09381 Extended Abstract Collection
Refinement Based Methods for the Construction of Dependable Systems
— Dagstuhl Seminar —

Jean-Raymond Abrial[1], Michael Butler[2], Rajeev Joshi[3], Elena Troubitsyna[4] and Jim C. P. Woodcock[5]

[1] ETH Zürich, CH
[2] University of Southampton, GB
mjb@ecs.soton.ac.uk
[3] Jet Propulsion Laboratory, USA
rajeev.joshi@jpl.nasa.gov
[4] Aabo Akademi University - Turku, FIN
elena.troubitsyna@abo.fi
[5] University of York, GB
jim.woodcock@york.ac.uk

Abstract. With our growing reliance on computers, the total societal costs of their failures are hard to underestimate. Nowadays computers control critical systems from various domains such as aerospace, automotive, railway, business etc. Obviously, such systems must have a high degree of dependability – a degree of trust that can be justifiably placed on them. Although the currently operating systems do have an acceptable level of dependability, we believe that they development process is still rather immature and ad-hoc. The constantly growing system complexity poses an increasing challenge on the system developers and requires significant improvement on the existing developing practice. To address this problem, we investigated how to establish a set of refinement-based engineering methods that can provide the designers with a systematic methodology for development of complex systems.

Keywords. Specification, refinement, verification, modelling, dependable systems

Executive summary

The seminar brought together academicians that are experts in the area of dependability and formal methods and industry practitioners that are working on developing dependable systems. The industry practitioners have described their experience and challenges posed by formal modeling and verification. The academicians tried to address these challenges while describing their research work. We seminar proceeded in a highly interactive manner and provided us with an excellent opportunity to share experience.

One of the outcomes of that seminar was the identification of the following list of challenging issues faced by industrial users of formal methods:

– Team-based development
– Dealing with heavy model re-factoring
– Linking requirements engineering and FMs
– Abstraction is difficult
– Refinement strategies are difficult to develop
– Guidelines for method and tool selection
– Keeping models and code in sync
– Real-time modelling
– Supporting reuse and variants
– Proof automation
– Proof reuse
– Handling complex data structures
– Code generation
– Test case generation
– Handling assumptions about the environment

The seminar has encouraged knowledge transfer between several major initiatives in the area of formal engineering of computer-based systems. We have got a good understanding of the advances made within the EU-funded project Deploy "Industrial deployment of system engineering methods providing high dependability and productivity". The project aims at integration of formal engineering methods into the existing development practice in such areas as automotive industry, railways, space and business domains. The participants described advantages and problems of refinement-based development using Event-B and Rodin tool platform. The advances made within the Grand Challenge in Verified Software initiative have been described by the researchers working on the Mondex system and a verified file store. Several large-scale experiments on system development and software verification were presented by the various researchers working in the software industry.

Discussions of such topics as foundations of program refinement, verification, theorem proving, techniques for ensuring dependability, automatic tool support for system development and verification, modeling concurrency and many others resulted in several new joint research initiatives and collaborative works.

This document consists of two parts: the first is a collection of short abstracts of talks and the second is the collection of extended abstracts.

Part 1. Short Abstracts

Refinement of programs of distributed agents

Egon Boerger (University of Pisa, IT)

We present a notion of program refinement and refinement correctness that works for stepwise refining programs to be used in runs of distributed agents. As case study we investigate an implementation of synchronous message passing by semaphores together with its correctness proof.

This is ongoing joint work with Iain Craig (Birmingham) and part of a larger project to model and verify operating system kernels.

Security specification: completeness, feasibility, refinement

Eerke Boiten (University of Kent, GB)

The formal methods and refinement community should be able to contribute to the specification and verification of security protocols. This talk describes a few of the essential differences, or problems. First, security properties go beyond functional correctness, and are fundamentally different for different applications.

Moreover, tomorrow's attacks may not be anticipated by yesterday's security properties. Second, notions of security may not be absolute: it may be good enough if guessing our secret is merely hard rather than impossible – and in some cases that may be provably the best we can get. Where does that leave us in wanting to provide security protocols "correct by construction"?

An overview of the Rodin toolset

Michael Butler (University of Southampton, GB)

Rodin is a toolset for the Event-B language and refinement method. The core functionality includes support for for static checking of models, generation of consistency and refinement proof obligations, and automatic and interactive proof. A key design consideration is support for the interaction between modellinig and proof. A further key design consideration is open architecture that enables extension to support additional modelling and analysis functionality. The toolset is implemented on Eclipse and is open source.

A roadmap for the Rodin toolset

Michael Butler (University of Southampton, GB)

Event-B is a formal method for system-level modelling and analysis. Key features of Event-B are the use of set theory as a modelling notation, the use of refinement to represent systems at different abstraction levels and the use of mathematical proof to verify consistency between refinement levels.

The Rodin Platform[6] is an Eclipse-based toolset for Event-B that provides effective support for refinement and mathematical proof.

Keep aspects of the are support for abstract modelling in Event-B; support for refinement proof; extensibility; open source.

To support modelliing and refinement proofs Rodin contains a modelling database surrounded by various plug-ins: a static checker, a proof obligation generator, automated and interactive provers.

The extensibility of the platform has allowed for the integration of various plug-ins such as a model-checker (ProB), animators, a UML-B transformer and a LaTeX generator. The database approach provides great flexibility, allowing the tool to be extended and adapted easily. It also facilitates incremental development and analysis of models.

The platform is open source, contributes to the Eclipse framework and uses the Eclipse extension mechanisms to enable the integration of plug-ins.

Joint work of: Abrial, Jean-Raymond; Butler, Michael; Hallerstede, Stefan; Voisin, Laurent

Challenges in Applying Formal Methods - SME view

Mathieu Clabaut (SYSTEREL Aix en Provence, FR)

This paper outlines past and foreseen challenges in applying both *classical B* and *event B* to design safety related systems in an SME.

On Proving with Event-B that a Pipelined Processor Model Implements its ISA Specification

John Colley (University of Southampton, GB)

Microprocessor pipelining is a well-established technique that improves performance and reduces power consumption by overlapping instruction execution. Verifying, however, that an implementation meets this ISA specification is complex and time-consuming.

One of the key verification issues that must be addressed is that of overlapping instruction execution. This can introduce hazards where, for instance, a new instruction reads the value from a register which will be written by an earlier instruction that has not yet completed.

Using Event-B's support for refinement with automated proof, a method is explored where the abstract machine represents directly an instruction from the ISA that specifies the effect that the instruction has on the microprocessor register file. Refinement is then used systematically to derive a concrete, pipelined execution of that instruction.

[6] Available from www.event-b.org

Microarchitectural considerations are raised to the specification level and design choices can be verified much earlier in the flow.

The method proposed therefore has the potential to be integrated into an existing high-level synthesis methodology, providing an automated design and verification flow from high-level specification to hardware.

Joint work of: Colley, John; Butler, Michael

Mechanising a correctness proof for a lock-free stack

John Derrick (Sheffield University, GB)

Concurrent objects are inherently complex to verify. In the late 80s and early 90s, Herlihy and Wing proposed linearizability as a correctness condition for concurrent objects, which - once proven - allows to reason about concurrent objects using pre- and postconditions only. A concurrent object is linearizable if all of its operations appear to take effect instantaneously some time between their invocation and return.

Here we discuss simulation-based proof conditions for linearizability and apply them to two a concurrent implementations of a lock-free stack. Similar to other approaches, we employ a theorem prover (here, KIV) to mechanize our proofs. Contrary to other approaches, we also use the prover to mechanically check that our proof obligations actually guarantee linearizability. This check employs the original ideas of Herlihy and Wing of verifying linearizability via possibilities.

Linearizability - deriving and mechanically verifying proof obligations

John Derrick (Sheffield University, GB)

Concurrent objects are inherently complex to verify. In the late 80s and early 90s, Herlihy and Wing proposed linearizability as a correctness condition for concurrent objects, which - once proven - allows to reason about concurrent objects using pre- and postconditions only. A concurrent object is linearizable if all of its operations appear to take effect instantaneously some time between their invocation and return.

In this paper we propose simulation-based proof conditions for linearizability which have been shown to be applicable to two concurrent implementations, a lock-free stack and a set with lock-coupling. Similar to other approaches, we employ a theorem prover (here, KIV) to mechanize our proofs. Contrary to other approaches, we also use the prover to mechanically check that our proof obligations actually guarantee linearizability. This check employs the original ideas of Herlihy and Wing of verifying linearizability via possibilities.

Joint work of: Derrick, John; Schellhorn, Gerhard; Wehrheim, Heike

Beetlz

Fintan Fairmichael (University College - Dublin, IE)

When a change to a system under development is motivated from new-found constraints, realisations, or changes at the implementation level, should we treat the change differently to a modification of the requirements? Can we develop in such a way that is both rigorous and flexible in the face of such changes? We advocate that our processes should design for reversibility from the implementation level all the way through to the highest level of abstraction. We will take a brief look at our recently developed tool, Beetlz, for checking and maintaining the consistency between BON specifications and the corresponding implementation in JML-annotated Java.

Formal modelling and refinement of OS kernels

Leo Freitas (University of York, GB)

During the POSIX pilot project on verified flash file stores, we realised the need for an underlying formalised kernel. This motivated work on modelling a simple and a separation real-time kernel for embedded devices, which we will present.

The work is also related to other pilot projects in the grand challenge, such as the verification of the real-time operating systems (e.g., FreeRTOS). The key difference from FreeRTOS is that, instead of trying to verify such successful product's code, we are modelling the kernel from the requirements down through to C code using Z and the refinement calculus. In this process, we found many interesting lemmas and general data structures that are useful for other domains and pilot projects. We will report here on the current state of this work and where is it going in the near future.

Towards Reasoned Modelling: Turning Proof Obligations into Modelling Guidance

Gudmund Grov (University of Edinburgh, GB)

The activities of formal modelling and reasoning are closely related.

But while the rigour of building formal models brings significant benefits, formal reasoning remains a major barrier to the wider acceptance of formalism within design. Here we propose reasoned modelling - a technique which aims to abstract away from the complexities of low-level proof obligations, and provide high-level modelling guidance to designers when proofs fail. Inspired by proof planning, the technique will combine modelling and reasoning patterns. We present the results of our initial investigations into reasoned modelling, and outline how we plan to realize our proposal.

An outline of a proposed system that learns from experts how to discharge proof obligations automatically

Gudmund Grov (University of Edinburgh, GB)

Most formal methods give rise to proof obligations (POs) which are putative lemmas that need proof.

Discharging these POs can become a bottleneck in the use of formal methods in practical applications.

It is our aim to increase the repertoire of techniques for reducing this bottleneck by tackling learning from proof attempts.

In many cases where a correct PO has not been discharged, an expert can easily see how to complete a proof.

We believe that it would be acceptable to rely on such expert intervention to do one proof if this would enable a system to kill off others “of the same form”.

Joint work of: Bundy, Alan; Grov, Gudmund; Jones, Cliff

A (small) improvement of Event-B?

Stefan Hallerstede (Universität Düsseldorf, DE)

Event-B and the Rodin tool use a number of simple techniques that make the modelling method around them effective in practical applications. We present two of these techniques, anticipation and witnesses. It is interesting how a couple of very simple techniques are so important for the method to work. Finally we propose a small enhancement of Event-B that would extend the use of witnesses.

Event-B Decomposition for Parallel Programs

Thai Son Hoang (ETH Zürich, CH)

We present here an approach for developing a parallel program combining refinement and decomposition techniques. This involves in the first step to abstractly specify the aim of the program, then subsequently introduce shared information between sub-processes via refinement. Afterwards, decomposition is applied to separate the resulting model into sub-models for different processes. These sub-models are later independently developed using refinement. Our approach aids the understanding of parallel programs and reduces the complexity in their proofs of correctness.

Joint work of: Hoang, Thai Son; Abrial, Jean-Raymond

Qualitative Reasoning for the Dining Philosophers

Thai Son Hoang (ETH Zürich, CH)

We continue our investigation of qualitative probabilistic reasoning in Event-B. In the past we have applied it protocol verification, in particular, the Firewire protocol. There is still some way to go to achieve a practical method for qualitative probabilistic reasoning. In this presentation we attempt the probabilistic solution to the dining philosophers problem to move further towards such a method.

Joint work of: Hallerstede Stefan; Hoang, Thai Son

Structuring Specifications with Modes

Alexei Iliasov (University of Newcastle, GB)

The two dependability means considered in this paper are rigorous design and fault tolerance. It can be complex to rigorously design some classes of systems, including fault tolerant ones, therefore appropriate abstractions are needed to better support system modelling and analysis. The abstraction proposed in this paper for this purpose is the notion of operation mode. Modes are formalised and their relation to a state-based formalism in a refinement approach is established. The use of modes for fault tolerant systems is then discussed and a case study presented. Using modes in state-based modelling allows us to improve system structuring, the elicitation of system assumptions and expected functionality, as well as requirement traceability.

Joint work of: Iliasov, Alexei; Alexander Romanovsky; Fernando Lufis Dotti

Launching Formal Methods into Space

Dubravka Ilic (Space Syst. Finland Ltd, FI)

This paper gives an overview of the experiences and so far known challenges in applying Event-B in the space domain.

Reasoned Modelling: Combining Proof and Modelling Patterns to Guide Systems Design

Andrew Ireland (Heriot-Watt-University Edinburgh, GB)

The activities of formal modelling and reasoning are closely related.

But while the rigour of building formal models brings significant benefits, formal reasoning remains a major barrier to the wider acceptance of formalism within design. Here we propose reasoned modelling - a technique which aims to abstract away from the complexities of low-level proof obligations, and provide high-level modelling guidance to designers when proofs fail. Inspired by proof planning, the technique will combine modelling and reasoning patterns. We present the results of our initial investigations into reasoned modelling, and outline how we plan to realize our proposal.

Joint work of: Ireland, Andrew; Grov, Gudmund

Refinement, Problems and Structures

Michael Jackson (The Open University - Milton Keynes, GB)

Refinement can be applied explicitly to the structure of the problem world, successive refinement steps taking account of problem domains in order of decreasing distance from the hardware/software machine. In this way system requirements can in principle be refined to software specifications.

The question immediately arises: Does the development deal with a single tree, or with a forest of refinements? For a realistic system only a forest is possible. Further questions then arise: How are the trees to be identified and separated? How are they to be recombined? What is the structure of the development process and its product? These are fundamental questions about problem analysis and solution.

Answers to these fundamental questions must recognise the importance of human understanding in software and system development. Formal tools are most effectively deployed within an intellectual framework based on principles of understanding.

Abstraction is all we've got

Clifford B. Jones (University of Newcastle, GB)

My talk was a condensed version of a submitted paper available as a report http://www.cs.ncl.ac.uk/publications/trs/papers/1166.pdf It's abstract follows:

This paper presents a formal development of a non-trivial parallel program: Simpson's implementation of asynchronous communication mechanisms
(ACMs). Although the correctness of this "4-slot algorithm" has been shown elsewhere, earlier proofs fail to offer much insight into the design.

The aims of this paper include both the presentation of an understandable (yet formal) design history of this one algorithm and teasing out of the techniques employed in the explanation for wider application. Among these techniques is using a "fiction of atomicity" as an aid to understanding the initial steps of development.

The rely-guarantee approach is, here, combined with notions of read/write frames and "phased" specifications; the atomicity assumptions implied by rely/guarantee conditions are realised by clever choices of data representation.

Verifying Large Probabilistic Models by 3-Valued Abstraction

Joost-Pieter Katoen (RWTH Aachen, DE)

Model checking of probabilistic models is used in many different areas such as performance and dependability evaluation, security protocols, randomized algorithms, and biological systems. Tools have been successfully applied to numerous case studies, but like in traditional model checking, the state explosion problem forms a serious limitation. Although many techniques from traditional model checking have been generalized towards probabilistic models such as BDDs and partial-order reduction, more aggressive reduction techniques are needed. In this talk, we introduce model checking of continuous-time Markov chains (CTMCs), present a three-valued abstraction technique, and present several examples to show its effectiveness when applied to huge, and even infinite CTMCs.

Verifying the Microsoft Hyper-V Hypervisor with VCC (with Thomas Santen)

Dirk Leinenbach (DFKI Saarbrücken, DE)

The European Microsoft Innovation Center (EMIC), the German Research Center for Artificial Intelligence (DFKI), and Saarland University cooperate in the Verisoft project on verifying the kernel of Hyper-V, Microsoft's server virtualization software. The first part of the talk presents VCC, an industrial-strength verification suite for low-level concurrent C code developed jointly by EMIC and Microsoft Research. The second part concentrates on the specifics of applying VCC for the verification of Hyper-V, e.g., the formal models required to express (and verify) hypvervisor correctness.

Joint work of: Leinenbach, Dirk; Santen, Thomas

Developing Tools for Formal Methods: Lessons and Outlook

Michael Leuschel (Universität Düsseldorf, DE)

We will present the ProB toolset for animation, model checking and refinement checking of B, Event-B, CSP and Z specifications.

We summarise recent developments surrounding the toolset, with successful application in the railway industry. We summarise our experience in developing tools for formal methods, and conclude with an overview of the major challenges for the future.

Formal Methods in the Automotive Sector - Challenges for Deployment

Felix Loesch (Robert Bosch GmbH - Stuttgart, DE)

The presentation will give a summary of the challenges for deployment of formal methods in the automotive sector.

Can we reuse qualitative proofs for quantitative security analysis?

Larissa Meinicke (Macquarie University - Sydney, AU)

One would ideally like Formal Methods for verifying and developing software applications that exhibit probabilistic behaviours and are subject to security requirements. One of the main goals of our current research is to integrate security, probability and modularity features into refinement-based Formal Methods.

A "first step" on this path was the introduction (Morgan 2006) of a "Shadow Model" that could be used to develop non-interference style protocols by refinement. Using this model it was shown how qualitative proofs could be performed using routine algebraic laws and calculations.

Since this model does not take probability into consideration, proofs of correctness using the Shadow Model may not be correct given the ability of an attacker to perform repeat-experiments or statistical analysis: correctness with respect to this more sophisticated form of attack need to be verified in more complex, probabilistic model extensions.

This leads us to wonder "under what circumstances do "Shadow proofs" guarantee correctness in such an extended model in which, say, hidden choices are replaced by uniform probabilistic choices"?

In this talk we explore the idea that "Shadow programs" –in which uncertainty is abstracted by nondeterministic choices– and "probabilistic Shadow programs" –in which uniform probabilistic choices take the place of purely nondeterministic choices– share many algebraic rules; and we show that this makes it possible to use qualitative proofs (like those already used in The Shadow to prove the correctness of a variety of security protocols) to verify security applications with respect to a notion of testing that allows statistical attacks.

Refinement-based guidelines for constructing algorithms

Dominique Mery (LORIA - Nancy, FR)

The *correct-by-construction* approach can be supported by a progressive and incremental process controlled by the refinement of models of programs. We explore the Event-B modelling language to illustrate the expression of our methodological proposal using proof-based patterns called guidelines. The main objective is to facilitate the correct-by-construction approach for designing classical sequential and distributed algorithms. We address the description of guidelines for the design of programs and algorithms and the integration of proof-based aspects using the RODIN platform. More precisely, we introduce several methodological steps identified during the development of case studies, and we propose auxiliary notions, such as refinement diagrams, for guiding users during problem analysis. A general structure characterizes the relationship between the contract, the Event B, and the developed algorithm using a specific application of Event B models and refinement. We simplify the translation of Event B models into algorithmic elements by promoting the use of recursive constructs. The resulting algorithm is proved to be sound with respect to the pre/post specification, namely, the contract.

Applications rely on a dynamic programming technique that illustrates the applicability of these patterns based on a call-as-event relationship. Each proof-based development is validated using the RODIN platform. Distributed algorithms are considered with respect to the local computation model based on a relabelling relation over graphs representing distributed systems. The VISIDIA toolbox provides facilities for simulating local computation models which can be easily modelled using Event B and the refinement.

Simple probabilistic proofs for simple probabilistic programs

C. Carroll Morgan (Univ. of New South Wales, AU)

While expectation transformers are fully general for proofs of programs with both demonic and probabilistic choice, in some cases an expectation invariant for a loop an be hard to find. This is particularly striking when the program has a simple intuitive justification, but the invariant is elusive, complex or unknown.

I'll propose a simplified reasoning technique, based on convex sets of distributions, which is appropriate only for purely probabilistic (ie deterministic, non-demonic) programs. Its justification will however be given in terms of expectation transformers. Thus simple- and complex arguments should be able to exist within the same system, each used where it is needed and appropriate.

With luck, some version of this should be amenable to event-B -style reasoning.

Formal Methods for Enterprise Applications - Challenges and Experiences

Andreas Roth (SAP Research - Darmstadt, DE)

We report on first experiences with applying formal specification and verification techniques in the development of enterprise applications - most of them within the context of the Deploy project. We highlight the challenges we see when applying refinement based methods in this area and our current approaches to address them.

Formal Methods in the Development of Business Software

Andreas Roth (SAP Research - Darmstadt, DE)

We discuss the suitability of Formal Methods for the development of business software as well as experiences and challenges in this area. Business software involves very different kinds of software: technical components, business applications, analytical applications. Our focus is on business applications which promise a good probability of a second use of Formal Methods. We investigate service choreography models, business object models, and business process models, as typical instances of models in this domain. Our approach is to take existing diagrammatic models and translate them to a formal language, e.g. Event-B. Though the approach works well, there are a number of challenges. These are especially the provision of good user feedback on prover or model checker results, the need for better automation, and a better usage of refinement.

The seed was spread out: An Overview of Formal Methods Application in Brazil

Aryldo G Russo Jr (AeS Group - Sao Paolo, BR)

The use of formal methods has constantly increased, although with basically two constraints: their use has been concentrated mostly in Europe, and they have been used only by big companies which are in charge of developing some safety critical applications and in some how are conected with academia projects. The aim of this talk is to present how formal methods have been applied in other parts of the world, mainly South America, by a small company headquarted in Brazil. It is splited in three parts. First, an introduction about the AeS Group, a small company that is been trying to apply Formal Methods in its projects. Second, some real industrial applications are presented (with some reasoning about why the used tool was selected) and how the formal method culture can drastically help the development process. And finally, some of the ongoing work (industrial and academic) that is been developed by the author and gaps identified in industry that can be fulfilled by extending the features of the actual tools.

Formal Foundation to Systematic Development of Simulink/Stateflow Models

Manoranjan Satpathy (General Motors - TCI - Bangalore, IN)

The Simulink/Stateflow (SL/SF) environment from Mathworks is widely used in industry for the development of control applications, especially in the automotive and aerospace domains. Such models are constructed by control engineers directly from the requirements, and the models are usually validated by simulation. This process can have the following deficiencies: (a) there is a large semantic gap between the informal requirements and the SL/SF design models and hence errors are likely to creep in, and (b) simulation alone may not discover errors/ambiguities in the model. Our research focuses on correct construction of Simulink/Stateflow models. We follow the Event-B method for the construction of hybrid control systems, and after the requirements are sufficiently refined, our method generates SL/SF models from the refined Event-B models. The SL/SF models so generated would be correct with respect to the requirements. Many verification/validation (V/V) infrastructures like Hardware-in-loop (HIL) testing, Plant-in-loop (PIL) testing and FlexRay bench have been built around Mathwork's SL/SF models. The SL/SF models that we generate out of the Event-B models can get the benefit of existing V/V infrastructure.

Joint work of: Satpathy, Manoranjan; Ramesh, S

Abstract Specification of the UBIFS File System for Flash Memory

Gerhard Schellhorn (Universität Augsburg, DE)

Flash memory is used in more and more applications (mp3 players, mass storage systems, automotive applications, space crafts) since it has certain advantages over magnetic disks like higher speed and resistance against kinetic shock.

However, to efficiently use flash memory, its specific characteristics demand the use of a specialized file system.

Development of a verified file system has been proposed as a case study in the context of the Grand Verification Challenge.

The talk describes a formal specification of the core data structures used in such a file system: an inode-based store for the data, index structures for efficient access, and a journal to deal with intermediate crashes. The model was based on an analysis of the UBIFS filesystem which was recently integrated into the Linux kernel.

The model is intended as an intermediate level of a refinement tower that starts with a standard (POSIX) specification of file system operations and ends with hardware operations according to the ONFI standard. We plan to develop such a tower, and the talk gives some challenges for the refinements of this effort.

Joint work of: Schellhorn, Gerhard; Schierl, Andreasl; Haneberg, Dominik; Reif, Wolfgang

Abstract Specification of the UBIFS File System for Flash Memory

Gerhard Schellhorn (Universität Augsburg, DE)

Today we see an increasing demand for flash memory because it has certain advantages like resistance against kinetic shock. However, reliable data storage also requires a specialized file system that can handle the limitations of flash memory. This paper develops a formal, abstract model for the UBIFS flash file system. We develop formal specifications for the core components of the file system: the inode-based file store, the flash index, its cached copy in the RAM and the journal to save the differences. We give an abstract specification of the interface operations of UBIFS and prove some of the most important properties using the interactive verification system KIV.

Joint work of: Schellhorn, Gerhard; Schierl, Andreas; Haneberg, Dominik; Reif, Wolfgang

Pattern-based Refinement of Confidentiality Requirements

Holger Schmidt (Universität Duisburg-Essen, DE)

We present an approach to security requirements engineering, which makes use of special kinds of problem frames that serve to structure, characterize, analyze, and solve software development problems in the area of software and system security.

In this paper, we focus on confidentiality problems. We enhance previously published work by formal behavioral frame descriptions, which enable software engineers to formally specify security requirements. Consequently, software engineers can prove that the envisaged solutions provide functional correctness and that the solutions fulfill the specified security requirements.

Full Paper:
http://swe.uni-duisburg-essen.de/en/members/schmidt/index.php

Modelling Finitary Fairness in Event B

Emil Sekerinski (McMaster University, CA)

In modelling concurrent systems, fair choice allows to abstract from scheduling policies of processes or from processor speeds. Refinement approaches like Event B support only the weaker notion of nondeterministic choice. We show how finitary weak fairness can be expressed in Event B. Compared to standard fairness, finitary fairness is a "more refined" model, that simplifies verification of liveness properties and is suitable for fault-tolerant distributed computing. A generic transformation of an Event B model with finitary fairness to a standard Event B model is suggested. The refinement process is illustrated with the development of the alternating bit protocol.

Joint work of: Sekerinski, Emil; Zhang, Tian

Ensuring Correctness of Network Applications with MIDAS

Kaisa Sere (Aabo Akademi University - Turku, FI)

Network applications and services are pervading our society in a unforeseen manner. On one hand, this creates an extraordinary market for innovative networked applications and information services. On the other hand, these applications and services tend to be more and more complex, while the human perception of the supporting network more acute, as we expect to be able to use networks anywhere and at any time. The framework that we propose in this paper sets out to alleviate the task of a network application developer in two directions. First, we employ modularity and stepwise development to address the inherent application complexity. Second, we employ a reasoning framework named MIDAS to ensure the correctness of the proposed modeling, both at the application and at the supporting network level.

Joint work of: Petre, Luigia; Sere, Kaisa; Waldén, Marina

Specifying Safety Requirements for a Railway Interlocking System (An example using refinement in UML-B)

Colin F. Snook (University of Southampton, GB)

We illustrate the use of UML-B to specify safety requirements in a railway interlocking system. Starting from a list of documented hazards, the example uses three refinement levels to concisely specify what is meant by a safe interlocking system. The refinements firstly introduce the basic domain concepts involved in an unsafe railway system, then document assumptions about the system that are relied upon to mitigate hazards, and finally specify the safety requirements that the proposed system must meet in order to avoid hazards. Hence, the model is progressively constrained from an unsafe one to a safe one.

Refinement-Based Specification and Verification

Maria Spichkova (TU München, DE)

The main focus of this approach is on interactive real-time systems. The approach is based on the methodology "Focus on Isabelle" where specification and verification/validation methodologies are treated as a single, joined, methodology with the main focus on the specification part – using coupling of the formal specification framework Focus in the generic theorem prover Isabelle/HOL.

By considering the framework "Focus on Isabelle", which is result of the coupling, we can influence the complexity of proofs already during the specification phase.

The presentation introduces how the ideas of specification groups, refinement-based verification as well as decomposition and refinement layers can be used to optimize the verification process, and which influences they have on the specification process.

Formal Development and Assessment of Dependable Control Systems

Elena Troubitsyna (Aabo Akademi University - Turku, FI)

Dependability is degree of reliance that can be justifiably placed on a computer-based system. It is a multi-facet system characteristic that encompasses safety, reliability, availability, maintainability and security. Dependability is impaired by faults that might propagate to a system level and result in a system failure. If a failure occurs then the system might cease to provide its services or provide them incorrectly. A set of techniques known as means for dependability aims at mitigating consequences of fault occurrence as well as avoiding and removing faults during the system design. Means for dependability include fault avoidance, fault tolerance, fault removal and fault forecasting. We argue that by interfacing refinement process with these techniques we would significantly enhance system development process. We present several examples to support our argument.

Joint work of: Troubitsyna, Elena; Laibinis, Linas; Tarasyk, Anton

Practical Experiences Constructing Working Virtual Machines

Stephen Wright (University of Bristol, GB)

The Instruction Set Architecture (ISA) of a computing machine is the definition of the binary instructions, registers, and memory space visible to an executing program. Despite there being many ISAs in existence, all share a set of core properties which have been tailored to their particular applications. An abstract model may capture these generic properties and be subsequently refined to a particular machine: this is a task to which the Event-B formal notation is well suited. The motivation for the work is the systematic specification of the ISAŠs behavior for all possible instruction sequences loaded into the machine, whether part of a correct program or an erroneous one.

The constructed model consists of two parts: a generic description of properties common to most ISAs, and refinement to particular ISAs. The generic part is incrementally constructed from a very trivial initial model, and the second part constructs a particular ISA from this. The complete refinement process is demonstrated by the creation and testing of Virtual Machines automatically generated as C source code via a translation tool, which was developed as part of the project.

The technique is demonstrated by refinement to the MIDAS (Microprocessor Instruction and Data Abstraction System) ISA specification. MIDAS is a specification capable of executing binary images compiled from the C language. It is intended to be representative of typical microprocessor ISAs, but using a minimal number of defined instructions in order to make a complete refinement practical. There are two variants: a stack-based machine and a randomly accessible register array machine. The two variants employ the same instruction codes, the differences being limited to register file behavior. Compiler tool chains for each variant have been developed.

Some numbers: the MIDAS ISAs have thirty four instructions; their complete refinements consist of over thirty steps, expanding a single initial event to over one hundred for each variant. This process involves the discharging of nearly five thousand proof obligations. Automatic translation yields over four thousand lines of C source code for each variant.

This presentation will give an overview of the MIDAS project as an example of a model refinement of sufficient scale, depth and detail to be suitable for automatic translation into a usable executable. The place of Event-B model construction within a wider development process is described. Various scaling issues are discussed, including editing and manipulation of Event-B notation via the user interface, discharging of the vast number of proof obligations, and the practical limits of Rodin as an Eclipse/Java platform. The modeling techniques used to mitigate some of these scaling issues are described, and promising emerging or proposed Event-B features highlighted. Suggestions for possible future enhancements are made.

Part 2. Extended Abstracts

Security specification: completeness, feasibility, refinement

Eerke Boiten

Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, UK.
E.A.Boiten@kent.ac.uk www.cs.kent.ac.uk/~eab2/

Abstract. The formal methods and refinement community should be able to contribute to the specification and verification of cryptography based security protocols. This paper describes a few of the challenges that arise in this context. These include: security properties which differ from one application to another, and as a consequence issues of specification completeness; approximate rather than absolute notions of security, and underlying theories which do not provide obvious methods for "correctness by construction".

1 Introduction: Commitment and Completeness

At a first glance, cryptographic protocols provide exactly the kind of problems that formal methods are most suitable for and perform best at: short programs (most fit on a single page), based on rich algebraic mathematics, whose correctness is highly critical. However, the mathematics and the notions of security (correctness) are very different from the usual formal methods repertoire.

As an example, consider the cryptographic primitive of *bit commitment*. This is an essential ingredient of many cryptographic applications, particularly to build up trust between different parties, e.g. in authentication, and for zero-knowledge proofs of knowledge. Commitment, informally, is like putting a value in a locked box. One party (the "committer") chooses a message, and transforms it in a way which makes sure they cannot later claim it was a different message (it is "binding"), and the other party (the "receiver") cannot see it (it is "hiding"), and passes the transformed message to the other party (it "commits"). After commitment has taken place, typically further interaction will follow. At some point the commitment may be opened, e.g. as a check for honesty. One then expects the original message to be retrieved ("correctness"). The three properties: hiding, binding, and correctness together constitute the specification of commitment. When the message is a single bit, it is called *bit commitment*.

Another method of specifying this functionality is in what is commonly called the "ideal model" [12], where parties can communicate securely with an incorruptable third party. In that model, the committer sends their message to the trusted intermediary, who then confirms to the receiver that some commitment is made. When the committer asks for the commitment to be opened, the intermediary sends the original message to the receiver. Security of an implementation in this model means: any attack that succeeds against the implementation is as likely to succeed against the ideal model scenario. Although we have omitted the formal details here, it appears as if hiding, binding and correctness are indeed guaranteed in this specification. Conversely, it would be difficult to come up with a simpler "ideal model" specification that satisfies those three properties. We will come back to this in detail in Section 2.

To consider a rather more complicated example, protocols for electronic voting have been studied for many years, leading to an extensive list of desirable security properties [11]:

> fairness, eligibility, individual verifiability, universal verifiability, vote-privacy, coercion-resistance, receipt-freeness

with many of these varying depending on whether computers and election officials can be trusted or not. Even though the cited work provides a lot of structure by establishing relations between all these properties, it would still be hard to be certain that this set of properties or any future extension completely covers all possible attacks on an electronic voting protocol [9].

In general, when stated security properties closely match attacks that have been envisioned, clearly any verification is relative to the set of attacks considered, and completeness remains an issue. The cryptographic security community is undecided as to whether ideal model specification addresses this completeness problem [9, 10].

2 Commitment and Feasibility

We examine commitment and its security properties in more detail here. The context in which commitment schemes must be understood is as part of a protocol. A *protocol* involves at least two parties and is an algorithmic prescription for a number of causally related communications between the involved parties, aiming to achieve a particular objective. A protocol may succeed, or it may fail. It fails when the exchange of messages stops prematurely, for example after one party observes that another party is not adhering to the protocol. It succeeds if the protocol has completed and none of the parties has declared failure explicitly. Parties which act according to the protocol's rules and aim to achieve the protocol objective are called "honest". If the protocol succeeds although the objective has *not* been achieved, this indicates a breach of security. The protocol is *expected* to fail if some of the parties act dishonestly – thus, it is never in the interest of a dishonest party to perform an action that is *guaranteed* to lead to the protocol's failing. A practically relevant expectation is that a cryptographic protocol has a fixed number of fixed size messages, where the numbers may depend on the sizes of any protocol parameters, or on a security parameter (such as a key size).

The bit commitment scheme consists of three phases: preparation, committed, and opened. In the preparation stage, no bit has been chosen yet; in the commited stage, the committer has chosen a bit b that they cannot change (binding), and that the receiver does not know (hiding); in the opened phase, the receiver knows that the committer originally committed to b. The transitions between phases are achieved by messages from the committer to the receiver.

A first, obviously broken, attempt at a protocol is where the committer sends out a value $commit(b)$ for a known function *commit*, and later the value b as an opening. Correctness is guaranteed, but hiding normally is not: the receiver can check immediately by "exhaustive" search whether they received $commit(0)$ or $commit(1)$. However, if $commit(0) = commit(1)$ then hiding is guaranteed, but binding is not.

The normal solution for this is *randomisation*. The traditional argument for the need for randomisation in cryptography is masking known distributions within plaintexts in encryption algorithms – this is another. There are two common views of probabilistic algorithms. One view is that they include explicit probabilistic choices "inside". This model fits best when e.g. considering the combination of non-deterministic and probabilistic specification [13]. The other view is to consider "deterministic extensions" of probabilistic algorithms: these are deterministic algorithms which take an additional argument (sampled from a given distribution) representing the actual probabilistic choices made. In the context of commitment, we need the latter view. Thus we end up with a new specification, where *commit* takes an additional argument, which is also sent at opening time to allow the receiver to verify correctness. However, due to the assumption of bounded sized messages, this additional argument is bounded, and thus both sides can attempt to cheat. When the committer has sent out $c = commit(0, r)$, he can search for r' such that $commit(0, r) = commit(1, r')$ in order to defeat the binding property: he could claim to have committed to 1 and provide r' as the evidence. Thus, such r' should not exist. However, in order to defeat hiding, the receiver can search exhaustively for r such that $c = commit(0, r)$ or $c = commit(1, r)$. One of these is guaranteed to succeed, so the only case where hiding succeeds because the cheating receiver has no information is when $c = commit(0, r) = commit(1, r')$ – exactly the case where binding fails. Thus, binding and hiding are contradictory properties, and a protocol of the suggested shape satisfying both cannot exist, despite the existence of an "ideal model" specification.

However, commitment *is* considered useful, even if practical schemes cannot live up to the ideal. The compromise of completely dropping one of the two crucial properties is clearly unacceptable – and we can do significantly better than that, by bringing in a computational notion of correctness. In summary: the ideal combination of "perfect" binding and hiding is not achievable; however, the literature shows that schemes exist which approximate both as closely as required, provided we assume (dishonest) parties to be constrained to bounded (polynomial) time. We will briefly describe approximate notions of correctness and refinement in Section 3.

A final aside about commitment relates to the notion of *universal composability* [7]. This is a formal methods inspired concept, of conditions which would allow compositional reasoning, in the sense that one could substitute the ideal model specification of a scheme instead of an implementation when reasoning about protocols built using the scheme. For commitment, it has been proved that an implementation satisfying this strong compositional notion of correctness cannot exist [8, 10].

3 Computational Correctness

Notions of statistical and computational correctness ("provable security") in cryptography are built on the idea that breaking a system may only need to be hard and unlikely, rather than theoretically impossible. For commitment this is the best that can be achieved; for other applications it may be more realistic and efficient. Attack models then include explicit probabilism, reflecting situations like it always being possible to correctly guess an encryption key, though with a very small probability only. Informally, the computational version of the hiding property is as follows. Let the randomising argument to *commit* have length n bits. Then, for any probabilistic algorithm with time complexity polynomial in n, the probability of it distinguishing outputs of *commit* for bit 0 with uniformly chosen randomising input, and similar for bit 1, is negligible (i.e., smaller in the limit than 1 divided by any positive polynomial). A similar definition exists for binding, and commitment schemes have been defined in the literature that achieve one security property in the absolute sense, and the other in the computational sense described here.

For more details of this and the reconstruction of the commitment primitive from a formal methods perspective, see the draft paper [4]. Clearly there is a connection between the notion of approximate correctness described here, and our notion of approximate refinement [6] – the draft paper also gives more details of that.

4 Towards Correctness by Construction

For a slightly more extensive discussion, see [5]. The state of the art for cryptographic protocols is that verification is done post-hoc only, with very little machine support. Proofs for provable security are hard: notations and theories such as probability theory and complexity theory do not have strong algebraic traditions or properties. In particular, proofs over "all probabilistic polynomial algorithms" have no induction principles to support them, so are typically carried out by contradiction and probabilistic reduction. ("If we had an efficient algorithm to break this cryptographic scheme, this could be used to solve a known difficult number-theoretic problem.") Promising approaches in this area include universal composability discussed above, and "game hopping" [2] supported by the CryptoVerif proving system [3].

On the formal methods side, recent developments in probabilistic refinement [13, 16], action refinement [1], and secrecy-preserving refinement [15, 14] contribute to solving the problem of finding refinement relations that will one day allow us to derive cryptographic protocols from abstract specifications, providing correctness by construction.

References

1. R. Banach and G. Schellhorn. On the refinement of atomic actions. *ENTCS*, 201:3–30, 2008. Proceedings BCS-FACS Refinement Workshop 2007.
2. M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In S. Vaudenay, editor, *EUROCRYPT*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426. Springer, 2006.
3. Bruno Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4):193–207, October–December 2008. Special issue IEEE Symposium on Security and Privacy 2006. Electronic version available at http://doi.ieeecomputersociety.org/10.1109/TDSC.2007.1005.
4. E.A. Boiten. Commitment: A challenge for formal methods, 2008. Draft paper, www.cs.kent.ac.uk/~eab2/crypto/commit.pdf.
5. E.A. Boiten. From ABZ to cryptography (abstract). In E. Börger, M. Butler, J.P. Bowen, and P. Boca, editors, *ABZ 2008*, volume 5238 of *LNCS*, page 353. Springer, September 2008. www.cs.kent.ac.uk/~eab2/crypto/abz.pdf.
6. E.A. Boiten and J. Derrick. Formal program development with approximations. In H. Treharne, S. King, M. Henson, and S. Schneider, editors, *ZB 2005*, volume 3455 of *Lecture Notes in Computer Science*, pages 375–393. Springer, 2005.
7. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000.
8. R. Canetti and M. Fischlin. Universally composable commitments. In J. Kilian, editor, *CRYPTO*, volume 2139 of *Lecture Notes in Computer Science*, pages 19–40. Springer, 2001.
9. R. Cramer and I. Damgård. Multiparty computation, an introduction. Material for a course on Cryptologic Protocol Theory, Aarhus University, `www.daimi.au.dk/~ivan/CPT.html` (last checked April 17, 2007), 2004.
10. A. Datta, A. Derek, J.C. Mitchell, A. Ramanathan, and A. Scedrov. Games and the impossibility of realizable ideal functionality. In S. Halevi and T. Rabin, editors, *TCC*, volume 3876 of *Lecture Notes in Computer Science*, pages 360–379. Springer, 2006.
11. S. Delaune, S. Kremer, and M.D. Ryan. Verifying privacy-type properties of electronic voting protocols. *Journal of Computer Security*, 17(4):435–487, 2009.
12. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC*, pages 218–229. ACM, 1987.
13. A. McIver and C. Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems.* Springer, 2004.
14. Carroll Morgan. How to brew-up a refinement ordering. *ENTCS*, 2009. Proceedings of Refine 2009, to appear.
15. Carroll Morgan. The shadow knows: Refinement and security in sequential programs. *Sci. Comput. Program.*, 74(8):629–653, 2009.
16. M. Ying. Reasoning about probabilistic sequential programs in a probabilistic logic. *Acta Informatica*, 39(5):315–389, 2003.

A roadmap for the Rodin toolset*

Jean-Raymond Abrial[1], Michael Butler[2], Stefan Hallerstede[3], and Laurent Voisin[4]

[1] Independent Consultant, France
[2] University of Southampton, UK
[3] Heinrich-Heine-Universität Düsseldorf, Germany
[4] Systerel, France

1 Event-B and the Rodin Platform

Event-B is a formal method for system-level modelling and analysis [?]. Key features of Event-B are the use of set theory as a modelling notation, the use of refinement to represent systems at different abstraction levels and the use of mathematical proof to verify consistency between refinement levels.

The Rodin Platform[5] [2] is an Eclipse-based [3] toolset for Event-B that provides effective support for refinement and mathematical proof. Keep aspects of the are

- support for abstract modelling in Event-B
- support for refinement proof
- extensibility
- open source

To support modelliing and refinement proofs Rodin contains a modelling database surrounded by various plug-ins: a static checker, a proof obligation generator, automated and interactive provers. The extensibility of the platform has allowed for the integration of various plug-ins such as a model-checker (ProB), animators, a UML-B transformer and a LaTeX generator. The database approach provides great flexibility, allowing the tool to be extended and adapted easily. It also facilitates incremental development and analysis of models. The platform is open source, contributes to the Eclipse framework and uses the Eclipse extension mechanisms to enable the integration of plug-ins.

For a fuller description of the Rodin tool see [1].

2 Roadmap

In its present form, Rodin provides a powerful and effective toolset for Event-B development and it has been validated by means of numerous medium-sized case studies. Naturally further improvements and extensions are required in order to improve the productivity of users further and in order to scale the application of the toolset to large industrial-scale developments. We outline the main extensions to Rodin that we have planned for a four year time frame. The outline descriptions of these planned extensions are grouped into sections 2.1 to 2.5 as follows.

* The continued development of the Rodin toolset is funded by the EU research project ICT 214158 DEPLOY (Industrial deployment of system engineering methods providing high dependability and productivity) www.deploy-project.eu. The toolset was originally developed as part of the project IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems).

[5] Available from www.event-b.org

2.1 Model construction

Rodin provides a structured editor for constructing and modifying models stored in the database. As mentioned above, this facilitates easy extension as well as incremental development and analysis of models. Rodin needs further improvement to make it easier to perform standard editing tasks such as text search, copy/paste and undo/redo. Rodin will be extended to provide refactoring facilities, such as identifier renaming, that can be applied not just to models but to proof obligations, proofs and other forms of elements. Better support for browsing refinement links between models will be provided, for example, allowing the refinements and abstractions of events to be followed down or up a refinement chain.

2.2 Scaling

Event extension: In many Event-B developments it is common to perform superposition refinement where existing model features are maintained and additional features are added (e.g., additional variables, invariants, events and additional guards and actions for existing events). Currently events can be inherited as a whole but not extended. Rodin will support event extension (or superposition) where only the additional features are defined in a refined event and the existing features are inherited.

Composition and decomposition: Composition and decomposition of models is essential for scalability. There are plans to support two styles of composition for Event-B in Rodin:

Shared variable composition Sub-models interact via shared variables
Shared event composition Sub-models interact via synchronisation over events

Rodin will be extended to provide support for composing models as well as decomposing models according to these styles. The proof obligation generator will be extended to enable independent refinement of sub-models.

Team-based development: Support for composition and decomposition will go some way towards enabling team-based development. But there will still be situations where a team needs to access a common set of models. Rodin will be extended to support concurrent modification of developments by providing viewing of change conflicts and automated merge of changes. It will provide support for version control. Support to analyse the impact of multiple user modifications on proof will be investigated.

2.3 Extending the proof obligations and theory

Proof obligations: Event-B models will be extended to include external variables. The proof obligation for such variables is that they must be preserved via a functional gluing invariant between abstract and concrete external variables. Other forms of proof obligations will also be added to support different paradigms (concurrent, distributed, sequential systems). These include proof obligations for preservation of event enabledness and richer variant structures(such as pointwise ordering and lexicographic ordering) for convergence proof obligations.

