shared Variable Analysis

Shared Variable Analysis (SVA) provides a way to analyse systems which use shared variables using formal model checking [Ros10]; and the Shared Variable Language (SVL) is an input language to SVA. In this paper we use the Uniform Candy Distribution Puzzle [TS], which has previously been written in CSP [IR08], to illustrate the strengths of SVA which will be applicable to the analysis of hardware models. The puzzle describes the passing and receiving of sweets between children. We represent the behaviour of a Child in the following SVL process:

$$
\begin{array}{ll}
Child(i) = \{iter\{ & sig(referee); \\
\quad int\ k; & childsSweets[i] := childsSweets[i] - k; \\
\quad k := childsSweets[i]/2; & if(childsSweets[i]\%2! = 0) then\{ \\
\quad sig(ready); & \quad childsSweets[i] := childsSweets[i] + 1\}; \\
\quad childsSweets[(i+1)\%N] := & isig(ch.(i+1)\%N, k); \\
\qquad childsSweets[(i+1)\%N] + k; & sig(passed)\ \}\}
\end{array}
$$

We then create a puzzle which contains three children and initialise the system so that child 0,1 and 2 have 0, 2 and 4 sweets respectively.

In [Ros10], signals are used as reporting events to enable CSP refinement specifications to be written. We use signals for reporting, but additionally to control execution. Signals synchronise the behaviour of children at certain points during their execution. Consequently, we are able to ensure that each child first acknowledges how many sweets she will be removing from her own pile at the beginning of a round. Once all children have read in their current number of sweets they proceed to add half of their sweet pile to their neighbouring child's pile. Again, once all

children have performed this step they will update their sweets pile by removing those passed to their neighbour from their count, and if they then have an odd number an extra sweet is added.

The specialized signal, *isig*, reports the current value of a shared variable. By using the *isig* we are able to use the CSP specification given by Rogennbach [IR08] in terms of this signal and hence demonstrate that the system will stabilise after nine passes have occurred and that each child with have four sweets.

2 Tailoring SVA for Hardware Models

From the VHDL behavioural semantics [IEE00], the puzzle is analogous with the three stages of a VHDL process firing; taking a snapshot of any VHDL signals used, performing some logical computation and then updating any signal that it has changed. A VHDL process fires when a signal it is sensitive to changes; as a result of this the cascaded firing of processes can occur as internal signals are updated. Only once all internal signals have stopped changing is the system considered to be in a stabilised state, waiting for some external stimuli to start processes firing again. At present there is no way to determine if the internal behaviour of a system has stabilised within SVA.

Roscoe describes a method for modelling of the StateMate semantics and consequently analysing StateMate state machines in [RW06]. These state machines distinguish between internal and external stimulus, and perform a variable number of *steps* to stabilise the internal behaviour before external stimulus can be accepted. Furthermore, [RW06] provides a methodology for tracking the passing of time, where time is incremented only after a series of *steps* have been performed. The concepts described in the StateMate compiler tie closely to the behavioural requirement of VHDL, which are, that internal behaviour is instantaneous and reaches a stable state before external stimulus may influence the model.

From our initial experiments with SVA and the functionality available within another CSP compiler we will adopt the ideas of time in Roscoe's StateMate compiler. This will allow us to identify, within an SVA model, when the processes have reached a stabilised state, and thus enable us to assert safety properties on a stabilised hardware system.

Bibliography

[IEE00] IEEE. Standard VHDL Language Reference Manual, IEEE Standard 1076. 2000.

[IR08] Y. Isobe, M. Roggenbach. Verifying the Uniform Candy Distribution Puzzle with CSP-Prover. In Gruner and Watson (eds.), *COLLOQUIUM and FESTSCHRIFT at Occasion of the 60th Birthday of Derrick Kourie*. University Pretoria, 2008.

[Ros10] A. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.

[RW06] A. W. Roscoe, Z. Wu. Verifying Statemate Statecharts Using CSP and FDR. In *Proceedings of ICFEM 2006*. 2006.

[TS] A. F. T. Bohman, O. Pikhurko, D. Sleator. Puzzle 6: Uniform candy distribution. http://www.cs.cmu.edu/puzzle/puzzle6.html

Generation of certifiably correct programs from formal models

Alexei Iliasov[1]

Newcastle University, England[1]

Abstract: On generating programs from with a correctness certificate.