Mathematical extensions: Rodin will be extended to support richer types such as record structures and user-defined data types including inductive data types. Appropriate automated and interactive proof support for richer types will be investigated and provded. Higher order provers should enable proof support for inductive datatypes. Users will be able to define operators of polymorphic type (but not use operator overloading) as well as parameterised predicate definitions. Support for disjointness constraints will be added.

2.4 Proof and model checking

Rodin provides an open architecture for proof in the form of a proof manager that can use a range of provers to discharge proofs and sub-proofs. The existing automated provers will be extended with more powerful decision procedures. The use of existing first order and higher order automated provers will be investigated. As mentioned already, higher order provers should enable proof support for inductive datatypes. The possibility of exploiting automated techniques such as SMT and SAT will be investigated. The facilities of the ProB model checker will be fully integrated into Rodin.

2.5 Animation

Prototype animation plug-ins already exist. The animation facilities will be extended to allow for greater automation of large animations to support regression testing of models. A clear API to the animation will be provided to allow for easy integration with graphical animation tools.

2.6 Process and productivity

Requirements Handling and Traceability: The interplay between informal requirements and formal modelling is crucial in system development and needs better tool support. Facilities for constructing structured requirements documents and for building links between informal and formal elements will be added to Rodin. These will support traceability between requirements and formal models. Support for recording validation of these links and for managing consistency under change to requirements and to formal models will be provided.

Document management: Currently, the B2Latex plug-in for Rodin generates a LaTeX version of an Event-B model. The structure of the document follows the structure of the model. For proper document generation tool support will be provided whereby users dictate the order in which parts of the model are presented. They should be able to write a document, structured according to their needs that includes parts of an Event-B project and that is automatically kept in synchrony with the models.

Automated model generation: Automatic generation of refinements will be investigated and appropriate tool support provided. More general modelling and refinement patterns, enabling greater reuse of modelling and refinement idioms, will be investigated and tool support provided. Code generation from models will be investigated. An indirect route for achieving code generation will be to generated classical B and use the existing code generators for classical B.

3 Tool development procedures

We are promoting a rigorous approach to the development of Rodin. Key features of Rodin, e.g., the static checker and the proof obligation generator, were specified formally before being implemented. Test procedures are developed in tandem with implementing. We are setting up rigorous specification and code review procedures. The development of new features should also follow this approach.

Many of the extensions listed above will first be implement as separate Eclipse plug-ins. When their general value and quality is assured, they will be incorporated into the platform release.

The management of platform release versions will be coordinated amongst the platform and plug-in developers. Facilities for importing existing developments into newer versions of Rodin will be provided. Support documentation and tutorial material for tool users and plug-in developers will continue to be improved and updated.

4 Concluding

Many of the Rodin extensions outlined above will be implemented as part of the DEPLOY project. However, we welcome support from other researchers and tool developers in elaborating and realising the roadmap. Furthermore, we anticipate that researchers will investigate and implement Rodin extensions that are not identified in our roadmap.

References

1. Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: An open toolset for modelling and reasoning in Event-B. http://deploy-eprints.ecs.soton.ac.uk/130/, 2009.
2. Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. An open extensible tool environment for Event-B. In Z. Liu and J. He, editors, *ICFEM 2006*, volume 4260, pages 588–605. Springer, 2006.
3. Erich Gamma and Kent Beck. *Contributing to Eclipse*. Addison Wesley, 2003.

Acknowledgements

We would like to thank the many contributors to the Rodin platform and plug-ins including Nicolas Beauger, Jens Bendisposto, Mathieu Clabaut, Kriangsak Damchoom, Andy Edmunds, Fabian Fritz, Andreas Fürst, Thai Son Hoang, Alexei Iliasov, Michael Jastram, Thierry Lecomte, Michael Leuschel, Issam Maamria, Christophe Métayer, Farhad Mehta, Carine Pascal, Antoine Requet, Abdolbaghi Rezazadeh, Mar Yah Said, Matthias Schmalz, Renato Silva, Colin Snook, Francois Terrier. In addition, Dominique Cansell and Cliff Jones provided valuable feedback on the design of the tool.

Challenges in Applying Formal Methods
An SME View

Mathieu Clabaut

Systerel, Aix-en-Provence, France
mathieu.clabaut@systerel.fr

Abstract. This paper outlines past and foreseen challenges in applying both *classical B* and *event B* to design safety related systems in an SME.

1 Activities

Systerel is an SME doing mainly fixed-price activities in the domain of real-time systems and safety-critical software ranging from on-board train speed controller to track-side automatic train protection. Systerel uses formal methods for some of these developments, most of which have been done with *classical B*.

Nowadays, we are doing some analysis and design of safety-critical systems with the help of *event B* while still developing safety-critical real-time embedded software.

2 Past Challenges: Classical B

2.1 At the Beginning

The earlier challenges met when first using the *classical B* method were mainly the following ones:

Planning proof workload was impossible to estimate, which was not in favour of formal method, since the workload of tests used in classical methods was far more easy to estimate.

Monitoring progress was very difficult. How to estimate the proof progress when a single unprovable proof obligation may require a break down of the whole architecture of your software?

Customer evaluation of formal method contribution was not always positive, notably with respect to the development costs: *Where is the return on investment?*

2.2 Now

Planning and monitoring is a bit easier as we have gained experience and domain knowledge. We thus do know where to apply formal methods and where not to.

We are now able to precisely draw the limits of software responsibility with respect to safety and then narrow down the formal properties which are to be modelled with B.

We also have collated somewhat usable metrics and rules of thumbs, and while still having to work until there is no more proof obligations left, we have a better confidence on our modelling principles for a given domain.

We claim better model designs which allow for simpler proofs and better reuse. The fact that a *model is designed to be proved* is now the core of our process.

Customer evaluation is now backed by some years of experience and nowadays, most of our customers reckon quality gains (albeit still internally disputed for some others). For those who master the formal process, using *classical B* is *less expensive than traditional safety-critical developments.*

2.3 Process

Our *classical B* process is composed of:

- Software requirements document, written by *small teams.*
- Formal design and proof done within *small teams.*
- Formal coding and proof done within *large teams.*
- Translation and compilation (automated).
- Integration and functional testing done within *large teams.*

This process is suitable for big industrial software with large teams of developers.

3 Today and Future Challenges: Event B

Nowadays, we are going a step further with the use of formal methods, by using *event B* to help the design of safety-critical systems.

3.1 Process

The starting point of our process, backed by our *classical B* experience, is that *the model has to be designed for proving it* which also implies that *event B* may not be convincing for analysis of existing systems (so called retro-modelling)[1] and may only show its usefulness for ab initio design, where one is allowed to tweak the design in order to *reject complexity* (with respect to safety proof).

The process in use today is made of the following steps:

- System requirement document (*small team*)
- Refinement plan (*small team*)

[1] But still more flexible than *classical B* where architectural constraints make retro-design in B even more difficult.

– Modelling / proving (*small team — basically, one person*)
– Testing (*To be defined...*)

The scalability of such a process is poor. New tools and concepts will definitely be needed to manage complexity and to allow team work on big models.

It is to be noted that despite these difficulties, *event B* proves to be very useful in designing a safety system.

3.2 Project Management

Our main challenge with the use of *event B* and the companion tool *Rodin* are about project management and convey the fact that:

– the desired system design is not known at the beginning of the process,
– heavy model refactoring are thus common.

Planning and monitoring is then a difficult task. How one can estimate the design and proof effort? How can one devise measurement to monitor design and proof progress?

Relation with customers on such a base are also not easy: reporting is no convincing given the lack of monitoring capabilities. The customer evaluation is then done on a poor basis: *"a model has been done, and then? ... "*

The ROI justification of the modelling work is still difficult, even if we are utterly convinced of its usefulness.

4 What's Needed

Roughly stated, the main needs are the following one:

1. Master the modelling process.
2. Reduce costs and delays.
3. Convince customers.

For this, we definitely need tools and methods for tackling model complexity (team based development, decomposition, mathematical extensions, patterns or automated refinements,...), for improving planning and monitoring and for improving customer evaluation and appreciation.

Some interest was also expressed by the space industry in being able to formalize requirements and the companion engineering process with the help of refinement-based formal methods. Maybe an issue to be tackled in the coming years?

Linearizability - deriving and mechanically verifying proof obligations

John Derrick[1], Gerhard Schellhorn[2], and Heike Wehrheim[3]

[1]Department of Computing, University of Sheffield, Sheffield, UK
J.Derrick@dcs.shef.ac.uk

[2]Universität Augsburg, Institut für Informatik, 86135 Augsburg, Germany
schellhorn@informatik.uni-augsburg.de

[3]Universität Paderborn, Institut für Informatik, 33098 Paderborn, Germany
wehrheim@uni-paderborn.de

Abstract. Concurrent objects are inherently complex to verify. In the late 80s and early 90s, Herlihy and Wing proposed *linearizability* as a correctness condition for concurrent objects, which - once proven - allows to reason about concurrent objects using pre- and postconditions only. A concurrent object is linearizable if all of its operations appear to take effect instantaneously some time between their invocation and return. In this paper we propose simulation-based proof conditions for linearizability and apply them to two concurrent implementations, a lock-free stack and a set with lock-coupling. Similar to other approaches, we employ a theorem prover (here, KIV) to mechanize our proofs. Contrary to other approaches, we also use the prover to mechanically check that our proof obligations actually guarantee linearizability. This check employs the original ideas of Herlihy and Wing of verifying linearizability via *possibilities*.

Keywords: Z, refinement, concurrent access, linearizability, non-atomic refinement, theorem proving, KIV.

1 Introduction

In 1987, Herlihy and Wing [10, 11] introduced the notion of linearizability as a correctness criterion for concurrent objects. Concurrent object are data structures (like sets, stacks, queues) shared by parallel processes. Implementations of concurrent objects usually apply fine-grained synchronisation schemes for access as to allow for a high degree of concurrency. Such synchronisation schemes might employ locking of single elements in the object (like a node in a linked list) but might as well completely dispose with locking. Such concurrent algorithms are intrinsically difficult to prove correct, and the down-side of the performance gain from permitting concurrency is the much harder verification problem: how can one verify that the implementation of a concurrent object is correct? Linearizability does not fix the "how" but defines "what" needs to be proven for correctness. Like serializability for database transactions, linearizability permits one to view operations on concurrent objects as though they occur in some sequential order [11]:

> Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response.

Unlike serializability, linearizability is however a *local* property, the proof of it can be carried out on individual objects. Once linearizability of a concurrent object has been shown, reasoning about it can be done in terms of the pre- and postconditions of operations alone.

Recently, a number of distributed algorithms have been shown to be linearizable, these works including [9, 4, 1, 12, 15, 2, 3] as well as our own [7, 8]. In these works, a number of different techniques are employed ranging from shape analysis or separation logic over rely-guarantee reasoning to simulation-based methods. The simulation-based methods show that an abstraction (or simulation or refinement) relation exists between the abstract specification and the concurrent implementation. The proofs employed in the above range from manual over partly theorem prover supported (e.g., with PVS) to automatic ones. Apart from our own [8], all of these papers, however, only argue at an informal level that their proof technique actually implies the original linearizability criterion of [11]. In this paper our aim is to give a formal theory that *relates* refinement theory and linearizability, and which has furthermore been fully mechanized using the interactive theorem prover KIV [14]. Two cases studies have been used to validate the approach: a concurrent implementation of a set and of a stack. These two examples differ in that while the first algorithm (taken from [15]) employs locks (on a small-grained scale), the second one [13] is a lock-free, non-blocking algorithm.

$$\begin{tikzcd}
\text{Linearizability} \\
\text{Data Refinement} \arrow[u, Rightarrow] \\
\text{Forward Simulation using Possibilities} \arrow[u, Rightarrow] \\
\text{Local proof obligations} \arrow[u, Rightarrow]
\end{tikzcd}$$

Fig. 1. The structure of the generic part of the proof of correctness

Our methodology of proving linearizability consists of two parts: A generic part which derives proof obligations, and an application specific part instantiating these to case studies.

The generic part of the methodology needs to show that our proof obligations guarantee linearizability. This is done in several steps as illustrated in Fig. 1. As a first step, we show that linearizability can be seen as a specialised instance of *data refinement* [5, 6]. Data refinement relates specifications on different levels of abstraction (abstract and concrete data types) with the aim of guaranteeing substitutability. It requires that any permitted observations of the concrete type are consistent with those that could be made of the abstract type. Normally these observations are the outputs of the data type, however, by extending the data types with *histories* (of operation executions) and adapting outputs to generate histories, we define a notion of data refinement which implies linearizability.

Data refinement is usually proven in a step-wise manner via forward or backward simulations [6]. To show that our proof obligations imply linearizability, we construct a forward simulation relation between the extended concrete and abstract data types out of our proof obligations. This turned out to be the hard part in the proof, and here we, in fact, re-used the original alternative proof technique for linearizability of Herlihy and Wing, that is of employing *possibilities*. This simulation relation then finally needs to be shown to indeed fulfill the conditions for forward simulations. All definitions and proof steps for the generic part have been formalised and mechanically verified using the theorem prover KIV.

In [] the application of the methodology to specific algortihms is illustrated by considering two case studies. The first is an implementation of a stack via a linked list [13]. It implements atomic push and pop operations as instructions to read, write and update local variables as well as the stack contents. The only atomic operations are the reading and writing of variables and an atomic *compare-and-swap* (atomically comparing the values of two variables plus setting a variable). This provides for a non-blocking algorithm on the stack, requiring no locking scheme at all. The second example, taken from [15], is an implementation of a set, again as a linked list but this time employing a lock-coupling scheme on consecutive nodes in the list. Besides reading and writing of variables, here locking (i.e. testing and setting a lock) is an atomic step. We show that our proof obligations hold on both examples, and again all proofs have been mechanized within KIV.

The methodology we have derived is sufficiently rich to be applicable to a number of algorithms. There are, however, a range of algorithms that are outside its range, and the extension of the methodology to them forms the basis of our continuing programme of work.

The first extension involves the location of non-determinism in an algorithm, and how that can be resolved in any implementation. This is pertinent since although forward simulations can be used to verify the majority of naturally occurring algorithms, in cases where the non-determinism is postponed in an implementation, *backward* simulations need to be used (and, in general, both can be needed). There are known cases of lock-free algorithms that need backward simulations, and one direction for this work is to provide the extension to this case.

Harder still are algorithms were the location of the linearization points is not static. Specifically, the linearization point of one operation can only be determined by the current progression of other operations in the data type. At present our formalisation of the *status* function cannot be used in such circumstances and we need to seek a tractable formalisation that can cope with all the generality.

References

1. J.-R. Abrial and D. Cansell. Formal Construction of a Non-blocking Concurrent Queue Algorithm (a Case Study in Atomicity). *Journal of Universal Computer Science*, 11(5):744–770, 2005.
2. Daphna Amit, Noam Rinetzky, Thomas W. Reps, Mooly Sagiv, and Eran Yahav. Comparison under abstraction for verifying linearizability. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 477–490. Springer, 2007.
3. Cristiano Calcagno, Matthew Parkinson, and Viktor Vafeiadis. Modular safety checking for fine-grained concurrency. In *SAS 2007*, volume 4634 of *LNCS*, pages 233–238. Springer, 2007.
4. R. Colvin, S. Doherty, and L. Groves. Verifying concurrent data structures by simulation. *ENTCS*, 137:93–110, 2005.
5. W. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.
6. J. Derrick and E. Boiten. *Refinement in Z and Object-Z: Foundations and Advanced Applications*. Springer, May 2001.
7. J. Derrick, G. Schellhorn, and H. Wehrheim. Proving linearizability via non-atomic refinement. In J. Davies and J. Gibbons, editors, *IFM*, volume 4591 of *Lecture Notes in Computer Science*, pages 195–214. Springer, 2007.
8. J. Derrick, G. Schellhorn, and H. Wehrheim. Mechanizing a correctness proof for a lock-free concurrent stack. In G. Barthe and F. de Boer, editors, *FMOODS 2008: Formal methods for open, object-based distributed systems*, volume 5051 of *LNCS*, pages 78–95. Springer, 2008.
9. S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In *FORTE 2004*, volume 3235 of *LNCS*, pages 97–114, 2004.
10. M. Herlihy and J. Wing. Axioms for concurrent objects. In *ACM Symposium on Principle of Programming Languages*, pages 13–26. ACM, 1987.
11. M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
12. Wim H. Hesselink. A criterion for atomicity revisited. *Acta Inf.*, 44(2):123–151, 2007.
13. Maged M. Michael and Michael L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared — memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.
14. W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications*, volume II: Systems and Implementation Techniques, chapter 1: Interactive Theorem Proving, pages 13 – 39. Kluwer Academic Publishers, Dordrecht, 1998.

15. Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 129–136, New York, NY, USA, 2006. ACM.

Formal modelling and refinement of OS kernels

Leo Freitas

Department of Computer Science, University of York, YO10 5DD, York, UK

Abstract. During the POSIX pilot project on verified flash file stores, we realised the need for an underlying formalised kernel. This motivated work on modelling a separation real-time kernel for embedded devices. The work is also related to other pilot projects in the grand challenge, such as the verification of the real-time operating systems (e.g., FreeRTOS). The key difference from FreeRTOS is that, instead of trying to verify such successful product's code, we are modelling the kernel from the requirements down through to C code using Z and the refinement calculus. In this process, we found many interesting lemmas and general data structures that are useful for other domains and pilot projects. We will summarise here on the current state of this work and where is it going in the near future.

Keywords: Grand challenge; OS kernels; theorem proving; verification

1 Introduction

Formal methods for software development allow the construction of an accurate characterisation of a problem domain that is firmly based on mathematics; by applying standard mathematical analyses, these methods can be used to prove the correctness of systems. The survey presented in [26] describes over 60 industrial projects, and discusses the effect formal methods have on time, cost, and quality. It shows that with tools backed by mature theory, formal methods are becoming cost effective, and their use is easier to justify, not as an academic or legal requirement, but as part of a business case. These recent advances in theory and tool support have inspired industrial and academic researchers to join up in an international Grand Challenge (GC) in Verified Software [12, 24]. Work has started with the creation of a Verified Software Repository (VSR) with two principal aims: (i) the construction of verified software components; and (ii) industrial-scale verification experiments to drive future research in the development of theory and tool support [2].

This extended abstract paper summarises an experiment undertaken as part of a pilot project on verifying operating system (OS) kernels within the GC. It explores the mechanisation of proofs of correctness of the formal specification and design of critical parts of operating systems kernels for real-time embedded systems constructed by Craig [5], such as its key data types [10]. This is not to be confused with another pilot project: the mechanisation of FreeRTOS [6], the real-time operating system. One key difference in Craig's kernel is the use of refinement from an abstract specification down to code. We have already mechanised the abstract parts of the scheduler and its scheduling policy [27, 23]. Since these models are all in the Z notation [20], it naturally follows that we use a Z tool, and for us that is the Z/Eves theorem prover (v. 2.4) [19, 18]. The choice is based on its ease of use, long previous experience and, most importantly for involving students, its gentle learning curve.

Related work. In 2006, the first VSR pilot project was undertaken on the verification of the Mondex smart card [21] to ITSEC Level 6 (Common Criteria Level 7) [14]. The work is reported in [16], where a summary of Mondex and its original development and certification are described [16, p.5–19]. The experiment mechanised the original manual proofs in Alloy [16, p.21–39], ASM [16, p.41–59], Event-B [16, p.61–77], OCL [16, p.79–100], π-calculus [15], Raise [16, p.101–116], and Z [16, p.117–139]. A second pilot project on POSIX compliant flash file stores followed [11]. A domain model with widely used terminology and well-understood requirements is needed, and we have based our mechanised domain model based on the formal refinement of OS kernel designs [5].

There are two other related GC pilot projects: FreeRTOS [6] and the Microsoft Hypervisor [4]. FreeRTOS is an open-source real-time embedded operating system written in pointer-rich C, and it does not have a specification, making it an attractive topic for research in formal analysis and top-down development. The extensive use of pointers offers two complementary challenges: (i) the annotation of the code with suitable assertions and the verification of the code against these assertions; and (ii) the top-down development of the code, starting from a suitable specification of its abstract behaviour. The goal of the Microsoft Hypervisor Verification Project is to develop an industrially viable verification methodology for low-level code, and to use this methodology to verify the functional correctness of the Microsoft Hypervisor [17]. The hypervisor is a 60kLoC C and assembler program that turns a multi-processor (MP) x64 machine into a number of virtual MP x64 machines.

2 Verified OS kernels pilot project

An OS kernel is a central component of most operating systems, providing an interface to the management of hardware and software resources, including memory, processors, and I/O devices. It offers this interface to application processes through inter-process communication mechanisms and system calls. Among its features, the most important are: low-level scheduling of processes; inter-process communication; process synchronisation; context switching; manipulation of process control blocks; hardware interrupt handling; process creation and destruction; *etc.*. Kernel development has a reputation for being a very difficult and complex programming task for two prime reasons. First, every computing system requires the OS kernel to provide correct functionality and good performance. Second, because the kernel cannot make (direct) use of the abstractions it provides (*e.g.,* processes, semaphores, *etc.*), which would make higher-level programming of embedded and real-time systems easier.

Our pilot project is inspired by Iain Craig's book on the formal refinement of OS kernels [5]. The objectives are to demonstrate feasibility of top-down OS kernel development using formal specification and verification, with refinement down to a C implementation. Craig uses the Z notation [20, 25, 13] for specification and refinement, and recording correctness arguments in hand-written proofs. Our pilot project investigates the tractability of mechanising all the models in each kernel development, including formalising all proofs. A key principle is to retain these models as far as possible, making changes only for correctness, not for easing the task of mechanisation.

Part of this investigation involves constructing prototype tool chains for the development process from specification through design and down to code. For the specification and verification we use Z theorem provers like Z/Eves [19]. Data refinement [25] links the abstract specification to a concrete design that is closer to code, and we again use a Z theorem prover. After that, we use the Z refinement calculus (ZRC) [3] to go down to the guarded command language. The invariants and pre- / postconditions for each programming statement are then converted to a formal annotation language for C, such as Spec# [1]. Finally, tools like Boogie and the Microsoft Verified C Compiler can be used to perform static and partial correctness analysis. All results, including models, lemmas, papers, tools, *etc.* are being curated in the VSR.

The pilot project is currently in an exploratory phase, having mechanised the whole of the simple kernel [27], and various parts of a separation kernel [23]. We have found some interesting issues, including missing and hidden invariants. Although

Craig's models have great insight from an OS engineer in necessary underlying data types, a series of mistakes are introduced, both clerical and on more substantial design decisions. On the other hand, despite these errors, the work is carried out using the refinement calculus [3] and goes down to a real ANSI-C implementation running on embedded processors, like the Intel IA32 architecture. With this work, our attempt is to straight up all that effort more rigourously.

In the kernel, processes can synchronise using counting semaphores, FIFO queues, and so on. They are defined as separate mathematical data type, later refined to a chain of process identifiers [10]. Message passing enable processes to exchange messages, where the discipline that *receivers wait* and *senders retry* is observed. System calls can be used to: create or terminate processes; retrieve process identifiers; send or receive synchronous messages; allocate and release semaphores; put processes to wait or to sleep, as well as signal them; *etc.*

The aim is to gain a more intimate understanding of the invariants for the kernel's basic data structures that are relevant to scheduler design. We take a step back from OS kernel design and verify its basic data structures to see which invariants are fundamental, and which can be relaxed and proved as properties instead. We have reports related to this paper with all definitions and proofs that can be found in [7–9].

3 Interesting lessons

Mechanisation has led to a deeper understanding of the kernel's components. We attempted simplifications by weakening some of the invariants that could have been derived as properties, in order to make our proofs simpler, but without compromising the specification. The lack of mechanisation in [5] led to missing invariants and other errors at the most crucial data structure in the kernel scheduler: its priority ready queue. This exercise shows the importance of tools in formal modelling in general, and theorem proving in particular, when one wants to provide greater levels of assurance. In practice, kernels use a matrix of priorities per sequences of identifiers, hence many problems of having a flat sequence might be simplified. This could be a good candidate for data refinement of the priority ready queue.

In fairness to Craig's original model, despite the mistakes mentioned, the sheer effort undertaken was considerable and worthwhile. His expertise in operating systems implementation is clear through the book. Luckily, many of the mistakes were consistent and easy to spot, which makes correcting them a simple task. And that is despite its serious consequences at times, like loosing scheduled processes FIFO ordering when priorities are the same. Overall, the exercise has proved worthwhile in establishing a solid foundation upon which one can build the top-level kernel components.

Overall, we tried to strike a balance between reusing good parts of the models and remodelling from scratch based on the intended goals to capture the underlying requirements. The motivation for doing that is to save important invariants already discovered and modelled by a domain expert.

3.1 Going back to the scheduler design

Besides the scheduler and its data structures, other familiar OS components are modelled, such as a global semaphore table, a synchronous message passing system, a process sleeping mechanism, and so on.

4 Conclusions

The Grand Challenge's pilot projects help us to learn the best ways to model various application domains and how to verify those models. The intention is to make it easier for the next team who want to work in the application domain. In that direction, a series of data types and useful lemmas are needed if one is to make progress in tackling the central problems with OS kernel scheduling.

The experiments that started with mechanising the refinement of simple OS kernel schedulers led to the mechanisation of a set of abstract data types useful for this kind of modelling in general. This in itself instigated thinking about general properties for injective functions, transitive closure, sequences, and started few reports [7–9] that are good candidates to become part of the VSR as reusable mathematical data types. In more detail: we already have most (95%) of [5, Ch. 3], which is discussed in [27, 10] and in here. Our model contains a series of declarations from Craig's book, and mechanically verified theorems. The general theories contain well over 120 theorems about various mathematical data types [7]. As a result of this work, our library of general theorems grew by 20 theorems; for the declared types, we needed 34 automation lemmas. The three components have a total of 145 schemas, type, and axiomatic declarations, with 44 precondition proofs, and 16 lemmas about the data type's properties. In total, these proofs were discharged with around 1540 proof commands, of which more than 2/3 were trivial, whereas the reminder 1/3 was divided in either creative steps involving quantifier's witnesses, or knowledge on how the tool works.

We improved the specification of most data structures used in the simple kernel and in the Separation Kernel described in [5]. The work incurred mostly in identifying useful properties about these data types and their use, as well as calculating the preconditions for each operation, and later proving data refinement about them. This mechanisation enabled both a better understanding of the various data structures, and a clearer definition of the Separation Kernel's scheduler specification use of it. As its use in [5] had modelling errors on data types, as well as the missing error cases uncovered here, we believe this to be an important contribution in building theories for formal modelling of OS kernels.

Future work We are currently writing up reports about the various parts of the simple kernel, and how they are woven together. We have one MSc student working on the modelling of the Separation Kernel, as well as the refinement of the core data structures presented here. Colleagues from another research group in Brazil are working on combining all the kernel's components into a single top-level user-interface. With that in place, we will start to apply the refinement calculus to derive the kernel's code. Another approach is to go bottom-up from the already available C-code up towards the refined specifications.

Acknowledgements We are grateful to QinetiQ Malvern for its long term support for our research. We are also grateful for the National Physical Laboratory, a more recent supporter of our research. We are thankful to Juan Perna Osmar Santos, and Andrius Velykis for fruitful discussions about properties of various parts of the kernel. We are also grateful to Iain Craig for producing an useful account of the formal specification and refinement of OS kernels.

References

1. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *In CASSIS*, volume 3362 of *LNCS*. Springer, 2004.
2. Juan Bicarregui, Tony Hoare, and Jim Woodcock. The verified software repository: a step towards the verifying compiler. *FACJ*, 18(2):143–151, 2006.
3. Ana Cavalcanti. *A Refinement Calculus for Z.* PhD thesis, Oxford, 1997.
4. Ernie Cohen. Validating the Microsoft Hypervisor. In Jayadev Misra et al., editors, *14th International Symposium on Formal Methods*, volume 4085 of *LNCS*, pages 81–81, Hamilton Canada, 2006. FM 2006, Springer.
5. Iain Craig. *Formal Refinement of OS Kernels.* Springer, 1^{st} edition, 2007.
6. FreeRTOS. www.freertos.org.

7. Leo Freitas. Extended Z mathematical toolkit. Technical Report CRG13, University of York, April 2008.
8. Leo Freitas. Formal model of a reusable *Chain* data type. Technical Report CRG14, University of York, April 2008.
9. Leo Freitas. Mechanising data-types for Kernel design in Z. Technical Report CRG15, University of York, March 2009.
10. Leo Freitas and Jim Woodcock. A Chain Datatype in Z. *International Journal of Software Informatics*, 2009. In Press.
11. Leo Freitas, Jim Woodcock, and Andrew Buterfield. POSIX and the Verification Grand Challenge: a Roadmap. In *IEEE Proceedings of* 13^{th} *ICECCS, Belfast*, pages 153–162. IEEE, April 2008.
12. Tony Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.
13. ISO/IEC 13568. *Information Technology—Z Formal Specification Notation—Syntax, Type System and Semantics.* ISO/IEC, first edition, 2002.
14. ITSEC. Information technology security evaluation criteria: primary harmonised criteria. Technical Report COM(90) 314, Commission of the European Communities, Jun. 1991. version 1.2.
15. Cliff Jones and Ken Pierce. What can the π-calculus tell us about the mondex purse system? In *12th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 300–306, New Zealand, 2007. IEEE.
16. Cliff Jones and Jim Woodcock, editors. *Formal Aspects of Computing — special issue on Mondex*, volume 20:1. Springer, Jan. 2008.
17. Mike Neil et al. Hypervisor Top Level Functional Specification v0.83. Technical report, Microsoft Coorporation, Dec. 2007.
18. Mark Saaltink. *Z/Eves 2.0 Math. Toolkit.* ORA, Oct. 1999. TR-99-5493-05b.
19. Mark Saaltink. *Z/Eves 2.0 User's Guide.* ORA Canada, 1999. TR-99-5493-06a.
20. J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 1998.
21. Susan Stepney et al. An Electronic Purse: Specification, Refinement, and Proof. PRG 126, Oxford University, Jul. 2000.
22. Susan Stepney et al. A z patterns catalogue vol 1. Technical Report YCS-349, University of York, 2003.
23. Andrius Velykis Formal Modelling of Separation Kernels MSc in Software Engineering, University of York, 2009.
24. Jim Woodcock. First steps in the verified software grand challenge. *IEEE Computer*, 39(10):57–64, 2006.
25. Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof.* Prentice Hall, 1996.
26. Jim Woodcock et al. Formal methods: practice and experience. *ACM Computing Surveys*, 2009. In Press.
27. Jim Woodcock, Leo Freitas, and Ian Craig. A Verified Simple Operating System Kernel. In *Workshop on the Verified Software Repository as part of FM Symposium*, Turku, Finland, 2008. Formal Methods Europe.

An outline of a proposed system that learns from experts how to discharge proof obligations automatically *(extended abstract)*

Alan Bundy[1], Gudmund Grov[1,2], and Cliff B. Jones[2]

[1] School of Informatics, University of Edinburgh
{bundy,ggrov}@staffmail.ed.ac.uk
[2] School of Computing Science, University of Newcastle
cliff.jones@ncl.ac.uk

1 Introduction

Many formal methods are "posit and prove" where a designer posits a specification, and then seeks to justify it. This justification is in the form of proof obligations (POs), putative lemmas that need proof. A large proportion of these can be discharged by automatic theorem provers, but there are still some that require user interaction (typically of the order of 5-20%). Discharging these POs can become a bottleneck in the use of formal methods in practical applications, and there are two approaches to dealing with them:

- Follow a *modelling strategy*: change the model/abstraction to a strategy that simplifies the proofs, thus increasing the number of automatically discharged POs.
- Follow a *proof strategy*: accept the challenging POs, and define a strategy for discharging them. Such a strategy must be sufficiently abstract to be able to discharge "similar POs". This is the approach we will take in our AI4FM project. Our aim here is to increase the repertoire of techniques for the proof-strategy approach by learning from proof attempts.

In many cases where a correct PO has not been discharged, an expert can easily see how to complete a proof. We believe that it would be acceptable to rely on such expert intervention to do one proof if this would enable a system to discharge others "of the same form". Specifically, we hope to build a system that will learn enough from one proof attempt to improve the chances of proving "similar" results automatically. By "proof attempt" we include things like the order of the steps explored by the user (not just the chain of steps in the final proof). Thus it is central to our goal that we find *high-level* strategies capable of cutting down the search space in proofs. By separating information about data structures and approaches to different patterns of POs, a taxonomy begins to evolve. A proof (attempt) might be seen to use "generalise induction hypothesis" (e.g. adding an argument to accumulate values) in a specific proof about, say, sequences; a future use of the same PO might involve a more complicated tree data structure — but if it has an extended induction rule, the same strategy might work. So our hypothesis is: *we believe that it is possible (to devise a high-level strategy language for proofs and) to extract strategies from successful proofs that will facilitate automatic proofs of related POs.*

We have previously outlined the strategy language in [1]. In this paper we attempt to show how the shape of proofs of one result helps in another proof – and also how a proof can be reused in another example. In AI4FM we are not restricting ourselves to one particular formal method, which is illustrated here by providing both VDM [2] and Event-B [3] examples. In §2 we will indicate what we would like to achive; this is followed by an example of tool constraints §3; before we discuss our solution and conclude in §4.

2 Examples indicating scope

This section indicates what we would like to achieve. The examples (taken from the literature) are just sketched here but a technical report version of this short paper will have more detail in appendices.

2.1 A simple example

A trivial teaching example in [2] uses two disjoint sets to record"students who do exercises"; the state in VDM notation is:

$$\begin{array}{l} \mathit{Studx} :: y : \mathit{Id}\text{-}\mathbf{set} \\ \qquad\qquad n : \mathit{Id}\text{-}\mathbf{set} \\ \mathbf{inv}\ (\mathit{mk}\text{-}\mathit{Studx}(y,n)) \triangleq y \cap n = \{\,\} \end{array}$$

A representation of *Studx* could be

$$\mathit{Studx}1 = \mathit{Id} \xrightarrow{m} \{\mathbf{Y}, \mathbf{N}\}$$

VDM reification proofs require that one defines a retrieve function from the concrete to the abstract state. In this case:

$$\mathit{retr}\text{-}\mathit{Studx} : \mathit{Studx}1 \to \mathit{Studx}$$

$$\begin{array}{lcl} \mathit{retr}\text{-}\mathit{Studx}(m) & \triangleq & \mathit{mk}\text{-}\mathit{Studx}(\{n \mid n \in \mathbf{dom}\, m \wedge m(n) = \mathbf{Y}\}, \\ & & \qquad\qquad\qquad \{n \mid n \in \mathbf{dom}\, m \wedge m(n) = \mathbf{N}\}) \end{array}$$

In order to prove the refinement correct, there are two POs on the types and retrieve function. The first is that the retrieve function must be total — this is trivial. Secondly, the "adequacy" PO has the form:

$$\forall a \in A \cdot \exists r \in R \cdot \mathit{retr}(r) = a$$

This is not quite as simple because it depends on the invariant on *Studx*.

The proof obligations for each operation are (i) the domain rule:

$$\forall r \in R \cdot \mathit{pre}\text{-}A(\mathit{retr}(r)) \ \Rightarrow\ \mathit{pre}\text{-}R(r)$$

and (ii) the result rule:

$$\begin{array}{l} \forall \overleftharpoon{r}, r \in R \cdot \\ \qquad \mathit{pre}\text{-}A(\mathit{retr}(\overleftharpoon{r})) \wedge \mathit{post}\text{-}R(\overleftharpoon{r}, r) \ \Rightarrow\ \mathit{post}\text{-}A(\mathit{retr}(\overleftharpoon{r}), \mathit{retr}(r)) \end{array}$$

For such a simple example, we would expect all of these to be discharged automatically or with minimal hand intervention.

2.2 A partitioning reification

The example here is less trivial than that in Section 2.1; the interest is that the first reification step is remarkably similar to the earlier one and thus makes it possible to indicate what we hope to achieve in AI4FM.

This example is derived from Chapter 11 of [2]. The idea is that there is some equivalence relation over X and elements are partitioned according to this relation. Here, the underlying state is a set of disjoint sets. Thus, the state is a partition and is represented by the following type:

$$Part = (X\text{-}\mathbf{set})\text{-}\mathbf{set}$$

$$\mathbf{inv}\ (p) \triangleq \{\,\} \notin p \wedge \forall s, t \in p \cdot s = t \vee s \cap t = \{\,\}$$

If we choose to use a representation of:

$$Keyed = X \xrightarrow{m} Key$$

with the following retrieve function:

$$retr\text{-}Part : Keyed \to Part$$

$$retr\text{-}Part(m) \quad \triangleq \quad \{\{d \mid d \in \mathbf{dom}\, m \wedge m(d) = k\} \mid k \in \mathbf{rng}\, m\}$$

The first steps (adequacy etc.) of the reification proof are (not quite obvious) generalisations of what is done in Section 2.1. Our “ambition” is that the proofs from that section would provide a sufficient strategy to generalise to this case. So here we have an example of the reuse “pattern” of reification proofs being what we hope to achieve.

The actual Fisher/Galler representation can be thought of as a *Forest*:

$$Forest = X \xrightarrow{m} X$$

$$\mathbf{inv}\ (m) \triangleq \forall s \subseteq \mathbf{dom}\, m \cdot s \neq \{\,\} \;\Rightarrow\; \exists e \in s \cdot m(e) \notin s \vee m(e) = e$$

This representation follows the Fisher/Galler inspiration that elements are equal iff they have the same root.[3] In this representation a root is therefore a mapping to itself:

$$roots : (X \xrightarrow{m} X) \to X\text{-}\mathbf{set}$$

$$roots(m) \quad \triangleq \quad \{x \mid m(x) = x\}$$

The retrieve function is:

$$retr\text{-}Keyed : Forest \to Keyed$$

$$retr\text{-}Keyed(f) \quad \triangleq \quad \{x \mapsto root(x, f) \mid x \in \mathbf{dom}\, f\}$$

where

$$root : X \times Forest \to X$$

$$root(e, f) \quad \triangleq \quad \mathsf{if}\ e \in roots(f)\ \mathsf{then}\ e\ \mathsf{else}\ root(f(e), f)$$

[3] This representation deviates from [2] with respect to the representation of roots: the type in [2] is “total” ($\bigcup p = X$) and the roots are simply those elements $x \notin \mathbf{dom}\, p$ where $p \in Forest$.

These POs are more difficult than the ones on the first reification because the inductive structure of *Forest* is not obvious. The “ambition” here is that the proofs of the first operation PO would provide a model for those that follow (including those on the operations). This is an example where the way an expert approaches the first operation should hopefully carry over to proofs about further operations.

Another exercise would be to look at the direct development from *Part* to *Forest*.

3 Example indicating tool constraints

This section illustrates current tool constraints. The example is developed within Event-B [3] and the Rodin toolset[4]. It models door lock states (e.g. locked, locking, unlocking, unlocking) – and, similar to §2.1, disjoint sub-sets of door identifiers (type *DOORS*) are represented as a function *doors*, from the type of door identifier *DOORS* into a type *DOORSTATE*, which enumerates all possible states.

In Event-B, the correspondance between the description and actual representation is formalised by a *gluing invariant* (comparable to the retrieve function in VDM), and in this example the gluing invariant is a conjunction – where each conjunct has the “form” illustrated by “the door-locking state”:

$$\mathit{doors\_locking} = \mathit{doors}^{-1}[\{\mathit{DoorLocking}\}].$$

Note that *doors_locking* is a (abstract) set of door identifiers. We will focus on the event where a door enters this state, and the description/abstract (left) and representation/concrete (right) events (with all non-relevant parts stripped) becomes:

```
EVENT lock_door $\triangleq$
  ANY $d, \cdots$
  WHERE
    $d \in \mathit{doors\_locking}, \cdots$
  THEN
    $\mathit{doors\_locked} := \mathit{doors\_locked} \cup \{d\}$
    $\mathit{doors\_locking} := \mathit{doors\_locking} \setminus \{d\}$
    $\cdots$
```

```
EVENT lock_door $\triangleq$
  REFINES lock_door
  ANY $d, \cdots$
  WHERE
    $\mathit{doors}(d) = \mathit{DoorLocking}, \cdots$
  THEN
    $\mathit{doors}(d) := \mathit{DoorLocked}$
    $\cdots$
```

The gluing invariant must be preserved by all events, and the proof obligation expressing this for the invariant over *lock_door* is not discharged automatically by the Rodin automatic provers:

$$\begin{aligned}&\Delta, \mathit{doors\_locking} = \mathit{doors}^{-1}[\{\mathit{DoorLocking}\}] \vdash \\ &\qquad \mathit{doors\_locking} \setminus \{d\} = (\mathit{doors} \mathbin{\lhd\!\!\!-} \{d \mapsto \mathit{DoorLocked}\})^{-1}[\{\mathit{DoorLocking}\}]\end{aligned}$$

[4] see http://www.event-b.org.

The proof proceeds by first proving the following intermediate lemma (i.e. applying the cut rule):

$$(doors \lhd \{d \mapsto DoorLocked\})^{-1}[\{DoorLocking\}] = doors^{-1}[\{DoorLocking\}] \setminus \{d\}$$

Both the intermediate lemma and the main goal are then proved automatically by the Rodin predicate prover (PP).

Note that the intermediate lemma could have been avoided by substituting the *doors_locking* term with $doors^{-1}[\{DoorLocking\}]$ using the equality in the hypothesis (this technique is known as *weak fertilization* in rippling [4]). The new goal is then equal to the intermediate lemma (modulo reflexitivity). However, the PP tactic was not able to prove this.

4 Discussion

The major challenge in being able to reuse proof strategies, as illustrated in §2 and §3 (the strategy described in §3 was used on all "similar" gluing invariant POs), is to design a sufficiently general-purpose and robust strategy language so that it can deal with unanticipated proof plans and patches that experts will devise. If we knew in advance what these plans and patches would be, we could include them in the theorem prover, so that the problematic POs would be discharged and would not require expert attention. For example, the strategy used in §2.1 is reused in §2.2.

The strategy language will combine a high-level proof strategy with a "vocabulary" of terms that might be instantiated in the separate theories of data structures stored in the system. The meta-language employed in our rippling/induction proof-planning work [4] provides an existence proof for such a strategy language. We refer to [1] for more details of how we envisage this strategy language.

Acknowledgements This work has been supported by UK EPSRC grants EP/E005713/1 and EP/E035329/1.

References

1. Bundy, A., Grov, G., Jones, C.B.: Learning from experts to aid the automation of proof search. In O'Reilly, L., Roggenbach, M., eds.: AVoCS'09 – PreProceedings of the Ninth International Workshop on Automated Verification of Critical Systems. Technical Report of Computer Science CSR-2-2009, Swansea University, Wales, UK (2009) 229–232 To appear in the EASST electronic publications.
2. Jones, C.B.: Systematic Software Development using VDM. Second edn. Prentice Hall International (1990)
3. Abrial, J.R.: Modelling in Event-B: System and Software Engineering. Cambridge University Press (2009) To be published.
4. Bundy, A., Basin, D., Hutter, D., Ireland, A.: Rippling: Meta-level Guidance for Mathematical Reasoning. Volume 56 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (2005)

A (Small) Improvement of Event-B?

Stefan Hallerstede

University of Düsseldorf
Germany
stefan.hallerstede@wanadoo.fr

Abstract. Event-B and the Rodin tool use a number of simple techniques that make the modelling method around them effective in practical applications. We present two of these techniques, anticipation and witnesses. It is interesting how a couple of very simple techniques are so important for the method to work. Finally we propose a small enhancement of Event-B that would extend the use of witnesses.

1 Introduction

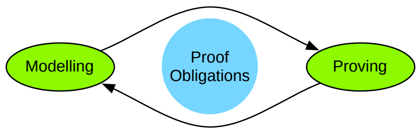


Fig. 1: Formal Reasoning in Event-B

We believe that the main purpose of modelling is reasoning, finding out why something works or why it does not. Reasoning should be formal in order to achieve a high degree of certainty about the corresponding claims we make. In the Event-B modelling method [1] reasoning is supported by formal proof. With each formal model we create a number of proof obligations is associated that, once discharged, establish certain properties of the model. Reasoning is not confined to carrying out a formal proof though. Whenever we fail to discharge some proof obligations, we modify the model and try to discharge the proof obligations of the modified model, and so on (Figure 1). This method of reasoning is supported by the Event-B formalism that has been designed to achieve a close correspondence between models and proof obligations. Event-B is intended for the modelling of complex systems. The large number of details of a complex system to be considered is introduced piecemeal by formal refinement [9]. We use refinement more as a technique to structure complex proofs, focusing less on preserving correctness along a sequence of models.

During the evolution of Event-B [1,3,2,4,5,6,7,8,12,15] many decisions and developments have been made to make Event-B an effective practical modelling method. Many of those are as simple as effective. In this paper we discuss two of them, the use of anticipated events [5] and of refinement witnesses [12].

In Section 2 we briefly discuss anticipation of events and in Section 3 we discuss refinement witnesses and suggest a small improvement of Event-B: to use witnesses also for non-deterministic assignments.[1]

[1] We do not present an introduction to Event-B. Introductions to Event-B can be found in the references mentioned above.

2 Anticipation

Anticipated events are described in [5] as a technique to couple events introduced during refinement with their variant and decouple them from variables. The approach solves the technical problem of finding a good ordering for a chain of refinements by relaxing the constraints on that order thus increasing the number of good orderings. Anticipated events can be used to avoid using lexicographic variants altogether,

$$(m \mapsto n) < (m' \mapsto n') \quad \Leftrightarrow \quad m < m' \vee (m = m' \wedge n < n') \quad .$$

Say, m is the variant of a machine M and n the variant of some refinement N of M. For an anticipated event f of M we would show that it does not increase m, that is, $m < m'$. Ultimately, event f has to be refined by a convergent event f in machine N, say. In machine N we have to show that f decreases the variant n, that is, $n < n'$. Following this technique we implicitly construct a lexicographic variant $(m \mapsto n)$, similar to the one shown above.

Anticipated events have turned out to be a very useful concept for modelling beyond the original purpose. Below are three examples of their practical use.
Counter abstraction: specification of an abstract timer. An abstract model of timer needs only express that after an arbitrary finite amount of time it will raise an alarm. The counter by which it could be implemented is irrelevant.

```
invariants                              anticipated event tick
  alarm ∈ BOOL                            when v = 0
  v ∈ 0 .. 1                              then alarm :∈ BOOL
  v = 1 ⇒ alarm = TRUE
                                        event one
event INITIALISATION                      when alarm = TRUE
  alarm :∈ BOOL                           then v := 1
  v := 0
```

In some refinement the timer can be implemented by a counter or in some other way that provides convergence.
Fewer variables: avoid introducing new variables. In Event-B new events must refine *skip*. This means that usually variables manipulated in a loop can not be variables from an abstract machine. The gcd algorithm, for instance, computes its result in variable q:

```
q ⟵ getgcd(x, y)                      event getgcd
  p, q := x, y;                           q := gcd(x, y)
  do p < q → q := q − p
  □ q < p → p := p − q                  anticipated event loop2
  od                                      q :∈ ℕ₁
```

In the corresponding abstract Event-B machine we model the body of the loop by an anticipated event *loop2*, specifying that in some refinement *loop2* is implemented by a convergent event that may modify variable q.

Decomposition of a proof: partial correctness and termination. When introducing a loop in a development of a sequential program, we have to prove that the loop body preserves the invariant and decreases a variant. If we make the loop body an anticipated event, we can prove invariant preservation at one stage and termination at a later stage. This reduces the complexity at each stage and separates concerns of invariant preservation from termination.

3 Witnesses

Originally, witnesses have been introduced in Event-B in order to decompose proof obligations [11] but also the methodical benefits had been recognised [12]. The reasoning underlying the use of witnesses is very simple: Let v be the abstract variables, $I(v)$ the abstract invariant, w the concrete variables, $J(v,w)$ the gluing invariant, p the abstract event parameters, $G(p,v)$ the abstract event guard, $S(p,v,v')$ the abstract event actions before-after predicate, q be the concrete event parameter, $H(q,w)$ the concrete event guard, and $T(q,w,w')$ the concrete event actions before-after predicate. With

$$\begin{array}{rcl} K(v,q,w,w') & \mathrel{\widehat{=}} & I(v) \wedge J(v,w) \wedge H(q,w) \wedge T(q,w,w') \\ L(p,v,w,w') & \mathrel{\widehat{=}} & G(p,v) \wedge S(p,v,v') \wedge J(v',w') \end{array}$$

the refinement proof obligation for the refinement of the abstract by the concrete event is thus:

$$K(v,q,w,w') \quad \Rightarrow \quad \exists\, p,\, v' \cdot L(p,v,w,w') \quad . \tag{1}$$

In the introduction we discussed the close correspondence between Event-B models and proof obligations. We see that the granularity of the correspondence could be improved greatly if (1) could be decomposed into three implications with conclusions $G(p,v)$, $S(p,v,v')$, and $J(v',w')$, for instance. Because of the existential quantification "$\exists\, p,\, v'$" this is not possible. In some step of the proof of (1) we would usually instantiate the bound identifiers p and v' by expressions r and u', subsequently, proving the conclusions $G(r,v)$, $S(r,v,u')$, and $J(u',w')$ separately. We can do this systematically for all refinements specifying *witnesses* $W(p,v,v',q,w,w')$ that serve to instantiate p and v'. Of course, we have to verify that the witnesses exist [2]

$$K(v,q,w,w') \quad \Rightarrow \quad \exists\, p,\, v' \cdot W(p,v,v',q,w,w') \quad . \tag{2}$$

Applying modus ponens in the premises of (1) and observing that p and v' do not occur free in (1) the following (3) implies (1)

$$K(v,q,w,w') \wedge W(p,v,v',q,w,w') \quad \Rightarrow \quad \exists\, p,\, v' \cdot L(p,v,w,w') \quad . \tag{3}$$

[2] This proof obligation is usually a simple consequence of the invariant and easily discharged.

Finally, we can instantiate "$p, v' := p, v'$" in the conclusion so that (4) implies (3), hence, also (1):

$$K(v,q,w,w') \wedge W(p,v,v',q,w,w') \quad \Rightarrow \quad L(p,v,w,w') \quad . \tag{4}$$

Implication (4) can now be decomposed and proved separately for each conjunct $G(p,v)$, $S(p,v,v')$, and $J(v',w')$ in the conclusion.[3] As an example of the use of witnesses we refine the abstract timer from the preceding section where witnesses are specified in the **with** clauses of the events:

```
invariants
  $time \in \mathbb{N}$
  $time = 0 \Rightarrow alarm = TRUE$
  $time > 0 \Rightarrow alarm = FALSE$

event INITIALISATION
  with $alarm' = FALSE$
  then
    $time :\in \mathbb{N}_1$
    $v := 0$

convergent event tick
  when $time > 0$
  with $alarm' = bool(time' = 0)$
  then $time := time - 1$

event one
  when $time = 0$
  then $v := 1$
```

More complicated examples of witnesses can be found, for instance, in [13]. Witnesses have turned out to be very valuable for explaining how refinement is achieved. They have also increased the potential of animation of Event-B models opening up an efficient possibility for refinement animation [14].

A small improvement: witnesses for non-deterministic choices. We have shown how witnesses are used to obtain proof obligations that are easier to handle than (1). In practice, we observe however, that the refinement condition $K(v,q,w,w') \wedge W(p,v,v',q,w,w') \Rightarrow S(p,v,v')$ that we have obtained from the decomposition often has itself a form similar to (1).

In order to be able to use witnesses for non-deterministic assignments, too, in the absence of explicit support, non-deterministic assignments can be modelled by means of guards [10,14] and distinguished by a labelling convention; for instance, guards are labelled grd$n$ and non-deterministic assignments chc$n$. However, this complicates support for certain proof obligations, in particular, deadlock freedom. (Relying on a labelling convention for proof obligation generation does not appear reliable.) The best solution would seem to treat non-deterministic assignments the same way as guards. An additional benefit would be that the need for primed identifiers in Event-B models would disappear, too, simplifying further the existing concept of witnesses described above.

4 Conclusion

The modelling method Event-B is effective in practice because of a number of simple techniques that have been incorporated into it. They usually originated as

[3] Note, that we have not renamed any identifiers in the conclusion which would have obscured the correspondence with model.

a solution to some technical problem but then proved to be useful in a much wider context. We believe that it is important to take note of such developments. If we keep record of the small improvements, we can mark the methodical progress we make and we avoid losing the knowledge about why Event-B works.

References

1. Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2009. To appear.
2. Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *Software Tools for Technology Transfer*, 2009. To appear.
3. Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. An open extensible tool environment for Event-B. In Z. Liu and J. He, editors, *ICFEM 2006*, volume 4260, pages 588–605. Springer, 2006.
4. Jean-Raymond Abrial and Dominique Cansell. Click'n'Prove: Interactive Proofs within Set Theory. In *Theorem Proving in Higher Order Logics*, volume 2758 of *LNCS*, pages 1–24, 2003.
5. Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. Refinement and Reachability in EventB. In Helen Treharne, Steve King, Martin Henson, and Steve Schneider, editors, *ZB 2005*, volume 3455 of *LNCS*, pages 222–241, 2005.
6. Jean-Raymond Abrial and Stefan Hallerstede. Refinement, Decomposition and Instantiation of Discrete Models: Application to Event-B. *Fundamentae Informatica*, 77(1-2):1–28, 2007.
7. Jean-Raymond Abrial and Louis Mussat. Introducing dynamic constraints in B. In Didier Bert, editor, *B'98 : The 2nd International B Conference*, volume 1393 of *LNCS*, pages 83–128. Springer, 1998.
8. Jean-Raymond Abrial and Louis Mussat. On using conditional definitions in formal theories. In D. Bert, J.P. Bowen, M.C. Henson, and K. Robinson, editors, *ZB 2002*, volume 2272 of *LNCS*, pages 242–269, 2002.
9. Ralph-Johan Back. Refinement Calculus II: Parallel and Reactive Programs. In J. W. deBakker, W. P. deRoever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 67–93. Springer, May 1989.
10. John Colley. Private communication, 2009.
11. Stefan Hallerstede. The Event-B Proof Obligation Generator. Technical report, ETH Zürich, 2005.
12. Stefan Hallerstede. Justifications for the Event-B Modelling Notation. In J. Julliand and O. Kouchnarenko, editors, *B 2007*, volume 4355 of *LNCS*, pages 49–63. Springer, 2007.
13. Stefan Hallerstede. Proving Quicksort correct in Event-B. In Eerke Boiten and John Derrick, editors, *Refine 2009*, ENTCS, 2009.
14. Stefan Hallerstede, Michael Leuschel, and Daniel Plagge. Refinement-Animation for Event-B — Towards a Method of Validation. In *ABZ 2010*, LNCS. Springer, 2007. 14 pages. To appear.
15. Christophe Métayer, Jean-Raymond-Abrial, and Laurent Vosin. Event-B Language. Technical report, ETH Zürich, 2005.

Qualitative Reasoning for the Dining Philosophers [⋆]
— *Extended Abstract* —

Stefan Hallerstede[1] and Thai Son Hoang[2]

[1] Institut für Informatik
Heinrich-Heine-Universität Düsseldorf
`halstefa@cs.uni-duesseldorf.de`
[2] Department of Computer Science
Swiss Federal Institute of Technology Zürich (ETH Zürich)
`htson@inf.ethz.ch`

Abstract. We continue our investigation of qualitative probabilistic reasoning in Event-B. In the past we have applied it protocol verification, in particular, the Firewire protocol. There is still some way to go to achieve a practical method for qualitative probabilistic reasoning, especially concerning with refinement. We describe here our attempt for a probabilistic solution to the dining philosophers problem, in order to move further towards such a method.
Keywords: Event-B, probability, qualitative reasoning, refinement.

1 Overview

Our motivation here is construct a proof for the probabilistic solution for the Dining Philosophers problem. The proof from McIver and Morgan [5] uses both fairness assumption and probabilistic arguments. We attempt here to reason using only qualitative reasoning, hence the fairness assumption is replaced by a probabilistic reasoning. Moreover, we want to construct a practical method for reasoning about this kind of system which should be simple to use. We now give an overview of some important properties of the Event-B method that we are going to use, in particular about different technique for proving convergent of events.

1.1 The Event-B Modelling Method

A development in Event-B [2] is a set of formal models. The models are built from expressions in a mathematical language, which are stored in a repository. Event-B models are organised in terms of the two basic constructs: *contexts* and *machines*. Contexts specify the static part of a model whereas machines specify the dynamic part. Contexts may contain *carrier sets*, *constants*, *axioms*, and *theorems*. *Machines* specify behavioural properties of Event-B models. Machines may contain *variables*, *invariants*, *theorems*, *events*, and *variants*. Variables v define the state of a machine. They are constrained by invariants $I(v)$. Possible state changes are described by events. Each event is composed of a *guard* $G(t, v)$ (the conjunction of one or more predicates) and

⋆ Part of this research was carried out within the European Commission ICT project 214158 DEPLOY (`http://www.deploy-project.eu/index.html`).

an *action* $S(t, v)$, where t are the *parameters* of the event. 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. An event can be represented by the term "**any** t **where** $G(t, v)$ **then** $S(t, v)$ **end**". We use the short form "**when** G(v) **then** S(v) **end**" when the event does not have any parameters, and we write "**begin** S(v) **end**" when, in addition, the event's guard equals *true*. A dedicated event of the last form is used for *initialisation*.

The action of an event is composed of one or more *assignments* of the form

$$x \ := \ E(t, v) \tag{1}$$
$$x \ :\in \ E(t, v) \tag{2}$$
$$x \ :| \ Q(t, v, x') \quad , \tag{3}$$

where x is a variable contained in v, $E(t, v)$ is an expression, and $Q(t, v, x')$ is a predicate. Assignments of the form (1) are *deterministic*, whereas the other two forms are *nondeterministic*. In (2), x (which must be a single variable) is assigned an element of a set. In (3), Q is a "before-after predicate", which relates the values x (before the action) and x' (afterwards). (3) is the most general form of assignment and nondeterministically selects an after-state x' satisfying Q and assigns it to x. Variables other than x are unchanged by the above assignments. There is also a side condition on the action of an event: the variables on the left-hand side of the assignments contained in the action must be disjoint.

Proof obligations serve to verify certain properties of machines. Formal definitions of all proof obligations are given in [1]. For a machine, we must prove *invariant preservation* and *feasibility* of events. *Invariant preservation* states that invariants hold whenever variables change their values. *Feasibility* state that the action of an event must be feasible whenever the event is enable.

Machine refinement provides a means to introduce details about the dynamic properties of a model [2]. For more details on the theory of refinement, we refer to the Action System formalism [3], which has inspired the development of Event-B.

A machine CM can refine another machine AM. We call AM the *abstract* machine and CM the *concrete* machine. The state of the abstract machine is related to the state of the concrete machine by a *gluing invariant* $J(v, w)$, where v are the variables of the abstract machine and w are the variables of the concrete machine.

Each event ea of the abstract machine is *refined* by one or more concrete events ec. Let the abstract event ea and concrete event ec be:

$$\mathsf{ea} \ \ \widehat{=} \ \ \textbf{any } t \textbf{ where } G(t, v) \textbf{ then } S(t, v) \textbf{ end} \tag{4}$$
$$\mathsf{ec} \ \ \widehat{=} \ \ \textbf{any } u \textbf{ where } H(u, w) \textbf{ then } T(u, w) \textbf{ end} \quad . \tag{5}$$

Somewhat simplified, we say that ec refines ea if the guard of ec is stronger than the guard of ea (*guard strengthening*), and the gluing invariant $J(v, w)$ establishes a simulation of ec by ea (*simulation*). Proving guard strengthening just amounts to proving an implication. For simulation, under the assumption of the invariants and of the concrete guard $H(u, w)$ we must show that it is possible to choose a value for the abstract parameter t such that the abstract guard holds and the gluing invariant $J(v, w)$ is re-established. The possible values for the abstract parameter are given as *witness* in ec with the keyword **with**. In the course of refinement, *new events* are often introduced into a model. New events must be proved to refine the implicit abstract event SKIP, which does nothing.

Moreover, it may be proved that events do not collectively diverge. In other words, the events cannot take control forever and hence one of the other events eventually occurs. To prove this, one gives a *variant* V, which maps a state w to a finite set. One then proves that each new event *strictly* decreases V.

Given an event

```
evt
  any  $x$  where
    $G(x, v)$
  then
    $v :| S(x, v, v')$
  end
```

ones has to prove

```
  Invariants and axioms
  $G(x, v)$
$\vdash$
  $\forall v' \cdot S(x, v, v') \Rightarrow$
      $V(v') \subset V(v)$
```

1.2 Qualitative Reasoning: Probabilistic Action

In our earlier work [4], we extend the Event-B with probabilistic action $v \oplus| \; S(v, v')$, and the notion of probabilistic (eventually) termination of events. This extension requires a slightly modification to the variant proof obligation: event *might* decrease the variant V.

Given an event

```
evt
  any  $x$  where
    $G(x, v)$
  then
    $v \oplus| \; S(x, v, v')$
  end
```

ones has to prove

```
  Invariants and axioms
  $G(x, v)$
$\vdash$
  $\exists v' \cdot S(x, v, v') \wedge$
      $V(v') \subset V(v)$
```

A great advantage of this approach is that the proof obligations still within first-order predicate logic hence we do not need to extend our proof system.

2 The Dining Philosophers

We summary the dining philosophers problem as follows:

- A number of philosophers sit at a round table.
- Between each adjacent pair of philosopher is a single fork.
- In order to eat, each philosopher need two forks on both sides.
- When hungry, a philosopher might want to pick up a fork, but this might already be taken by his neighbouring philosopher.
- There is a possibility of deadlock or livelock.

There are various solution for the problem, including some deterministic solutions, e.g. using a waiter to break symmetry. We consider here a symmetric probabilistic solution as described in [5]. The algorithm for each philosopher is summarised in Figure 1. The table describes possible state changes for a particular philosopher. The only probabilistic choice that the philosopher made is when deciding either to pick up the left fork first or the right fork first.

The proofs from [5] using the fairness assumption which prove the Pseudo loop on the left to terminate. By replacing the fairness assumption with a probabilistic one, we attempt to verify that the Pseudo loop on the right to terminate probabilistically.

```
Some philosophers are hungry;
while "No philosopher is eating" do
  Schedule one of the philosopher fairly
end
```

```
Some philosophers are hungry;
while "No philosopher is eating" do
  Schedule one of the philosopher probabilistically
end
```

Our initial model is as follows, where h, t and e represent the set of *hungry*, *thinking* and *eating* philosophers, respectively.

Action	Before	After
Becomes hungry	t	h
Decides	h	$l\ {}_{0.5}\!\oplus\ r$
Picks first left fork (if left fork available)	l	L
Picks first right fork (if right fork available)	r	R
Eats by pick up right (if right fork available)	L	E
Drops left fork (if right fork not available)	L	h
Eats by pick up left (if left fork available)	R	E
Drops right fork (if left fork not available)	R	h
Thinks	E	t

t	thinking
h	hungry
l	left first
r	right first
L	hold left only
R	hold right only
E	eating
	(holding both fo

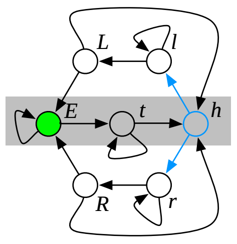


Fig. 1. Actions of a philosophers

```
variables:  $h, t, e$
```

```
invariants:
    $partition(P, h, t, e)$
```

```
init
  begin
    $h, t :| partition(P, h', t') \land h' \neq \varnothing$
    $e := \varnothing$
  end
```

```
eats
  any  $p$  where
    $p \in h$
  then
    $e := e \cup \{p\}$
    $h := h \setminus \{p\}$
  end
```

```
thinks
  any  $p$  where
    $p \in e$
  then
    $t := t \cup \{p\}$
    $e := e \setminus \{p\}$
  end
```

```
getsHungry
  any  $p$  where
    $p \in t$
  then
    $h := h \cup \{p\}$
    $t := t \setminus \{p\}$
  end
```

Our idea for developing the algorithm is as follows. Using refinement, we introduce the details of the approach with different set of philosophers, e.g. holding forks, picking forks. As a result, more events are introduced into the development. Our termination is established by the following arguments:

– Prove that events other than eats are (probabilistic) convergent.
– System is deadlock-free.

We formalise the lexicographic variant as presented in [5] in our Event-B development. The decrement of the variant can be split into different phases depending on the state of the system, where phase A is when no philosophers holding forks; phase BL (resp. BR) is when there are some philosophers holding left forks (resp. right forks); and phase C_i is when there are some philosophers holding left forks and some philosophers hold right forks. In particular, the convergent proof in phase BL and BR using probabilistic termination argument as mentioned earlier in Section 1.2.

$$\begin{tikzcd}
 & BL \arrow[dr] & & & \\
A \arrow[ur] \arrow[dr] & & C_n \arrow[r] & C_{n-1} \arrow[r, dotted] & C_0 \\
 & BR \arrow[ur] & & &
\end{tikzcd}$$

We focus now on the probabilistic convergent argument in phases C_i. In particular, we have the events of the following forms:

```
chooseLeft
  any  $p$  where
    $p \in l$
    ...
  then
    ...
  end
```

```
dropLeft
  any  $p$  where
    $p \in L$
    ...
  then
    ...
  end
```

...

The difficulty here is that the probabilistic choice is associated with the parameter p for the philosophers. In particular, our reasoning must take into account all the actions that a philosopher can do. For our probabilistic termination argument, we have to prove the following: There exists a philosopher such that he can always act, and any action that he made decreases the variant. At the moment, our proof obligations cannot express the above condition, hence we need to extend the proof obligation rule.

```
evt_i
  any  $t$  where
    $G_i(t, v)$
  then
    $v :| Q_i(t, v, v')$
  end
```

```
variant:  $V$
```

```
witness:  $W(t, v)$
```

- Sketch of probabilistic termination witness for t, say $W(t, v)$.
- Sketch of the proof obligations.
  1. Existent of witness: $I(v) \Rightarrow (\exists t \cdot W(t, v))$.
  2. Given the witness, at least one probabilistic event is enable.
     $I(v) \wedge W(t, v) \Rightarrow G_1(t, v) \vee \ldots \vee G_n(t, v)$
  3. For any probabilistic event evt_i, it decreases the variant V: $I(v) \wedge W(t, v) \wedge G_i(t, v) \wedge Q_i(t, v, v') \Rightarrow V(v') \subset V(v)$

3 Conclusions

We sketch here an extension to our qualitative reasoning for proving probabilistic termination of an algorithm for the dining philosophers problem. Our proposed extension includes a new interpretation for probabilistic choice between events' parameters and new proof obligations which are practical for having tool support.

Moreover, for future work, because of the fact that refinement can reduce non-determinism, qualitative termination is not necessary preserved through refinement in general. We need to have additional proof obligation(s) for preserving qualitative termination, however, any approach should be simple and usable.

References

1. J-R. Abrial. *Modeling in Event-B: System and Software Design*. CUP, 2009. To appear.
2. J-R. Abrial and S. Hallerstede. Refinement, Decomposition and Instantiation of Discrete Models: Application to Event-B. *Fundamentae Informatica*, 2006.
3. R-J Back. Refinement Calculus II: Parallel and Reactive Programs. In *Stepwise Refinement of Distributed Systems*, 1989.
4. S. Hallerstede and T.S. Hoang. Qualitative probabilistic modelling in event-b. In J. Davies and J. Gibbons, editors, *IFM*, volume 4591 of *LNCS*, pages 293–312. Springer, 2007.
5. A. McIver and C. Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer, 2005.

Event-B Decomposition for Parallel Programs — *Extended Abstract* —[⋆]

Thai Son Hoang and Jean-Raymond Abrial

Deparment of Computer Science,
Swiss Federal Institute of Technology Zurich (ETH-Zurich),
CH-8092, Zurich, Switzerland
htson@inf.ethz.ch, jrabrial@neuf.fr

Abstract. We present here an approach for developing a parallel program combining *refinement* and *decomposition* techniques. This involves in the first step to abstractly specify the aim of the program, then subsequently introduce shared information between sub-processes via refinement. Afterwards, decomposition is applied to separate the resulting model into sub-models for different processes. These sub-models are later independently developed using refinement. Our approach aids the understanding of parallel programs and reduces the complexity in their proofs of correctness.

Keywords: Event-B, parallel programs, decomposition, refinement.

1 Introduction

There are a number of methods for proving the correctness of parallel programs [4]. Our main contribution is an approach applying the technique of refinement and decomposition in Event-B [1]. The approach contains four steps as follows.

1. Starts with an abstract specification *in-one-shot* giving the purpose of the program.
2. Refines this abstract specification by introducing details about the *shared variables*.
3. Decomposes the model in the previous step to split the model into several (abstract) sub-models for processes.
4. Refines each sub-model in the previous step independently.

In the last step, each sub-model can be seen as a new abstract specification, hence application of steps 2, 3 and 4 can be repeated again. The novelty of our approach is in step 2 where we specify shared information between processes. This information has dual purpose. Firstly, it contains the necessary guarantee condition from each process to establish the final result. Secondly, it also gives the condition for which each process can rely on in further development. This decision, i.e. to have this step early in our development, takes advantage of decomposition technique and results in simpler models and reduces the complexity of proving programs. This is the main advantage of our method over existing approaches. More information on related work is in Section 4.

⋆ Part of this research was carried out within the European Commission ICT project 214158 DEPLOY (http://www.deploy-project.eu/index.html).

2 The Event-B Modelling Method

A development in Event-B [3] is a set of formal models. Event-B has a semantics based on transition systems and simulation between such systems, described in [2]. We will not describe in detail, just high-lights some important points for Event-B semantics.

Event-B models are organised in terms of the two basic constructs: *contexts* and *machines*. Contexts specify the static part of a model whereas machines specify the dynamic part. Contexts may contain *carrier sets*, *constants*, *axioms*, and *theorems*.

Machines specify behavioural properties of Event-B models. Machines may contain *variables*, *invariants*, *theorems*, *events*, and *variants*. Variables v define the state of a machine. They are constrained by invariants $I(v)$. Possible state changes are described by events. Each event is composed of a *guard* $G(t, v)$ (the conjunction of one or more predicates) and an *action* $S(t, v)$, where t are the *parameters* of the event. *Proof obligations* serve to verify certain properties of machines. Given a machine, we need to prove the following obligations:

- **Invariant preservation**: invariants hold whenever variables change their values.
- **Feasibility**: For an event the action is feasible whenever the guard is enable.

Machine refinement provides a means to introduce details about the dynamic properties of a model [3]. A machine CM can refine another machine AM. We call AM the *abstract* machine and CM the *concrete* machine. The state of the abstract machine is related to the state of the concrete machine by a *gluing invariant* $J(v, w)$, where v are the variables of the abstract machine and w are the variables of the concrete machine.

Each event ea of the abstract machine is *refined* by one or more concrete events ec. Somewhat simplified, we say that ec refines ea if the guard of ec is stronger than the guard of ea (*guard strengthening*), and the gluing invariant $J(v, w)$ establishes a simulation of ec by ea (*simulation*).

In the course of refinement, *new events* are often introduced into a model. New events must be proved to refine the implicit abstract event SKIP, which does nothing. Moreover, it may be proved that new events do not collectively diverge. In other words, the new events cannot take control forever and hence one of the old events can occur.

2.1 Shared Variable Decomposition

The idea of decomposition is to split a large model into smaller sub-models which can be handled more comfortably than the whole: one should be able to refine these sub-models independently. More precisely, if one starts from an initial (large) model, say M, decomposition allows us to separate this model into several sub-models $\mathsf{M}_1 \cdots \mathsf{M}_i$. These sub-models can then be refined independently yielding $\mathsf{N}_1 \cdots \mathsf{N}_i$. The correctness of the decomposition technique guarantees that the model N, obtained by re-composing $\mathsf{N}_1 \cdots \mathsf{N}_i$, is a refinement of the original model M. This process is illustrated in the following diagram:

$$\boxed{\begin{array}{ccccc} & \text{Decomposition} & \text{Refinement} & \text{Re-composition} & \\ \mathsf{M} & \rightarrow & \left\{\begin{array}{ccc} \mathsf{M}_1 & \rightarrow & \mathsf{N}_1 \\ & \cdots & \\ \mathsf{M}_i & \rightarrow & \mathsf{N}_i \end{array}\right\} & \rightarrow & \mathsf{N} \end{array}}$$

Generation of sub-models using shared variable decomposition Given a certain model M with events $\mathsf{e}_1(\mathsf{a})$, $\mathsf{e}_2(\mathsf{a},\mathsf{c})$, $\mathsf{e}_3(\mathsf{b},\mathsf{c})$, $\mathsf{e}_4(\mathsf{b})$,[1] we would like to decompose M into two separate models: M_1 dealing with events e_1 and e_2; and M_2 dealing with events e_3 and e_4. By giving the above *event partition*, we must also perform a certain *variable distribution*. This distribution can be derived directly from the information about the partitioning of events and the set of variables that they access.

Moreover, in each sub-model, we need to have a number of *external events* to simulate how shared variables are handled in the non-decomposed model. These events are abstract versions of the corresponding internal events and use only the shared variables. In our example, M_1 will have an external event corresponding to e_3 (beside the internal events e_1 and e_2. Symmetrically, M_2 will have an external event corresponding to e_2. Similar to shared variables, *external events* cannot be further refined.

We also present a practical construction of the external event given its original event. This is illustrated below for an external event $(\mathsf{ext\_})\mathsf{e}_2$ in sub-model M_2. Intuitively, this event is the *projection* of the original event, i.e. e_2, on the state of the sub-model M_2.

```
e₂
  any  $t$  where
    $G(t, a, c)$
  then
    $a, c :\mid Q(t, a, c, a', c')$
  end
```

```
(ext_)e₂
  any  $t, a$  where
    $G(t, a, c)$
  then
    $c :\mid \exists a' \cdot Q(t, a, c, a', c')$
  end
```

More detail on shared variable decomposition in Event-B can be found in [1].

3 Example. The FindP Program

Our running example is a standard problem in the literature for parallel programs. The purpose of the *FindP* program is to find the first index k of an array $ARRAY$, if there is one, satisfies some property P. Otherwise, if this index does not exist, i.e. none of the array elements satisfy P, the program returns $M + 1$, where M is the size of the array.

The pseudo-code for the main program is given below (on the left) and for each process (presented here $process_1$ on the right)

```
$index1, index2 := min(PART1), min(PART2);$
$publish1, publish2 := M + 1, M + 1;$
$\mathbf{process_1} \mid\mid \mathbf{process_2};$
$result := min(\{publish1, publish2\})$
```

```
while $index1 < min(\{publish1, publish2\})$ do
  if $ARRAY(index1) = \mathrm{TRUE}$ then $publish1 := index1$
  else $index1 :=$ the-next-index-in-PART1-or-M+1 end
end
```

The machine-checked version of the development can be found on the web [5]. We summarise our strategy for developing this program as follows.

Initial model specifies the result of the algorithm directly.
First refinement introduces the local indices of processes.
Decomposition step splits the model into sub-models corresponding to different processes: $main$, $process_1$, $process_2$.

We continue with further refinement steps for $process_1$; $process_2$ should be developed in symmetrical fashion. Futher development of the $main$ process is straightforward and is not of our interest here.

First sub-refinement introduces the local index of the process.
Second sub-refinement introduces the read value of the process.
Third sub-refinement introduces the address counter for scheduling of events.

[1] Note that the variables appeared in brackets denote those that are *accessed* by these events.

4 Related Work

The problem of verifying the *FindP* program has been tackled using different methods, e.g., Owicki/Gries' *interference-free* [9] and Jones' *rely/guarantee* approach [7].

The work of Owicki/Gries [9] extends Hoare's deductive system for sequential programs [6] in order to prove the correctness of parallel programs. Their proofs of correctness for parallel statements centre around the notion of *interference-free* which is defined as follows. Given a proof of Hoare's triple $\{P\}\ S\ \{Q\}$ and a statement T with precondition $pre(T)$, T does not interfere with $\{P\}\ S\ \{Q\}$ if

InfFree1 $\{Q \wedge pre(T)\}\ T\ \{Q\}$, i.e. T maintains the post-condition Q, and
InfFree2 for any sub-statement S' of S, $\{pre(S') \wedge pre(T)\}\ T\ \{pre(S')\}$.

Within our approach, the above two conditions are verified during the development of the model at various refinement levels. At the abstract level before decomposing, S and T are some events of the models and the post-condition Q are just some invariants. For example, S are some events belonging to $process_1$ and T are events belonging to $process_2$, Q are the invariants that state the outcome of $process_1$, e.g. **inv1_1–inv1_5**. We have to prove that these invariants are maintained by any events T and this corresponds to condition **InfFree1**. Furthermore, during the sub-refinement of a process, sub-statements S' of S are introduced. At the same time, new invariants are added and these invariants correspond to the preconditions $pre(S')$ in the proof of $\{P\}\ S\ \{Q\}$ using Hoare's deductive system. Hence the condition **InfFree2** is verified by proving that events T (now becoming external events) maintain the new invariants.

This is somewhat not surprising, since in our approach, the role of external events is to keep the information about the possible changes on shared variables by different processes. During the refinement of a sub-process, we need to take into account the effect of these external events so that they do not "interfere" with the development of this sub-process. The main advantage of our approach over the work from Owicki/Gries is that these external events are at the abstract level rather than concrete statements as defined in the *interference-free* conditions. This reduces the complexity of the verification process.

Comparing to the Owicki/Gries approach, our method is closer to the *rely/guarantee* approach of Jones [7]. The approach extends the notion of Hoare's triple $\{P\}\ S\ \{Q\}$ to encode the rely condition R and guarantee condition G. By definition, a condition $\{P, R\}\ S\ \{G, Q\}$ is satisfied by S if: under the assumptions that S starts in state satisfies the precondition P, and any external transition satisfies the rely condition R; then S ensures that any internal transition of S satisfies the guarantee condition G, and if S terminates then the final state satisfies postcondition Q.

We focus on an example rule for parallel composition.

$$\textbf{PAR-I}\quad \frac{\begin{array}{ll} R \vee G_1 \Rightarrow R_2 & (\textbf{RG1}) \\ R \vee G_2 \Rightarrow R_1 & (\textbf{RG2}) \\ G_1 \vee G_2 \Rightarrow G & (\textbf{RG3}) \\ \{P, R_1\} S_1 \{G_1, Q_1\} & (\textbf{RG4}) \\ \{P, R_2\} S_2 \{G_2, Q_2\} & (\textbf{RG5}) \end{array}}{\{P, R\}\ S_1 \mid\mid S_2\ \{G, Q_1 \wedge Q_2\}}$$

The rule is interpreted as follows. Statement $S_1 \mid\mid S_2$ satisfies $\{P, R\}\ S_1 \mid\mid S_2\ \{G, Q_1 \wedge Q_2\}$ if the following conditions are met. Firstly, both "global" rely condition R and the guarantee condition of one statement ensure the rely condition of the other (**RG1** and **RG2**). Secondly, both guarantee conditions of the two statements ensure the global guarantee condition G (**RG3**). Lastly, S_1 and S_2 independently satisfy their corresponding rely/guarantee condition (**RG4** and **RG5**)

Note that both rely and guarantee conditions are relations over two states. They are indeed similar to events in Event-B which correspond to a relations over pre-/post-states. Moreover, the implication between rely/guarantee conditions is the same as event refinement. Within our approach, a pair of internal/external events encodes rely/guarantee conditions where the rely condition corresponds to the external event and the guarantee condition corresponds to the internal event. The generation of external events guarantees that they are the abstractions of the corresponding internal events. In fact, our generation of sub-models as described in Section 2.1 guarantees that the resulting sub-models satisfy the parallel composition rule. This is the advantage of our approach over *rely/guarantee* method. In fact the external events are the strongest possible condition that the other process can rely on. In practise, the rely/guarantee conditions could be more abstract, e.g. requires that values of some variables decrease monotonically [8].

5 Conclusion

Our approach introduces the possible *interaction* between processes early in the development in order to take the advantage of decomposition. This is different from the approach where one develops processes according to the implementation of the process with possible *cheating* (e.g. one process directly looks into the value of the other process), and subsequently refines the model until there is no more cheating. This approach has been proposed in [2] and is used in many other examples. Applying this approach without using decomposition, the two processes are developed together, hence the development also has higher complexity comparing to our approach.

The key point in our development using decomposition lies in the model that is being decomposed, where we have to abstractly specify the effect of the two future processes on shared variables. We use the overall intended result of the program to help us to *derive* the requirement on the future processes.

References

1. J-R. Abrial. Event model decomposition. Technical Report 626, ETH Zurich, May 2009.
2. J-R. Abrial. *Modeling in Event-B: System and Software Design*. CUP, 2009. To appear.
3. J-R. Abrial and S. Hallerstede. Refinement, decomposition and instantiation of discrete models: Application to Event-B. *Fundamentae Informatica*, 2006.
4. W. P. de Roever, F. S. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge Tracts in Theoretical Computer Science. CUP, 2001.
5. T.S. Hoang. FindP development using decomposition. `http://deploy-eprints.ecs.soton.ac.uk/154/`, 2009.
6. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 1969.
7. C.B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 1983.
8. C.B. Jones. Splitting atoms safely. *Theor. Comput. Sci.*, 2007.
9. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Inf.*, 1976.

Structuring Specifications with Modes

Alexei Iliasov, Alexander Romanovsky
Center for Software Reliability
Newcastle University
England, United Kingdom
Email: {alexei.iliasov, alexander.romanovsky}@newcastle.ac.uk

Fernando Luís Dotti
Faculty of Informatics
Pontifical Catholic University of Rio Grande do Sul
Porto Alegre - RS - Brazil
Email: fernando.dotti@pucrs.br

***Abstract*—The two dependability means considered in this paper are rigorous design and fault tolerance. It can be complex to rigorously design some classes of systems, including fault tolerant ones, therefore appropriate abstractions are needed to better support system modelling and analysis. The abstraction proposed in this paper for this purpose is the notion of operation mode. Modes are formalised and their relation to a state-based formalism in a refinement approach is established. The use of modes for fault tolerant systems is then discussed and a case study presented. Using modes in state-based modelling allows us to improve system structuring, the elicitation of system assumptions and expected functionality, as well as requirement traceability.**

***Keywords*-operation modes; fault-tolerance; formal specification; Event-B.**

I. Introduction

Systems are dependable if they deliver service that can be justifiably trusted [1]. Building such systems is a challenging task, typically conducted by employing various dependability means. In this paper we are particularly interested in the means of two types: rigorous design and fault tolerance.

Rigorous design (or fault prevention) is often used to justify system trustworthiness by preventing introduction of faults into system. This can be done by employing formal modelling and analysis. The known problem with this approach is its scalability. A way to improve it is through the development of abstractions and formal techniques tailored to classes of systems.

System dependability cannot be achieved by only trying to build perfect systems, any critical system has to face abnormal situations (including malfunctioning devices, wearing hardware and software defects) and deal with them properly. This is achieved by integrating appropriate fault tolerance means into the system. Unfortunately the situation is not satisfactory here: as reported by F. Cristian [2], field experience with telephone switching systems showed that up to two thirds of system failures were due to design faults in exception handling or recovery algorithms. Other evidences of inadequate use or construction of fault-tolerance mechanisms are reported in [3].

Several authors have investigated fault-tolerance modelling using different specification formalisms and verification approaches (e.g. [4], [5]). However, the identification and support of suitable abstractions for formal design of fault tolerant systems is still an open issue. Such abstractions have to, at the one side, be amenable to representation using a formal specification language, and, on the other side, offer the way to model and reason about (i) states - the characterization of normal and erroneous states, and state manipulation to reach consistency are inherent to fault tolerant systems; (ii) structure - separation of normal and abnormal (fault tolerant) behaviour is to be supported, as well as the representation of control structures for different fault tolerance mechanisms; and (iii) system properties under changing conditions - the explicit statement of possible working conditions, addressing fault assumptions, and assured system properties under these conditions are also to be supported.

In this paper the known concept of 'operation mode' [6] is revisited - we use modes to structure system specification to facilitate rigorous design and to integrate fault tolerance. Due to the use of modes in different types of systems, such as real-time [6], avionic and space systems [7], [8], the approach is useful for building wide classes of dependable systems.

We use term mode in the same sense as [6]: both as partitions of the state space, representing different working conditions of the system, and as a way to define control information, imposing structure on the operation of the system. In Section II, modes are defined to allow the modeller to explicitly state the property that must be respected, called guarantee, in each working system condition, called assumption. In Section III, mode refinement is discussed, allowing detalisation of the mode system. The use of modes together with a state-based formal method is discussed in Section V. Mode refinement is performed hand in hand with the refinement of the respective formal model and offer a way for layered definition and reasoning about system properties. This helps to assure that the properties are easily traceable to requirements. Another advantage of a refinement approach is that it offers a strategy obtain a correct implementation from the formal model. Moreover, theorem proving strategies and tools sometimes offer an attractive option to model-checking as they avoid the state-space problem.

Section IV discusses the use of modes in the design of fault-tolerant systems. Using the model of a cruise control system, Section VI exemplifies both mode refinement and the use of modes to specify fault tolerant systems. Related work and conclusions are presented in Sections VII and VIII.

This work is partially supported by the ICT DEPLOY IP and the EPSRC/UK TrAmS platform grant. Fernando L. Dotti is supported by CNPq/Brazil grant 200806/2008-4.

II. Operation Modes

Operation modes help to reason about system behaviour by focusing on the principal system properties observed under different situations. In this approach, a system is seen as a set of modes partitioning the system functionality over differing operating conditions. The term *assumption* is used to denote the different operating conditions and *guarantee* denotes the functionality ensured by the system under the corresponding assumption. A system may switch from one mode to another in a number of ways characterised by *mode transition*. A mode is a pair A/G where:

- $A(v)$ is an assumption - a predicate over the current system state;
- $G(v, v')$ is the guarantee, a relation over the current and next states of the system; and

Vector v is the set of variables, characterising a system state and constrained by an invariant $I(v)$. The purpose of an invariant $I(v)$ is to limit the possible states of v by excluding undesirable or unsafe states. It also defines types for variables v. To limit the scope of discussion, it is assumed that a system is only in one mode at a time. Mode overlapping and mode interference bring a number of interesting challenges that cannot be sufficiently addressed in this paper due to space limitations. Formally, it is required that mode assumptions are mutually exclusive and exhaustive in respect to a model invariant, as below. $\oplus$ is a set partitioning operator.

$$I(v) = A_1(v) \oplus \cdots \oplus \; A_n(v) \tag{1}$$

Mode switching is realised with mode transitions. A mode transition is an atomic step switching system from one source to one destination mode. It is convenient to characterise a mode transition by a pair of assumptions - the assumptions of source and of destination modes. Assuming that mode is assigned an index, a mode transition from A_i/G_i to A_j/G_j is a relation on mode indices $i \rightsquigarrow j$.

A system starts executing one of initiating transitions $\top \rightsquigarrow k$. The transition switches the system on and places it into some system mode A_k/G_k. A system terminates by executing one of terminating transitions $t \rightsquigarrow \bot$ [1]. Mode transitions $i \rightsquigarrow \top$ and $\bot \rightsquigarrow j$ are not allowed. Also, it is required that during its lifetime a system enters at least in one operation mode and thus transition $\top \rightsquigarrow \bot$ is not possible. There can be any number of initiating and terminating mode transitions.

There are certain restrictions on the way mode assumptions and guarantees are formulated. One obvious condition is that a guarantee may not require or permit a mode to violate an invariant, that is, the states described by a guarantee must be wholly included into valid model states:

$$I(v) \wedge A(v) \wedge G(v, v') \Rightarrow I(v') \tag{2}$$

[1]Not every system has to have this transition: a control system would be typically designed as never aborting.

The assumption and guarantee of a mode must be non-contradictory. I.e. a mode should permit a concrete implementation:

$$\exists v, v' \cdot (I(v) \wedge A(v) \Rightarrow G(v, v')) \tag{3}$$

A system is characterised by a collection of modes and a vector of mode transitions:

$$\begin{array}{c} A_1/G_1 \\ \dots \\ A_n/G_n \\ i \rightsquigarrow j \\ \dots \end{array} \tag{4}$$

The state of a system described using operation modes is a tuple (m, v) where m is the index of a current operation mode and v is the current system state. Mode index helps to clarify how mode switching is done although it may be computed from v alone due to condition 1. The evolution of a system like above is understood as follows. While it is in some mode m the state of model variables evolves so that the next state is any state v' satisfying both the corresponding guarantee $G(v, v')$ and the modes assumption $A(v')$:

$$\boxed{\text{internal}}\ \frac{A_m(v) \wedge G_m(v, v') \wedge A_m(v')}{\langle m, v\rangle \rightarrow \langle m, v'\rangle}$$

If there is a mode transition originating from a current mode, the transition could be enabled to switch the system to a new mode.

$$\boxed{\text{switching}}\ \frac{m \rightsquigarrow n \wedge A_m(v) \wedge A_n(v')}{\langle m, v\rangle \rightarrow \langle n, v'\rangle}$$

These two activities compete with each other: at each step a non-deterministic choice is made between the two. An initiating transition is a special case: it must find an initial system state without being able to refer to any previous state:

$$\boxed{\text{start}}\ \frac{\top \rightsquigarrow k \wedge A_k(v)}{\langle \top, \textit{undef}\rangle \rightarrow \langle k, v\rangle}$$

where *undef* denotes a system state prior to the execution of an initiating transition. System termination is addressed by the *switching* rule above. Note that all of the three rules also assume that an invariant holds in current and new states: $I(v) \wedge I(v')$. This is a corrolary of conditions 1 and 2.

III. Mode Refinement

Refinement is formal technique for transitioning from an abstract model to a concrete one [9]. Terms abstract and concrete are relative here: a concrete model of one step is another's step abstract model. There are a number of benefits in apply refinement in model construction: it combats complexity by splitting design process into a number of simple steps; it helps to organise the process of modelling by allowing a modeller to focus on one aspect of a model a time; it makes proofs easier as for each refinement one only has to proof the correctness of new behaviour[2].

At a very general level, refinement is a partial order relation on model universe. This relation is denoted as $\sqsubseteq$ and it

[2]Strictly speaking, this only applies to cases when refinement is monotonic. However, all the popular formal methods enjoy this property and heavily rely on it.

is reflexive, transitive and antisymmetric. For the operation modes mechanism the refinement technique is used to gradually evolve a system description by adding or replacing modes and transitions. Such evolution is completely formal in a sense that a refined model may be used in place of its abstraction.

Refinement itself is a combination of a number of techniques: data refinement, when data types are changed and data structures are introduced; behavioural refinement, when system behaviour becomes more deterministic and also described in a finer level of details; and superposition refinement (or model elaboration), when new functionality is added without changing an existing model. All the three are applicable and discussed for modes in the following.

a) Data Refinement: With data refinement, the vector of model variables v is changed to some new vector u and model invariant $I(v)$ is replaced with new invariant $J(v, u)$, often called a *gluing invariant*. The mentioning of old variables v in new invariant J allows modeller to expresses a linking relation between the state of concrete and abstract models.

b) Behavioural Refinement: Behaviour refinement details the mode view on a system. One case is changing a mode assumption or guarantee or both. It is postulated mode assumption cannot be strengthened during refinement. This is based on understanding that an assumption is a requirement of a mode to its environment. As a system developer cannot assume control over the environment of a modelled system, a stronger requirement to an environment may not be realisable. On the other hand, a weaker requirement to an environment means that a system is more robust as it would remain operational in a wider range of environments. Symmetrically, a mode guarantee cannot be weakened as a mode guarantee is understood as a contract of a mode with the rest of a system and the system environment. In other words, weakening a mode guarantee could violate expectations of another system part. The following condition summarises this refinement rule:

$$\begin{array}{l} A(v)/G(v,v') \sqsubseteq A'(u)/G'(u,u'), \\ \text{iff} \begin{cases} I(v) \wedge J(v,u) \wedge A(v) \Rightarrow A'(u) \\ J(v,u) \wedge G'(u,u') \Rightarrow G(v,v') \end{cases} \end{array} \tag{5}$$

Another case is when an abstract mode is a modelling abstraction for several concrete modes. Thus, a single mode in an abstract model evolves into a two or more concrete modes. The general rule for such refinement step is that the combination of new modes must be a refinement of an abstract mode. In more concrete terms, a disjunction of concrete mode assumptions must be not stronger than the abstract mode assumption and the disjunction of concrete guarantee must be not weaker than the abstract guarantee:

$$\begin{array}{l} A(v)/G(v,v') \sqsubseteq \begin{array}{l} A_1(u)/G_1(u,u') \\ A_2(u)/G_2(u,u') \end{array}, \\ \text{iff} \begin{cases} I(v) \wedge J(v,u) \wedge A(v) \Rightarrow A_1(u) \vee A_2(u) \\ I(v) \wedge J(v,u) \wedge G_1(u,u') \vee G_2(u,u') \Rightarrow G(v,v') \end{cases} \end{array} \tag{6}$$

c) Superposition Refinement: Sometimes it is needed to add new modes without having to split an abstract mode. This is accomplished using superposition refinement. With superposition refinement, a refined model contains additional modes. Assumptions and guarantees of these modes must be expressed on new variables (variables for u that are not mapped onto abstract variables v). Formally, this is possible by refining an implicit skip mode $false/true$. This is the weakest form of a mode and it can be refined into any other mode.

d) Refinement of Transitions: A refinement of a mode or an introduction of a new mode requires changes to mode transitions. The general rule is that a transition present in an abstract model must have a corresponding transition in a refined model and no new transitions may appear. Changing mode assumptions and guarantees does not affect mode transitions. Splitting a mode into sub-modes, however, leads to the distribution of the mode transitions associated with the refined mode among the new modes. Thus, if a mode with a transition is split into two new modes, the transition can be associated with any one of the new modes or both.

e) Visual Notation: To assist in application of the operation modes approach, a simple visual notation is proposed. It is loosely based on modechards [6]. A mode is represented by a box with mode name; a mode transition is an arrow connecting two modes. The direction of an arrow indicates the previous and next modes in a transition. Special modes $\top$ and $\bot$ are omitted in a diagram so that initiating and terminating transitions appear to be connected with a single mode. This is also how they can be distinguished from other transitions. Refinement is expressed by nesting boxes. Figure 1(B) presents a mode M1 refined into modes M1.1 and M1.2. The mode transitions depicted are only one possibility. A refined diagram with an outgoing arrow from an abstract mode is equivalent to having outgoing arrows from each of the concrete modes (this feature is used in the case study).