Keywords: code generation, Hoare logic, formal modelling, Event-B

1 Introduction

A development process based on the application of formal notations and verifications techniques potentially delivers a system that is free from engineering defects. A typical formal development starts with a comprehensive requirements document, proceeds with a modelling stage where the requirements are formalised and transformed into implementable steps, and completes with the construction of a final product, e.g., a program or a hardware description. An automated code generator transforms models into runnable software quickly, consistently, reproducibly and with a lower rate of errors than is if coded manually. Most code generators, however, do not offer the guarantee of an error-free result. Commonly, a code generator is a fairly large program constructed informally and producing an output that is not (at least formally) traced to an input. This undermines the value of using formal methods in safety-critical domains. Industrial standards to the development of safety-critical systems, such as IEC 61508, require that for any tool used in a development there is a sufficiently strong justification. Such a justification could be an extensive prior experience with the tool or a formal certification by a relevant certification authority. As there are not many formal modelling tools that have been around for decades, the prior use is rarely an option for formal method adopters. These leaves just two opportunities: certify a code generator for the use in a given domain, or completely ignore the code generating activity in the safety case and verify the generated software as if it were constructed informally. The latter case leaves no reason for using formal modelling in the first place. In the former case, a code generator itself must be constructed accordingly to the relevant industrial standards for safety-critical software which means a higher cost, a longer development cycle and, possibly, tips the balance away from the use of formal modelling. In addition, certification requirements vary considerably between the certification bodies of differing nations and industries.

We propose an approach where instead of attempting to justify the use of a code generator a user places no trust whatsoever in the code generation stage but, through a code generator, obtains software that is certifiable without any further effort. The essence of the approach is in the transformation of a formal model into runnable software that is demonstratively correct in respect to a given set of verification criteria, coming from a requirements document. The technique is generally known as proof carrying code [Nec97]. In our approach, all the correctness guarantees are embedded in the resultant program; intermediate formal models are disregarded for the purpose of product certification and the design and functioning of a code generator are deemed irrelevant.

To evaluate the idea, we have implemented to a proof of concept tool for code generation from Event-B models. The tool, called B2H5, is available for evaluation together with a set of sample problems [Rod11]. B2H5 works in the context of a single Event-B machine and, optionally, a Flow diagram [Ili11]. The former describes the possibles computation steps, variables and their types, and global invariant properties. The latter defines an algorithmic structure of a target program and may be used to express additional verification constraints. The output is code annotated with a Hoare logic proof scheme. To this end, we have defined a custom Hoare logic with most of the inspiration from [OG76]. Proof annotations are automatically derived from event definitions and Flow diagram assertions and hence the tool is completely automatic. Event-B and Flow proof obligations are recorded as evidence of the satisfaction of side conditions of the inference rules of the logic. For instance, the Event-B invariant satisfaction proof obligation supplies the proof for a side condition in the global correctness rule of the Hoare logic. The tool is able to emit JML [BCC+05] code where non-deterministic statements (derived from non-deterministic actions or event parameters) are replaced with method calls. Such methods are annotated with pre- and post-conditions but have empty bodies. It is an obligation for a programmer to fill in the missing code; verifications tools would ensure that the method conditions are satisfied by the added code.

The related work may be loosely structured into approaches to verifying compilers and certifying compilers. A verifying compiler, for instance [GHZG99], implements a provably correct translation procedure for transforming a high-level programming language into machine code. The correctness is defined in terms of the observable behaviour of a program and, possibly, additional annotations. A certifying compiler generates a program together with a proof of its correctness. A notable example is the proof-carrying code technique [Nec97]. Our approach belongs to this group.

Bibliography

[BCC+05] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, E. Poll. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.* 7:212–232, June 2005.

[GHZG99] T. Gaul, A. Heberle, W. Zimmermann, W. Goerigk. Construction of verified software systems with program-checking: An application to compiler back-ends. 1999.

[Ili11] A. Iliasov. Use case scenarios as verification conditions: Event-B/Flow approach. In *Serene 2011*. 2011.

[Nec97] G. C. Necula. Proof-Carrying Code. In Jones (ed.), *Proceedings of the Symposium on Principles of Programming Languages*. Pp. 106–119. ACM Press, Paris, France, 1997.

[OG76] S. Owicki, D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica* 6:319–340, 1976.

[Rod11] Rodin platform plug-in. B2H5. 2011. Available at http://iliasov.org/b2h5/.