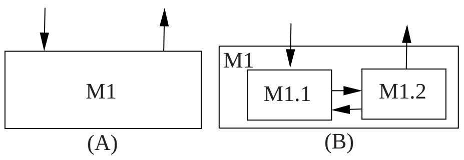


Fig. 1. Mode Diagrams.

IV. Modes for Fault Tolerant Systems

The use of modes together with a refinement approach, as introduced in the previous sections, offers suitable abstractions to modelling and reasoning about fault tolerant systems, as discussed in the following.

Due to the use of a state-based approach, state representation, manipulation and reasoning becomes natural. The support provided by modes allows to partition the state space into normal and erroneous: mode assumptions allow this separation to be declared and erroneous states made explicit. Refinement allows further definition of erroneous states into more specific ones. Assumptions on normal and erroneous states can be suitably associated to modes in charge of performing normal system operation and fault tolerance measures, respectively.

Generally speaking, a recovery mode should be associated with a particular normal mode, which it recovers, and mode switching is in some sense reminiscent to calling an exception handler in programming languages. Error detection is immediate, embedded in the erroneous state assumption of a recovery mode. As soon as a state transition leads to the characterization of an erroneous state, the recovery mode is enabled. A more concrete view is to consider the existence of a detection mechanism, which is active during normal operation. In such case the detection mechanism affects the state used in the assumptions of normal and recovery modes. By refinement one could start with the first and reach the second, more detailed model. Any of the possibilities allow switching to recovery mode from any normal mode state. For reasoning purposes, one can introduce the possibility of fault occurrences in parallel with the model. In an event based formalism this takes the form of an enabled event that affects the state to satisfy the erroneous state assumption.

The recovery mode has access to the state of the respective normal mode. Analogously to assumptions, guarantees associated to normal or recovery modes assist to define properties of the system in absence or presence of errors, respectively. Depending on the severity of the detected error, the recovery mode guarantees may assert that the recovery procedure: (i) successfully recovers the state and thus switches back to normal mode to proceed execution (Figure 2(B) or (C)); (ii) provides degraded service in cases where full functionality is not recoverable (Figure 2(D)); (iii) fails to recover, in which case measures to stop safely may be taken (Figure 2(A) and part of (D)). Using the graphical notation introduced in the previous section, the following configurations exemplify some possible use of modes for fault tolerance.

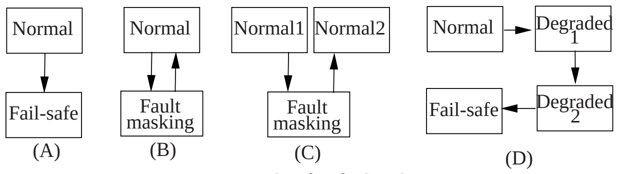


Fig. 2. Modes for fault tolerance.

V. Operation Modes for Event-B

The operation modes method is not intended to be used as a modelling method on its own as it lacks the facilities for expressing detailed design. The schematic nature of the approach makes it it well suited to integration with an existing formalism. One such case is presented in this section. A well known formalism - Event-B - is extended with operation modes. The rules for deriving formal conditions for reasoning about a combination modes and Event-B models are presented.

Event-B is a state-based formalism closely related to Classical B [10] and Action Systems [11]. The step-wise refinement approach is the corner stone of the Event-B development method. The combination of model elaboration, atomicity refinement and data refinement helps to formally transition from high-level architectural models to very detailed, executable specifications ready for code generation.

An extensive tool support through the Rodin Platform makes Event-B especially attractive [12]. An integrated Eclipse-based development environment is actively developed, well-supported, and open to third-party extensions in the form of Eclipse plug-ins. The main verification technique is theorem proving supported by a collection of powerful theorem provers. The development environment is also equipped with model checking capabilities.

An Event-B model is defined by a tuple (c, s, P, v, I, R_I, E) where c and s are constants and sets known in the model; v is a vector of model variables; $P(c, s)$ is a collection of axioms constraining c and s. I is a model invariant limiting the possible states of v: $I(c, s, v)$. The combination of P and I should characterise a non-empty collection of suitable constants, sets and model states: $\exists c, s, v \cdot P(c, s) \wedge I(c, s, v)$. The purpose of an invariant is to express model safety properties (that is, unsafe states may not be reached). In Event-B an invariant is also used to deduce model variable types. R_I is an initialisation action computing initial values for the model variables; it is typically given in the form of a predicate constraining next values of model variables without, however, referring to previous values - $R_I(c, s, v')$. Finally, E is a set of model *events*. An event is a guarded command:

$$H(c, s, v) \to S(c, s, v, v') \tag{7}$$

where $H(c, s, v)$ is an event guard and $S(c, s, v, v')$ is a before-after predicate. An event may fire as soon as the condition of its guard is satisfied and no other event executes at the same time. In case there is more than one enabled event at a certain state, the demonic choice semantics is applies. The result of an event execution is some new model state v'. The semantics of an Event-B model is usually given in the form of proof semantics, based on Dijkstra's work on weakest precondition [13]. A collection of proof obligations is generated from the definition of the model and these must be discharged in order to demonstrate that the model is correct.

Putting it as a requirement that an enabled event produces a new state v' satisfying a model invariant, the following would define the model *consistency* condition: whenever an event on an initialisation action is attempted there exists a suitable new state v' such that a model invariant is maintained - $I(v')$. This is usually stated as two separate proof obligations: a feasibility obligation requiring the existence of (any) new state v' and the invariant satisfaction obligation showing that any new state v' maintains an invariant. The invariant satisfaction obligation requires that a new state produced by an event must satisfies a model invariant:

$$I(c, s, v) \wedge P(c, s) \wedge H(c, s, v) \wedge S(c, s, v, v') \Rightarrow I(c, s, v') \tag{8}$$

An event must also be feasible, in a sense that an appropriate new state v' must exist for some given current state v:

$$I(c, s, v) \wedge P(c, s) \wedge H(c, s, v) \Rightarrow \exists v' \cdot S(c, s, v, v') \tag{9}$$

Conceptually, operation modes and Event-B models are related by requiring that every mode and mode transition has a suitable implementation in an Event-B model. A mode is related to a non-empty subset of Event-B model events and mode transition is mapped into a single Event-B event:

$$\begin{array}{l} A_1/G_1 \mapsto E_1 \\ \quad \dots \\ A_n/G_n \mapsto E_n \\ (i \rightsquigarrow j) \mapsto E_k \\ \quad \dots \end{array} \tag{10}$$

Event sets $E_1, \dots, E_n$ may overlap but may not be identical. The latter is due to the fact that two modes $A_i/G_i \mapsto E$ and $A_j/G_j \mapsto E$ are equivalent to a single mode $A_i \vee A_i/G_i \wedge G_j \mapsto E$ and thus there is no advantage in allowing configurations where modes have identical event sets. The mapping between transitions and events is one-to-many: a transition is mapped into a non-empty set of events. Each event associated with a transition must properly implement the transition, that is, it must be proven it gets enabled in a stated assumed by a source mode and establishes a state corresponding to the assumption of a target mode. To establish mapping, for some transition $(i \rightsquigarrow j) \mapsto E_k$ it is required to demonstrate the following:

$$\forall e \cdot (e \in E_k \wedge I(c,s,v) \wedge H_e(c,s,v) \wedge S_e(c,s,v,v') \Rightarrow A_i(v) \wedge A_j(v')) \tag{11}$$

The composition of modes and Event-B clarifies how a system evolves when it is in a mode, how mode switching is done and the way system is initialised. The old *internal* rule is changed to reflect the way a new system state is computed: assuming that a system is mode $A_i/G_i \mapsto E_i$ and the current state is valid ($I(v)$ holds) and satisfies the mode assumption (A_i holds) the next state is some state v' such that mode guarantee $G(v,v')$ holds along with before-after predicate $R_e(v,v')$ of one of enabled ($H_e(v)$) mode events ($e \in E_i$):

$$\boxed{\text{internal}_1}\ \frac{I(v) \wedge A_m(v) \wedge G_m(v,v') \wedge A_m(v') \quad \exists e \cdot e \in E_i \wedge H_e(v) \wedge R_e(v,v')}{\langle m, w \rangle \rightarrow \langle m, w' \rangle}$$

The above states that an execution cannot progress if none of the events establishes a mode guarantee or there is no enabled event. To ensure that in a given mode a system evolves correctly, it is required to show for every mode event that the event establishes mode guarantee and the event guard is compatible with the mode assumption. Rules switching_1 and start_1 are analogously obtained from rules switching and start in Section II. The rule above gives a rise to a number of conditions on Event-B. Firstly, all the events of a mode must satisfy the mode guarantee provided the mode assumption holds:

$$I(v) \wedge A(v) \wedge H(v) \wedge R(v,v') \Rightarrow G(v,v') \tag{12}$$

Also, the partitioning of the events into modes must be in an agreement with the event guards. When event is enabled then the assumption of its mode must hold. Since an event is potentially associated with multiple modes, the disjunction of all the relevant assumptions must hold:

$$\begin{array}{l} H(v) \Rightarrow A_1(v) \vee \dots \vee A_k(v) \\ A_{k+1}(v) \vee \dots \vee A_n(v) \Rightarrow \neg H(v) \end{array} \tag{13}$$

where $A_1, \dots, A_k$ are the assumptions of the modes containing an event with guard $H(v)$ and $A_{k+1}, \dots, A_n$ are those not containing the event.

It is required to show that a system is always able to progress once it is in a given mode. For this, it must be shown that there is always at least one enabled event among the events of the mode:

$$I(v) \wedge A(v) \Rightarrow H_1(v) \vee \dots \vee H_n(v) \tag{14}$$

Provided the three conditions above are discharged, it is guaranteed that, once in a given mode, a system would unfailingly progress in accordance with the mode conditions for the system lifetime or until the system transitions into a different mode.

a) Operation Modes and Event-B Co-refinement: The cornerstone of the Event-B development method is a gradual, refinement-based, model detailing. To refine model M one constructs a new model M' such that at a certain level of observation new model is at least as good as the old one. Formally, this is demonstrated by constructing a refinement mapping between M' and M that would show that for any valid state of M' there is a corresponding state in M. In Event-B, this is accomplished by discharging a number of refinement proof obligations formulated for each model event. As refinement in Event-B is monotonic, a model refinement could be constructed by changing only a part of a model and demonstrating the relevant conditions for just that part. Event-B refinement is a combination of data, superposition, behavioural and atomicity refinement. Atomicity refinement permits introduction of a finer level of atomic steps needed to realise a given functionality. What would appear as an one atomic event in an abstract model may be refined into a complex of events with all the properties of the abstraction retained. The Event-B notion of data refinement follows the same generic style used for operation modes data refinement.

Event-B behavioural refinement allows a modeller to replace an event guard and event before-after predicate. The rules linking abstract and concrete guards and before-after predicates are as follows. The guard of the concrete version of an event must be stronger than its abstract counterpart:

$$P(s,c) \wedge I(s,c,v) \wedge J(s,c,v,u) \wedge H(s,c,u) \Rightarrow G(s,c,v) \tag{15}$$

A new before-after predicate must be a stronger version of its abstraction:

$$\begin{array}{l} P(s,c) \wedge I(s,c,v) \wedge J(s,c,v,u) \wedge H(s,c,u) \wedge \\ \quad S(s,c,u,u') \Rightarrow v' \cdot (R(s,c,v,v') \wedge J(s,c,v',u')) \end{array} \tag{16}$$

An event may be split into two or more events. In this case, the refinement relation is proved for each new event in the same manner for as for on-to-one event refinement. New events may be introduced but may only update new variables. Standard consistency conditions apply.

A composition of operation modes and Event-B models has to be refined in such a manner that it obeys both operation mode refinement and Event-B refinement. For rule 5, it is required that a refined operation mode is made of events refining events from an abstract mode and also each event from the abstract mode is present as a copy or a refined event in the refined mode.

$$A(v)/G(v,v') \mapsto E \sqsubseteq A'(u)/G'(u,u') \mapsto E', \\ \text{iff} \begin{cases} I(v) \wedge J(v,u) \wedge A(v) \Rightarrow A'(u) \\ I(v) \wedge J(v,u) \wedge G'(u,u') \Rightarrow G(v,v') \\ \forall e \cdot e \in E' \Rightarrow \exists a \cdot a \in E \wedge e \sqsubseteq a \\ \forall e \cdot e \in E \Rightarrow \exists a \cdot a \in E' \wedge a \sqsubseteq e \end{cases} \tag{17}$$

Rule 6 for refinement of modes into a collection of new modes is changed in a similar manner.

$$A(v)/G(v,v') \mapsto E \sqsubseteq \begin{matrix} A_1(u)/G_1(u,u') \mapsto E_1 \\ A_2(u)/G_2(u,u') \mapsto E_2 \end{matrix}, \\ \text{iff} \begin{cases} I(v) \wedge J(v,u) \wedge A(v) \Rightarrow A_1(u) \vee A_2(u) \\ I(v) \wedge J(v,u) \wedge G_1(u,u') \vee G_2(u,u') \Rightarrow G(v,v') \\ \forall e \cdot e \in E_1 \cup E_2 \Rightarrow \exists a \cdot a \in E \wedge e \sqsubseteq a \\ \forall e \cdot e \in E \Rightarrow \exists a \cdot a \in E_1 \cup E_2 \wedge a \sqsubseteq e \end{cases} \tag{18}$$

Conditions 17 and 18 state how mode refinement is related to Event-B model refinement. They are the basis for generating proof obligations that would determine the correspondence between an Event-B model and a modes model.

b) Tool Support for Modes Modelling: As already mentioned, the Rodin platform supports modelling and reasoning with Event-B models. Extensions to the Rodin platform can be integrated with: tool interface, modelling process and verification infrastructure. An extension providing the support for modelling with modes would let a designer to visually construct a modes model and would take care of generating the proof obligations required to demonstrate the correspondence between the modes model and the associated Event-B model (defined by relation (10)). Proof obligations are delegated to the proof infrastructure of the Platform that passes them on to one or of automated theorem provers and also an interactive prover should a theorem prover find a problem or fail to discharge a proof obligation.

VI. Cruise Control Case Study

A simplified version of one of the DEPLOY case studies [14] developed in cooperation with industrial partners, the case study illustrates the application of the proposed technique to the development of a cruise control system.

The purpose of the system is to assist a driver in reaching and maintaining some predefined speed. Due to the nature of the system, a lot of attention is given to the interaction of a driver, cruise control and the controlled parts of a car. In the current modelling we assume an idealised car and idealised driving conditions such that the car always responds to the commands and the actual speed is updated according to the control system commands.

a) A Mode for the Ignition Cycle: At the most abstract level we introduce mode *IGNITION_CYCLE* to represent the activity from the instant the ignition is turned on to the instant it is turned off. The initial model includes: the state of ignition (on/off), modelled by a boolean flag ig; the current speed of the car (a modelling approximation of an actual car speed), stored in variable sa; a safe speed limit $speedLimit$ above which the car should not be in any case; and a safe speed variation $maxSpeedVariation$. No memory is retained about the states in the previous ignition cycle. Initially, the current speed is zero and ignition is off: $sa \in 0 \wedge ig \in FALSE$.

Independently of the operation of the car - by the driver or by the cruise control - the following has to be ensured during an ignition cycle (we present the intuition in the first line and a formal representation of the same assumptions and guarantees, based on the variables introduced, in the second line).

mode	assumption	guarantee
IGNITION_ CYCLE	ignition is on	keep speed under limit and (ac/de)celarate safely
	$ig = true$	$(sa < speedLimit) \wedge (\lvert sa' - sa\rvert < maxSpeedVariation)$

Figure 3(A) presents the diagram of the system. At this level of abstraction it is composed only by the *IGNITION_CYCLE* mode. An event happens in the system that establishes the assumption for that mode: $ignitionOn$. While ignition is on, the corresponding guarantees have to be ensured. Another event may change the conditions of the system and the assumptions for this mode may become false: $ignitionOff$.

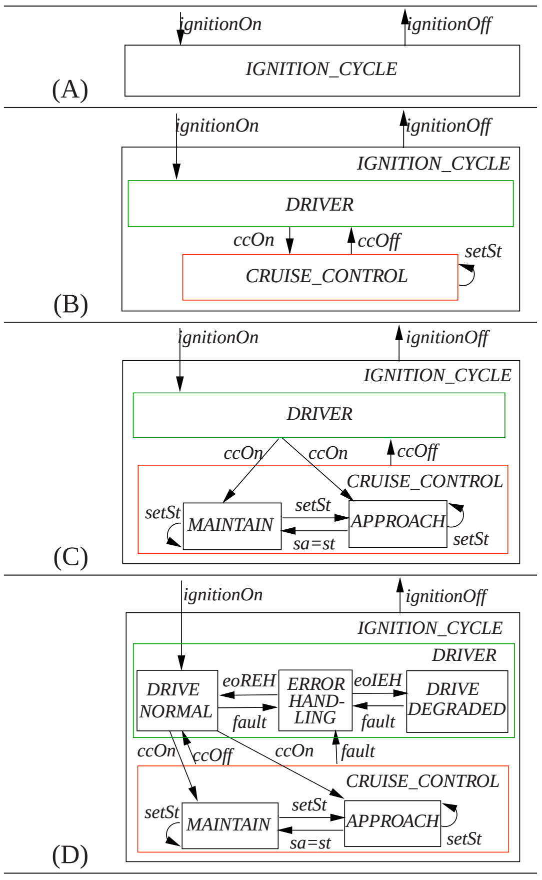


Fig. 3. Mode refinement sequence for the Cruise Control System.

b) DRIVER and CRUISE_CONTROL Modes: When the ignition is turned on, control is with the driver. While the ignition is on, control can be passed from the driver to the cruise control and back. It is assumed that a driver has two buttons on a control panel: the *on* button switches on the cruise control; the *off* button returns to the driving mode. A third input is available to set the target speed to be achieved by the cruise control. The system is naturally represented with two modes: *DRIVER* corresponding to the activity when cruise control is off and *CRUISE_CONTROL* when cruise control is active. The *on/off* buttons mentioned are mapped to transition events $ccOn$ and $ccOff$. The diagram in Figure 3(B) depicts the two possible modes during an ignition cycle.

This refinement introduces: the state of cruise control (on/off), modelled by boolean flag cc; the target speed that a cruise control is to achieve and maintain, represented by variable st; an allowance interval isp that determines how much actual speed could deviate from a target speed when cruise control tries to maintain a target speed. Initially, the target speed is undefined and cruise control is off: $st \in \mathbb{N} \wedge cc \in FALSE$. The description of the modes:

mode	assumption	guarantee
DRIVER	ignition cycle assumptions and cruise control off	ignition cycle guarantees
	$ig = true$ $\wedge$ $cc = false$	$(sa < speedLimit) \wedge (\lvert sa' - sa\rvert < maxSpeedVariation)$
CRUISE_ CONTROL	ignition cycle assumptions and cruise control on	ignition cycle guarantees and maintain target speed or approach target speed
	$ig = true$ $\wedge$ $cc = true$	$(sa < speedLimit) \wedge (\lvert sa' - sa\rvert < maxSpeedVariation) \wedge (\lvert sa' - st'\rvert \leq isp \vee \lvert sa' - st'\rvert < \lvert sa - st\rvert)$

c) Refining the CRUISE_CONTROL Mode: If the difference between current (sa) and target (st) speeds is within an acceptable error interval (isp), the cruise control works to *MAINTAIN* the current speed. Otherwise, it employs different procedures to *APPROACH* the target speed, characterizing two modes refining *CRUISE_CONTROL* with assumptions and guarantees are as follows.

mode	assumption	guarantee
APPROACH	cruise control assumptions and speed not close to target	cruise control guarantees and approach target speed
	$ig = true \wedge$ $cc = true \wedge$ $\lvert sa' - st'\rvert > isp$	$(sa < speedLimit) \wedge (\lvert sa' - sa\rvert < maxSpeedVariation) \wedge (\lvert sa' - st'\rvert < \lvert sa - st\rvert)$
MAINTAIN	cruise control assumptions and speed close to target	cruise control guarantees and maintain target speed
	$ig = true \wedge$ $cc = true \wedge$ $\lvert sa' - st'\rvert \leq isp$	$(sa < speedLimit) \wedge (\lvert sa' - sa\rvert < maxSpeedVariation) \wedge (\lvert sa' - st'\rvert \leq isp)$

Figure 3(C) depicts these modes. Switching from *DRIVER* to *CRUISE_CONTROL* may either establish the assumptions of *APPROACH* or *MAINTAIN*, depending on the difference between st and sa. In either of these two modes the cruise control can be switched off and the control returned to the driver.

d) Error handling: at any time failures of the surrounding components (e.g. airbag activated, low energy in battery, etc.) may happen and affect the cruise control system. These faults are typically signaled to the cruise control system as erroneous conditions. The conditions can be either reversible or irreversible: the reversible errors results in the control to be returned to the driver and handling measures to be undertaken, so that the cruise control becomes available again; the irreversible ones are handled but the cruise control becomes unavailable during the ignition cycle.

When an error is detected it is registered in an $error$ variable. We introduce a normal (*DRIVE_NORMAL*), a degraded (*DRIVE_DEGRADED*) and an error handling mode (*ERROR_HANDLING*). If an error is signaled in any of the system modes, the system switches to *ERROR_HANDLING*, where control is with the driver. Eventually error handling reestablishes *DRIVE_NORMAL*, with full functionality available, or switches to *DRIVE_DEGRADED* mode where the cruise control is not available. This exemplifies situations (C) and (D) of Figure 2. Figure 3(D) shows these modes. An $eHand$ variable registers that error handling is taking place. The following table shows the assume/guarantee conditions for the modes introduced. Note that although these modes have same guarantees, they have different transition possibilities. After error handling, the system continues in degraded or normal mode. From error handling and degraded modes it is not possible to turn the cruise control on.

mode	assumption	guarantee
DRIVE_ NORMAL	driver assumptions and no error	driver guarantees and cruise control available
	$ig = true \wedge$ $cc = false \wedge$ $error = false$	$(sa < speedLimit) \wedge (\lvert sa' - sa\rvert < maxSpeedVariation)$
ERROR_ HAND- LING	driver assumptions and error and handling not finished	driver guarantees and cruise control not available and recovery measures restore normal mode or swich to degraded mode
	$ig = true \wedge$ $cc = false \wedge$ $error = true \wedge$ $eHand = true$	$(sa < speedLimit) \wedge (\lvert sa' - sa\rvert < maxSpeedVariation)$
DRIVE_ DEGRA- DED	driver assumptions and error and handling finished	driver guarantees and cruise control not available
	$ig = true \wedge$ $cc = false \wedge$ $error = true \wedge$ $eHand = false$	$(sa < speedLimit) \wedge (\lvert sa' - sa\rvert < maxSpeedVariation) \wedge$

VII. Related Work

Several applications, structured in modes, can be found in the literature. Papers [7] and [8] show how to formally model and analyse modal space and avionic systems. In [15] the extension of an Automated Highway System to tolerate several kinds of faults is discussed, and modes are used to characterize degraded operation. A classic case study showing the use of formal methods, the Steam Boiler Control [16], is based on the notion of operation modes. More recent examples on the extensive use of modes for the specification of airspace, transportation and automotive systems can be found in [14]. Such contributions focus on specific applications and not on general means to model and reason using modes.

In [17] the authors discuss characteristics of mode-driven distributed applications and a software architecture with extensions to mode-driven fault-tolerance. An infrastructure is proposed to support mode-driven fault tolerance in run time. In [18], the representation of degraded service outcomes and exceptional modes of operation using UML use cases, activity diagrams and state charts is discussed. Formal modelling and reasoning is not discussed in these contributions.

In [6] a specification language for real-time systems, called Modechart, is presented. In [19] the author discusses issues related to mode changes and scheduling for hard real-time systems. The general notion of modes in these papers is analogous to the one discussed here, however their focus is on the specification and analysis of timing properties of systems. Functional properties are not discussed.

In the context of refinement based methods, the most related work found is by Back and von Wright [20], where guarantees (of an action system) are introduced to reason about the parallel composition of action systems. Guarantees of composed action systems have to mutually respect the invariants. Since there is no notion of assumptions (they are embedded in the invariants), the flexibility of changing assumptions, allowing different modes and mode switching, is not offered.

Finally, Jones, Hayes and Jackson, in [21], address the problem of obtaining a starting specification for systems that interact with the physical world, like control systems. A method is discussed that leads the designer to explicitly state rely conditions (to be compared with assumptions) about the physical world before deriving a first specification of the system. The notion of 'layer' is briefly discussed. A layer is associated to a set of rely/guarantee predicates and could be compared to a mode. Different layers could be used to state the behaviour under distinct conditions. Fault tolerance is briefly mentioned, where one could have assumptions to characterise absence or presence of faults.

VIII. Conclusions

In this paper the notions of modes and mode refinement are formally defined and their representations in a state-base formalism (Event-B) are established. These notions allow explicit characterization of various system conditions, through expressing assumptions, and the properties of the system working under such conditions, through the use of guarantees. The complexity of design is reduced by structuring systems using modes and by detailing this design using refinement. This approach makes it easier for the developers to map requirements to models and to trace requirements. More specifically, the approach suits well for dealing with fault-tolerance requirements: assumptions allow the explicit mapping of the error coverage provided by the system, whereas guarantees and mode switching configurations allow the explicit mapping of requirements for different levels of fault-tolerance.

In addition to developing a tool support, in the near future we plan to investigate mode hierarchy (nesting), to express recursive structuring for fault tolerance [22], mode concurrency, where further work is needed to support concurrent modes acting on shared states, and state consistency during distributed execution of modes.

References

[1] J.-C. Laprie, B. Randell, A. Avizienis, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable Secur. Comput.*, vol. 1, no. 1, pp. 11–33, 2004.

[2] F. Cristian, *Exception handling*, T. Anderson, Ed. Blackwell Scientific Publications, 1989.

[3] A. Romanovsky, "A looming fault tolerance software crisis?" *SIGSOFT Softw. Eng. Notes*, vol. 32, no. 2, pp. 1–4, 2007.

[4] J. Peleska, "Formal methods and the development of dependable systems - habilitationsschrift," Institut für Informatik und Praktische Mathematik der Christian-Albrechts-Universität su Kiel, Tech. Rep. 9612, 1996.

[5] F. C. Gärtner, "Transformational approaches to the specification and verification of fault-tolerant systems: formal background and classification," *Journal of Univ. Computer Science*, vol. 5, no. 10, pp. 668–692, 1999.

[6] F. Jahanian and A. Mok, "Modechart: A specification language for real-time systems," *IEEE Transactions on Software Engineering*, vol. 20, no. 12, pp. 933–947, 1994.

[7] R. W. Butler, "Nasa tech. memo. 110255 an introduction to requirements capture using pvs: Specification of a simple autopilot," p. 29, 1996.

[8] S. P. Miller, "Specifying the mode logic of a flight guidance system in core and scr," in *FMSP '98: Proc. of the 2nd workshop on Formal methods in software practice*. New York, USA: ACM, 1998, pp. 44–53.

[9] R.-J. J. Back and J. V. Wright, *Refinement Calculus: A Systematic Introduction*. Springer-Verlag New York, Inc., 1998.

[10] J. R. Abrial, *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 2005.

[11] R.-J. Back and K. Sere, "Stepwise Refinement of Action Systems," in *Proceedings of the International Conference on Mathematics of Program Construction, 375th Anniv. of the Groningen Univ.*, J. L. A. van de Snepscheut, Ed. London, UK: Springer-Verlag, 1989, pp. 115–138.

[12] "Event-b and the rodin platform," http://www.event-b.org/ (last accessed 8 March 2009). Rodin Development is supported by European Union ICT Projects DEPLOY (2008 to 2012) and RODIN (2004 to 2007).

[13] E. Dijkstra, *A Discipline of Programming*. Prentice-Hall Int., 1976.

[14] J.-R. Abrial, J. Bryans, M. Butler, J. Falampin, T. S. Hoang, D. Ilic, T. Latvala, C. Rossa, A. Roth, and K. Varpaaniemi, "Report on knowledge transfer - deploy deliverable d5," p. 321, February 2009.

[15] J. Lygeros, D. N. Godbole, and M. E. Broucke, "Design of an extended architecture for degraded modes of operation of ivhs," in *In American Control Conference*, 1995, pp. 3592–3596.

[16] J.-R. Abrial, E. Börger, and H. Langmaack, Eds., *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control (the book grow out of a Dagstuhl Seminar, June 1995)*, ser. Lecture Notes in Computer Science, vol. 1165. Springer, 1996.

[17] D. Srivastava and P. Narasimhan, "Architectural support for mode-driven fault tolerance in distributed applications," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–7, 2005.

[18] S. Mustafiz, J. Kienzle, and A. Berlizev, "Addressing degraded service outcomes and exceptional modes of operation in behavioural models," in *SERENE '08: Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems*. New York, NY, USA: ACM, 2008, pp. 19–28.

[19] G. Fohler, "Realizing changes of operational modes with a pre run-time scheduled hard real-time system," in *In Proceedings of the Second International Workshop on Responsive Computer Systems*. Springer Verlag, 1992, pp. 287–300.

[20] R.-J. Back and J. von Wright, "Compositional action system refinement," *Formal Asp. Comput.*, vol. 15, no. 2-3, pp. 103–117, 2003.

[21] C. B. Jones, I. J. Hayes, and M. A. Jackson, "Deriving specifications for systems that are connected to the physical world," in *Formal Methods and Hybrid Real-Time Systems*, 2007, pp. 364–390.

[22] P. A. Lee and T. Anderson, *Fault Tolerance: Principles and Practice*, J. C. Laprie, A. Avizienis, and H. Kopetz, Eds. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1990.

On Event-B and Control Flow

A. Iliasov

Centre for Software Reliability, Newcastle University, UK
alexei.iliasov@newcastle.ac.uk

Abstract. The paper presents an extension of the Event-B method with a viewpoint portraying control flow properties of a model. The novelty of the work is in relying solely on theorem proving to demonstrate the consistency of control flow and main Event-B specification. The focus is placed on the practicality of working with such an extension and also on achieving proof economy. A detailed formal treatment of the method is presented and illustrated with a case study. A proof of concept implementation for the RODIN platform is briefly discussed.

1 Introduction

Event-B [1–3] is a general-purpose specification language and is a close relative of the popular B-Method [4](or Classical B). Its distinctive feature is relying on the event-based specification paradigm. An Event-B model is a collection of events where the next event is selected non-deterministically among the currently enabled events. Event-B facilitates construction of models with a large number of rather simple events. Theorem proving is the primary verification technique and, crucially, almost all the correctness conditions (proof obligations) are formulated on per-event basis. This makes Event-B very friendly to automated theorem provers. High rate of verification automation is extremely important and it makes Event-B one of the few practical proof-based formalisms.

However, there are some downsides in following pure event-based paradigm. Not all systems are naturally expressed in this style. Often the information about event ordering has to be embedded into guards and event actions. This results in an entanglement of control flow and functional specification with an additional downside of extra model variables.

There are a number of reasons to consider an extension of Event-B with an event ordering mechanism:

- for some problems the information about event ordering is an essential part of requirements; it comes as a natural expectation to be able to adequately reproduce these in a model;
- explicit control flow may help to prove properties related to event ordering;
- sequential code generation requires some form of control flow information;
- since event ordering could restrict the non-determinism in event selection, model checking is likely to be more efficient for a composition of a machine with event ordering information;

– a potential for a visual presentation based on control flow information;
– bridging the gap between high-level workflow and architectural languages, and Event-B.

In this paper we discuss an extension of Event-B with a mechanism to reason about event ordering. The practical issues, like verification means, the integration with the Event-B development process and the tooling support are given the highest priority. Unlike much of the work on combining state-based and process-bases specification methods [5–8] our proposal is based on theorem proving rather than model checking. We demonstrate that the proposal is realistic and presents distinct practical advantages with a proof-of-concept tool realising the technique.

2 Flow Model

The Flow View extends Event-B with a facility for defining event ordering. A flow is an expression written in a special language resembling those used in process algebras, such CSP [10]. The basic element of the language is an event. Events in a flow are the same events as in an Event-B machine. Events are characterised by an event label and may have parameters (in flow analysis these are treated as an integral part of an event label). The following is the summary of the constructs forming the flow language:

$e.a$	event with label e and arguments a
$p; q$	sequential composition
$p \|\|_E q$	parallel composition synchronised on events from E
$p \sqcap q$	choice
$*(p)$	terminating loop
$'start, 'stop, 'skip$	initialisation, termination and stuttering events

where p and q are flow expressions. Events starting with $'$ bear special meaning. $'start$ is a shortcut for Event-B event INITIALISATION, $'stop$ is an assumed termination event and $'skip$ corresponds to an implicit Event-B *skip* event.

An essential part of the flow mechanism is the notion of a *partial flow* expression (or simply partial flow). There are situations when it is not necessary to mention all the machine events in a flow. For example, one may want to state flow for a part model corresponding to the current refinement step or simply focus on a part where flow reasoning is required. The notion of partial flow becomes clear if one thinks of a flow expression as a set of conditions formulated on a machine. A partial flow is then a more relaxed version of a complete flow.

There are some basic well-formedness requirements to a flow. Event $'start$ corresponding to the initialisation event of a machine may not be composed with other events using choice and parallel composition. Also, it may only occur on the left-hand side of a sequential composition. This restriction is due to the fact that the initialisation event is a special case in Event-B. It has no guard and is always a first event to run. Since flows may be partial, initialisation event may be omitted from a flow. The termination event $'stop$ also needs special treatment. This event is not present explicitly in a machine and the following Event-B definition is implied if the event is present in a flow expression: $'stop = \texttt{when}\ \neg(G_1 \vee \cdots \vee G_n)\ \texttt{then skip end}$. Event $'stop$ is enabled when all other events are disabled; it executes infinitely but keeps the state intact so that a machine cannot get into a state when anything else is enabled. Since this event diverges it is not possible to have any other event to follow $'stop$. Hence, $'stop$ may not occur on the right-hand side of a sequential composition. For the same reason, a parallel composition with $'stop$ is disallowed. It is possible, however, to have a choice between $'stop$ and another event (including $'start$).

In a composition of a flow and machine, the flow loop construct $*(p)$ would correspond to a loop on machine events. It is the responsibility of the Event-B part to demonstrate the convergence of a loop. This is a standard part of model analysis in the RODIN Event-B environment. Later we discuss how to improve the strategy of demonstrating convergence in Event-B by using the information contained in a flow attached to a machine.

In the context of Event-B models, the parallel composition may only be applied to certain kind of events. We require that for any set of parallel events (as defined in a flow expression) there exists a well-formed Event-B event that simulates all the possible interleavings of the parallel events. This condition results in a number of syntactic requirements to machine events.

Let $rd(e)$ return the set of all variables read by event e. These are the model variables referenced in the event guard and the event actions. Likewise, $wr(e)$ is a set of variables updated by event e. These are the variables found on the left-hand side of substitutions in an event body. Events that are potentially concurrent are called independent events.

Definition 1. Independent events. *Events that do not have read/write and write/write conflict are independent. The conflicts are defined as follows:*

- Read/write conflict. *A pair of events have a read conflict if one updates the variables read by another. This is denoted as* $rdcfl(e_1, e_2) = rd(e_1) \cap wr(e_2) \neq \oslash \vee rd(e_2) \cap wr(e_1) \neq \oslash$.
- Write/write conflict. *Events updating the same variable have a write conflict:* $wrcfl(e_1, e_2) = wr(e_1) \cap wr(e_2) \neq \oslash$.

Set E of events is independent, denoted as $ind(E)$, if for every event pair (a, b) from E the following holds: $a \neq b \implies \neg rdcfl(a, b) \wedge \neg wrcfl(a, b)$

The condition $ind(\dots)$ must be established for all possible event pairs composed with the parallel composition operator.

3 Semantics

This section discusses the semantics of the flow language and the way to integrate it with Event-B. In particular we show how to reason about flow and machine consistency in the terms of machine properties rather than flow or machine traces. But first we use the traces semantics to formally integrate flows with Event-B. The following defines the traces of a flow expression.

$$
\begin{array}{ll}
\mathit{traces}('\mathit{skip}) & \widehat{=} \{\langle\rangle\} \\
\mathit{traces}('\mathit{start}) & \widehat{=} \{\langle '\mathit{start}\rangle\} \\
\mathit{traces}('\mathit{stop}) & \widehat{=} \{s \mid n \in \mathbb{N} \wedge s \leq \langle '\mathit{stop}\rangle^n\} \\
\mathit{traces}(e_i.a) & \widehat{=} \{\langle e_i.a\rangle\} \\
\mathit{traces}(p;q) & \widehat{=} \{s^\frown z \mid s^\frown z \in \mathit{traces}(p) \wedge z = \langle '\mathit{stop}\rangle\} \cup \\
 & \quad \{s^\frown t \mid s^\frown z \in \mathit{traces}(p) \wedge t \in \mathit{traces}(q) \wedge z \neq \langle '\mathit{stop}\rangle\} \\
\mathit{traces}(p|q) & \widehat{=} \mathit{traces}(p) \cup \mathit{traces}(q) \\
\mathit{traces}(*(p)) & \widehat{=} \mathit{traces}(p|(p;*(p))) \\
\mathit{traces}(p\|_E q) & \widehat{=} \{\bigcup(s\overline{\|}_E t \mid s \in \mathit{traces}(p) \wedge t \in \mathit{traces}(q)\}
\end{array}
$$

Here $s \leq t$ states that trace s is a prefix of trace t; $\alpha(x)$ is an alphabet of x (set of all events occurring in x). The parallel composition operator is defined as a collection of possible event interleavings:

$$
\begin{array}{lll}
\langle\rangle\overline{\|}_E\langle\rangle & \widehat{=} \{\langle\rangle\} & \\
\langle a\rangle^\frown p\overline{\|}_E\langle\rangle & \widehat{=} \oslash & a \in E \\
\langle a\rangle^\frown p\overline{\|}_E\langle\rangle & \widehat{=} \{\langle a\rangle^\frown s | s \in (p\overline{\|}_E\langle\rangle)\} & a \notin E \\
\langle a\rangle^\frown p\overline{\|}_E\langle b\rangle^\frown q & \widehat{=} \oslash & a \in E \wedge b \in E \wedge a \neq b \\
\langle a\rangle^\frown p\overline{\|}_E\langle b\rangle^\frown q & \widehat{=} \{\langle a\rangle^\frown s | s \in (p\overline{\|}_E q)\} & a \in E \wedge b \in E \wedge a = b \\
\langle a\rangle^\frown p\overline{\|}_E\langle b\rangle^\frown q & \widehat{=} \{\langle b\rangle^\frown s | s \in (\langle a\rangle^\frown p\overline{\|}_E q)\} & a \in E \wedge b \notin E \\
\langle a\rangle^\frown p\overline{\|}_E\langle b\rangle^\frown q & \widehat{=} \{\langle a\rangle^\frown s | s \in (p\overline{\|}_E\langle b\rangle^\frown q)\} \cup & a \notin E \wedge b \notin E \\
 & \quad \{\langle b\rangle^\frown s | s \in (\langle a\rangle^\frown p\overline{\|}_E q)\} &
\end{array}
$$

$p\overline{\|}_E q$ constructs all the possible interleavings of p and q while respecting the synchronisation on common events E.

3.1 Event-B Trace Semantics

In this section we briefly present how traces of an Event-B model are constructed. Much more detailed treatment of the subject is given in [11] and [12].

An elementary step of a machine interpretation is the computation of the set of next states for some current event. For some event e the next states are found by selecting a set of suitable values for the event parameters and using them to characterise the possible next states v'. An Event-B machine may be understood as a relation $T : \mathit{Event} \leftrightarrow S \leftrightarrow S$: $T \overset{\mathrm{df}}{=} \exists p_e \cdot (G_e(p_e, v) \wedge S_e(p_e, v, v'))$. Here p_e, G_e, S_e are the event parameters, guard and before-after predicate. T is a predicate characterising a relation on system states: it is a total function from events to relations on states. A next event would start from a state produced by a previous event. This is expressed with the sequential composition operator ";": $e_1; e_2 = \forall v_1 \cdot T(e_1)[v_1/v'] \wedge T(e_2)[v_1/v]$. v_1 is a vector of fresh names used to record the final state of e_1 and pass it on to e_2. The concept of sequential composition can be generalised to a chain of events. Operator seq performs a sequential composition over an event list: $\mathit{seq}(\langle\rangle) = \mathit{id}(S)$ and $\mathit{seq}(\langle e\rangle t) = T(e); \mathit{seq}(t)$. From these definitions, the traces of a machine are formulated as all possible traces reachable from the initial machine state Init:

$$\mathit{traces}(M) = \{t \mid \mathit{seq}(t)[\mathit{Init}] \neq \oslash\}$$

In the next section we use the traces semantics of flows and Event-B to define the consistency conditions for a model combining a flow expression and an Event-B machine.

3.2 Flow/Machine Consistency

The minimal requirement to a given pair of a flow and machine is that the two agree on deadlocks and divergences. To account for partial flows it is required to consider a situation when only a part of a machine traces is specified by a flow. A flow trace starting with $'start$ and eventually reaching $stop$ would match a complete machine trace if it matches any trace at all.

Definition 2. Flow consistency. *A flow f is consistent with a given machine m if it is possible to find a machine trace that contains some flow trace: $\exists t, hd, tl \cdot t \in traces(f) \wedge hd^\frown t^\frown tl \in traces(m)$.*

One important case of a flow and machine combination is when flow event ordering and event guards together define a concrete, implementable event ordering. Individually, both flow expression and machine still may have non-deterministic event choice. Such a property is essential for code generation and sometimes is a desired property of a model. While choice related non-determinism must be resolved, non-deterministic event ordering may still be present due to the parallel composition operator. To distinguish between these two cases we use the notion of interleaving equivalence.

Two traces are said to be interleave equivalent if one can be obtained from another by swapping events in a pair of independent events. This is formulated using the following relation on traces: $s\ Re\ t \Leftrightarrow s = t \vee \exists a, b, hd, tl \cdot (hd^\frown \langle a, b \rangle^\frown tl \in t \wedge hd^\frown \langle b, a \rangle^\frown tl \in s \wedge ind(\{a, b\}))$. Traces s and t are said to be interleave equivalent if $s\ Re^*\ t$ where Re^* is a transitive closure of Re.

Definition 3. Concrete flow. *The traces contained in the intersection of a* concrete flow *and machine traces are interleave equivalent.*

Having these definitions does not lead to practical means of establishing flow properties especially since it is our intention is to use theorem proving to reason about a combination of a flow and machine. In the rest of the section we discuss how to transition from statements about traces of flows and machines to equivalent conditions on machine variables, events guards and event actions. First, some mathematical context is presented. This gives a basis for theorems reformulating the definitions of consistent and concrete flows in terms of machine properties. In its turn, this gives a foundation for deriving proof obligations.

In a general case, an event may be preceded by any configuration of choice and parallel composition. Let us consider the following example: $((a \| b) | (c \| d)); z$. Event z gets enabled as soon as both a and b or c and d terminate. One has to show that for any possible situation (that is, the first and the second branch of the choice) it is possible to pass control to z. Even more complex case is demonstrated by the following expression: $((a \| b) | (c \| d)); ((e \| f) | (g \| h))$. For this, one also has to consider a multitude of options on the right-hand side. The notions of *entry and exit points* are introduced to reason about events actively involved in passing control in a sequential composition. These are defined as follows:

$$\begin{array}{llll}
EN(e) & = \{\{e\}\} & EX(e) & = \{\{e\}\} \\
EN('skip) & = \{\{\}\} & EX('skip) & = \{\{\}\} \\
EN('start) & = \{\{'start\}\} & EX('start) & = \{\{'start\}\} \\
EN('stop) & = \{\{'stop\}\} & EX('stop) & = \{\{'stop\}\} \\
EN(p;q) & = EN(p) \quad p \neq' skip & EX(p;q) & = EX(q) \quad q \neq' skip \\
EN('skip;q) & = EN(q) & EX(p;'skip) & = EX(p) \\
EN(p|q) & = EN(p) \cup EN(q) & EX(p|q) & = EX(p) \cup EX(q) \\
EN(p\|q) & = \{EN(p) \cup EN(q)\} & EX(p\|q) & = \{EX(p) \cup EX(q)\} \\
EN(*(p)) & = EN(p) & EX(*(p)) & = EX(p)
\end{array}$$

where $EN(x)$ is a set of entry points of a flow expression x. Correspondingly, $EX(x)$ denotes the set of exit points. Note that entry and exits points are set of sets. The reason is that a combination of parallel composition and choice results in a set of event clusters. For example the set of entry points of $((a\|b)|(c\|d)); z$ is $\{\{a, b\}, \{c, d\}\}$. This set contains two entry points $\{a, b\}$ and $\{c, d\}$ where each entry points is set itself denoting a complex entry point of a parallel composition construct.

Independent events may be *merged* into a single event[1]. Indeed, since independent events are conflict free and can be executed in any order there is nothing that prevents an existence of a single event that would have the same effect as possible interleavings of the independent events. This is a purely abstract construction. There is, of course, no need to actually introduce merged events in a model.

Definition 4. Operator $merge(a, b)$. *The operator constructs a single event from the definitions of events a and b. It is well-defined only when a and b are independent. For some events a and b,*

$a = \texttt{any}\ p\ \texttt{where}\ G(p, d)\ \texttt{then}\ S(p, d, w')\ \texttt{end}$

$b = \texttt{any}\ q\ \texttt{where}\ H(q, g)\ \texttt{then}\ R(q, g, u')\ \texttt{end}$

a merged event takes the following general form:

$a = \texttt{any}\ p, q\ \texttt{where}\ G(p, v) \wedge H(q, v)\ \texttt{then}\ S(p, v, v') \wedge R(q, v, v')\ \texttt{end}$

Constant c and set s are omitted but implied in guards and before-after predicates.

Since only independent events may be merged, the resultant merged event enjoys a number of properties. It is enabled when both its donor events are enabled and simulates the effect of interleaving the merged events. A merged event is feasible as long as its individual donor events are feasible. It is straightforward to see that the state observed after executing a merged event is the same state as one would observe after executing both donor events in any order. Event merging is a special case of *event fusion* [13].

[1] Event-B uses event merging as a refinement technique. This has nothing to do with our definition of merging.

Definition 5. Operator $s \fatsemi_m t$. *This operator defines the consistency conditions for a sequential composition where control is passed from a collection of exit points s to a collection of entry points t. The operator type is*

$\fatsemi : M \times \mathbb{P}(\mathbb{P}(Event)) \times \mathbb{P}(\mathbb{P}(Event)) \to BOOL$

where M is an Event-B model and Event is a set of model events; the second and the third parameters are some exit and entry points. The strategy is to construct an Event-B event implementing what is essentially a sequential composition of s and t. The feasibility conditions for the event would demonstrate the well-formedness of a sequential composition.

Let us first consider a simple case of a composition of two events when $s = \{\{e_1\}\}$ *and* $t = \{\{e_2\}\}$. *Events* e_1 *and* e_2 *are defined as follows (these definitions come from an Event-B machine that is supplied as the first parameter to operator):*

$e_1 = \textit{any}\ p\ \textit{where}\ G(p, v)\ \textit{then}\ S(p, v, v')\ \textit{end}$
$e_2 = \textit{any}\ q\ \textit{where}\ H(q, v)\ \textit{then}\ R(q, v, v')\ \textit{end}$

A composed event $"e_1; e_2"$ *is an event with the same guard as* e_1 *and the after state of* e_2 *when executed after executing* e_1*:*

$"e_1; e_2" = \textit{any}\ p\ \textit{where}\ G(p, v)\ \textit{then}\ S(p, v, v'); (\exists q \cdot H(q, v) \land R(q, v, v'))\ \textit{end}$

Here we introduce operator ; *for the sequential composition of event actions*[2]. *It can be reduced to a simple action using the following definition:*

$S_0(p, v, v'); S_1(p, v, v') \mathrel{\widehat{=}} \exists\ v_1\ \cdot\ S_0(p, v, v_1)\ \land\ S_1(p, v_1, v')$

Now we are ready to define the meaning of $\fatsemi_m$ *when, as a special case, it is applied to a pair of events:* $e_1 \hat{\fatsemi}_m e_2 = \mathsf{FIS}("e_1; e_2")$, *where* $\mathsf{FIS}(e)$ *is an Event-B event feasibility condition (see Section* **??** *and also [3]).*

The next step is to reduce the general form of $\fatsemi$ *to the simple case above. For this we consider all the pairs from a Cartesian product of s and t while also reducing the multiple exit and entry points introduced by the parallel composition construct to a single event:* $s \fatsemi_m t = \forall d, f \cdot (d, f) \in s \times t \Rightarrow mergeall(d) \hat{\fatsemi}_m mergeall(f)$, *where* $mergeall(x)$ *is a following generalisation of merge:*

$$mergeall(x) = \begin{cases} e & x = \{e\} \\ merge(hd, mergeall(tl)) & x = \{hd\} \cup tl \land tl = x \setminus hd \end{cases}$$

Finally, we are ready to approach the problem of checking flow/machine consistency. Using the $\fatsemi$ operator, the problem is reduced to a number of conditions on Event-B machine events. Importantly, they all are expressed in first-order logic as they are essentially various instance of the Event-B feasibility proof obligation. The last remaining step is to lift $\fatsemi$ to the level of a model composed of a machine and flow.

Definition 6. Predicate *cons. This predicate defines the consistency conditions for a combination of a flow and machine. Its type is* $cons : F \times M \to BOOL$ *and the definition is as follows:*

[2] Classical B defines a similar operator to compose actions[4].

$$\begin{array}{ll}
cons(ev, m) & = true \\
cons(p; q, m) & = cons(p, m) \wedge cons(q, m) \wedge (EX(p) \mathbin{;}_m EN(q)) \\
cons(p|q, m) & = cons(p, m) \wedge cons(q, m) \\
cons(p\|q, m) & = cons(p, m) \wedge cons(q, m) \\
cons(*(p), m) & = cons(p, m)
\end{array}$$

where ev is either a machine event one of the predefined events ('skip, 'start or 'stop).

Now we are able to state the flow consistency as a condition on machine elements.

Theorem 1. *A flow f is consistent with a machine m provided $cons(f, m)$ holds.*

Proof. Firstly, either a flow or machine may diverge at different points without giving an option to continue with a non-divergent trace. For a flow this could only happen when there is a transition into *'stop* event (flow loops always agree with machine event loops on divergences since a flow loop covers both terminating and non-terminating machine loops). In other words, there is an instance of sequential composition $p; q$ such that $\{'stop\} \in EN(q)$. For a machine, a divergence on traces happens when an event infinitely enables itself while keeping all other events disabled. The conditions introduced by *cons* guarantee that any sequential composition is consistent and thus a divergent event may not be found in the entry points of the right-hand side of a sequential composition. Then, assuming that flow and machine traces agree on deadlocks, such an event may only be *'stop*. Hence, the satisfaction of $cons(f, m)$ establishes the fact that traces of f and traces of m agree on divergences.

Secondly, there is a possibility that a combination of a flow and machine reveals deadlocks that were not present in either flow or machine alone. The only source of such deadlocks is a sequential composition that is not well-formed. However, $cons(f, m)$ states that this may not be the case.

One interpretation of an Event-B machine is that of a loop made of machine events and preceded by the initialisation event. In the flow language this is expressed as $'start; *(e_1| \ldots |e_k)$. This expression gives rise to a consistency condition requiring that there is an enabled event after the initialisation event. It is straightforward to see that machines shown to be deadlock free or refining a deadlock free abstract machine are always consistent with this flow.

Theorem 2. *A consistent flow f, containing 'start in its traces, is concrete with the respect to machine m if for every instance of the sequential composition $p; q$ the following condition holds: $\forall s, t \cdot \{s, t\} \in EN(q) \wedge s \neq t \implies \neg(EX(p) \mathbin{;}_m s \wedge EX(p) \mathbin{;}_m t)$*

Proof. Let us consider two traces of f: d and g, $d \in traces(f), g \in traces(f)$ such that they are prefixes of some machine traces: $\exists md, mg \cdot d \leq md \wedge g \leq mg$. d and g are necessarily prefixes since *'start* is included in the flow expression f. Should it not be possible to find two machine traces then the theorem condition is trivially satisfied. Let us assume that d and g are not interleave equivalent: $\neg(d\ Re^*\ g)$. Then it is possible to find two distinct, non-independent events a and b, $a \neq b, \neg ind(a,b)$ where $\exists hd \cdot hd^\frown \langle a \rangle \leq d \wedge hd^\frown \langle b \rangle \leq g \wedge \#hd > 0$ and $\#x$ denotes the length of trace x. The two traces record the same event occurrences until a point when a is recorded in one and b is recorded in another. Since the theorem condition requires that f uses $'start$ it is known that $\langle 'start \rangle \leq hd$ and thus hd is not empty. Prefix hd corresponds to some flow expression fp such that $traces(fp) = hd$ (it is not, however, necessarily a part of f as it might be just one possible trace of a parallel composition in f). The fact that d and g disagree on events a and b necessarily requires that pf is followed by a choice construct that among its entry points has a and b. Thus, machine definition would have to satisfy the following condition: $EX(fp) \mathbin{⨾}_m \{a\} \wedge EX(fp) \mathbin{⨾}_m \{b\}$. Let us consider the theorem condition where let $p = fp$ and $\{a,b\} \in EX(q)$. Then $\neg(EX(fp) \mathbin{⨾}_m \{a\} \wedge EX(fp) \mathbin{⨾}_m \{b\})$. The contradiction proves the theorem.

These two theorems show how to reason about flow and machine consistency in terms of conditions o machine elements. Next we show how derive conditions that could be used as proof obligations in the automated reasoning framework of Rodin Platform[14].

3.3 Proof Obligations

For a combination of a flow and machine we would like to be able to demonstrate that the flow is consistent or concrete (the latter requires the former). The general strategy is split an overall proof into a collection of simpler conditions.

For flow consistency, a suitable way to do this is to analyse each instance of sequential composition individually as suggested by the condition of Theorem 1 (see Definition 6 for operator $cons$). For an instance of a sequential composition, from Definition 5 we have the following feasibility condition for a composed event.

$$I(v) \wedge G(p,v) \vdash$$
$$\exists v' \cdot (S(p,v,v'); (\exists q \cdot H(q,v) \wedge R(q,v,v'))) \vdash$$
$$\exists\ v_1\ \cdot\ (S(p,v,v_1)\ \wedge\ \exists q \cdot H(q,v_1) \wedge R(q,v_1,v')))$$

The condition is far too complex in the current form. A more compact one could be found. Let us first assume that the composed events are feasible on their own. This gives the following two axioms.

$\text{axm1} : I(v) \wedge G(p,v) \vdash \exists v' \cdot S(p,v,v')$
$\text{axm2} : I(v) \wedge H(q,v) \vdash \exists v' \cdot R(q,v,v')$

Applying axiom axm1, the feasibility condition for a composed event is simplified to the following:

$$I(v) \wedge G(p,v) \wedge S(p,v,v_1) \vdash \exists\ q\ \cdot\ H(q,v_1) \wedge R(q,v_1,v')$$

With the help of the second axiom we are able to remove $R(q,v_1,v')$ clause from the goal:

$$I(v) \wedge G(p,v) \wedge S(p,v,v_1) \vdash \exists\ q \cdot\ H(q,v_1)$$

Finally, extending the above with the consideration of model constants and sets, the following proof obligation is formulated.

$$P(c,s) \wedge I(c,s,v) \wedge G(c,s,p_e,v) \wedge S(c,s,p_e,v,v') \vdash H(c,s,q,v)$$

Here G and S are the guard and before-after predicate (actions) of what is possibly a result of merging several model events. The proof obligation demonstrates that an event characterised by G and S is able to pass control to another (possibly merged) event with guard H for any possible state permitted by G.

The axioms we have rely upon are sound since they are a part of model consistency proof obligations that are to be discharge for every Event-B model[3].

With a similar procedure we are able to find a practical form of a proof obligation for demonstrating that a flow is concrete. The following proof obligation requires that for a given instance $p; q$ of a sequential composition the choice branches in q, if there any, are mutually exclusive.

$$P(c,s) \wedge I(c,s,v) \wedge G(c,s,p,v) \wedge S(c,s,p,v,v') \vdash$$
$$\bigwedge_{\{s,t\} \in EN(q) \wedge s \neq t} \neg(H_s(c,s,q_s,v') \wedge H_t(c,s,q_t,v'))$$

Here H_s and H_t are the guards of possibly merged events. The goal in this proof obligation may become lengthy in some extreme case when there is a choice on a large number of events. However, since the goal is in conjunctive form is relatively straightforward for a prover to apply case analysis.

3.4 Example

In this section we consider a combination of a simple Event-B model and flow expression. An emphasis is made on using sequential event composition as it is the construct requiring the consistency proof obligations.

The example is a sluice with two doors connecting areas with dramatically different pressures. The pressure difference makes it unsafe to open a door unless the pressure is levelled between the areas connected by the door. The purpose of the system is to adjust the pressure in the sluice area and control the door locks to allow a user to get safely through the sluice.

The model has three variables: $d1 \in DR$ and $d2 \in DR$ are the door states; $pr \in PR$ is the current pressure in the sluice area. A door is either closed or open: $DR = \{OP, CL\}$ and pressure is low or high: $PR = \{HIGH, LOW\}$. Initially, the doors are shut and the pressure is set to low.

A model has a number of invariants expressing the safety properties of the system: a door may be opened only if the pressures in the locations it connects is equalised; at most one door is open at any moment; the pressure can only be switched on when the doors are closed. Model events control the doors and a device regulating the sluice pressure:

$open1$ = when $d1 = CL \wedge pr = LOW$ then $d1 := OP$ end
$close1$ = when $d1 = OP$ then $d1 := CL$ end
$open2$ = when $d2 = CL \wedge pr = HIGH$ then $d2 := OP$ end
$close2$ = when $d2 = OP$ then $d2 := CL$ end
pr_low = when $d1 = CL \wedge d2 = CL \wedge pr = HIGH$ then $pr := LOW$ end
pr_high = when $d1 = CL \wedge d2 = CL \wedge pr = LOW$ then $pr := HIGH$ end

Finally, the following flow expression is used. It describes a sequence of steps needed to let a user through the sluice starting from an area adjoining door 1 ($d1$): $pr\_low; open1; close1; pr\_high; open2; close2$

Let us see how we can check that this specification is consistent with the flow expression. For each instance of sequential composition ($pr\_low; open1$, *open*1; *close*1 and so on) it is needed to show that condition (**??**) holds. For example, for *pr_low*; *open*1 it is:

$$\begin{cases} d1 = CL \wedge d2 = CL \\ pr' = LOW \wedge d1' = d1 \wedge d2' = d2 \end{cases} \vdash d1' = CL \wedge pr' = LOW$$

The condition is trivially true. Another proof obligation, generated from *open*1; *close*1, also trivially holds: $d1 = CL \wedge pr = LOW \wedge pr' = pr \wedge d1' = OP \wedge d2' = d2 \vdash d1' = OP$

The next case presents some difficulties. When trying to demonstrate that event *close*1 always enables *pr_high* we find that there is not enough information to discharge the proof obligation:

$$\begin{cases} d1 = OP \wedge pr' = pr \\ d1' = CL \wedge d2' = d2 \end{cases} \vdash d1' = CL \wedge d2' = CL \wedge pr' = LOW$$

The problem here is that the guard of event *close*1, although strong enough to satisfy safety properties, is too weak for the flow. By strengthening the guard with the additional clauses $d2 = CL \wedge pr = LOW$ we are able to discharge the proof obligation.

3.5 Collecting Additional Hypothesis

There is a way to discharge proof obligations like in the example above without strengthening event guards. Indeed, by looking at the flow expression one should notice that *close*1 is always preceded by *pr_low* and thus may only be enabled when $pr = LOW$. Likewise, since *close*1 always follows *open*1 and the second door is always closed in the after-states of *open*1 (due to the safety invariant of the model requiring that at most one door is open a time) it is known that the condition $d2 = CL$ is always true for states when *close*1 is enabled. Hence all the information that was introduced into proofs by strengthening event guards is already present in a model. To benefit from this information it must be collected and added in the form of hypothesis to flow proof obligations.

Let v_{i-1} be a model state preceding state v_i and state v_n be the most recent previous state preceding the current state v. Also, let $H_i(v_1, \dots, v_n, v)$ be the current collection of hypothesis for some event a. Then for an instance of sequential composition $a; b$ the collection of hypothesis available in the after-state of b is computed as

$$H_{i+1} = H_i(v_1, \dots, v_n, v_{n+1}) \wedge G(v_{n+1}) \wedge S(v_{n+1}, v)$$

where G and S are the guard and actions of b. It is straightforward to generalise this basic procedure to the complete flow language. However, there an issue of filtering out irrelevant hypothesis as a large number of hypothesis slows down some provers.

3.6 Flow Refinement

We use the traces refinement notion [10] to define the refinement relation for flow expressions. To keep flow events in agreement with machine events, some renaming is applied before comparing flow traces:

$$f_a \sqsubseteq f_c \Leftrightarrow traces(R^*(f_c \setminus E_n)) \subseteq traces(f_a)$$

property	*definition*	*description*
eventually	$a\ F^*\ b$	after a eventually b
reachable	$'start\ F^*\ b$	b is reachable
always reachable	$\forall e \cdot' start\ F^*\ e \Rightarrow e\ F^*\ b$	b is always reachable
liveness	$\forall e \cdot' start\ F^*\ e \Rightarrow \exists n \cdot \{b\} = F^n(e)$	b keeps happening

Fig. 1. Flow properties

where $x \setminus S$ removes all occurrences of events from S in traces of x; E_n is a set of new events introduced in machine refinement (these events refine an implicit *skip* event of an abstract machine); R^* is a function mapping concrete event labels into the labels of abstract events. Note that since a flow selects one of the possible traces of a machine, the combination of a consistent flow and a machine exhibits the failure-divergence refinement in respect to the pair of abstract flow and a machine.This is due to the fact that Event-B refinement is a case of the failure-divergence refinement [12].

3.7 Reasoning about Flows

A flow expression may be seen as a directed graph. Its vertices are model events and edges are the transitions connecting events in a flow expression. Computing the transitive closure of such graph, one is able to check statements like "*after event a eventually event b*" or "*event x is reachable*". Let F be a graph constructed from a flow expression: $F : Event \leftrightarrow Event$. Then "*after event a eventually event b*" is understood as $a\ F^*\ b$ and "*event x is reachable*" becomes $'start\ F^*\ x$. One is also able to check that event x is always reachable by stating that it can be reached from any event that in its turn is reachable from the initialisation event: $\forall e \cdot e \in F^*('start) \Rightarrow e\ F^*\ x$. With a similar technique it is possible to express liveness properties to check that something good keeps happening throughout a system lifetime (Figure 1).

Since flow properties are checked at the level of a flow and a flow may have more traces than a machine, not all flow properties automatically hold for a combination of a flow and machine. It this light, formulating flow properties may seem a vain exercise. However, flows give a considerable advantage in model checking by reducing a model state space. Since validating flow properties is computationally cheap and the user gets an instant feedback, it is more effecient to first constrain a flow expression and then apply model checking on combination of a flow and Event-B machine.

4 Conclusions

In our view, the ability to reason about event ordering is a useful addition to the Event-B method. It helps to construct models with rich control flow properties and it also makes such models more readable. Unlike the existing work in this area, it relies solely on theorem proving. It uses practical and scalable proof obligations that are handled well by automated theorem provers. The approach benefits from the existing tool support with a proof-of-the-concept tool implemented for the RODIN platform [14].

We attempted to solve the problem of unmanageable proofs resulting from a sequential composition of actions. For instance, in Classical B, actions within operations and events may be composed using operator ;, e.g., $a := a + 1; b := a + 1$. This is interpreted as applying the second action in the context of the first one. Unfortunately, the verification of sequential action composition is not compositional and all the composed actions must be analysed as a single logical statement. With flows, we make use of event guards to do localised reasoning where possible. In fact, in all the case studies attempted so far, it was possible to show flow consistency by strengthening event guards and adding new invariants with most of the proof obligations discharged automatically. This is despite the fact that in some example there were rather long chains of sequentially composed events (14 for the final refinement of the sluice control). The role of guards in analysing flow consistency is similar to the use of assertions in VDM [16] and refinement calculus [17]. Yet in our case, guards retain their primary role in the analysis of event feasibility, invariant preservation and refinement.

We have presented a three-step verification approach where one first establishes independently the well-formedness of a flow and consistency (and possibly refinement) of a machine and then checks the consistency of a machine and flow combination. In addition to the consistency condition, there is a possibility to generate proof obligations that would ensure that a flow is suitable for deriving an executable program. We are investigating some additional proof obligations.

The introduction of a flow is a step towards constructing runnable sequential code from Event-B models. The addition of a flow to a machine converts an event-triggered, data-driven Event-B model into a a sequence of assignments and control structures, such *if* and *while*. It is possible that flows could play the role of B0 intermediate language [4] of Classical B for the Event-B method.

The proposed mechanism has been implemented as an extension of RODIN platform [14]. The platform is an Eclipse-based integrated environment for constructing Event-B developments. It provides means for model manipulation (editing, pretty-printing, exporting, etc.) and verification. The platform is responsible for generating proof obligations demonstrating model consistency and also the refinement obligations if a model happens to be a refinement of another model. Proof obligations are handed over to a collection of theorem provers. Any unproved obligations has to be analysed in an integrated interactive prover. We considered it essential to make the flow extension a natural part of an Event-B development method. The flow editing is done within the Platform's machine and thus appears a natural part of a model. Flow proof obligations are automatically generated from a flow expression attached to a machine. Syntactic checks and flow refinement checks are also done automatically in a background while a user works with a model. A number of case studies carried with the tool demonstrated that, on average, flows account for 10 % to 25 % of interactive proof obligations.

There is a substantial amount of work based on the Morgan's [12] failure-divergence semantics for event-based systems discussing the integration of state-based and process-based formalisms [19–21, 8, 22]. Their main difference from our approach is that consistency analysis is carried out with a help of process algebraic reasoning.

Our flow language lacks many constructs found in notations like CSP and CCS. In particular there are no communication primitives. It would be hard to justify a message passing mechanism for a single machine but it becomes an interesting possibility should a flow be able to relate several machines. The combination of CSP and Classical B has been investigated in [20] while the CSP style message passing was used to compose Event-B machines[13].

References

1. J. R. Abrial and L. Mussat, "Introducing Dynamic Constraints in B," in *Second International B Conference.* LNCS 1393, Springer-Verlag, April 1998, pp. 83–128.
2. J.-R. Abrial, "Event Driven Sequential Program Construction," 2000, available at http://www.matisse.qinetiq.com.
3. C. Metayer, J. Abrial, and L. Voisin, Eds., *Rodin Deliverable D7: Event B language.* Project IST-511599, School of Computing Science, Newcastle University, 2005.
4. J. R. Abrial, *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 2005.
5. H.Treharne and S.Schneider, "How to Drive a B Machine," 2000, pp. 188–208.
6. M.Butler and M.Leuschel, "Combining CSP and B for Specification and Property Verification," 2005, pp. 221–236.
7. C. Fischer and H. Wehrheim, "Model-Checking CSP-OZ Specifications with FDR," in *IFM '99: Proceedings of the 1st International Conference on Integrated Formal Methods*, K. Araki, A. Galloway, and K. Taguchi, Eds. London, UK: Springer-Verlag, 1999, pp. 315–334.
8. J. Woodcock and A. Cavalcanti, "The Semantics of Circus," in *ZB '02: Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B.* London, UK: Springer-Verlag, 2002, pp. 184–203.
9. R.-J. Back and K. Sere, "Stepwise Refinement of Action Systems," in *Proceedings of the International Conference on Mathematics of Program Construction, 375th Anniversary of the Groningen University*, J. L. A. van de Snepscheut, Ed. London, UK: Springer-Verlag, 1989, pp. 115–138.
10. C. A. R. Hoare, "Communicating Sequential Processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.
11. M. Butler, "A CSP Approach to Action Systems. phd thesis." 1992.
12. C. Morgan, "Of wp and CSP," pp. 319–326, 1990.
13. M. Butler, "Decomposition Structures for Event-B," in *Integrated Formal Methods iFM2009, Springer, LNCS 5423*, vol. LNCS, no. 5423. Springer, February 2009.
14. "Event-B and RODIN Platform," http://www.event-b.org, 2004.
15. M. Leuschel and M. Butler, "ProB: A model checker for B," in *FME 2003: Formal Methods*, ser. LNCS 2805, K. Araki, S. Gnesi, and D. Mandrioli, Eds. Springer-Verlag, 2003, pp. 855–874.
16. C. B. Jones, *Systematic software development using VDM.* Prentice Hall International (UK) Ltd., 1986.

17. R.-J. J. Back and J. V. Wright, *Refinement Calculus: A Systematic Introduction.* Springer-Verlag New York, Inc., 1998.
18. M. J. Butler, "Event Ordering in Action Systems," in *Proc. Int. Refinement Workshop / Formal Methods Pacific'98, Springer Series in Discrete Mathematics and Theoretical Computer Science*, J. Grundy, M. Schwenke, and T. Vickers, Eds. Springer-Verlag, Berlin, 1998, pp. 61–80.
19. M. Leuschel and M. Butler, "Combining CSP and B for Specification and Property Verification," A. T. John Fitzgerald, Ian Hayes, Ed. Springer-Verlag, LNCS 3582, January 2005, pp. 221–236.
20. M. J. Butler, "An Approach to the Design of Distributed Systems with B AMN," in *Proc. 10th Int. Conf. of Z Users: The Z Formal Specification Notation (ZUM), LNCS 1212*, J. Bowen, M. Hinchey, and D. Till, Eds. Springer-Verlag, Berlin, April 1997, pp. 223–241.
21. S. Schneider, , S. Schneider, and H. Treharne, "Verifying Controlled Components," in *In Proc. IFM.* Springer, 2004, pp. 87–107.
22. C. Fischer, "CSP-OZ: a combination of object-Z and CSP," in *FMOODS '97: Proceedings of the IFIP TC6 WG6.1 international workshop on Formal methods for open object-based distributed systems.* London, UK, UK: Chapman & Hall, Ltd., 1997, pp. 423–438.

Launching Formal Methods into Space

Dubravka Ilić

Space Systems Finland
02200, Espoo, Kappelitie 6, Finland
dubravka.ilic@ssf.fi

Abstract. This paper gives an overview of the experiences and so far known challenges in applying Event-B in the space domain.

1 Rational

Formal methods are traditionally used for reasoning about software correctness. However, they are still not widely accepted in engineering practice. One of the goals of Deploy (Industrial deployment of system engineering methods providing high dependability and productivity) project is deployment of formal engineering methods within the space industries. Development of safety critical software in the space sector aims at producing high quality software that ensures the safety of human lives, achieves mission objectives, and correctly operates valuable instruments.

Space Systems Finland (SSF) is a software engineering company specialized in high reliability embedded software. As the space industrial partner within Deploy, SSF investigates ways of advancing engineering methods for dependable systems by introducing formal modelling framework Event-B into an ongoing space project, while still maintaining compliance with the existing standards and regulations in the space sector.

2 Description of the conducted work

As an effort to promote the reuse of on-board and ground systems European Space Agency (ESA) developed a standard for packet telemetry (TM) and telecommand (TC) - PUS. It defines a set of standard service models with the corresponding structures of the associated telemetry and telecommand packets. Various missions then can choose to implement those standard PUS services that best conform to their specific requirements. The TC/TM handling software has been and will be an inevitable part of many space software projects. This is the reason why it has been given the main focus of Event-B modelling activities based on the ongoing MIXS/SIXS project of the BepiColombo satellite.

The MIXS/SIXS On-Board software (OBSW) consists of five different software components, as shown in Fig. 1: The Core Software (CSW) and four application software components controlling the specific instruments (SIXS and MIXS). CSW is a common interface software for the application SW. It works as a TC/TM interface with the BepiColombo platform. The MIXS/SIXS instrument SW receives telecommands via a dedicated hardware link. Also, similarly as for TC reception, there is a TM delivery service.

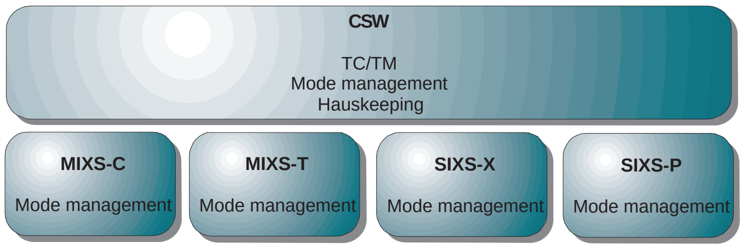


Fig. 1. High-level architecture

Specifying the TC/TM handling. All relevant concepts of TC/TM processing are introduced in a number of refinement steps, where the initial focus is on modelling generated TC/TMs and their associated statuses (with respect to their validation and execution activities) and the ways TC validation and execution are conducted. Naturally, these descriptions are incorporated into the specifications step-by-step, building up the more detailed TC validation/TC execution and TM reporting chain, and leaving some other behavioural aspects underspecified. Upon specifying a very basic system behaviour regarding TC/TM handling, further models introduce different standard TC services and some special services responsible for commanding the instruments. This, in turn, resulted in refining already introduced TC validation/execution while taking into account the purpose and the functionality of a TC under validation/execution.

Described architectural division of the on-board software components becomes visible after introducing management of SW component modes. For every component, mode commands (telecommands) are executed according to specified transition diagrams. The central part in managing operating modes belongs to CSW. Its operating modes and mode transitions triggered by dedicated telecommands are governed by system level mode transition diagram, meaning that all mode transitions made on the level of other components have to be synchronized with this upper-level (system) modes, as shown in Fig. 2. The transition diagrams give an overall picture of system behaviour and are especially suited to be modelled in this framework. In addition, introducing mode management seems as a reasonable step since TC/TM processing is dependant on and, at the same time, affects the modes. In fact, TC execution is limited to certain modes and, moreover, it is TC execution that explicitly induces mode changes. Mode changes can also be induced by FDIR mechanisms, but that is currently not foreseen in the scope of this study.

Introducing mode management on the level of CSW and all the other four subcomponents requires significant attention when it comes to modelling their dependencies. Expressing them in a model and proving them is of vast importance. This way we can guarantee the synchronization between different system level modes and the instrument software modes.

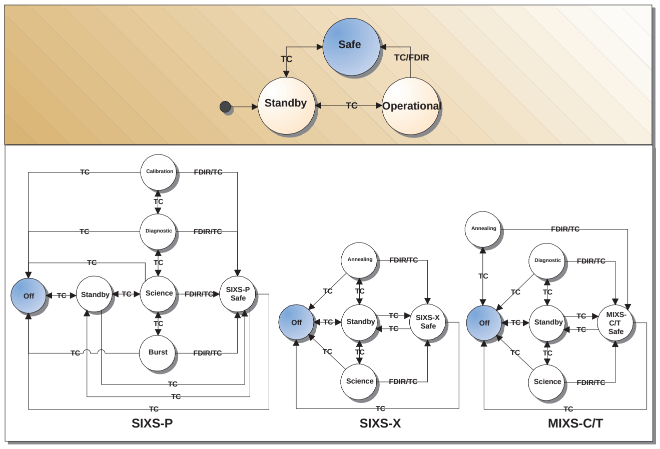
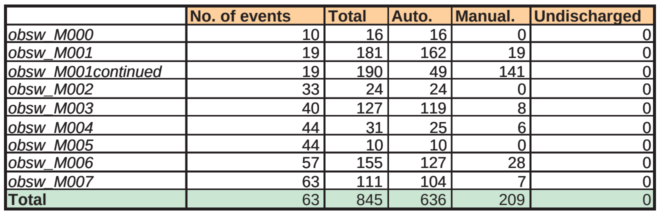


Fig. 2. Mode synchronization of components

3 Achieved results

Currently, the pilot models cover app. 18% of mainly functional requirements. Even though this is only a small percentage of the original requirements, the resulting models are quite complex. Despite this, our experience suggests that the required modelling efforts are not very alarming. The only really time-consuming activity in the pilot development is proofs - a considerable amount of time is needed for producing a proof, even when it is needed only to reuse an existing proof. Hence our interest has moved during the project from quantitative goals such as requirements coverage to qualitative goals such as better understanding of the Event-B proof methodology.

The resulting formal development consists of eight refinement steps, with altogether 1000 generated proof obligations (155 POs for axioms and theorems in the context), which have all been discharged either automatically or manually. The proof statistics is shown in Fig. 3.



	No. of events	Total	Auto.	Manual.	Undischarged
obsw_M000	10	16	16	0	0
obsw_M001	19	181	162	19	0
obsw_M001continued	19	190	49	141	0
obsw_M002	33	24	24	0	0
obsw_M003	40	127	119	8	0
obsw_M004	44	31	25	6	0
obsw_M005	44	10	10	0	0
obsw_M006	57	155	127	28	0
obsw_M007	63	111	104	7	0
Total	63	845	636	209	0

Fig. 3. Proof statistics

Work which has been carried out so far exposed several challenging tool and methodology needs, such as a need to support: team work, modularity - which would allow to easier manage model complexity, proof reusability. In addition, there is an ongoing initiative within the Deploy project to provide a support for code generation, which, if guided according to the space standards, limits the choice of a programming language to C and Ada and imposes more stringent requirements such as: no recursion, no dynamic memory allocation, etc. These are still to be investigated in the frame of the future work.

Reasoned Modelling: Combining Proof and Modelling Patterns to Guide Systems Design

Extended Abstract

Andrew Ireland[1] and Gudmund Grov[2]

[1] School of Mathematical & Computer Sciences, Heriot-Watt University, Edinburgh, EH14 4AS, UK. `A.Ireland@hw.ac.uk`
[2] School of Informatics, University of Edinburgh, Informatics Forum, Edinburgh, EH8 9AB, UK. `ggrov@inf.ed.ac.uk`

1 Introduction

The proliferation of embedded software systems brings significant economic and social benefits. But with these benefits comes risks, e.g. project over-runs and software vulnerabilities. In older engineering disciplines, such civil, mechanical and construction, these risks are mitigated by the use of *scientific methods* – methods that support the rigorous analysis of design decisions before construction gets underway. Within software engineering, formal modelling and reasoning provide a basis for such rigorous analysis.

Tools that support formal modelling typically require expert practitioners that understand the close relationship that exists between proof and modelling. They use proof intuitions to inform their modelling decisions, and their intuitions as modellers to assist with reasoning. This reliance on expert practitioners, in particular the need for expertise in proof, is a major barrier to increasing the accessibility of formalism within design.

We believe this barrier can be significantly reduced by building tools that allow practitioners to focus more on modelling, and less on proof. Our aim is to contribute to the development of such tools. Here we propose *reasoned modelling* - a technique which aims to abstract away from the complexities of low-level proof obligations, and provide high-level modelling guidance to designers. To illustrate, here are a couple of scenarios of what reasoned modelling is targeting:

- When verifying properties of a system model, the low-level analysis of failed-proof attempts may suggest the need for alternative system properties, while knowledge of a designer's intentions may assist in ranking these alternatives – *our aim is to automate such low-level analysis and rank alternative modelling suggestions.*
- During the evolution of a system model, an incorrect refinement will give rise to proof failures. Knowledge of common patterns of refinement combined with an expectation as to how proof should proceed can assist in overcoming such failures – *our aim is to automate such assistance by combining common patterns of refinement with proof-failure analysis.*

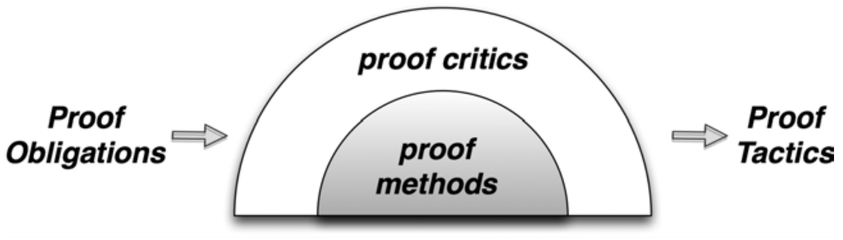


Fig. 1. Basic proof planning architecture

As well as increasing the accessibility of formal modelling, we believe that providing automatic guidance will also lead to productivity gains. Note that our aim is to only provide guidance, leaving the decision making to the designer.

2 Reasoning patterns and proof planning

Our starting point is *proof planning*, an AI technique for automating the search for proofs through the use of high-level proof outlines, known as *proof plans* [3]. A proof plan is defined in terms of proof *methods*, where a method specifies the applicability of a general purpose proof *tactic* [5]. Proof planning uses methods to build a customized tactic for a given proof obligation (conjecture).

Central to proof planning is the *proof critics* mechanism. While methods represent common patterns of reasoning, critics define patchable exceptions [6]. Both methods and critics are heuristic in nature – where the heuristics are represented within a meta-language. Proof critics provide a degree of flexibility in the application of proof plans, and have been used successfully in automating inductive conjecture generalization and lemma discovery [4, 7], as well as automatic loop invariant discovery [10, 8]. Using abductive reasoning, proof critics have also been developed for patching faulty conjectures [11]. The basic architecture of a proof planning system is presented in Fig 1.

3 From proof planning to reasoned modelling

Our proposal builds directly upon the proof planning ideas. Specifically, we are extending the proof critics mechanism with a modelling dimension to give *reasoned modelling critics*. These new style critics will have access to models as well as proof obligations. Analogous to the meta-language used by methods and critics, we are developing a complementary meta-language for modelling. This will enable designers to annotate models with their “design intentions”, as hinted at in §1. Such “meta-data” is typically not directly supported by formal modelling notations. Currently we are investigating concepts of *priority* and *flexibility* within the context of design choices. In a forthcoming paper [9], we describe in

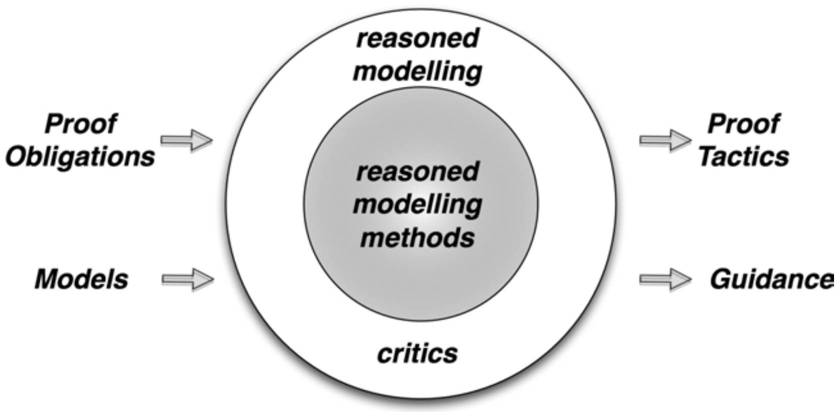


Fig. 2. Basic reasoned modelling architecture

detail how our concept of priority can be used to rank modelling suggestions that arise from the analysis of failed system invariant proofs. Common patterns of refinement are also being explored through what we call *reasoned modelling methods*. In particular, we are investigating guidance based upon the use of common patterns of refinement, combined with proof-failure analysis.

With access to meta-level knowledge of design intentions and proof, our reasoned modelling critics and methods will generate the kind of guidance outlined in §1. Our architecture for reasoned modelling, as given in Fig 2, can be seen as a natural evolution of the basic proof planning architecture.

4 Future work and conclusions

While the proposal outlined above is generic, we are currently developing our ideas using Event-B, a formal framework for modelling discrete complex systems [1]. Event-B promotes an incremental style of formal modelling, where each step of a development is underpinned by formal reasoning. Specifically, we are working with the Rodin tool-set [2] which mechanizes Event-B. Our aim is to develop a reasoned modelling plug-in for Rodin.

Acknowledgements This research is supported by EPSRC grants EP/F037058 and EP/E005713. Thanks go to Ewen Maclean, Maria Teresa Llano Rodriguez, Alan Bundy, Michael Butler, Jean-Raymond Abrial and Cliff Jones for their feedback and encouragement with this work.

References

1. J.-R. Abrial. *Modelling in Event-B: System and Software Engineering.* To be published by Cambridge University Press, 2009.
2. J.-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An open extensible tool environment for Event-B. In *International Conference on Formal Engineering Methods (ICFEM)*, Lecture Notes in Computer Science. Springer-Verlag, 2006.
3. A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.
4. A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning.* Cambridge University Press, 2005.
5. M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF - A mechanised logic of computation*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, 1979.
6. A. Ireland. The use of planning critics in mechanizing inductive proofs. In A. Voronkov, editor, *International Conference on Logic Programming and Automated Reasoning (LPAR'92), St. Petersburg*, Lecture Notes in Artificial Intelligence No. 624, pages 178–189. Springer-Verlag, 1992. Also available from Edinburgh as DAI Research Paper 592.
7. A. Ireland and A. Bundy. Automatic verification of functions with accumulating parameters. *Journal of Functional Programming: Special Issue on Theorem Proving & Functional Programming*, 9(2):225–245, March 1999. A longer version is available from Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/97/11.
8. A. Ireland, B. J. Ellis, A. Cook, R. Chapman, and J. Barnes. An integrated approach to high integrity software verification. *Journal of Automated Reasoning: Special Issue on Empirically Successful Automated Reasoning*, 36(4):379–410, 2006.
9. A. Ireland, G. Grov, and M. Butler. Reasoned modelling critics: turning failed proofs into modelling guidance. In *Proceedings of ABZ 2010*, LNCS. Springer, 2010. To appear.
10. A. Ireland and J. Stark. Proof planning for strategy development. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):65–97, February 2001. An earlier version is available as Research Memo RM/00/3, Dept. of Computing and Electrical Engineering, Heriot-Watt University.
11. R. Monroy, A. Bundy, and A. Ireland. Proof Plans for the Correction of False Conjectures. In F. Pfenning, editor, *5th International Conference on Logic Programming and Automated Reasoning, LPAR'94*, Lecture Notes in Artificial Intelligence, v. 822, pages 54–68, Kiev, Ukraine, 1994. Springer-Verlag. Also available from Edinburgh as DAI Research Paper 681.

Refinement, Problems and Structures

Michael Jackson

The Open University
jacksonma@acm.org

Introduction

Refinement is a powerful reasoning tool in software development. success in its use for program construction encourages the hope of equal success in the construction of dependable computer-based systems, especially in safety-critical applications. These are systems in which the computer, with the software it executes, forms only one component of the whole system. The other components are other engineered artifacts and systems of many kinds, parts of the natural world, and human beings who are users or operators of the system or in any other way participate in its behaviour.

The criterion of dependability of such a system is not to be judged in the software alone, but rather in the behaviour of the whole system. This behaviour is constrained in two ways. First, it is constrained by the given properties, characteristics and behaviours of the relevant parts of the world. For example: in a lift control system the physical arrangement of the shaft prevents the lift car from travelling from the second floor to the fourth without passing the third; in the same system the electrical and mechanical equipment ensures that if the hoist motor is switched on, with upwards polarity, the lift car will rise in the shaft. Second, system behaviour is constrained by execution of the software, monitoring and controlling the relevant parts of world both through interfaces to parts directly connected to the computer, and through the interactions of those parts with remoter parts. The lift system is dependable if the whole system dependably satisfies its requirements: in response to users' requests it carries them safely and efficiently from floor to floor.

Developing systems of this kind is different from developing programs. In program development the goal is to achieve a certain input-output behaviour of the computer by constructing a text in a programming language. Effectively, the programming language and the semantics of its execution are formally defined. The input-output behaviour to be achieved—the program specification—is also formal. In refinement-based development the program text is developed from the formal specification, each step justified by the known formal properties of the programming language.

In system development, by contrast, the requirement to be satisfied is a certain behaviour of the whole system: the computer's input-output behaviour is merely an instrumental means to this end. Usually the requirement itself is not formally specified, so the root of the refinement tree is not well-defined. Further, the relevant parts of the system outside the computer—the *problem world domains*—are not themselves formal. Their given behaviours and properties can be formally described, but a formal description can never be more than an approximation: appeal to the formal description is therefore never fully reliable. Further yet, systems are increasingly expected to embody a proliferation of functional features whose mutual interactions add another order of complexity.

Contrivances and Operational Principles

We shall regard systems of the kind we are concerned with as *contrivances*, in the sense explained by the physical chemist and philosopher Michael Polanyi. One broad class of contrivances comprises physical inventions such as clocks, telephones, locomotives and cameras. A contrivance is characterised by its *parts*, interacting according to the *operational principle* of the contrivance to achieve its *purpose*.

Figure 1 depicts the parts and purpose of the pendulum clock invented by Christiaan Huygens in 1656 and constructed by Salomon Coster in 1657. The rectangles represent the physical parts of the clock and the progression of time, the solid lines connecting them representing interactions among the parts. The dashed oval represents the purpose of the clock, which is to ensure that the positions of the hands on the clock face correspond to the passage of time (the arrowhead indicating that the purpose is to constrain the hands to match the time, not vice versa).

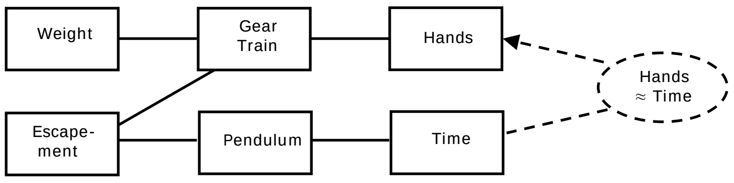


Fig. 1. A pendulum clock

The operational principle of the clock is readily explained. The falling weight drives the gear train, which drives the hands and the escapement. Rotation of the escapement shaft is constrained by the pendulum, each swing releasing one tooth of the escapement and receiving an impulse to maintain the pendulum's momentum. Gear train rotation is therefore proportional to the number of swings. The swing period is roughly constant, so the hand positions correspond to elapsed time.

Polanyi points out that the operational principle is expressed in terms of what he calls the *logic of contrivance*, and that this is distinct from scientific knowledge. It explains "how its characteristic parts—its organs—fulfil their special function in combining to an overall operation which achieves the purpose of the machine. It describes how each organ acts on another organ within this context." Scientific knowledge comes into play only within the given structure of the design of the contrivance and its known operational principle. Science can then explain the success or failure of a particular clock or a particular design, and calculate physical values that will allow the contrivance to fulfil its purpose. What is the swing period of the pendulum? Why do we expect it to be nearly constant? Are the gear ratios between the escapement and the hands correctly calculated? Does the escapement deliver an impulse large enough to counteract the slowing of the pendulum by friction and air resistance? Is the weight heavy enough to overcome the gear train friction? All these are questions of science, but they are posed only in reference to the contrivance and its operational principle. The operational principle tells us how the system is intended to work, and science tells us whether it will actually do so, and how well.

A contrivance is designed to achieve its purpose in a specific context, and is not expected to operate successfully outside that context. The pendulum clock must be stably positioned, in calm air and in a vertical orientation, on the earth's surface; it cannot operate successfully in a moving carriage, or in a ship at sea. The constancy of the pendulum's period depends also on a stable ambient temperature: in summer the clock ran slower as the pendulum period increased as its length expanded in the heat.

Systems As Contrivances and Problems

Systems of the kind we are considering can be similarly regarded as contrivances. Figure 2 shows the configuration of a system whose purpose is to ensure orderly and safe traffic of vehicles and pedestrians at a very complex road crossing.

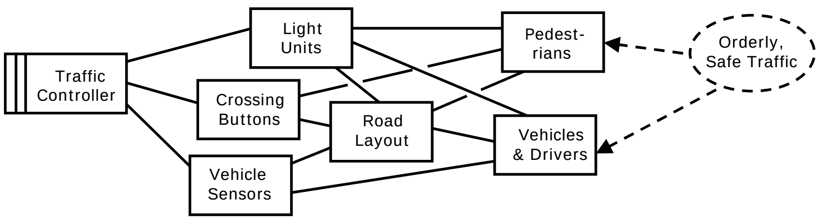


Fig. 2. A traffic control system

Again the rectangles represent parts of the system and the oval represents the system's purpose. The striped rectangle represents the computer and its software, directly connected to the light units, the pedestrian crossing buttons and the vehicle sensors embedded in the road.

Unlike the depiction of the pendulum clock in Figure 1, this representation depicts not only the system and its parts and purpose, but also a development *problem*. The problem is to discover, invent, or specify a Traffic Controller part whose external behaviour—by its interactions with the light units, crossing buttons and sensors—will ensure satisfaction of the system's purpose. To solve this problem the developer must investigate and understand—and then respect and exploit—the given properties of the other parts of the system. How fast do the pedestrians make their way across the various crossings? How fast do the cars go? How do the various streams of traffic intersect? What is the protocol for operating the light units? How reliably do drivers stop at red lights? These given properties, like the properties of the clock parts, depend on the specific context assumed for the system. If the junction is near an old-age home some pedestrians may walk very slowly; if one of the feeder roads is a motorway many vehicles can be expected to be driving above the legal speed limit; if an industrial plant is nearby there will be an unusually high number of very large vehicles; and so on.

Simplistically, we can imagine that the Traffic Controller specification and code can be developed by a refinement progression that moves across the diagram from right to left, from the requirement to the external behaviour of the machine. First the required (orderly and safe) behaviour of the pedestrians and the vehicles and drivers is refined. Relying on the given properties of pedestrians and vehicles, this behaviour is then refined into a required behaviour of the light units, crossing buttons and sensors. Relying on the properties of these devices, a final refinement produces a required external behaviour of the Traffic Controller machine. At each refinement step the problem is *progressed* [2,3] towards a software specification.

Problem Characteristics for Simplicity

For any particular system, feasibility of the simplistic view of a refinement process mentioned in the preceding section depends on several factors. The most important factor is the simplicity of the system, evidenced in a simple operational principle: the system must exhibit certain *unities* of purpose and structure. Some of these unities, with illustrative counterexamples of complexity, are:

- Unity of purpose: fails for an air traffic control system in which a certain horizontal and vertical separation is to be maintained, but if that proves impossible some other rule is to be applied.
- Unity of problem domain role: fails for a customer support system in which staff perform operational tasks and are also assigned as ‘personal assistants’ to changing groups of customers.
- Unity of problem domain properties: fails for a lift-control system in which the lift service function depends on faultless equipment behaviour and the safety function depends on diagnosis of equipment faults.
- Unity of system context: fails for a railway operations system in which train scheduling must assume fixed track configuration and availability and the track maintenance function must manage changes to track configuration.

A system exhibiting these and other unities has a simple operational principle. The intended working of the system can be explained in a simple traversal of the system configuration, saying at each step of the traversal how one part behaves and how it interacts with its neighbours. Scientific—or mathematical—knowledge and reasoning are invoked within this structure to validate the explanation in detail, both locally and end-to-end, and to calculate the behaviour that the machine part of the system must have if it is to ensure satisfaction of the requirement.

Decomposition and Recombination

We address complexity by decomposing a complex development problem into *subproblems*. A subproblem has the form of a problem, with its own machine, problem world, purpose and operational principle; being a problem in this sense, it can also be viewed as a system. A successful decomposition produces simple subproblems exhibiting the unities of the kind mentioned in the preceding section.

Subproblem decomposition produces *projections* of the problem. Projections may be chosen in many different dimensions. A projection in space may separate consideration of distinct problem domains: in the traffic control system, determining the locations of vehicles from the sensors does not involve the pedestrian crossing buttons. A projection in time may separate distinct phases and modes of system operation: in an avionics system, control of an aircraft while it is taxiing is separated from control of take-off. A projection in context may separate control of the lift equipment into assumed-healthy and potentially-faulty.

In any decomposition the complexity of the resulting parts has two sources: the inherent complexity of the part itself, taken in isolation; and the complexity due to its interactions with other parts. Traditional approaches to decomposition conflate these two sources, often frustrating the goal of simplicity. For example, program decomposition into a procedure call hierarchy is *embedded decomposition*, in which the parts are embedded in the whole by precisely matched call interfaces. The structure of a relational database schema results from *jigsaw decomposition*, in which the whole consists only of its parts, which must fit together by matching key values.

For systems of the kind we are considering, a *loose decomposition* is preferable, in which the task of identifying the parts is clearly separated from the task of recombining them into a whole. This separation permits a productive *oversimplification* of subproblems, in which each can be analysed in a restricted context in which the unities are preserved. Only in the later stage of recombination are the subproblem interactions addressed, and solutions devised for any difficulties they may pose. For example, in the railway operations system the train scheduling and track maintenance subproblems are first considered separately, oversimplified by their respective contexts of constant track configuration and complete absence of trains. Only when these subproblems are well understood is their recombination addressed. In designing the recombination it may, of course, be necessary to modify either or both of the oversimplified subproblems; but the need for this modification and its design can be more reliably and confidently carried out at this later stage.

The development process can be seen to have a top-down decomposition facet, and a bottom-up recombination facet. Locally, decomposition must logically precede recombination, but globally the two facets may be interleaved. In this process, structure, and human comprehension of operational principles, provides the essential framework. Formal techniques can be effectively deployed within this framework, in analysing and designing individual subproblem structures and their recombination.

References

[1] Michael Polanyi; Personal Knowledge: Towards a Post-Critical Philosophy; Routledge and Kegan Paul, 1958 and U Chicago Press, 1974.

[2] Robert Seater, Daniel Jackson and Rohit Gheyi; Requirement Progression in Problem Frames: deriving specifications from requirements; Requirements Engineering 12, 2 pages 77-102, April 2007.

[3] Zhi Li, Jon G Hall and Lucia Rapanotti; From requirements to specifications: a formal approach; Proceedings of the 2006 International Workshop on Advances and Applications of Problem Frames, pages 65-70, Shanghai 2006.

Abstraction is all we've got: auxiliary variables considered harmful *(extended abstract)*

Cliff B. Jones

School of Computing Science, University of Newcastle
cliff.jones@ncl.ac.uk

1 Introduction

In the best cases, formal proofs of programs are just writing down a designer's intuition in a notation that can be manipulated by a fixed set of rules. Recording the steps of development in a layered series of design decisions can provide real insight (to reader and writer) into why a program works and what can safely be changed in the future.

Concurrent programs present special challenges for either informal or formal justification/understanding. One approach that has had some success is to use rely/guarantee conditions. These have the property that they are "compositional" in the sense that one can reason about one step of development at a time and not risk having to redo the whole development later.

So called "auxiliary variables" (i) are often used in reasoning about concurrent programs; (ii) can be useful; but (iii) can also be undesirable in that they undermine the hard won property of compositionality. This short paper explores the issue of auxiliary variables and tries to set concerns about overuse in a wider context; it concludes with an attempt to recommend constraints on their use.

2 Abstractions

The point of departure is that writing (concurrent) programs is hard. This is the sense of the title "abstraction is all we've got". Layers of abstraction can convey intuition; and subsequently help maintenance. The role of "formality" is to help check each step. In particular a "posit and prove" approach can significantly reduce the "scrap and rework" which robs program development of productivity.

This section reviews (mostly well known) abstraction ideas.

2.1 Procedural abstraction

Approaches like VDM [Jon90], B [Abr96] or Event-B [Abr10] use pre and post conditions to postpone giving details of "how" to compute some particular result. Some form of "operation decomposition rule" can then be used to introduce design decisions in a staged way. Such rules follow Hoare's axiomatic approach but it is worth noting that there are nuances especially concerning termination and relational post conditions. For what follows, it is also pertinent to observe that some sets of development rules deliberately forgo expressiveness to inculcate a style of writing specifications.

2.2 Recording interference

The idea of procedural abstraction can be extended to shared variable concurrent programs in a number of ways. Rely/guarantee conditions [Jon81,Jon96] offer one way of preserving compositionality in the presence of interference. There is a wide range of ways of presenting such rules and I prefer to refer to rely/guarantee "thinking" to embrace different details. What is common is that there is likely to be some expressive weakness in any particular form of rely/guarantee rules and the connection of this with auxiliary variables is returned to in Section 4.

2.3 Data abstraction

The idea of using abstract objects to provide short and perspicuous specifications of non-trivial systems is well known and I am proud of having given it prominence in the first book on program development (as opposed to language description) on VDM [Jon80] and have moved it ahead of procedural abstraction in later VDM books.

The development method is also well known and is referred to as "data reification" in VDM. There are actually two different rules used in [Jon90] but this touches on auxiliary variables and the point is expanded in Section 4 below.

2.4 Link between rely/guarantee thinking and data reification

There is an interesting interplay between the successful use of rely/guarantee thinking and data reification. Oddly, I did not spot this until after having done a number of developments (it is first discussed in [Jon06]). Essentially, it is often possible to use an intermediate abstraction with rely/guarantee conditions but for these to only be realisable in an efficient way by choosing a suitable representation. The cited paper pinpoints how several examples employ guarantee conditions which could be realised by "locking" but how a careful choice of data representation can uncover a lock free algorithm. This point is reinforced in Section 3.

2.5 Fiction of atomicity

Space does not permit a long explanation of this idea here but its connection with two Dagstuhl workshops (in 2004 and 2006 — see [JLRW05,CJ07]) on "Atomicity" justifies a brief mention. The idea of "splitting (software) atoms safely" is discussed further in [Jon07].

3 Simpson's "four slot" ACM implementation

Asynchronous communication mechanisms (ACMs) are non-blocking algorithms to cope with high speed data transfer. Hugo Simpson presented [Sim97] an extremely ingenious "four-slot" implementation. Several authors have attempted proofs of Simpson's algorithm; our aim has been to give a rational reconstruction that provides insight into the design.

The first attempt is published as [JP08]; this includes the essential abstraction layers:

– Σ^a initial abstraction: interfering sub-operations with a data abstraction of unbounded memory
– Σ^i is a data reification to a finite collection of "slots" but which still uses a "fiction of (data) atomicity"
– Σ^r is a representation in terms of which Simpson's code can be understood — here the only atomicity assumption is that single bits can be changed atomically

Unfortunately, doing proofs in more detail showed up two flaws in [JP08]; we have now submitted a revision to a journal (available as [JP09]). Interestingly, I at first thought that I would have to use auxiliary variables to overcome the problems in the earlier development. I didn't — and the "triumph of abstraction" is interesting.

In both the development steps from Σ^a to Σ^i and that from Σ^i to Σ^r, there is a need to discuss all of the values of a shared variable that could occur during execution.[1] In [JP09] a notation for "possible" values has been introduced and this is proving valuable elsewhere.

The other problem concerned mutual exclusion — but of an interesting flavour. Classically, the term "mutual exclusion" refers to making sure that multiple threads can't be in particular regions at the same time. In Simpson's implementation, there is no locking or constraint on progress and one needs to establish that when the two threads are in the same region they are accessing different slots. We call this "mutual data exclusion". My co-author Ken Pierce and I have taken different paths to reason about this: in [Pie09], he uses auxiliary variables; in [JP09], I have shown how the issue of mutual data exclusion can be handled in the Σ^i abstraction and inherited into the final code. Thus the abstraction can be made to clarify what is going on with a key concept. This leads to one of the key observations in Section 4.

4 Auxiliary variables

I'd always prefer a neat abstraction to coding something in auxiliary variables. My aversion to coding probably dates all the way back to the earlier of my two

[1] The flaw in [JP08] was to cover only initial and final values where in fact there could be more than one change.

spells at the IBM Lab in Vienna. Peter Lucas had come up with a "twin machine" proof of the equivalence of two formulations of the block concept of ALGOL-like languages: his idea was all about "ghost variables". My contribution[2] was to observe that a homomorphic "retrieve function" led to clearer arguments providing a suitable abstract specification could be found.[3]

In 1977, I proposed a specific test for "implementation bias" (see [Jon90, §9]). It is also true that there are specifications where one is forced to use what might be considered to be "bias" once some non-determinism has been decided. The important point is that one can nearly always find a better abstraction to avoid bias.

Given the bias in my views against ghost variables, it should be clear why I prefer the abstract argument about mutual data exclusion in [JP09] over the use of an argument with auxiliary variables. There is, however, a deeper point here. John Reynolds observed verbally at MFPS in 2005 that separation logic was for showing the avoidance of races and rely/guarantee conditions were for reasoning about race conditions. This appeared to be a very neat characterisation. In Section 3 it has been shown that rely/guarantee conditions *used on an abstract level* facilitate showing that data races are avoided. Clearly, there is scope for further research here! (In general, a number of researchers are looking at the links between separation logic and rely/guarantee thinking — a useful entry point to this discussion is [Vaf07].)

It is worth reverting to the comment in Section 2 that rely/guarantee conditions are expressively weak: the use of a single relation limits what one can say. For a while, I wondered if this is what was forcing us to consider auxiliary variables. Closer investigation has shown that the problem is already present in the Owicki/Gries approach (see [Jon09]).

My current position on auxiliary variables is that their use is sometimes justified to distinguish *where* in execution an assertion is true (i.e. permits s_l to make assertions specific to phases of s_r) but that they should be used carefully (lest compositionality is compromised).

Acknowledgements

Clearly, some of this work is collaborative with my co-author Ken Pierce. There are many other collaborators mentioned in the two papers [JP09] and [Jon09]. This extended abstract summarises the talk given at the Dagstuhl event on "Refinement Based Methods for the Construction of Dependable Systems". A fuller version of this paper has been submitted as [Jon09]; a technical report version of a related joint paper is already available as [JP09].

My research is supported by the UK EPSRC Platform Grant "TrAmS" and the EU-IP Project "DEPLOY".

[2] To save space, see [Jon01] for the history and citations.

[3] The observation that early (VDL) operational semantic descriptions of languages over used auxiliary data is much more general than this specific example.

References

[Abr96] J.-R. Abrial. *The B-Book: Assigning programs to meanings*. Cambridge University Press, 1996.

[Abr10] J.-R. Abrial. *The Event-B Book*. Cambridge University Press, 2010.

[CJ07] J.W. Coleman and C.B. Jones. Atomicity: A unifying concept in computer science. *Journal of Universal Computer Science*, 13(8):1042–1043, 2007.

[JLRW05] C. B. Jones, D. Lomet, A. Romanovsky, and G. Weikum. The atomic manifesto. *Journal of Universal Computer Science*, 11(5):636–650, 2005.

[Jon80] C. B. Jones. *Software Development: A Rigorous Approach*. Prentice Hall International, 1980.

[Jon81] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Printed as: Programming Research Group, Technical Monograph 25.

[Jon90] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990.

[Jon96] C. B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, March 1996.

[Jon01] C. B. Jones. The transition from VDL to VDM. *Journal of Universal Computer Science*, 7(8):631–640, 2001.

[Jon06] C. B. Jones. An approach to splitting atoms safely. *Electronic Notes in Theoretical Computer Science, MFPS XXI, 21st Annual Conference of Mathematical Foundations of Programming Semantics*, 155:43–60, 2006.

[Jon07] C. B. Jones. Splitting atoms safely. *Theoretical Computer Science*, 357:109–119, 2007.

[Jon09] Cliff B. Jones. The role of auxiliary variables in the formal development of concurrent programs. In Cliff Jones, Bill Roscoe, and Ken Wood, editors, *Reflections on the work of C. A. R. Hoare*, page submitted. Springer, 2009.

[JP08] Cliff B. Jones and Ken G. Pierce. Splitting atoms with rely/guarantee conditions coupled with data reification. In *ABZ2008*, volume LNCS 5238, pages 360–377, 2008.

[JP09] Cliff B. Jones and Ken G. Pierce. Elucidating concurrent algorithms via layers of abstraction and reification. Technical Report CS-TR-1166, School of Computing Science, Newcastle University, 2009.

[Pie09] Ken Pierce. *Enhancing the Useability of Rely-Guaranteee Conditions for Atomicity Refinement*. PhD thesis, University of Newcastle upon Tyne, submitted 2009.

[Sim97] H. R. Simpson. New algorithms for asynchronous communication. *IEE, Proceedings of Computer Digital Technology*, 144(4):227–231, 1997.

[Vaf07] Viktor Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007.

Verifying the Microsoft Hyper-V Hypervisor with VCC

Dirk Leinenbach[1], Thomas Santen[2]

[1] German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany
dirk.leinenbach@dfki.de
[2] European Microsoft Innovation Center, Aachen, Germany
thomas.santen@microsoft.com

Abstract. VCC is an industrial-strength verification suite for the formal verification of concurrent, low-level C code. It is being developed by Microsoft Research, Redmond, and the European Microsoft Innovation Center, Aachen. The development is driven by two applications from the Verisoft XT[1] project: the Microsoft Hyper-V Hypervisor and SYSGO's PikeOS micro kernel [1].
This paper outlines the Hypervisor verification project. It gives a brief overview on the Hypervisor focusing on verification related challenges this kind of low-level software poses. It discusses how the design of VCC addresses these challenges, and highlights some specific issues of the Hypervisor verification and how they can be solved with VCC.

Keywords. C code verification, system verification, virtualization

1 The Microsoft Hypervisor

The Hypervisor verification project aims at developing an industrially viable software verification tool for concurrent, low-level C code – VCC [2] – and using it for the functional verification of Microsoft's Hyper-V Hypervisor. This effort is an ongoing collaborative research project between the European Microsoft Innovation Center (EMIC), the German Research Center for Artificial Intelligence (DFKI), Microsoft Research, and Saarland University in the Verisoft XT project [3].

The Hypervisor is a relatively thin layer of software (100 KLOC of C, 5 KLOC of assembly) that runs directly on x64 hardware. It turns a single real multi-processor x64 machine with virtualization extensions into a number of virtual multi-processor x64 machines without virtualization extensions but with additional machine instructions (hypercalls) to create and manage other virtual machines. The Hypervisor is divided into two strata. The lower stratum forms a small multi-processor operating system, complete with hardware abstraction layer, kernel, memory manager, and scheduler (but no device drivers). The higher stratum runs in each thread an "application" that simulates an x64 machine; the observable effect of a machine instruction executed by a guest operating system on the virtualized machine basically is the same as on the real machine.

For the most part, a virtual machine is simulated by simply running the real hardware. However, the virtual machines run under an additional level of memory address

[1] Work partially funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft XT project under grant 01 IS 07 008.

translation so that each of them can see its own zero-based, contiguous physical memory. This extra level of virtual address translation is accomplished by using *shadow page tables* which combine the two levels of address translation. The shadow page tables, along with the hardware *translation lookaside buffers* (TLBs), implement *virtual* TLBs. TLB simulation is the most important factor in system performance. Thus, the Hypervisor uses a very complex and highly optimized shadow page table algorithm which leverages the degrees of freedom given by the hardware TLB semantics. The correctness of the concurrent shadow page table algorithm is extremely subtle because translations in the TLB are not automatically flushed in response to edits to page tables stored in memory and translations are gathered asynchronously and non-atomically (requiring multiple reads and writes to traverse the page tables), creating races with guest (system) code that operates on the page tables.

The verification of a piece of software like the Hypervisor – which was not written with formal verification in mind – poses a number of challenges to a verification tool:

1. In an industrial process, developers and testers must drive the verification process such that the annotations evolve with the code. Thus, verification should be primarily driven by assertions stated at the level of code itself, rather than by guidance provided to interactive theorem provers. Even if special verification engineers add the annotations initially, they should eventually be maintainable by suitably trained developers.
2. The Hypervisor is written in C, which has only a weak, easily circumvented type system and explicit memory (de)allocation, so memory safety has to be explicitly verified.
3. The Hypervisor is a concurrent piece of software and makes heavy use of lock-free synchronization. Still, support for the verification of lock-free code should not add a burden to the verification of lock-protected or thread-local code.
4. The Hypervisor explicitly manages its own virtual memory. The verification methodology needs a way to argue about the correct setup and maintenance of page tables and TLBs at the lower layers of the Hypervisor whilst abstracting from these properties at the higher layers.
5. A typical way to prove properties of a concurrent data type is to show that it simulates some simpler type. To keep annotations tightly integrated with the code, a way of proving concurrent simulation in the code itself is needed.
6. Part of the Hypervisor is written in assembly code. An integrated verification of C and assembly code is needed, which addresses the subtle interactions between the two and the implications on hardware resources.
7. The code base of the Hypervisor is fixed and cannot be changed just to make verification simple.

2 VCC

VCC – a comprehensive overview can be found in [2] – is geared towards sound, functional verification of concurrent, low-level C code. Specifications and annotations are embedded into the C code itself; so, they can evolve with the code – ideally maintained by developers – and serve as a *formal* documentation of the implementation. Using conditional compilation, the annotations are hidden from standard C compilers.

VCC performs static modular analysis, in which each function is verified in isolation. VCC translates annotated C programs into the Boogie language [4]. Then, the Boogie tool generates verification conditions and passes them to the first order theorem prover Z3 [5]. If Z3 is not able to verify the verification conditions, several diagnostic tools are available for convenient debugging of the program and the annotations [2]. In the last resort it is possible to prove verification conditions interactively in the theorem prover Isabelle / HOL [6].

Although C is not typesafe, most code in a well-written C system adheres to a strict type discipline. Taking advantage of this fact, VCC implements a Spec#-style object and ownership model [7]; in particular, the VCC memory model [8] tracks where the "valid" typed memory objects are. Objects may coincide with (pointers to) C structures, but also with sub-structures and arrays. On each memory reference and pointer dereference, there is an implicit assertion that resulting object is valid. System invariants guarantee that valid objects of the same type with different addresses do not overlap, so they behave like objects in a modern (typesafe) object oriented system. Type definitions can be annotated with invariants, which are one- or two-state predicates on data, describing the properties of "valid" instances of the type, and how they can evolve from one system state to the next. Objects can be *closed* or *open*; initially, new objects are open, and VCC checks the invariant of an object at the transition from open to closed. Thus, it can ensure the system invariant that in each state all closed objects fulfill their invariants.

In addition to function contracts and invariants, VCC allows augmenting the operational code of a program with ghost (specification) code and data. Ghost code is seen only by the static verifier, not the regular compiler; ghost code must not affect the control flow of the real code. One application of ghost code is maintaining abstractions of operational data, e.g., representing a list as a set. If the ghost data is marked volatile this allows for atomic update of the ghost, even if the underlying data structure is not updated atomically. Ghost code also establishes and maintains the closedness and ownership relations of objects and provides existential witnesses to the first order prover.

As in some other concurrency methodologies (e.g., [9]), the ownership model of VCC allows a thread to perform sequential writes only to data that it owns, and sequential reads only to data that it owns or can prove is not changing. VCC accommodates concurrent access to data that is marked as volatile even if the data is not owned by the current thread (using operations guaranteed to be atomic on the given platform), leveraging the observation that a correct concurrent program typically can only race on volatile data. Updates of volatile data are required to preserve invariants but are otherwise unconstrained.

Concurrent programs implicitly deal with chunks of knowledge about the shared state. For example, a program attempting to acquire a spin lock via an access to a volatile state variable must "know" that the spin lock is still allocated and fulfills its invariant (i.e., is closed) [10]. But such knowledge is ephemeral – it could be broken concurrently by other threads – so passing knowledge to a function in the form of a precondition is too weak. Instead, VCC provides a special kind of ghost objects called *claims*. A claim is associated with a number of objects: it guarantees (via references counters) that those objects stay allocated and closed as long as claims to them exist. Claims can guarantee additional properties of the claimed objects, given that these properties are fulfilled at creation time of the claim and are stable with respect to the objects' invariants.

3 Verifying the Hypervisor

Implementation correctness of concurrent systems is usually verified by proving a simulation theorem between the implementation and an abstract model. Often, such theorems are formalized by adding an (external) universally quantified state variable which is updated whenever the implementation state changes. This is adequate when proving simulation theorems about abstract programs, e.g., a transition system. In our case, implementation updates are scattered throughout the code base and we want to keep annotations close to the code. Thus, our abstract model is implemented as C ghost code, keeping code and annotations closely together and allowing us to establish the simulation theorem within VCC without the need of external tools. The coupling relation between the implementation and the abstract model is formalized as a single state invariant between the implementation and the ghost data of the top-level model. Explicit updates of the ghost state provide existential witnesses to the prover where necessary.

For each guest (also called partition), the (volatile) ghost state for the top-level specification contains, among others, general information about the partition (privileges, IPC state), a set of x64 processor states, and the memory content. The formal C specification of the x64 architecture and the top-level model are being developed at Saarland University. In addition to the top-level simulation proof, the x64 model is also used for the low-level correctness proofs of the Hypervisor implementation, in particular for the verification of assembly code [11].

A model of the x64 architecture cannot be specified completely with (deterministic) C functions because components outside the processor core can change their state non-deterministically: the TLB is allowed to cache new translations on its own initiative, the memory might be changed by other processor cores in the same x64 multi-processor machine, or the state of the APIC might change due to interrupt requests from devices or other processor cores. We model such non-deterministic behavior by two-state invariants which restrict the legal transitions of the model. Most of the time, these invariants correspond to the two-state invariants of the x64 model. Additional invariants specify the execution of Hyper-V specific instructions and hypercalls.

Eventually, verifying the coupling invariant between implementation and top-level model in VCC will ensure that the implementation transitions are covered by corresponding *admissible* transitions of the top-level model, i.e., that the Hypervisor implementation correctly simulates the top-level model.

As of July 2009, the Hypervisor verification is still ongoing. Data structure invariants are in place and the larger part of public interfaces is specified. Several hundred functions have been verified with VCC. We are confident that VCC is powerful enough to successfully verify all functions. The specifications provide a detailed documentation that is provably in sync with the code, which is an added value of the exercise in itself. The Hypervisor is part of a released product with very low defect density. Therefore, we did not expect to find many bugs in the code, and indeed less than a handful have been found during the verification process. All of them are very unlikely to let the Hypervisor fail in practical operation. Applying the now available technology early in the process of a product development could help to prevent defects and reduce the effort to meet high quality bars.

Acknowledgment. The results and ideas described in this paper have been developed by a lot of people. Thanks to everyone: Artem Alekhin, Eyad Alkassar, Mike Barnett, Nikolaj Bjørner, Sebastian Bogan, Sascha Böhme, Matko Botinĉan, Vladimir Boyarinov, Ernie Cohen, Markus Dahlweid, Ulan Degenbaev, Lieven Desmet, Sebastian Fillinger, Mark Hillebrand, Tom In der Rieden, Bruno Langenstein, K. Rustan M. Leino, Wolfgang Manousek, Stefan Maus, Michał Moskal, Leonardo de Moura, Andreas Nonnengart, Steven Obua, Wolfgang Paul, Hristo Pentchev, Elena Petrova, Norbert Schirmer, Sabine Schmaltz, Wolfram Schulte, Peter-Michael Seidel, Andrey Shadrin, Stephan Tobies, Alexandra Tsyban, Sergey Tverdyshev, Herman Venter, and Burkhart Wolff.

References

1. Baumann, C., Beckert, B., Blasum, H., Bormer, T.: Better avionics software reliability by code verification – A glance at code verification methodology in the Verisoft XT project. In: Embedded World 2009 Conference, Franzis Verlag (2009)
2. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In Berghofer, S., Nipkow, T., Urban, C., Wenzel, M., eds.: Theorem Proving in Higher Order Logics (TPHOLs 2009). Volume 5674 of LNCS., Munich, Germany, Springer (2009) 23–42 VCC is available at `http://vcc.codeplex.com`.
3. Verisoft XT: The Verisoft XT project. `http://www.verisoftxt.de` (2007)
4. Barnett, M., Chang, B.Y.E., Deline, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P., eds.: FMCO 2005. Volume 4111 of LNCS., Springer (2006) 364–387
5. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In Ramakrishnan, C.R., Rehof, J., eds.: TACAS 2008. Volume 4963 of LNCS., Springer (2008) 337–340
6. Böhme, S., Moskal, M., Schulte, W., Wolff, B.: HOL-Boogie: An interactive prover-backend for the Verifiying C Compiler. Journal of Automated Reasoning (2009)
7. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In Barthe, G., Burdy, L., Huisman, M., Lanet, J.L., Muntean, T., eds.: CASSIS 2004. Volume 3362 of LNCS., Springer (2004) 49–69
8. Cohen, E., Moskal, M., Schulte, W., Tobies, S.: A precise yet efficient memory model for C. In: 4th International Workshop on Systems Software Verification (SSV 2009). Volume 254 of ENTCS., Elsevier Science B.V. (2009) 85–103
9. Jacobs, B., Piessens, F., Leino, K.R.M., Schulte, W.: Safe concurrency for aggregate objects with invariants. In Aichernig, B.K., Beckert, B., eds.: SEFM 2005, IEEE (2005) 137–147
10. Hillebrand, M.A., Leinenbach, D.C.: Formal verification of a reader-writer lock implementation in C. In: 4th International Workshop on Systems Software Verification (SSV 2009). Volume 254 of ENTCS., Elsevier Science B.V. (2009) 123–141 Source code available at `http://www.verisoftxt.de/PublicationPage.html`.
11. Maus, S., Moskal, M., Schulte, W.: Vx86: x86 assembler simulated in C powered by automated theorem proving. In Meseguer, J., Roşu, G., eds.: Algebraic Methodology and Software Technology (AMAST 2008). Volume 5140 of LNCS., Urbana, IL, USA, Springer (2008) 284–298

Refinement-based guidelines for constructing algorithms*

Dominique Méry[1]

Université Henri Poincaré Nancy 1 and LORIA
BP 239
54506 Vandœuvre-lès-Nancy, France
mery@loria.fr

1 Introduction

The *correct-by-construction* approach can be supported by a progressive and incremental process controlled by the refinement of models of programs. We explore the EVENT B modelling language to illustrate the expression of our methodological proposal using proof-based patterns called guidelines. The main objective is to facilitate the correct-by-construction approach for designing classical sequential and distributed algorithms [1–7]. We address the description of guidelines for the design of programs and algorithms and the integration of proof-based aspects using the RODIN platform [8]. More precisely, we introduce several methodological steps identified during the development of case studies, and we propose auxiliary notions, such as refinement diagrams, for guiding users during problem analysis. A general structure characterizes the relationship between the contract, the EVENT B , and the developed algorithm using a specific application of EVENT B models and refinement. We simplify the translation of EVENT B models into algorithmic elements by promoting the use of recursive constructs. The resulting algorithm is proved to be sound with respect to the pre/post specification, namely, the contract. Applications rely on a dynamic programming technique that illustrates the applicability of these patterns based on a call-as-event relationship. Each proof-based development is validated using the RODIN platform. Distributed algorithms are considered with respect to the local computation model [9] based on a relabelling relation over graphs representing distributed systems. The VISIDIA toolbox [10] provides facilities for simulating local computation models which can be easily modelled using EVENT B and the refinement.

2 The call-as-event principle for sequential algorithms

We combine concepts of the EVENT B method and programming paradigms by defining a proof-based pattern based on the call-as-event principle [3]. Proof-

* This work is supported by grant No. ANR-06-SETI-015-03 awarded by the Agence Nationale de la Recherche. The project RIMEL produces the VISIDIA system developed by the team of Mohammed Mosbah, Yves Métivier et Pierre Castéran from the LABRI laboratory.

based development constitutes a very powerful framework for constructing procedures, programs, and systems, but it requires the use of formal techniques and formal languages. Consequently, the introduction of patterns or general structures for helping users to obtain correct systems with minimal effort is a very important direction. We have proposed a pattern as a means to structure refinement-based development. We suggest the definition of the pattern as a structure to fill with models or with actions to perform. We consider that it is like a design calculus [11], being based on validation through logical actions. However, the issue is to provide the basic concepts of pattern design and pattern use.

Our main goal is to facilitate the use of the Event B modelling language by proposing techniques and tools. Proofs were improved, even though RODIN has unexpected features. The general structure states the link between programming objects and modelling objects and provides a way to map an Event B model to a sequential program. Refinement is the step that introduces control points and guarantees the correctness of the resulting algorithm. We have formalized, generalized, and illustrated the technique introduced by Cansell and Méry in [12], and unspecified details in their paper have been made more precise [3]. The technique of development is a top-down approach, which is clearly well known from the earlier works of Dijkstra[13, 14] and uses refinement to control the correctness of the resulting algorithm. It relies on a more fundamental issue related to the notion of the *problem to solve.* A specific diagram is used to organize the refinement, and we call it a refinement diagram [3]. It is very similar to a proof lattice and provides a way of deriving the total correctness of the resulting algorithm. The translation of Event B models into sequential programs has already been proposed by Abrial, but our models correspond to acyclic refinement diagrams, as opposed to those implicitly used by Abrial. We have an automatic transformation of the refinement model into an algorithm. The call-as-event guideline is summarized by the following diagram:

$$\begin{tikzcd}
\mathtt{CALL} \arrow[r, "call-as-event"] \arrow[d, "call"'] & \mathrm{PREPOST} \arrow[r, "\mathrm{SEES}"] \arrow[d, "\mathrm{REFINEMENT}"'] & \mathrm{PB} \arrow[d, "\mathrm{EXTENDS}"'] \\
\mathtt{PROCEDURE} & \mathrm{M} \arrow[l, "mapping"'] \arrow[r, "\mathrm{SEES}"] & \mathrm{CM}
\end{tikzcd}$$

- `CALL` is the call of the `PROCEDURE`
- PREPOST is the machine containing the events stating the pre- and post-conditions of `CALL` and `PROCEDURE`, and M is the refinement machine of PREPOST, with events including control points defined in CM.
- The *call-as-event* transformation produces a model PREPOST and a context PB from `CALL`.
- The *mapping* transformation allows us to derive an algorithmic procedure that can be mechanized.
- `PROCEDURE` is a node corresponding to a procedure derived from the refinement model M. `CALL` is an instantiation of `PROCEDURE` using parameters x and y.

– M is a refinement model of PREPOST, which is transformed into `PROCEDURE` by applying structuring rules. It may contain events corresponding to calls of other procedures.

We have used the technique in the teaching [7] of the design of algorithms. Surprisingly, students discovered invariants using the theorem prover of the RODIN platform. In fact, we think that it is the recursive nature of the development that made the discovery of invariants easier. Proof obligations were not very difficult because we inherit the recursiveness of the structure of the problem. Floyd's algorithm is not a trivial algorithm, but proof obligations were not difficult to prove.

3 The local computation model for deriving correct-by-construction distributed algorithms

ViSiDiA [10] is a tool for implementing, simulating, testing and visualizing distributed algorithms. It is based s on the use of graph relabelling systems to encode distributed algorithms and to prove their correctness. A distributed algorithm in the local computation model [15, 16] is simply given by some (possibly infinite but always recursive) set of rules like for instance, the leader election:

– $\overset{d}{\bullet}\!\!-\!\!\!-\!\!\!-\!\!\!-\!\!\overset{1}{\bullet} \longrightarrow \overset{d-1}{\bullet}\!\!-\!\!\!-\!\!\!-\!\!\!-\!\!\overset{NE}{\bullet}$

– $\overset{0}{\bullet} \longrightarrow \overset{E}{\bullet}$

A run of the algorithm consists in applying the relabelling rules specified by the algorithm until no rule is applicable, which terminates the execution. The relabelling rules are applied asynchronously and nondeterministically, which means that given the initial labelling usually many different runs are possible. The distributed aspect comes from the fact that two consecutive non-overlapping steps may be applied in any order and in particular in parallel. The local computation model is easily expressed in the Event B framework. The methodology leads to fill the following diagram from the problem analysis:

$$\begin{tikzcd}
\textbf{PROBLEM} \arrow[r, "formalization"] \arrow[dd, "derivation"'] & \text{M0} \arrow[r, "\text{SEES}"] \arrow[d, "\text{REFINES}"'] & \text{C} \arrow[r, "\text{LINK}"] & \textbf{COQ-library} \\
& \text{M1} \arrow[d, "\text{REFINES}"'] & & \\
\textbf{V} & \text{VM1} \arrow[l, "mapping"'] & &
\end{tikzcd}$$

– The context C states properties of graphs.

- The machine *M0* expresses the problem to solve by events stating a relation between the initial states and the final states, for instance, the election of a leader.
- The refinement of *M0* into *M1* expresses that the machine *M1* expresses the inductive property allowing to express the computation in the local computation model.
- The next refinement of *M1* is a refinement for producing a set of events corresponding to the set of relabelling rules.
- *V* is derived from *VM1*; *mapping* checks that *VM1* can be translated into VISIDIA [10].

We have obtained an extension of the call-as-event guideline by integrating the local computation model into the EVENT B modelling language.

4 Concluding Remarks

The development of programs is carried out either using bottom-up techniques, or top-down techniques and our main goal is to develop correct programs from specifications using refinement and proofs. More precisely, we explore the use of *proof-based patterns* for aiding and assisting *programmers* ready to use formal techniques in the development of programs. The development is supported by Event B models related by semantical relationship which guarantee correctness properties. Modeling is a very challenging task and its complexity increase when dealing with proof obligations checking development steps or refinement steps. Proof-based patterns intend to make easier the programmer's life. According to Christopher Alexander [17] dealing with architectural problems, *each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.*. Gamma et all [18] introduces design patterns, which systematically name, motivate, and explain a general design that addresses a recurring design problem in object-oriented systems. A design pattern describes the problem, the solution, when to apply the solution, and gives implementation hints and examples.

Considering our general problem of correct-by-construction programs (or systems), we limit our scope to problems solved by sequential and distributed algorithms based on paradigms. Discovering patterns is based on practical case studies However, we are not defining patterns in an object-oriented framework but only reusing the idea of adding values to Event B by proof-based patterns. Finally, our paper intends to list methodological guidelines for developing Event B models and for producing programs. The current paper has summarized two main tasks: the development of sequential algorithm using the call-as-event guideline and the development of distributed algorithm expressed in the VISIDIA environment. Further work will integrate the development of distributed algorithms by listing new patterns for the design of correct-by-construction distributed algorithms, especially by integrating VISIDIA and EVENT B .

References

1. Rehm, J.: Proved development of the real-time properties of the ieee 1394 root contention protocol with the event b method. International Journal on Software Tools for Technology Transfer (STTT) (2009)
2. Benaïssa, N., Méry, D.: Cryptologic protocols analysis using proof-based patterns. In Marchuk, A., ed.: Seventh International Andrei Ershov Memorial Conference PERSPECTIVES OF SYSTEM INFORMATICS, Novosibirsk, Akademgorodok, Russia, A.P. Ershov Institute of Informatics Systems & Novosibirsk State University (15-19 June 2009)
3. Méry, D.: Refinement-based guidelines for algorithmic systems. International Journal of Software and Informatics **3**(2-3) (June/September 2009) 197–239
4. Benaïssa, N., Méry, D.: Cryptologic protocols analysis using Event B. In Marchuk, A., ed.: Seventh International Andrei Ershov Memorial Conference PERSPECTIVES OF SYSTEM INFORMATICS. Volume to appear of Lectures Notes in Computer Science., Springer (15-19 June 2010)
5. Cansell, D., Méry, D.: Designing old and new distributed algorithms by replaying an incremental proof-based development. In Abrial, J.R., Glässer, U., eds.: Rigorous Methods for Software Construction and Analysis - Papers Dedicated to Egon Börger on the Occasion of His 60th Birthday. Number 5115 in Lectures Notes in Computer Science (2010)
6. Cansell, D., Méry, D., Proch, C.: System-on-chip design by proof-based refinement. International Journal on Software Tools for Technology Transfer (STTT) **11** (03 2009) 217–238
7. Méry, D.: A simple refinement-based method for constructing algorithms. SIGCSE Bull. **41**(2) (2009) 51–59
8. RODIN, P.: The rodin project: Rigorous open development environment for complex systems. http://rodin-b-sharp.sourceforge.net/ (2006)
9. Chalopin, J., Godard, E., Métivier, Y.: Local terminations and distributed computability in anonymous networks. In Taubenfeld, G., ed.: DISC. Volume 5218 of Lecture Notes in Computer Science., Springer (2008) 47–62
10. Mosbah, M.: Visidia. http://visidia.labri.fr (2009)
11. Jaehnichen, S., Hussain, F.A., Weber, M.: Program development by transformation and refinement. In: An international workshop on Advanced programming environments
12. Cansell, D., Méry, D.: Proved-patterns-based development for structured programs. In Diekert, V., Volkov, M.V., Voronkov, A., eds.: CSR. Volume 4649 of Lecture Notes in Computer Science., Springer (2007) 104–114
13. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)
14. Morgan, C.: Programming from Specifications. Prentice Hall International Series in Computer Science. Prentice Hall (1990)
15. Chyalopin, J., Métivier, Y.: On the power of synchronisation between adjacent processes. Technical report, LABRI (2009)
16. Chalopin, J., Métivier, Y.: An efficient message passing algorithm based on Mazurkiewicz s algorithm. Fundamenta Informaticae **80**(1) (2007) 221–246
17. Alexander, C., Ishikawa, S., Silverstein, M.: A pattern language: towns, buildings, construction. Oxford University Press (1977)
18. Gamma, E., Helm, R., Johnson, R., Vlissides, R., Gamma, P.: Design Patterns : Elements of Reusable Object-Oriented Software design Patterns. Addison-Wesley Professional Computing (1997)

Formal Methods in the Development of Business Software

Andreas Roth

SAP Research CEC Darmstadt, Germany

Abstract. We discuss the suitability of Formal Methods for the development of business software as well as experiences and challenges in this area. Business software involves very different kinds of software: technical components, business applications, analytical applications. Our focus is on business applications which promise a good probability of a second use of Formal Methods. We investigate service choreography models, business object models, and business process models, as typical instances of models in this domain. Our approach is to take existing diagrammatic models and translate them to a formal language, e.g. Event-B. Though the approach works well, there are a number of challenges. These are especially the provision of good user feedback on prover or model checker results, the need for better automation, and a better usage of refinement.

Business software is any software which helps companies improve their business. In contrast to typical areas of application of Formal Methods, this area has little, e.g. safety critical, aspects which have been significant incentives to make the use of formal methods attractive, as e.g. in the transportation industry. On the other hand business software is often highly mission critical and customers expect high qualitative software which is efficiently developed — goals Formal Methods could be helpful to achieve. Nevertheless Formal Methods are (to our knowledge) not used routinely in the development of business software.

In this short paper we highlight the potential usage of Formal Methods in this area, the special challenges, and our experiences with applying Formal Methods in the context of the Deploy project.

1 Business Software

Business software is a wide field, covering

- technical components (e.g. process integration, master data management, etc.)
- business applications (e.g. Enterprise Resource Management (ERP), Customer Relations Management (CRM), Supply Chain Management (SCM), etc.)
- industry solutions (adaptations of business applications for e.g. Banking, Automotive, Chemicals, Retail, Public Sector, etc.)

– analytical applications (tools for collecting, integrating, analysing, interpreting and presenting business data)

Formal Methods could play a role in all of these areas, but business application development is a—though so far widely ignored—promising domain of Formal Method application. The main reason is that business application development incorporates the routine development of hundreds of structurally similar components. This allows us to focus on domain-specific requirements and helps to come from one application of Formal Methods to the next one because the method can be adapted to these specifics. There is thus a high potential of "second usage" of Formal Methods in that area. Moreover, business applications incorporate interesting "business logic" of processes without dealing (in general) with complex algorithms. The reasoning on a "model-level", without the need to go formally down to source code, is already a welcome support to further increase development efficiency. Applying Formal Methods in this area seems thus to be relevant, sustainable, and feasible.

Finally, a good tradition of (informal) model-based development exists [1] which can be built on. Especially model content which is very expensive to collect is often already present and can be re-used via generating formal models from existing ones. The re-use and integration of such already existing methods seems to us a key factor to achieve deployment of Formal Methods in the presence of successful and established development processes.

2 Applying Formal Methods to Business Applications

We are conducting[1] a number of deployment studies with the help of which we investigate the use of Formal Methods for Business Applications.

2.1 Choreography Models

The allowed ordering of messages exchanged between independent service components is described in (diagrammatic) message choreography models [2]. These models are similar to extended finite state machines and consist of a global model defining the messages exchanged from a global perspective and a local model which describes the send- and receive events of the involved components.

We have formalised these models by providing an automatic translation from them to Event-B [3]. We have implemented the following analyses of the Event-B model:

– Checking the local enforceability by proving a refinement relation between the global and a (general) local model.
– Checking for the absence of inconsumable messages [4], i.e. messages which cannot be received at all times when they are being transmitted.
– Checking the local models against consistency with component models (business object models, see below).

[1] in the context of the EC-funded Deploy project, `www.deploy-project.eu`

– Interactively simulating the choreography model by stepping through the diagram.
– Deriving test cases with a certain model coverage (model-based integration testing) which can then be executed on an implementation of the model [5].

These analyses have been efficiently implemented based on the in-house Eclipse plugin of the choreography model editor and the open Rodin platform [6]. We made use of the Rodin provers and the ProB model checker [7].

2.2 Business Object Models

Business objects (e.g. a purchase order, a sales order, an invoice, a delivery, a project) are at the core of business applications. Typically these objects are structured as a tree (e.g. a purchase order consists of a root node and a number of (ordered) item nodes). Each of the nodes has a lifecycle where user actions (e.g. create, approve, release) lead a path through a set of status (created, approved, released) of the business object nodes.

We are conducting experiments with modelling business objects as Event-B models and started to automatically translate models in a proprietary in-house notation to Event-B. Our goals are to (1) prove properties about the models such as “if an invoice is posted all items of the invoice are released” and to (2) prove (via refinement) that the business object conforms to a given message choreography.

2.3 Business Process Models

Business process models are usually the starting point when designing business applications. They model the (cross-organisational) processes which should be supported by business applications. There are a number of notations for modeling business processes, e.g. BPMN [8], which are mostly not backed by a formal semantics.

We are experimenting with modelling business processes in Event-B with the goal to (1) automatically translate from an established process modelling language into Event-B and (2) prove properties such as timed data consistency (i.e. consistency among distributed components that is achieved not immediately but after a certain time span) which go beyond classical properties in the verification of business processes.

3 Experiences and Challenges

In all experiments mentioned above it was possible to model the given problem with the help of the chosen formal modelling language Event-B. It was also possible to analyse the model.

The analysis was based on proving the proof obligations generated by the Event-B modelling platform Rodin with the help of the provers provided there.

Though techniques like automatically generating invariants through templates [9] helped in closing a considerable amount of proofs automatically, the degree of proof automation is currently not so high that Formal Methods unexperienced users could do the verification work. We investigate whether pattern-based approaches [10] can further increase proof automation.

Analyses with the help of the model checker ProB were very helpful during the creation of the models, especially the interactive simulation. We got most positive feedback about the possibility to derive test suites from the models, achieved from using ProB. Using this approach would require least overhead in adapting quality assurance processes in the company because testing is an established QA technique.

A major drawback of our approach to generate a formal model from existing models is that users expect that the feedback from the analysis tools are incorporated in the source model. With the help of the open Rodin platform we could create plugins which extract model checking results, animation snapshots, or even proof results, and mark-up the original diagrammatic model for specific typical cases. However to deal with this in general is a notorious difficult endeavour which requires further investigations.

A further issue is that the central concept of refinement intrinsic to approaches like Event-B is not present in the original diagrammatic models. Therefore our generated Event-B models were initially flat (consisting of one or two layers of refinement). The power of refinement-based approaches to introduce complexity (of proofs) incrementally might thus not be fully exploited yet. Our strategy will be to make use of modelling features (like sub-processes in BPMN) to derive a suitable refinement structure or to introduce these into the modelling language (like done in UML-B [11]). An important aspect to this is that Event-B users need stronger guidelines and documentation from the Event-B community on how to obtain a good refinement hierarchy and what the main rules are to change a refinement hierarchy during a development – this will be a necessary precondition to find good generic refinement strategies for our domain-specific problems.

4 Summary

We have given a classification of Enterprise Software and investigated the usability of formal development methods for enterprise applications. Our experiments on choreography modelling, business object modelling, and business process modelling use Event-B as formal specification language being the result of an automatic model transformation from diagrammatic in-house languages. Experiments were largely positive, but major challenges remain. These are: providing adequate user feedback on prover or model checker results, increasing automation, and better exploiting refinement.

References

1. Kätker, S., Patig, S.: Model-driven development of serviceoriented business application systems. In Hansen, H.R., Karagiannis, D., Fill, H.G., eds.: Wirtschaftsinformatik (1). Volume 246 of books@ocg.at., Österreichische Computer Gesellschaft (2009) 171–180
2. Wieczorek, S., Roth, A., Stefanescu, A., Kozyura, V., Charfi, A., Kraft, F.M., Schieferdecker, I.: Viewpoints for modeling choreographies in service-oriented architectures. In: Proceedings of the 8th IEEE/IFIP Conference on Software Architecture (WICSA'09), IEEE Computer Society (2009)
3. Abrial, J.R., Hallerstede, S.: Refinement, decomposition, and instantiation of discrete models: Application to Event-B. Fundamenta Informaticae **77**(1-2) (2007) 1–28
4. Kozyura, V., Roth, A., Wei, W.: Local enforceability and inconsumable messages in choreography models. In Dranidis, D., Stamatopoulou, I., eds.: Proceedings of 4th South-East European Workshop on Formal Methods (SEEFM'09), IEEE (2009)
5. Wieczorek, S., Kozyura, V., Roth, A., Leuschel, M., Bendisposto, J., Plagge, D., Schieferdecker, I.: Applying model checking to generate model-based integration tests from choreography models. In: Proceedings of the 21st IFIP Int. Conference on Testing of Communicating Systems (TESTCOM'09). LNCS, Springer (2009)
6. Abrial, J.R., Butler, M.J., Hallerstede, S., Voisin, L.: A roadmap for the rodin toolset. In Börger, E., Butler, M.J., Bowen, J.P., Boca, P., eds.: ABZ. Volume 5238 of Lecture Notes in Computer Science., Springer (2008) 347
7. Leuschel, M., Butler, M.J.: ProB: an automated analysis toolset for the B method. STTT **10**(2) (2008) 185–203
8. Object Management Group: Business Process Modeling Notation (BPMN) 1.1, Final Adopted Specification. http://www.bpmn.org/ (2008)
9. Kozyura, V., Roth, A.: Generation of gluing invariants for checking local enforceability of message choreographies. In Jastram, M., Laibinis, L., Lösch, F., Mazzara, M., eds.: Proceedings of Deploy Technical Workshop 2009, Newcastle University, Technical Report (2009)
10. Hoang, T., Fuerst, A., Abrial, J.R.: Event-b patterns and their tool support. In: SEFM, IEEE Computer Society (2009)
11. Said, M.Y., Butler, M.J., Snook, C.F.: Language and tool support for class and state machine refinement in uml-b. In Cavalcanti, A., Dams, D., eds.: FM. Volume 5850 of Lecture Notes in Computer Science., Springer (2009) 579–595

The seed was spread out: The State of Practice of Formal Methods outside europe

Aryldo G Russo Jr.

AeS Group & Research Institute of State of São Paulo (IPT),
agrj@aes.com.br

Abstract. The use of formal methods has constantly increased, although with basically two constraints: their use has been concentrated mostly in Europe, and they have been used only by big companies which are in charge of developing some safety critical applications and in some how are conected with academia projects. The aim of this talk is to present how formal methods have been applied in other parts of the world, mainly South America, and the Far East. It's splited in four parts. First, an introduction about the AeS Group, a small company that's been trying to apply Formal Methods in its projects. Second, a personal comparison of some formal method tools, namely: Atelier B[1], RODIN[2], and SCADE[3]. The comparison methodology is based on three different points of view: capability, that is, how these tools can satisfy project constraints, usability, which is basically the difficulty the user faces when trying to use the tool, and adequacy to the current development process. Third, some real industrial applications and how the formal method culture can drastically help the development process. And finally, some of the ongoing work that it's been developend by the author and gaps identified in industry that can be fulfilled by extending the features of the actual tools.

1 Introduction

The primary objective of this paper is to present the current State of Practice of Formal Methods in countries outside Europe, namely, Brazil and Korea. The general utilization of formal methods is presented focusing in how the principles that guide the formal methods usage can help in the software development process and not specifically the use of a particular method.

Initially, a background information about the reason to start working with Formal Methods, and the involvement of AeS group with academia is given. Then, in section 2, a general scenario of how these methods are being used nowadays in Brazil and Korea and particularly some industrial areas where formal methods are currently applied are shown. A brief description of the methods that support each of these tools is also presented.

In section 3, a comparison of three tools, namely, AtelierB[1], RODIN[2] and SCADE[3] is presented. This comparison is based on three aspects, tool capability, usability and fitting to current development process. Some real application of these tools are also shown in section 4 as well as the work that was performed to change the way that industry was used to think about software development, even in safety critical areas.

At the end, in section 5 the author presents some gaps that, from his personal point of view, can be fulfilled with some new or in phase of development, plugins and language extensions.

1.1 The AeS Group

The AeS Group has developed railway sub-systems since 1998. Among the systems developed by the group, the door system became one of the most important in the railway market due mainly to the architecture used (modular, and with distributed processing) and, since this kind of system deals with human lives, the strong concern of the group with reliability and safety.

During the development process, four versions of the main controller (called CGP) were created and, at each iteration, additional safety features were incorporated, using different techniques, such as hardware redundancy where different sources are employed to activate an output (for example, some safety outputs have to be activated both from a software command and from an external solicitation). Safety and reliability studies were performed, and all the identified potential weak points revealed were mitigated to prevent, or at least minimize, the hazard effects. In the current version of the equipment, even after applying all these hardware techniques, safety issues, as well as the software (firmware) correctness, robustness and failure avoidance remain to be fulfilled.

Due to the advances in technology, many safety functions that were handled by hardware are now responsibility of the embedded software. This fact triggered motivation to use formal methods in standards relevant to software safety [4]. Some standards can be followed to increase the equipment safety level. One of them is the IEC 61508 [5]. This standard presents four levels of safety, the so called Safety Integrity Levels - SIL, and above level 2, the use of formal method is required or suggested to achieve a certain level of completeness, robustness, and safety, that grows as the level grows. The goal of using formal methods is to produce an unambiguous and consistent specification that is as complete, error-free and with less contradictions as possible, however simple to verify.

To address the group concern with safety, the AeS group decided to identify a formal method that would best fit the current CGP SIL 3-level requirements and railway industry standard practices and standards (as is the case of CENELEC EN 50128[6]).

Based on these previous information, and the constraints such as, the size of the company (at that date, AeS counted only with 15 employees, and most of them working on administrative tasks) and the lack of deep knowledge of the method itself, the AeS group decided, first, to study and use the B method[7] and, second, to look for assistance from academia, which was obtained from two Brazilian Universities (Universidade de São Paulo and Universidade do Rio Grande do Norte).

From that time, and after facing several pitfalls, AeS Group has acquired a reputation as a company that has the needed know-how to develop safety critical applications, and, nowadays, it is in charge of several training courses around the world teaching software development process for safety critical applications based on a formal method mind.

Nowadays, AeS Group has also the support of DEPLOY project and some universities like University of Southampton, and University of York, besides companies like ClearSy and Esterel.

1.2 Technological Research Institute of the State of São Paulo (IPT)

Anchored by previously performed studies, but with some reluctance, the author decided to finally initiate a ”formal” dedication in the Formal Methods field, and choose the Technological Research Institute of State of São Paulo (IPT) as starting point. During the last years, some articles were developed at IPT but the relationship with other research and academic centers was the main incentive to study the application of these methods in real world systems.

In the mean time, the author joined the Software Requirements Specification Laboratory (SoftREL). The main goal of SoftREL is to create, deploy and disseminate a research environment for post-graduate IPT students and other researchers, helping them to develop academic research and artifacts related to software requirements engineering. The laboratory intends to create academic and industrial partnerships aiming at the development of engineering techniques and tools required to deliver more reliable computer systems.

1.3 General scenario

Some information about the current software engineering process applied to safety-related developments is given below in order to picture out the differences in formal method applications outside Europe.

Basicaly, the software development process presented an recomended by IEC 61508[5] (The IEC 61508 has 7 parts, and the part 3 is related to the software development. The chapther of this part that present the software development process, shows that the recomended process is based on the ”V model” like the one shown in figure 1) is well known in South American companies, but since the time to market is, usually, extremely short, those recommendations are frequently put aside, and the ”craft” process is followed.

This ”craft” process consists basically in receiving the primary specification, performing the coding phase relying on the personal expertise, and making as few tests as possible. This is a good scenario to try to better the process through the use of formal methods without changing the manual tasks.

Those processes (presented in the standard) are barely known at the Far East, meaning Korea[1]. As presented in [8], the adoption of the recommendations

[1] this assumption is based on the author experience and based on a population of 4 big korean companies engaged in railway field

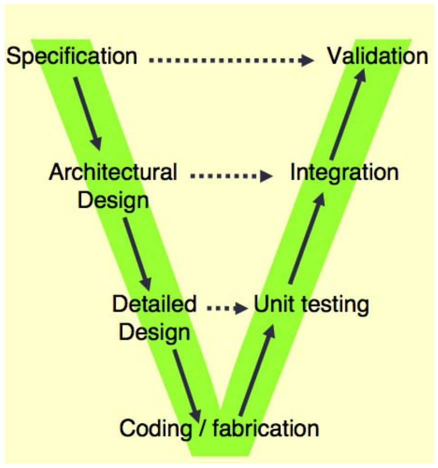


Fig. 1. V Model - Software Development Model

referenced in the software development process is in its infancy phase, meaning that even the standard understanding is not clear enough.

Based on the "V model" reference presented in figure 1, it is possible to point out:[2]

- Companies in South America
  - They are aware of the whole process
  - They usually rely on tests to guarantee the expected behavior
  - The transition takes place directly from NL specification to the code phases, some times, through an intermediate phase, based usually in UML specifications.
  - apparently, there is no traceability methodology
- Companies in the Far East - South Korea
  - The coding phase starts right after receiving the specification
  - Quality is the main concern, but no defined process is used to ensure that
  - They rely on experienced professionals to reach the desired quality level
  - clearly, there are only three phases: specification, coding and integration tests

Based on this view, it becomes clear that formal methods can not be applied straightforward. Before, it is necessary to create a better culture on software development process.

In order to create such culture, the advantages of using formal methods should be shown, and the the formal thinking must be integrated in the normal development process in small doses. In this way companies will be led to

[2] those points are based on author's feelings

accept that the development process can be speed up, and the costs of software development can be less than in normal process, mainly based on tests, where problems or errors are discovered only at the end of development.

2 Where formal methods (could be) are used

Many different industrial areas, where safety and reliability issues are highly important characteristics, have used, or at least have tried formal methods in order to increase their confidence that those requirements are met. Those industries are, mainly, Nuclear[9], Medical devices[10], Avionics, Aerospacial and transportation [11]. Some examples are the emergency contention measures in nuclear power plants, health support devices in medical applications, automatic pilot on avionics, positioning systems in aerospacial and signaling systems in tranportation just to cite a few.

This means that there is plenty of space for the adoption of supporting tools that could help either the development process (either system or software) in the sense of automatizing some parts of it, and also, in some cases, for speeding up those development tasks that are difficult to perform, while the developer uses his efforts in other more conceptual phases.

Unfortunately, even if the referenced industrial areas exist all around the world, the application of formal methods is not the true reality in South America and Far East (Korean) companies that work on those fields. It is something not easy to explain, as, in theory, the standards that should be followed by all those companies are the same, as for example IEC 61508[5](related to general functional safety, then, not field specific), DO-178b[12] for avionics and EN50128[6] for railways. All of these standards highly recommend the use of formal methods either in the specification phase or in the design phase in order to achieve high levels of the so called Safety Integrity Levels[13].

In order to change this scenario, the distance between mathematical notation and the normal procedures used so far has to be shortened, and for that some highly desired characteristics should be included in the current tools in order to reflect the activities that are normally performed in those industries.

Fortunately, it might not be so difficult as, at least, the development model that has been adopted in those industries (V model, in figure 1) is not different from the model used in a formal model development.

The focus of this paper is in the railway field, the author's area of application. The B formal method is the most frequently used in this field as mentioned in several works like [11] and [14], Recently, the Esterel[15] formal method began to be used as well, and the support tool for this method, SCADE[3] was certified as capable to produce safety code up to SIL4.

In any field of application, formal methods, and their related tools, can help in the development process replacing the human interaction of the phases (see figure 1): Detailed design, coding and unit testing, by an automated process, and consequently, can help to speed up the development process and to better the "quality" of the final product.

Moreover, with some effort, formal methods could help even during the very early phases as a support tool to verify the specification and to guarantee the transiction consistency to the later phases.

2.1 Formal methods

A brief description of the formal methods mentioned before is given in this section.

B method The B method for software development [7] is based on the *Abstract Machine Notation* (AMN) and the use of formally proved refinements up to a specification sufficiently concrete that programming code can automatically be generated from it. Its mathematical basis consists of first-order logic, integer arithmetic and set theory. Industrial tools for the development of B based projects have been available for a while now.

Event B Method Event B is a formal method used to model discrete systems. It is based on the B method[7] and has adopted some ideas from Action Systems. As in B, the semantic of this method is also based on proof obligations, though, the principles presented in the section 2.1 are valid in this case (with only some small differences). More about Event B language and method can be found in [16]

Esterel Language Esterel Language is a kind of syscronous language for reactive systems, as other languages as Lustre [17], and Statecharts [18]. Esterel originated from a joint INRIA-ENSMP project on the semantics of parallelism. As a result of the formal approach applied in Esterel Language, is it possible to generate efficient code from the models developed. More information about Esterel language can be found in [19]

3 Tool comparison

In order to verify how the current tools can be modified to reflect the industrial needs, a brief comparison of some existent tools is presented. This comparison is restricted to some tools that have already been used in the author's application field, that is, railways application. Those tools are, Atelier B, RODIN and SCADE.

3.1 Methodology

The tools are classified according the author's personal feelings based on comments obtained in trainings, and serveral other applications like revalidations and system development based on B (EventB method) during the last 3 years.

It is prudent to establish the difference in maturity of these tools. While SCADE and AtelierB have been in the market for a long time, RODIN is about to be released in its first official version (version 1.0). Thus, the manufacturers of the first two have already received many feedbacks from their industrial users helping them to change the directions when the users were not satisfied (as occurred with AtelierB case, where after a lot of complaints about the user interface, its GUI was completely changed), while the last one has not have time yet to receive or to implement completely such feedbacks.

The comparison methodology was based on three aspects, as follows:

– *capability:* the verification of how these tools can satisfy project constraints, like, for example, parallel behaiviors, time treatment, etc...
– *usability:* basically, which is the difficulty the user faces when trying to use the tool
– *adequacy to the current development process :* how the tool can better fit in the process without causing too many changes in the way it was performed so far

To make a classification of these aspects the author used a simple ranking method, as follows:

– *1 Very dificult* - It was not possible to achieve any intended task, meaning that a strong knowledge of the tool or of the method is needed
– *2 Medium* - Some task could be achieved, with or withoud dificulties. A basic knowledge is needed.
– *3 easy* - It was easy to achieve any task. No previous knowledge of the tool or method is needed.

The results are presented in table 1. It's important to cite that some points used in the justification are related to the method supported by the tool and not to the tool itself, meaning that there is also space for a language extansion as well.

3.2 Chart comparison

Aspect	**capability**	**usability**	**adaptation**	***Results***
AtelierB	2	1	2	*5*
RODIN	2	2	1	*5*
SCADE	2	3	3	*8*

Table 1. Comparison table

Those results were obtained considering the points below:

– AtelierB

  - the capability to solve the project constraints is not so bad, but it is necessary to know a lot of the formal language and constructs to be able to have easy proof obligations. Moreover, the time dependency needs to be discretized in order to model it.
  - although, the version 4 of AtelierB supplies a real better usability, all comments received so far are based on the previous version where the lack of a good User Interface makes its usage painful.
  - since it allows to go from the specification to the code it can be considered as a good tool for that purpose, but as the interactions during the middle phases (refinements) are some times, painful, it can not receive the higher grade.
- RODIN
  - since it is not so different from AtelierB, similar results are shown, i.e. the capability to solve the project constraints is not so bad, but it is necessary to know a lot of the formal language and constructs to be able to have easy proof obligations. Some extentions are being developed to treat time constraints[20], but it was not incorporated in the language yet, so the same problems about time in AtelierB should be considered here also.
  - the way that RODIN was constructed is quite helpful for a non experienced person, as it is only necessary to fill down some fields to have a basic specification, but the lack of text editor that could help more experienced person and speed up the specification process lowers its classification
  - the lack of possibilities of decomposition at the moment of the evaluation and the ability to help only in the system specification phase,make of RODIN a difficult tool to be used in the current process.
- SCADE
  - even based on a different concept, where formal methods are behind the scene, it has a great capability to deal with project constraints, but some formal background is still needed to construct correct models.
  - as it was built from the very beginning to be an industrial tool; its usability is its strongest point, with a good interface and a lot of fancy features that captivate the user. A lot of things can be done based on templates and patters, what helps a lot as well
  - Besides the capability to go from the specification to the code, it has also some other complementary tools which help in important auxiliary tasks in the project such as requirement management, traceability, etc..

4 Experiences

With experience in formal methods both as a practitioner and as a researcher the author has tried during the last 3 years to introduce formal methods in the projects he has worked on. He concluded that even if formal methods can not fulfill all industrial needs they can help a lot to better model the development process and the resultant product (or software).

Three different projects that the author has been working on recently and the achievements so far are summarized below:

4.1 Signaling system

European companies that develop signaling systems for railway applications are known as some of a few that use formal methods during the development process. It is also true that, not all of their branches around the world follow the same concept. During 2008 the author participated in a revalidation process of a signaling system using B method and its associate tool, AtelierB.

As, for this kind of system, there is always a start point, i.e., the new project is, usually, based on a previous one, the task consisted in implementing new functions and then revalidating all the system (the standard IEC 61508, for systems classified as SIL4, requires that at the moment any function is, changed, the whole system has to be revalidated).

As the system was previously developed in B, this kind of task became a trivial one, not only because the B method was used but also because the related tools (AtelierB in this case) are powerful enough to keep track of the changes and reprove only what is really needed. Basically, the changes were applied in the abstract model, and after that they were reflected in the refinements and implementation. New proof obligations were generated and the affected older ones were reapplied. Even though the project was big enough for a real world comparison, where more than 4000 proof obligations were generated, but around 95% were proved automatically.

As a result of the complete process no failures where detected after the deployment of the system. The associated costs in this development were less than in a traditional process as there were no needs of maintenance changes and the necessary time dedicated to testing was really short. But again, this job was performed in a company that has been using formal methods for a long time.

4.2 Door system

In order to verify the consistency of a door system specification, RODIN was used as a proof of concept, and it was possible to show the benefits of this approach to the final customer. This job was developed based on a small portion of the natural language specification, but at least 3 contradictions or inconsistencies on it could be verified. The objective was to help the door system manufacturer to rewrite the specification based on the result of the verification of the formal model.

The natural language specification is more than 100 pages long, and the needed information is spread out over all this specification. The following two statements of the specification show one of the contradictions that were found:

- The train is not allowed to move while at least one door is open;
- If the emergency buttom is pressed, the respective door must open when the train speed is under 6 km/h.

In this example it is easy to notice the contradiction, moreover because both sentences were placed together, but those statements were spread out in the specification, so the direct comparison was not so clear.

In the example, the contradiction refers to the behavior of the door, which should not open until the train is completely stopped, but which also should open in an emergency situation when the speed of the train is under 6 km/h.

The machine in figure 4.2, represents this specification, and the PO in figure 2 represent the contradiction.

MACHINE Open_contradiction
VARIABLES

`train_stoped` boolean. when the train is stoped it's value is TRUE
`train_low_speed` boolean. when the train speed is below 6km/h it value is TRUE
`door_authorization` boolean. when the train is allowed to open doors it's value is TRUE
`emergency_buttom` boolean. if the buttom is pressed, it's value is TRUE

`open_comand` boolean. if true, command the opening
`train_speed` NAT. real speed

INVARIANTS

inv1 : $train\_stoped \in BOOL$
inv2 : $door\_authorization \in BOOL$
inv3 : $train\_low\_speed \in BOOL$
inv4 : $emergency\_buttom \in BOOL$
inv5 : $train\_stoped = TRUE \Rightarrow door\_authorization = TRUE$
inv6 : $train\_stoped = FALSE \Rightarrow door\_authorization = FALSE$
inv7 : $train\_stoped = TRUE \Rightarrow train\_low\_speed = TRUE$
inv9 : $open\_comand \in BOOL$
inv10 : $train\_speed \in \mathbb{N}$
inv11 : $door\_authorization = FALSE \Rightarrow open\_comand = FALSE$

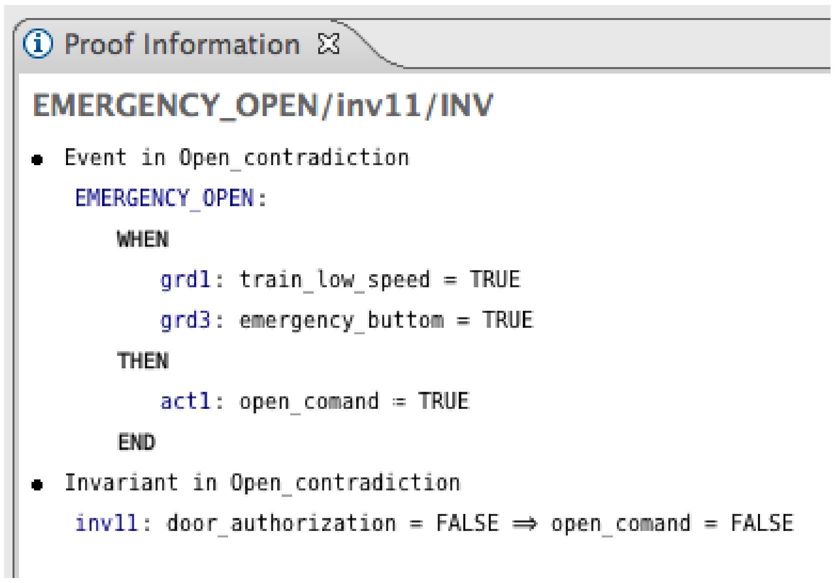
EVENTS
Initialisation
 begin
  act1 : $door\_authorization := TRUE$
  act2 : $train\_stoped := TRUE$
  act3 : $train\_low\_speed := TRUE$
  act4 : $emergency\_buttom := FALSE$
  act5 : $open\_comand := FALSE$
  act6 : $train\_speed := 0$
 end
Event $EMERGENCY\_OPEN \widehat{=}$
 when
  grd1 : $train\_low\_speed = TRUE$
  grd3 : $emergency\_buttom = TRUE$
 then

```
        act1 : open_comand := TRUE
    end
Event   LOW_SPEED_MONITOR ≙
    when
        grd1 : train_speed ≤ 6
    then
        act1 : train_low_speed := TRUE
    end
Event   ZERO_SPEED_MONITOR ≙
    when
        grd1 : train_speed = 0
    then
        act1 : train_stoped := TRUE
        act2 : train_low_speed := TRUE
        act3 : door_authorization := TRUE
    end
Event   AUTHORIZARION_RELEASE ≙
    when
        grd1 : train_speed > 0
    then
        act1 : door_authorization := FALSE
        act2 : train_stoped := FALSE
        act3 : open_comand := FALSE
    end
Event   LOW_SPEED_RELEASE ≙
    when
        grd1 : train_speed > 6
    then
        act1 : train_low_speed := FALSE
        act2 : train_stoped := FALSE
        act3 : door_authorization := FALSE
        act4 : open_comand := FALSE
    end
END
```

It's clear that to discharge this PO, (figure 2) it is not a question of correcting the model, but the natural language specification must be changed to avoid this kind of ambiguities or contradictions.

In this case three different approaches or options were proposed, as follows:

1. The train is not allowed to move when at least one door is open, *unless in a emergency situation*;
2. The train is not allowed to move *over 6 km/h* when at least one door is open;
3. If the emergency buttom is pressed, the respective door must open when the train *stops*



```
Proof Information

EMERGENCY_OPEN/inv11/INV

• Event in Open_contradiction
  EMERGENCY_OPEN:
     WHEN
        grd1: train_low_speed = TRUE
        grd3: emergency_buttom = TRUE
     THEN
        act1: open_comand ≔ TRUE
     END
• Invariant in Open_contradiction
  inv11: door_authorization = FALSE ⇒ open_comand = FALSE
```

Fig. 2. PO to be discharged

The first option was choosen by the custumer, and the specification and model were changed to reflect this new constraint.

This simple example helped to present the formal method benefits, stating the impossibility to introduce ambiguities and contradictions.

The objective now is to try to represent the complete specification of one train sub-system (probably the door system) in Event-B[21], and reformulate the natural language specification in a better representation. At least, pointing out the items that need to be revised to create a more consistent specification.

4.3 Platform screen doors

Platform Screen Doors, aka PSD, is a door system that is installed in the platform stations to avoid people to fall down to the track. The safety related issues are even higher than for the train door system as, people get used with it, and a dangerous situation can lead to severe accidents[11]. For example, if the train departures with doors in PSD open, the train can easily hit someone.

This kind of system is being installed in Metro São Paulo, Brazil, and a company from Korea was hired to develop and install the system. The same standards must be applied in order to guarantee that the desired safety level (in this case SIL 3[22]) will be met.

Besides the safety constraints (that by themselves are a huge problem) there is no room to rework as the whole system has to be in operation at the end of 2009. As the first phases of the "V model", take a lot of time, while in a formal process, there will be no time for many tests at the end. The IEC 62279[23] can be used as support standard to guide on the necessary documentation that should be generated to prove that the needed care was taken during the development process. The documentation that needs to be generated is the following:

1. System Requirements Specification
2. System Safety Requirements Specification
3. System Architecture Description
4. System Safety Plan
5. Sw Configuration Management Plan
6. Sw Verification Plan
7. Sw Integration Test Plan
8. Sw/Hw Integration Test Plan
9. Sw Requirements Specification
10. Sw Requirements Verification Report
11. Sw Architecture Specification
12. Sw Design Specification
13. Sw Arch. and Design Verification Report
14. Sw Module Design Specification
15. Sw Module Test Specification
16. Sw Module Verification Report
17. Sw Source Code Verification Report
18. Sw Module Test Report
19. Sw Integration Test Report
20. Sw/Hw Integration Test Report
21. Sw Validation Report

Moreover, it is requested that a formal method should be used from the detailed specification to the unit tests, as mentioned in section 2.

Another problem that was faced is the lack of knowledge on formal methods and development process by the team in charge of the project. In Korea there is no culture of a structured process[3]

Based on all of these considerations, SCADE tool was selected to help on these tasks. As mentioned in section 3, SCADE seems to be the best choice for non-formal method people.

Items 10 to 19 from the list presented before can be performed, either automatically or with the support of SCADE tool. All the formalism is performed behind the scenes, so the user can feel comfortable in developing what is really needed from his point of view.

Even with this simplistic view, it was possible to verify that the formalism, and moreover, the capability to model checking and theorem proving, helped to better the quality and consistency of the generated documentation and to verify missing points and inconsistencies. On the other hand, for more complex systems, or systems where the number of variables grows exponentially, or a lot

[3] to not be a so strong argument, this is a fact based on the author experience and could be noted at those 4 companies where the author has been work during the last year

of arrays and matrix are used, it's possible to note some bottle necks in it's usage, like poor performance from the generated code, for example.

As an example, two functions that should be modeled, based on the first requirement specification of one of the PSD system equipments are shown.

The equipment is called, PCM, and it is in charge of control the door open and close functions while in manual mode, what means, while PCM is enabled.

The two extractions from the Software Requirement Specification are as follows:

- *open command* If PCM is enabled, and the OPEN buttom is pressed longer than 1 second, the OPEN command has to be generated.
- *close command* If PCM is enabled, and the CLOSE buttom is pressed longer than 1 second, the CLOSE command has to be generated.

Using SCADE, it was modeled like figure 3

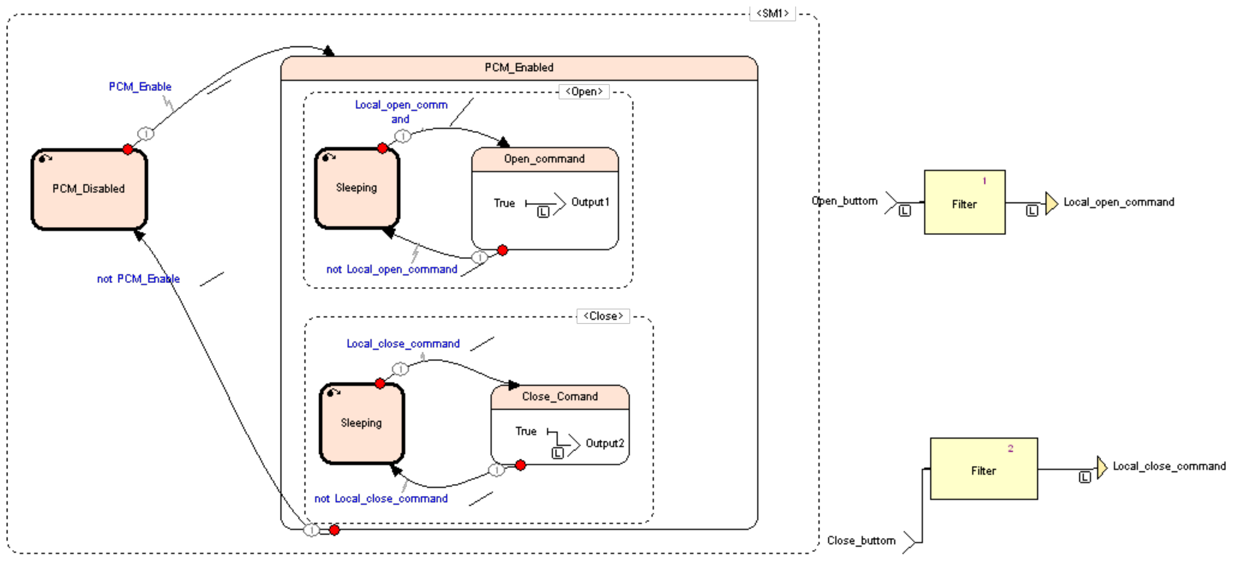


Fig. 3. SCADE model

Again, it is easy to realize that there are a lot of missing information and contradictions. For example, it is not possible to say, based only in the specification whether it is correct or not, to generate a close command while the open command is present, and vice versa. There is no information to determine when the command (doesn't matter open or close) should be turned off. One can figure out a lot of different needs, this is not the objective.

The main objective here was to present that a simple way to formalize the development process, whether or not, with heavy formal methods, helps a lot to find this kind of problems.

It's an ongoing project, and the author hopes to present some strong evidences to support these assumptions.

4.4 considerations

Some other points should be enhanced. Despite all odds, and as pointed in [24], it was not necessary to have someone with strong knowledge in mathematics, although the basic concepts were needed. Moreover, it was not necessary a big team in none of the described projects in order to successfully carry on the project. In the second project cited before, just one person did all the work.

In all of these projects, the most difficult task, and the one that took more time was the requirement elicitation and analysis. Even if it is not directly related to formal methods, the process adopted is important and the goal to build a formal model helps during the classification and elaboration of each requirement forcing them to be complete and non ambiguous.

At the end, the time (and money) that is spent in the earlier phases of the development process is greater than in a normal development, but the time (and much money) that is spent in tests and rework is definitely less. In the case of the second example (Door system), even using the formal methodology only as a support tool, the resulting test cases were much more effective, and the period of tests was shortened by 2 months (from 6 months to 4 months).

Based on these experiences, some features that could be included in the supporting tools, mainly the RODIN platform are summarized in section 5.

5 Gaps or needs

In this section some expectations about the future of supporting tools (mainly RODIN) are summarized and some characteristics that are considered necessary are pointed out. Most of them are being prepared, but some key points should be enhanced.

- *Requirements*, It is a fact that requirement problems are responsible for more than 40% of the total problems in a project 4. Then, this is the most important feature that should be integrated to RODIN platform. Some characteristics might be helpful:
  - the development of a "natural language dictionary", that would be used to rewrite the specification in a way it could be better understood.
  - the conversion of this redefined specification directly into the abstract model, avoiding with that the insertion of human errors
  - the creation of a tool based on the formal model that could write back a natural language specification. This is crucial when modeling the model manually and at the end there is the need to present it to the customer for approval.
  - the development, (better than a simple "natural language dictionary") of a methodology to annotate the requirement files allowing verifying the coverage of this requirement and helping traceability.

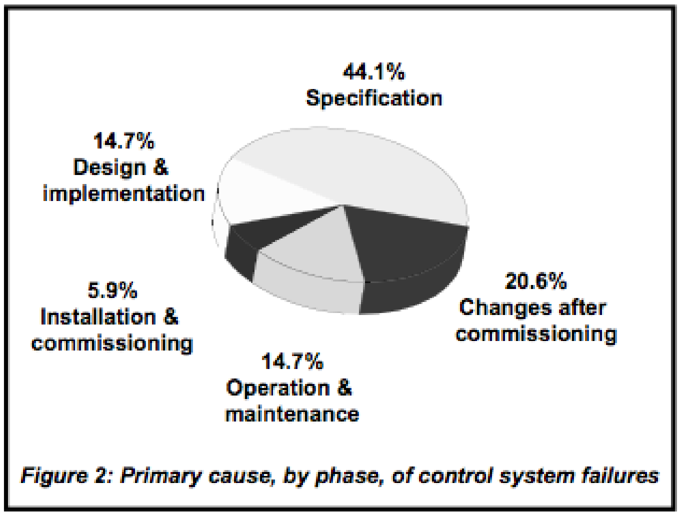


Fig. 4. Requirement problems from [22]

- *Traceability* Also related to requirements, RODIN platform should have the capability to:
  - to be able to track forwards, that is, when something is changed in the abstract model, it would be good if RODIN platform could point out the possible refinements that should be verified in order to meet the changes.
  - in the same way, it should be able to track backwards, and point where to verify if changes were made (intentionally or not) in the refinement machines.
  - still more crucial, it would be good if RODIN platform allowed to track back and forward all the requirements and any changes could be highlighted. Moreover, with this ability, it would be possible to verify if all requirements were fulfilled or not.
- *intermadiate languages* - This has already been done by UMLB plugin, but an interesting feature seems to be missing. Besides the ability to create state machines, for example, the ability to execute these models would be gratefully appreciated. With that, it would be possible to verify if the assumptions are correct, with no need to go inside the proof obligations.
- *test case generation* This seems to be one of the biggest gaps in industry right now. All generated tests are based on specialist feelings, and usually, what is tested is not exactly what should be. As a result, after a long time testing the system, at the moment it is set to operate some failure occurs, and the test generation phase has to begin again in order to address that specific failure. This routine happens several times until the product can be finally released.

  The Proof Obligations are strongly pointed (at least by the author fellings) as the basic source for generating test cases that are necessary and sufficient.

If those proofs are necessary and sufficient to validate the specification, why not use those proofs to generate the test case scenarios?

6 Conclusion

From the data presented in this paper the author concludes that: The application of formal methods in industry is growing, however most of the times as a result of some projects involving academia and industry, like DEPLOY project.

It is clear that outside Europe, formal methods usage is still incipient, and more effort in showing the benefits of that use is needed. In order to facilitate this approach we need tools that do not scare the customer in a first sight, otherwise the fear not to perform a good job will be always greater than the possibility of creating better products.

If these barriers could be broken, the use of formal methods would spread out really fast.

A great step could be done with the introduction of the features presented above.

If the managers are open minded, and admit waiting a bit more at the beginning of the development to see real results, (light or heavy) formal methods application could be more cost-effective and could, at the end, decrease the costs of the whole project by decreasing the costs in test and maintenance phases.

7 Aknowledgements

I'd like to thank to all univiersities and companies cited in this paper for their support providing me recomendations and tools.

Finally I'd like to thank Mr. Hrvoje Belani, who devoted his time to review this paper and mostly for his comments that helped me to include substantial points in this work.

References

1. ClearSy: Atelierb 1
2. Butler, M., Hallerstede, S.: The rodin formal modelling tool. deploy-eprints.ecs.soton.ac.uk 1
3. Esterel: Getting started with scade. (Sep 2007) 1–148 1, 5
4. Bowen, J.P., Stavridou, V.: The industrial take-up of formal methods in safety-critical and other areas: A perspective. In: FME '93: Industrial-Strength Formal Methods, First International Symposium of Formal Methods Europe. Volume 670 of Lecture Notes in Computer Science., Odense, Denmark, Springer (1993) 183–195 2
5. Commission, I.E.: IEC 61508 - Functional safety of electrical/electronic/programmable electronic safety-related systems. International Electrotechnical Commission Standards (1998) 2, 3, 5

6. CENELEC: Software for Railways Control and Protection Systems. EN 50128. (1995) 2, 5
7. Abrial, J.: The b-book: Assigning programs to meanings. books.google.com (Jan 1996) 2, 6
8. Hwang, J., Jo, H., Jeong, R.: Analysis of safety properties for vital system communication protocol. Electrical Machines and Systems, 2007. ICEMS. International Conference on (2007) 1767–1771 3
9. Abrial, J.: Formal methods: Theory becoming practice. Journal of Universal Computer Science (Jan 2007) 5
10. Jetley, R., Iyer, S., Jones, P.: A formal methods approach to medical device review. COMPUTER (Jan 2006) 5
11. Lecomte, T., Servat, T., Pouzancre, G.: Formal methods in safety-critical railway systems. Proc. Brazilian Symposium on Formal Methods: SMBF (Jan 2007) 5, 12
12. RTCA, I.: DO-178B, Software Considerations in Airborne Systems and Equipment Certification. (1992) 5
13. Squair, M.: Issues in the application of software safety standards. Proceedings of the 10th Australian workshop on Safety critical systems and software-Volume 55 (2006) 13–26 5
14. Bernardeschi, C., Fantechi, A., Gnesi, S., Larosa, S.: A formal verification environment for railway signaling system design. Formal Methods in System Design (Jan 1998) 5
15. Boussinot, F., Simone, R.D., ENSMP-CMA, V.: The esterel language. Proceedings of the IEEE (Jan 1991) 5
16. Abrial, C., Voisin, L.: Event-b language 6
17. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow language lustre. Proceedings of the IEEE **79**(9) (1991) 1304–1320 6
18. Harel. . . , D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming (Jan 1987) 6
19. Berry, G.: The foundations of esterel. Foundations Of Computing Series (2000) 425–454 6
20. Bicarregui, J., Arenas, A., Aziz, B., Massonet, P., Ponsard, C.: Towards modelling obligations in event-b. ABZ2008 (May 2008) 14 8
21. Metayer, C., Voisin, L.: The event-b mathematical language 12
22. Bell, R.: Introduction to iec 61508. Proceedings of the 10th Australian workshop on Safety . . . (Jan 2006) 12, 16
23. Commission, I.E.: IEC 62279 Railway Applications Communications, Signalling and Processing Systems Software for Railway Control and Protection Systems. International Electrotechnical Commission Standards (2002) 12
24. Bowen, J., Hinchey, M.: Ten commandments of formal methods ten years later. COMPUTER (Jan 2006) 15

Formal Foundation to Systematic Development of Simulink/Stateflow Models

Manoranjan Satpathy and S. Ramesh

India Science Lab, GM Technical Centre India Pvt Ltd, Bangalore-560066, INDIA
Email: {manoranjan.satpathy, s.ramesh}@gm.com

1 Introduction

Refer to Figure 1(A) which outlines the current development process of automotive control systems. Design models are directly obtained from the feature requirements but in an *ad hoc* manner. Simulink/Stateflow (SL/SF) is the primary modeling notation for the development of hybrid control systems; UML based modeling is used for discrete control applications. In industry, sometimes, C-code is directly developed from requirements. We focus here on the development of design models from feature requirements. We also assume the SL/SF modeling framework. The current process has limitations:

- The semantic gap between the requirements and the design models is high; therefore, obtaining SL/SF model directly from the requirements can be prone to errors.
- The process is primarily manual, a trial-and-error approach is used, and often it is based on experience.
- Requirements and intermediate steps are primarily mental models having no concrete representations.
- A lot of simulation and testing are performed *a posteriori* to have a confidence in the design.

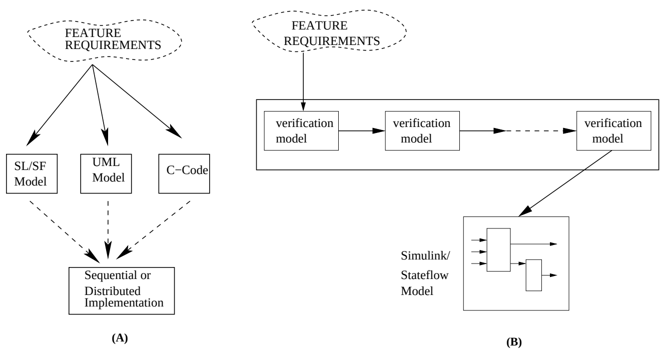


Fig. 1. (A) Current Development Process, (B) Design models from VMs

The aim of this paper is to systematize and formalize the *feature-to-design* phase; the characteristics of this process are:

– Through a chain of incremental steps requirements are guided towards the design. Each step produces an abstract model which we term as a verification model (VM); the primary purpose behind VMs is formal verification.
– Each VM has a clear mathematical representation.
– From a VM, we obtain the next one in the chain by applying correctness preserving transformations.
– A VM may have non-determinism which allows multiple design choices.
– Incremental steps are manual, but rigorous verification is performed using tool support; the technique used is theorem proving.
– Once a VM is sufficiently detailed, a design model can be derived from it.
– Design models are correct-by-construction; so, no additional validation.

We use the Event-B method for the construction of automotive control systems; Event-B models will be our VMs. We obtain a SL/SF model from a detailed Event-B model. Many verification/validation (V/V) infrastructures like Hardware-in-loop (HIL) testing, Plant-in-loop (PIL) testing and FlexRay bench have been built around Mathwork's SL/SF models. The SL/SF models that we generate can get the benefit of existing V/V infrastructure. Our main contributions are: (a) use of Event-B method in obtaining SL/SF models, and (b) the new notion of RRM diagrams play a key role in this development process (RRM stands for Requirement, Refinement and Modeling).

2 The Method

A variant of Cruise Controller is our running example. A vehicle can be in cruise mode only when an external switch called `cc_switch` is on; however, during the cruise mode, the vehicle also need to satisfy other conditions (*cc_cond*): (a) the throttle and the brake pedals are not disturbed, and (b) the vehicle speed is not less than the minimum cruise speed. When the vehicle is in cruise mode the driver can press the set button to bring the vehicle to cruise active state, and the current vehicle speed becomes the cruise speed. When in active state, the throttle angle for the cruise speed is computed by the controller; otherwise, it is determined by the driver. When in cruise active state, the cruise speed can be increased (by 1 KM/h) by tapping the `accl` button once, or it can be decreased (by 1KM/h) by tapping the *dccl* button once. These buttons can be tapped multiple times. When the *cc_cond* becomes false, the cruise state becomes inactive. When cc_switch is on and *cc_cond* is true, tapping the resume button engages the vehicle again in cruise active state, and the last cruise speed becomes the current cruise speed.

2.1 Development using Verification Models

In Figure 2, M_0 is the initial abstract model. $THETA_d$ and $THETA\_c$ are respectively the throttle angles produced by the driver and the controller. When

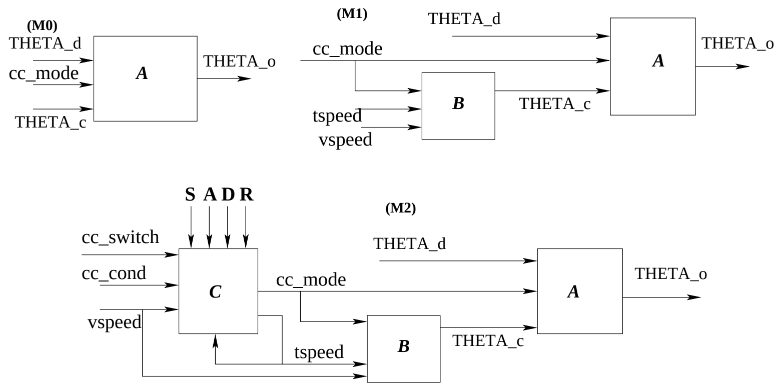


Fig. 2. Development steps as RRM diagrams-I

the value of *cc_mode* (cruise control mode) is active, then $THETA_c$ is produced as the output; otherwise $THETA_d$ is the output. This model captures a requirement fragment. Model M_1 is a refinement of M_0 which considers additional requirements. When *cc_mode* is active, $THETA_c$ computation depends on *vspeed* (vehicle speed) and *tspeed* (target cruise speed). We have assumed a proportional controller; therefore, increase in $THETA_c$ is proportional to ($tspeed - vspeed$).

The next refinement focuses on the computation of the *cc_mode*. The inputs are *cc_switch*, *cc_cond*, *vspeed* and *tspeed*. When *cc_mode* is active, the value of *tspeed* is also computed. The additional external inputs are the status of the `set (S), accl (A), dccl (D)` and the `resume (R)` buttons. In Figure 2, block **C** of M_2 depicts the computation of *cc_mode* and *tspeed*. M_2 is next refined to obtain M_3 (Figure 3). In M_2, *cc_cond* is a free input but in reality, it is computed in terms of environment inputs like brake pedal (*brP*), throttle pedal (*thP*) and the *vspeed*; M_3 makes this computation explicit.

In model M_4, the plant is introduced which relates the throttle output $THETA_o$ and the vehicle speed; refer to Figure 3. M_5 is the final refinement in which plant input of M_4 is replaced with sensed value of the speed (*sensed_speed*); $THETA_o$ is the actuator input to the plant.

2.2 RRM Diagrams

We have explained the development process by using a chain of diagrams, which we term as *RRM Diagrams*. Let us consider model M_1 as a RRM diagram (Figure 2). Block **A** captures and model the requirement that when CC mode is active, the throttle angle produced by the controller is the output; otherwise the throttle angle determined by the driver is the output. This requirement description could be attached to block **A**. Block **B** captures and model the requirement that when CC is active, $THETA_c$ depends on *vspeed* and *tspeed*. In summary, requirement fragments can be partitioned across the blocks and the wires. An informal mapping can be established between the requirements in the require-

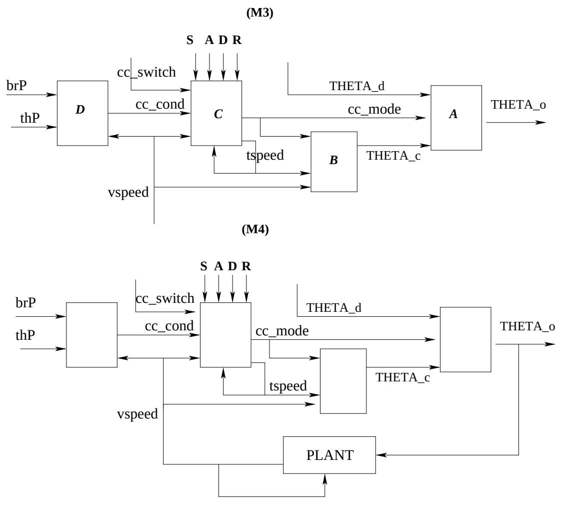


Fig. 3. Development steps as RRM diagrams-II

ment document and those in the RRM diagram and thus completeness issues can be checked. Requirement fragments when distributed across blocks presents the user with a clear understanding of the modeling activity going on. And thus the RRM diagrams can be a mechanism for keeping the user/requirement engineer in sync with the designer. This constitute the requirement viewpoint of the RRM diagrams.

A RRM diagram has a formal presentation as an Event-B model. The model is proved against invariant preservation and deadlock-freedom. Also, the diagram guides us as to what is the next possible refinement. For instance, in M_1, block **B** identifies that none of its inputs are environmental inputs, and therefore each input can be refined. In M_1, *cc_mode* is a free input but not an environmental input; so, block **C** is created in the next refinement which computes *cc_mode* in terms of other environmental or intermediate inputs. In summary, RRM diagram always can guide as to what would be the next possible refinement, and when to stop the refinement process. Refinement amounts to addition of new blocks and new links, or splitting a block into multiple blocks and links.

RRM diagrams have a modeling view point. A RRM diagram indicates to what extent the design has progressed. Note also that a RRM diagram resembles a SL/SF model; the logic and the meaning of blocks and links in a SL/SF model is evident from the structure of the corresponding RRM diagram.

3 SL/SF model generation

A detailed Event-B model can be used to derive a SL/SF model. Figure 1(B) shows the complete process. Our method considers both the Event-B model and

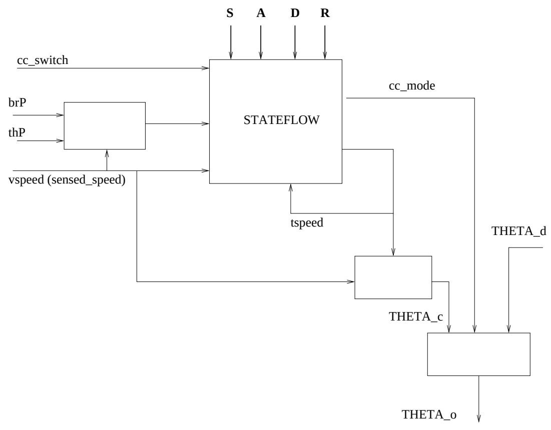


Fig. 4. Outline of SL/SF model for CC obtained from Event-B models

the corresponding RRM diagram together to derive the SL/SF model. We now present a set of guidelines.

- Inputs to the RRM diagram become the inputs to the SL/SF model. Input variables receive non-deterministic values through non-deterministic events in the Event-B model.
- There is an one-to-many mapping between the RRM diagram blocks and the events in the Event-B specification. If a block is mapped exactly to one event, then this event becomes a Simulink block; the internal details of the block is guided by the computation in the event.
- A block is associated with many events in the Event-B specification and each event is of the form $f(T_1, \ldots) \rightarrow (T_1, \ldots)$ meaning that one input and one output share a type. If T_1 is enumerated then the block is a candidate for a Stateflow block in the Simulink model. The enumerated type(s) determine the states and, the concerned events determine the transitions.
- A block in RRM diagram is mapped to multiple events but the event do not satisfy the criteria just discussed, then they become Simulink blocks. The Simulink block structures are determined by the event condition and action.

Figure 4 shows the SL/SF model which is obtained from the final refinement of the CC in Event-B. This model is structurally similar to the corresponding RRM diagram.

Consider the block **D** in M_4; this block corresponds to a single event in the Event-B model which computes $brP \wedge thP \wedge vspeed \geq 25$ This computation can be trivially translated to Simulink. Refer to block **C** of the RRM diagram in Figure 3. This block is mapped to 10 events in the Event-B model. Since the events satisfy the Stateflow criteria – each event takes an element of $\{set, active, inactive\}$ and outputs an element of same type and in addition there are other actions – we build a Stateflow model for these events.

4 Summary

1. In a SL/SF model, wires between blocks carry signals. It means that there is no complicated data sharing between the blocks, only atomic data. The events in the Event-B model are so refined that there is no complex data transfer between them. The RRM diagrams are also designed keeping this in mind.
2. The RRM diagrams play a major role in the development process. They determine the sequencing structure, what should be the next refinement, and furthermore, they are used in SL/SF model generation. The mapping between the RRM diagram blocks and the Event-B events determines the nature of the SL/SF block. We believe the Event-B method along with the notion of RRM diagrams can solve the problem of obtaining correct SL/SF models.
3. Up to the final refinement, the development process is correct-by-construction. Assuming that the translation step from the final refined model to the SL/SF model is correct, the latter model is correct with respect to the requirements.
4. The requirement capture/development steps remove any ambiguities or inconsistencies in the requirements. The RRM diagrams also help a user in addressing the issue of incompleteness.
5. The semantics of Stateflow in general unsafe and could be ambiguous. That is why restricted subsets are defined [4]. Since the events in an Event-B are unambiguous, in translating them to Stateflow, the translator is expected to use safe features.

References

1. Abrial J.R., Hallerstede S.(2006). Refinement, Decomposition, and Instantiation of discrete models, Fundamentae Informatica, 2006.
2. Abrial J.R., Butler M., Hallerstede S., Voisin L. (2006). An open extensible tool environment for Event-B, Proceedings of ICFEM 2006, Macau.
3. The Mathworks, http://www.mathworks.com
4. Scaife N, Sofronis C, Caspi P, Tripakis S, Maraninchi F. 2004. Defining and translating a safe subset of Simulink/Stateflow into Lustre, ACM EMSOFT'04.

Abstract Specification of the UBIFS File System for Flash Memory

Andreas Schierl, Gerhard Schellhorn, Dominik Haneberg, and Wolfgang Reif

Lehrstuhl für Softwaretechnik und Programmiersprachen,
Universität Augsburg, D-86135 Augsburg, Germany
{schierl,schellhorn,haneberg,reif}@informatik.uni-augsburg.de

Abstract. Today we see an increasing demand for flash memory because it has certain advantages like resistance against kinetic shock. However, reliable data storage also requires a specialized file system that can handle the limitations of flash memory. This paper develops a formal, abstract model for the UBIFS flash file system. We develop formal specifications for the core components of the file system: the inode-based file store, the flash index, its cached copy in the RAM and the journal to save the differences. We give an abstract specification of the interface operations of UBIFS and prove some of the most important properties using the interactive verification system KIV.

1 Introduction

Flash memory has become popular in recent years as a robust medium to store data. It has significant advantages compared to traditional hard disks, in particular shock resistance. Therefore it is popular in digital audio players, digital cameras and mobile phones. Flash memory is also getting more and more important in embedded systems where space restrictions rule out magnetic drives, as well as in mass storage systems (solid state disk storage systems like the RamSan-5000 from Texas Memory Systems) since it has shorter access times than hard disks.

Flash memory has different characteristics when compared to a traditional hard disk. In brief, flash memory cannot be overwritten, but only erased in blocks whose size is typically around 64k. Erasing a block is slow and can only be done a limited number (typically 10^5 to 10^7) of times before memory wears out. Therefore it should be done evenly ("wear leveling") and as seldom as possible. In particular if memory is filled to 90% with static data, it must be ensured that even the static data is moved around. Otherwise all writes would happen on the remaining 10%, reducing life time significantly. These characteristics imply that standard file systems cannot be used with flash memory directly.

Two solutions are possible: either a flash translation layer is implemented (typically in hardware) emulating a standard hard disk. This is the standard solution used e.g. in USB flash drives. It has the advantage that any file system can be used on top (e.g. NTFS or ext2). On the other hand, the characteristics of file systems (e.g. partitioning of the data into the content of files, directory trees, or other meta data like journals etc.) cannot be effectively exploited using this solution.

Therefore a number of flash file systems (abbreviated FFS in the following) has been developed, that optimize the file system structure to be used with flash memory. Many of these FFS are proprietary (see [7] for an overview). A very recent development is UBIFS [10], which was added to the Linux kernel last year.

Increased use of flash memory in safety-critical applications has led Joshi and Holzmann [11] from the NASA Jet Propulsion Laboratory in 2007 to propose the verification of a FFS as a new challenge in Hoare's verification Grand Challenge [9]. Their goal was a verified FFS for use in future missions. NASA already uses flash memory in spacecraft, e.g. on the Mars Exploration Rovers. This already had nearly disastrous consequences as the Mars Rover Spirit was almost lost due to an anomaly in the software access to the flash store [15].

A roadmap to solving the challenge has been published in [6]. This paper presents our first steps towards solving this challenge. There has been other work on the formal specification in the context of the Grand Challenge. The papers [3], [5], [14] give top-level models of tree-structured file systems with POSIX-like operations. Butterfield and Woodcock [2] have started bottom-up with a formal specification of the ONFI standard [4] of flash memory. The most elaborate work we are aware of is the one by Kang and Jackson [12] using Alloy which gives a vertical prototype and discusses high-level recovery operations.

2 An Abstract Specification of UBIFS

Our approach described in detail in [16] is middle-out, since our main goal was to understand the critical requirements of an efficient, real implementation. Therefore we have analyzed the code of UBIFS (ca. 35.000 loc), and developed an abstract, formal model from it. Although the resulting model is still *very* abstract and leaves out a lot of relevant details, it already covers some of the important aspects of any FFS implementation. These are:

1. Updates on flash are out-of-place because overwriting is impossible.
2. Like most traditional file systems the FFS is structured as a graph of inodes.
3. For efficiency, the main index data structure is cached in RAM as well as stored on the flash drive.
4. Due to e.g. a system crash the RAM index can always get lost. The FFS stores a *journal* to recover from such a crash with a *replay* operation.
5. Care has been taken that the elementary assignments in the file system operations will map to atomic updates in the final implementation, to ensure that all intermediate states will be recoverable.

Our model is on the level of the Linux VFS (virtual file system switch) which is the abstract interface to the UBIFS implementation as well as to the implementation of other traditional file systems under Linux (like ext3 or reiserfs). It therefore offers the standard operations of this interface: creating inodes for files and directories, reading and writing a page of a file, truncating and renaming files, creating and removing hardlinks. The operations are given in an abstract programming language using a notation similar to ASMs [8], [1].

We have verified three properties for the file system:

- **Functional Correctness of the Operations.** We proved that all specified operations terminate and fulfill postconditions about their results.
- **Consistency of the File System.** We proved that all operations preserve consistency of the file system. Consistency specifies several well-formedness properties, e.g. that every entry in the RAM index always points to an existing inode in the main store.
- **Correctness of the Replay Process.** We proved that the replay operation which is called after a crash is able to restore a consistent file system, losing as little data as possible.

The full specification and all proofs are available on the Web [13].

3 Conclusion and Outlook

The work of this paper defines a first abstract model of the essential data structures needed in a FFS. It defines the four central data structures: the file store which stores node-structured data, the flash index, its cached copy in the RAM and the journal. Based on these, we have specified the most relevant interface operations and verified some core properties.

Our model should be of general interest for the development of a correct flash file system, since variants of the data structures and operations we describe should be relevant for every realistic, efficient implementation.

In future work we intend to use the model as an intermediate layer in the development of a sequence of refinements, which starts with an abstract POSIX specification and ends with an implementation based on a specification of flash memory based on the ONFI-Standard. The development of such refinements will offer many challenges and we finish sketching some of the most important ones.

- **Concurrency.** In a real implementation operations on files are often executed concurrently. Therefore, as an example writing a file consists of several page writes which may be interleaved with other operations (even with another write operation to the same file!).
- **Caching** All real file systems use caching to gain efficiency. A page may still be written in the cache only, when the write operation has already finished.
- **Efficient implementation of index structures**. Our model abstracts the index structure to a simple map, while in reality a B$^+$-tree is used. This representation allows to optimize updating the index stored on flash by only updating relevant branches (out of place), which makes it a "wandering tree".

– **Quality of wear leveling**. Our model has abstracted from erase blocks, and therefore does not have code for wear leveling and garbage collection. Lower level models will include this code, and one of the import verification tasks is to show that wear leveling is effective. This looks possible for UBIFS, since its wear leveling strategy is based on counting erase cycles.

References

1. E. Börger and R. F. Stärk. *Abstract State Machines—A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
2. A. Butterfield and J. Woodcock. Formalising flash memory: First steps. In *Proc. of the 12th IEEE Int. Conf. on Engineering Complex Computer Systems (ICECCS)*, pages 251–260, Washington DC, USA, 2007. IEEE Comp. Soc.
3. K. Damchoom, M. Butler, and J.-R. Abrial. Modelling and proof of a tree-structured file system in Event-B and Rodin. In *Proc. of the 10th Int. Conf. on Formal Methods and Sw. Eng. (ICFEM)*, pages 25–44. Springer LNCS 5256, 2008.
4. Hynix Semiconductor et al. *Open NAND Flash Interface Specification Technical Report Revision 1.0*, December 2006. URL: www.onfi.org.
5. M.A. Ferreira, S.S. Silva, and J.N. Oliveira. Verifying Intel flash file system core specification. In *Modelling and Analysis in VDM: Proceedings of the Fourth VDM/Overture Workshop*. School of Computing Science, Newcastle University, 2008. Technical Report CS-TR-1099.
6. L. Freitas, J. Woodcock, and A. Butterfield. Posix and the verification grand challenge: A roadmap. In *ICECCS '08: Proc. of the 13th IEEE Int. Conf. on Eng. of Complex Computer Systems*, pages 153–162, Washington, DC, USA, 2008.
7. E. Gal and S. Toledo. Algorithms and data structures for flash memory. *ACM computing surveys*, pages 138–163, 2005.
8. Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9 – 36. Oxford Univ. Press, 1995.
9. C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.
10. A. Hunter. A brief introduction to the design of UBIFS. URL: http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf, 2008.
11. R. Joshi and G. J. Holzmann. A mini challenge: build a verifiable filesystem. *Formal Aspects of Computing*, 19(2), June 2007.
12. E. Kang and D. Jackson. Formal modelling and analysis of a flash filesystem in Alloy. In *Proceedings of ABZ 2008*, pages 294 – 308. Springer LNCS 5238, 2008.
13. Web presentation of the Flash File System Case Study in KIV, 2009. URL: http://www.informatik.uni-augsburg.de/swt/projects/flash.html.
14. J.N. Oliveira. Extended Static Checking by Calculation Using the Pointfree Transform. In *LerNet ALFA Summer School 2008*, Springer LNCS 5520, 2008.
15. G. Reeves and T. Neilson. The Mars Rover Spirit FLASH anomaly. In *Aerospace Conference, 2005 IEEE*, pages 4186–4199, March 2005.
16. A. Schierl, G. Schellhorn, D. Haneberg, and W. Reif. Abstract Specification of the UBIFS File System for Flash Memory. In *Proc. of FM*, LNCS 5850, pages 190–206. Springer-Verlag, 2009.

Pattern-based Refinement of Confidentiality Requirements

Holger Schmidt

University Duisburg-Essen, Germany, Faculty of Engineering, Department of Computer Science and Applied Cognitive Science, Workgroup Software Engineering, holger.schmidt@uni-duisburg-essen.de

Problem Description

The software development principle of *stepwise refinement* is popular in software engineering, and is also well supported by *formal notations*. When performing stepwise refinement, software is developed by creating intermediate levels of abstraction. Starting with the requirements, an *abstract specification* is constructed, which is refined by a more *concrete implementation*. Then, the implementation must be verified against the specification, and further refinement steps are accomplished until the desired level of abstraction is reached.

In this paper, we consider the refinement of *confidentiality requirements*. In contrast to, e.g., integrity requirements, which are functional requirements on the correctness of data and thus typically preserved under refinement, this is generally not true for confidentiality requirements (see [9] for details). According to ISO/IEC 13335-1:2004 [1], "confidentiality is the property that information is not made available or disclosed to unauthorized individuals, entities, or processes". These unauthorized subjects are considered as the malicious part of the environment, which threatens an ICT-system. The goal of the software to be built is to protect the ICT-system against attacks that stem from this malicious environment.

Confidentiality requirements can be enforced using different kinds of countermeasures such as security mechanisms (e.g., encryption algorithms), physical protection mechanisms (e.g., security guards that control access to server rooms), and organizational means (e.g., user instructions concerning password security). In general, the adequacy of a countermeasure to establish confidentiality requirements depends on the confidentiality requirement and the environment in which the ICT-system is embedded. For example, a password-based encryption mechanism might work properly if applied in a private environment, but it is rather not suited for a public environment.

From the software engineering point of view, we must analyze the adequacy of security mechanisms such as encryption and access control mechanisms at a very early stage of software development. The goal of this analysis is to find an answer to the question "Does the selected security mechanisms work *properly* in the intended operational environment so that the confidentiality requirement is established?"

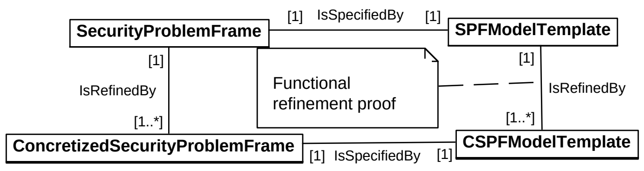


Fig. 1. Refinement on Pattern Level

Solution Approach

In earlier publications (cf. [2, 3]), we introduced a security engineering process named *SEPP* that focuses on the early phases of software development. The basic idea is to make use of special patterns defined for structuring, characterizing, and analyzing *problems* that occur frequently in security engineering. Similar patterns for functional requirements have been proposed by Jackson [5]. They are called *problem frames*. Accordingly, our patterns are named *security problem frames* (SPF). Furthermore, for each of these frames, we define a set of *concretized security problem frames* (CSPF) that take into account generic security mechanisms to prepare the ground for solving a given security problem. For example, we defined an SPF that describes the problem class of confidentially transmitting data over an insecure network, and we specified a CSPF that uses symmetric encryption to solve such problems.

According to [10], we consider the step from the instantiation of an SPF to the instantiation of a corresponding CSPF as a *formal* refinement step that not only *preserves* the *functional correctness* but also the *confidentiality requirement*. This *confidentiality-preserving refinement* step is supported on the level of patterns as described in Fig. 1. Since (C)SPFs are denoted rather informally as diagrams, we complement those (C)SPF that deal with confidentiality requirements by *model templates*. They constitute formal CSP (Communicating Sequential Processes) [4] models of the corresponding (C)SPFs. Each of these templates is deadlock- and livelock-free, which we proved using the model-checker FDR2 (Failure-Divergence Refinement) [6]. For each pair consisting of an SPF model template and a corresponding CSPF model template, we proved a failure-divergence refinement using FDR2.

Moreover, confidentiality-preserving refinement is supported on the level of instances as described in Fig. 2. The model templates are instantiated based on the instances of the corresponding (C)SPFs. We refer to instantiated SPF model templates as *SPF model instances* and to instantiated CSPF model templates as *CSPF model instances*. To prove that a CSPF model instance refines an SPF model instance in a confidentiality-preserving way, we must accomplish the following steps:

1. Prove that the CSPF model instance failure-divergence refines the SPF model instance.
2. Specify the confidentiality requirement based on the SPF model instance.

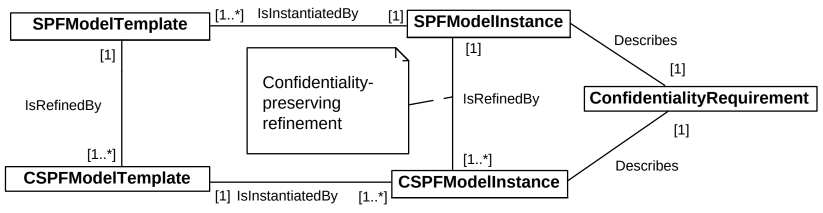


Fig. 2. Refinement on Instance Level

3. Prove that the SPF model instance fulfills the confidentiality requirement.
4. Prove that the CSPF model instance fulfills the confidentiality requirement initially defined for the SPF model instance.

Since the failure-divergence refinement is already proven on the pattern level, the effort for the proof on the instance level in the first step should be remarkably reduced. Moreover, this proof is tool-supported by FDR2.

In the second step, the SPF model instance is used to formally express the confidentiality requirement as an information flow property. Here, we use the framework for the specification of probabilistic and possibilistic confidentiality requirements by Santen [8]. Since confidentiality requirements are predicates on *sets* of traces, they cannot be modeled in CSP, and thus cannot be verified using FDR2. Nevertheless, we can mathematically specify a confidentiality requirement and prove in the third step that a given SPF model satisfies the requirement. Note that there does not exist *the* information flow property that allows us to express every informal confidentiality requirement. Instead, an adequate property depends on the confidentiality requirement that it formalizes. Mantel [7] gives a comprehensive overview of possibilistic information flow properties.

In step four, we must show that no leaks are introduced in the CSPF model instance that do not exist in the SPF model. For this proof, we re-abstract the CSPF model instance, i.e., we hide the characteristics of the used security mechanism. Then, we can prove that the confidentiality requirement holds following the same procedure as for the SPF model instance.

A comprehensive description of this pattern-based approach to confidentiality-preserving refinement can be found in [10].

Conclusions

Confidentiality requirements can be expressed based on communication defined by instances of our model templates. Given an SPF model instance and a corresponding CSPF model instance, we can formally prove a refinement that preserves the confidentiality requirement. The main benefits of this approach are:

– The model templates underlay the (C)SPFs with formal behavior descriptions to gain an unambiguous comprehension of the frames and to clarify their semantics.

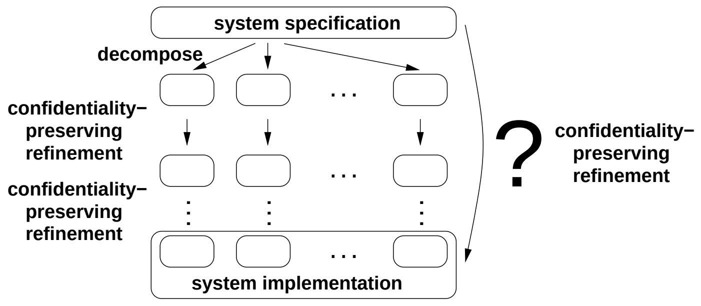

system specification
decompose
confidentiality–preserving refinement
confidentiality–preserving refinement
?
confidentiality–preserving refinement
system implementation

Fig. 3. Compositionality of Confidentiality-Preserving Refinement

- Based on instances of model templates, confidentiality requirements are expressed as information flow properties in a well-defined way.
- As a prerequisite for software development based on stepwise refinement, instances of the model templates enable us to prove that the step from SPF instances to corresponding CSPF instances is a functionally correct refinement, which preserves the confidentiality requirements. The two refinement proofs show if an instantiated CSPF indeed solves a security problem defined by an SPF instance in a given environment.

Following the divide & conquer concept, a software development problem is decomposed into subproblems that are solved independently. Afterwards, the sub-solutions must be composed to obtain a solution for the initial software development problem. As illustrated in Fig. 3, an important question in such a scenario is: Is confidentiality-preserving refinement compositional? In general, it is not compositional. But, we intend to find special software architectures that guarantee compositionality.

References

[1] Information technology – Security techniques – Management of information and communications technology security – Part 1: Concepts and models for information and communications technology security management (ISO/IEC 13335-1:2004), 2004.

[2] D. Hatebur, M. Heisel, and H. Schmidt. A security engineering process based on patterns. In *Proceedings of the International Workshop on Secure Systems Methodologies using Patterns (SPatterns)*, pages 734–738. IEEE Computer Society, 2007.

[3] D. Hatebur, M. Heisel, and H. Schmidt. Analysis and component-based realization of security requirements. In *Proceedings of the International Conference on Availability, Reliability and Security (AReS)*, pages 195–203. IEEE Computer Society, 2008.

[4] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall PTR, 1986. `http://www.usingcsp.com`.

[5] M. Jackson. *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley, 2001.

[6] F. S. E. Limited. Failures-divergence refinement (FDR2) 2.83, 2009. `http://www.fsel.com`.

[7] H. Mantel. *A Uniform Framework for the Formal Specification and Verification of Information Flow Security*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, Juli 2003.

[8] T. Santen. Preservation of probabilistic information flow under refinement. *Information and Computation*, 206(2-4):213–249, April 2008.

[9] T. Santen, M. Heisel, and A. Pfitzmann. Confidentiality-preserving refinement is compositional – sometimes. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS) (LNCS 2502)*, LNCS 2502, pages 194–211. Springer, 2002.

[10] H. Schmidt. Pattern-based confidentiality-preserving refinement. In *Proceedings of the International Symposium on Engineering Secure Software and Systems (ESSoS) (LNCS 5429)*, LNCS 5429, pages 43–59. Springer, 2009.

Finitary Fairness in Event-B

Emil Sekerinski and Tian Zhang

Department of Computing and Software, McMaster University, Canada
{emil, zhangt26}@mcmaster.ca

1 Introduction

In the design of concurrent systems, fairness allows to abstract from scheduling policies of in multi-process systems and from processor speeds in multi-processor systems. In Event-B, like in action systems, the choice among events is nondeterministic [1, 2]. Fairness restricts this nondeterministic. In this paper we propose a way to express fairness in Event-B. Finitary fairness has been proposed as a way of further restricting standard fairness [3]. It is a “more realistic” notion of fairness, it allows some systems to be modelled for which standard fairness is not sufficient, and it is more easily used for proving properties in Event-B than standard fairness. We give a general transformation from an Event-B model, in which some events are marked as fair, into an equivalent plain Event-B model. A theoretical justification is given. A similar transformation was proposed in [3], but does not lead to “equivalent” computations. The contribution of this paper is this new transformation.

2 Motivation

Consider the event system in Fig. 1, which is taken from [3]. Both events L and R have no guards and are thus always enabled. Both events are specified to be fair. A *schedule* of an event system is a sequence of names of events that can occur in an execution (which is going to be made precise shortly). A schedule can be a finite or an infinite sequence; in the example all possible schedules are infinite. For example, a schedule could start with:

$$LRRLRLLLRR\ldots$$

Fairness of L implies that a schedule cannot contain an infinite sequence of R's. A schedule is *bounded* if for some natural number k, no fair event is neglected more than k times consecutively. Finitary fairness of an event system means that all schedules are bounded. For the example, a schedule in which the number of consecutive R's continues to increase is not bounded:

$$LRLRRLRRRLRRRRL\ldots$$

Suppose the events belong to different processes. A *scheduler* is an automaton with event names as the alphabet. For above schedule to be generated by an automaton, the automaton would need to count the number of R's and would need an unboundedly large state. Conversely, if the schedule is bounded, only finite state is needed. Thus the

```
invariants
    $x \in BOOL$
    $y \in \mathbb{N}$
initialisation
    $x, y := TRUE, 0$
fair event L
    $x := NOT\ x$
fair event R
    $y := y + 1$
```

Fig. 1. Event system with two fair events.

bounded schedules are exactly the languages of finite state schedulers. Since any practical scheduler uses a fixed amount of memory, finitary fairness is not only an adequate, but a more precise abstraction from scheduling policies than standard fairness.

Suppose that the events are executed on different processors; the speeds of the processors may differ and may vary. Finitary fairness implies that the speeds of the processors may not drift apart unboundedly. Alur and Henzinger formalize this claim in terms of timed transition systems [3]. Again, finitary fairness allows a more precise abstraction of multiprocessor systems.

Since finitary fairness is more restrictive than standard fairness, one can expect more properties to hold under finitary fairness. For example, the event system of Fig. 1 will eventually reach a state in which $x = TRUE \wedge \neg powerOf2(y)$ holds: if this property would always be false, then L must be scheduled only when $powerOf2(y)$ holds, for increasing values of y, but that is impossible in a bounded schedule.

The finitary restriction can be used for modelling *unknown delays* of timed systems. In a distributed consensus, processes have to agree on a common output value, but each process may fail and not deliver a value. This can be solved using finitary fairness, as shown in [3], but cannot be solved using standard fairness only [4].

Proof rules for the termination of events in presence of fairness can get involved: not all events must decrease the variant. It is sufficient if events that don't decrease the variant keep those fair events that do decrease the variant enabled, as by fairness these will eventually be taken. The proof rule requires that an invariant is specified for each event, e.g. as used in [5] for the refinement of action systems. This would require the proof rules of Event-B to be significantly expanded.

The alternative that we follow is to transform an event system by replacing fair events with regular events and introducing an explicit scheduler [6, 2]. The standard proof rules of Event-B can then be applied. Figure 2 illustrates this. The event system of (a) is supposed to eventually terminate as x is set initially some natural number and fair event R decrements x. In the transformed event system fairness is achieved by introducing a counter c that is decremented each time the (regular) event L is taken. This eventually forces R to be taken as L becomes disabled when c reaches zero. When R is taken, c is set again to a new positive value. In (b) the counter can does not have an upper bound, but still event R will eventually be taken; this ensures standard fairness. In

```
invariants
  $x \in \mathbb{N}$
initialisation
  $x :\in \mathbb{N}$
event $L$
  when
    $x > 0$
  then
    $skip$
  end
fair event $R$
  when
    $x > 0$
  then
    $x := x - 1$
  end
```

(a)

```
invariants
  $x \in \mathbb{N}$
  $c \in \mathbb{N}$
initialisation
  $x :\in \mathbb{N}$
  $c :\in \mathbb{N}_1$
event $L$
  when
    $x > 0$
    $c > 0$
  then
    $c := c - 1$
  end
event $R$
  when
    $x > 0$
  then
    $x := x - 1$
    $c :\in \mathbb{N}_1$
  end
```

(b)

```
invariants
  $x \in \mathbb{N}$
  $c \in 0 .. b$
initialisation
  $x :\in \mathbb{N}$
  $c :\in 1 .. B$
event $L$
  when
    $x > 0$
    $c > 0$
  then
    $c := c - 1$
  end
event $R$
  when
    $x > 0$
  then
    $x := x - 1$
    $c :\in 1 .. B$
  end
```

(c)

Fig. 2. (a) Event system with fair event R. (b) Counter c is used to ensure finitary fairness of R. (c) Counter c is used to ensure standard fairness of R

(c) this counter be at most B, hence B gives an upper bound of how many times event R can be ignored before it must be taken; this ensures finitary fairness.

A further reason for preferring finitary fairness is that it can simplify proofs of termination. For a set of events to terminate, there must exist a variant, a function from the state to a well-founded domain, and all events have to decrease the variant. For proving the termination of the event system in Fig. 2 (c), following variant with natural numbers as the well-founded domain is sufficient:

variant

$$x * (B + 1) + c$$

Event L decreases the variant by decreasing c. Event R decreases the variant by decreasing x; while c may increase, as c is at most B, the variant is still decreased. A similar variant cannot be given for the event system in (b). Natural numbers as the well-founded domain are not sufficient with standard fairness.

3 Fair Event Systems

A *fair event system P* is a structure (Q, E, T, I, F) where

– Q is a set of states,

– E is a set of events,
– T is a set of transitions, relations over $Q \times Q$ indexed by E,
– I is the set of initial states, $I \subseteq Q$,
– F is a set of fair events, $F \subseteq E$

We write $T(e)$ for the transition relation of event e. A *computation* p of P is a finite or infinite maximal sequence of states and events alternating, written

$$p = \sigma_0 \xrightarrow{e_0} \sigma_1 \xrightarrow{e_1} \sigma_2 \xrightarrow{e_2} \cdots$$

such that $\sigma_i \in Q$, $e_i \in E$, $\sigma_0 \in I$, and $\sigma_i \mapsto \sigma_{i+1} \in T(e_i)$. That is, states σ_i and σ_{i+1} must be in relation $T(e_i)$. A computation is a finite sequence, or is *terminating*, if it ends with a state s_n that is not in the domain of any transition relation, $\forall e \in E \cdot s_n \notin dom(T(e))$. Otherwise it is an infinite sequence, or is *nonterminating*.

The *schedule* of a computation p is the projection of the sequence p to only the events; the *trace* of p is the projection of p to only the states, i.e. for p as above:

$$\begin{aligned} schedule(p) &= e_0 e_1 e_2 \ldots \\ trace(p) &= \sigma_0 \sigma_1 \sigma_2 \ldots \end{aligned}$$

We write $schedule_i(p)$ for e_i, the i-th event of computation p and $trace_i(p)$ for σ_i, the i-th state of computation p. The *guard* of an event is the domain of its relation, $grd(e) = dom(T(e))$; an event is *enabled* in a state if the state is in its guard, otherwise *disabled*. A computation p is *bounded* if it is finite or if for some $k \in \mathbb{N}$, for all fair events $e \in F$, event e cannot be enabled for more than k consecutive states without being taken, formally:

$$\forall i \in \mathbb{N} \cdot \exists j \in i \,..\, i + k \cdot schedule_j(p) = e \vee trace_j(p) \notin grd(e)$$

When considering finitary fairness, we are interested only in the bounded computations. This definition of fair event systems generalizes that of transitions systems in [3] by indexing the transitions with the events and by allowing only some events to be fair.

An Event-B model defines the set of states through the variables and invariants, the transition relations through guards and generalized substitutions, and the initial states through the initialization. Thus fair event systems are an abstract representation of Event-B models, in which we additionally allow some events to be specified as fair.

4 The Finitary Weakly Fair Transformation

Let $P = (Q, E, T, I, F)$ be a fair event system. We assume that $E = \{e_1, \ldots, e_n\}$ and that F is the subset $\{e_1, \ldots, e_m\}$ with $m \leq n$. The *finitary weakly fair transformation* $FWF(P)$ ensures finitary fairness by introducing counter variables $c_1, \ldots, c_m$, one for each fair event. The counters indicate the priority of events. Once the counter of an event reaches zero, that event must be tested: if it is enabled, it must be taken, otherwise it is skipped. The counters are kept distinct, therefore only one counter can be zero. The counters are initialized to values between 1 and B. On every transition, the guards of all fair events must be tested: if an event is enabled, its counter must be decreased, otherwise its counter is reset to a value between 1 and B. Formally, $FWF(P) = (Q', E, T', I', \varnothing)$ where for some $B \geq m$:

– $Q' = Q \times \mathbb{N}^m$
– For every event $e_i \in E$, $(\sigma, c_1, \ldots, c_m) \mapsto (\sigma', c'_1, \ldots, c'_m) \in T'(e_i)$ if:
  1. if e_i is a regular event, $e_i \in E - F$, then
     $\sigma \mapsto \sigma' \in T(e_i) \wedge$
     $(\wedge j \in 1 \mathinner{..} m \cdot c_j > 0 \wedge ((\sigma \in grd(e_j) \wedge c'_j = c_j - 1) \vee (\sigma \notin grd(e_j) \wedge c'_j \in 1 \mathinner{..} B)))$
  2. if e_i is a fair event, $e_i \in F$, then
     $\sigma \mapsto \sigma' \in T(e_i) \wedge$
     $(\wedge j \in 1..m-\{i\} \cdot c_j > 0 \wedge ((\sigma \in grd(e_j) \wedge c'_j = c_j - 1) \vee (\sigma \notin grd(e_j) \wedge c'_j \in 1..B)))$
     $\vee$
     $(c_i = 0 \wedge \sigma \notin grd(e_i) \wedge \sigma' = \sigma \wedge c'_i \in 1 \mathinner{..} B)$
  3. $distinct(c'_1, \ldots, c'_n)$
– I' is such that $(\sigma, c_1, \ldots, c_n) \in I'$ if
  1. $\sigma \in I \wedge (\wedge j \in 1 \mathinner{..} m \cdot c_j \in 1 \mathinner{..} B)$
  2. $distinct(c_1, \ldots, c_n)$

All counters of the finitary fair transformation are between 0 and B and are distinct, i.e. for all computations p of $FWF(P)$ and for all natural numbers i with $0 \leq i < |trace(p)|$:

$$trace_i(p) = (\sigma, c_1, \ldots, c_n) \Rightarrow c_1 \in 0 \mathinner{..} B \wedge \ldots \wedge c_n \in 0 \mathinner{..} B \wedge distinct(c_1, \ldots, c_n) \quad (1)$$

This property follows by induction over i: with $FWF(P) = (Q', E, T', I', \varnothing)$ the initial states I' satisfy (1) and transitions T' preserve (1). In the transformation of fair events a case analysis is needed: when the counter of an event reaches zero, the event's transition is take if enabled, otherwise not. This case analysis leads to splitting a fair event E into E and E' in the transformed system. For a fair event system P, we call the schedules of the computations of P simply the schedules of P and the traces of computations of P simply the traces of P. The schedules of P and $FWF(P)$ are necessarily different, as $FWF(P)$ contains the auxiliary primed events. The *restriction* of a sequence s onto a set S is the subsequence of s containing only elements of S. Following theorem justifies the finitary fair transformation.

Theorem 1. *For a fair event system P, the schedules of FWF(P) restricted to the events of P are exactly the bounded schedules of P.*

References

1. Métayer, C., Abrial, J.R., Voisin, L.: Event-B Language, in RODIN Project Deliverable 3.2. (2005)
2. Back, R.J.R.: Refinement calculus, part ii: Parallel and reactive programs. In deBakker, J.W., deRoever, W.P., Rozenberg, G., eds.: REX Workshop on Stepwise Refinement of Distributed Systems - Models, Formalisms, Correctness. Lecture Notes in Computer Science 430, Mook, The Netherlands, Springer Verlag (1989) 67–93
3. Alur, R., Henzinger, T.A.: Finitary fairness. ACM Trans. Program. Lang. Syst. **20**(6) (1998) 1171–1194
4. Fischer, M., Lynch, N., Paterson, M.: Impossibility of distributed consensus with one faulty process. Journal of the ACM **32** (1985) 374–382
5. Back, R., Xu, Q.: Refinement of fair action systems. Acta Informatica **35**(2) (1998) 131–165
6. Apt, K.R., Olderog, E.R.: Proof rules and transformations dealing with fairness. Sci. Comput. Program. **3**(1) (1983) 65–100

Specifying Safety Requirements for a Railway Interlocking System[1]

Colin Snook,

University of Southampton,
Southampton,
U.K.
cfs@ecs.soton.ac.uk

Abstract. We illustrate the use of UML-B to specify safety requirements in a railway interlocking system. Starting from a list of documented hazards, the example uses three refinement levels to concisely specify what is meant by a safe interlocking system. The refinements firstly introduce the basic domain concepts involved in an unsafe railway system, then document assumptions about the system that are relied upon to mitigate hazards, and finally specify the safety requirements that the proposed system must meet in order to avoid hazards. Hence, the model is progressively constrained from an unsafe one to a safe one.

Keywords: Safety Requirements, Refinement, UML-B, Event-B.

1 Introduction

Train Interlocking systems control railway signals and points so that trains travelling upon a rail network are controlled to reach their destinations without collision. The fact that the interlocking system has the responsibility for avoiding collisions means it is a safety-critical control component. It is usual for a hazard analysis to be performed when a safety-critical control component is about to be designed. The purpose of a hazard analysis is to examine all the possible kinds of accidents that could result in injury to humans, analyse the scenarios that could lead to those accidents and thereby identify hazards which must be avoided.

It is possible that some of the identified hazards may be outside the scope of the proposed control component. Others are hazards that the control component must avoid and lead to 'safety requirements'. In the first case it is important to precisely define the limitations, assumptions and scope of the control component and in the second it is important to identify the safety requirements.

In this paper we illustrate how a hazard analysis could be used to drive abstract formal modelling that segregates these two important goals. The model could then be

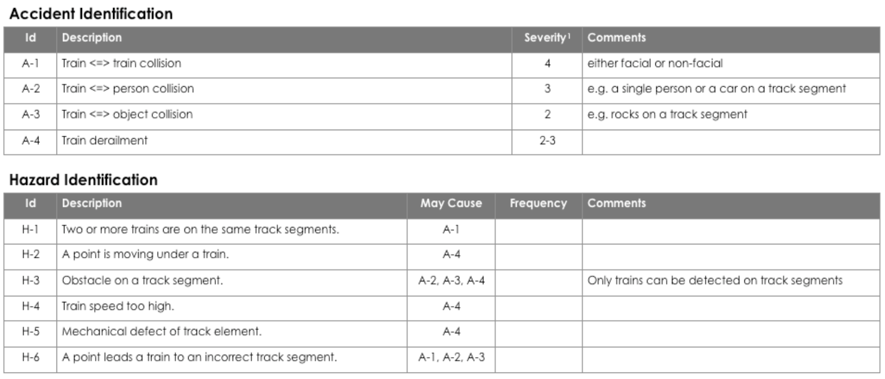
[1] This work is derived from a deliverable of the EU framework 7 project, INESS (INegrated European Signalling System) [No: 218575]. It uses notations and tools developed in the EU framework 7 project, Deploy.

used via refinement to elaborate a specification and design of the control component that is proven to be safe according to the safety requirements.

The model uses the UML-B [1] notation which is a graphical front-end for the Event-B [2] language and tools. The Rodin tool set [2] includes a prover which automatically attempts to prove properties about a model whenever it is saved. The properties that are proved include invariant preservation (which may include the safety invariants we are interested in here as well as simple typing properties) and that a refinement behaves in a way that was permitted by the model it refines. Proofs might not succeed either because the model is incorrect or because the proof is too difficult for the automatic prover. The Rodin toolset includes an interactive prover where the user can attempt to guide the automatic prover.

2 Formal Model of a Safe System

In this section we introduce an example formal model to formalise the top level safety requirements for railway interlocking systems. The formalisation process starts from an existing hazard analysis [3] developed for an interlocking system.



Accident Identification

Id	Description	Severity[1]	Comments
A-1	Train <=> train collision	4	either facial or non-facial
A-2	Train <=> person collision	3	e.g. a single person or a car on a track segment
A-3	Train <=> object collision	2	e.g. rocks on a track segment
A-4	Train derailment	2-3	

Hazard Identification

Id	Description	May Cause	Frequency	Comments
H-1	Two or more trains are on the same track segments.	A-1		
H-2	A point is moving under a train.	A-4		
H-3	Obstacle on a track segment.	A-2, A-3, A-4		Only trains can be detected on track segments
H-4	Train speed too high.	A-4		
H-5	Mechanical defect of track element.	A-4		
H-6	A point leads a train to an incorrect track segment.	A-1, A-2, A-3		

The model consists of the following 3 refinement steps:

- Introduce domain concepts required to describe the 6 hazards above. The idea of this first stage is to describe what would happen if the system was not made to behave safely. If we animated this model we would observe trains crashing and being derailed. The point of this stage is just to introduce the minimal ‘vocabulary’ that will be used in later models;
- Introduce assumptions relating to domain concepts not controlled by the interlocking systems and limitations of the interlocking system. The point of this stage is to explicitly describe the safety properties that won’t be dealt with by the interlocking system. This can include assumptions we make about things outside the control of the interlocking system, for example, the actions of a driver or hazards that the interlocking system cannot address and therefore have to be risked. By modelling these things explicitly we are forced to consider them and allow domain experts the chance to object to them;

- Introduce safety requirements of the interlocking system. This final stage addresses the safety requirements that the interlocking system must satisfy. They are expressed as invariant properties that must always hold and a behavioural model that satisfies these invariants. This behavioural model could be further refined until an interlocking system is represented. The Rodin proof tools will ensure that this interlocking system is a safe system.

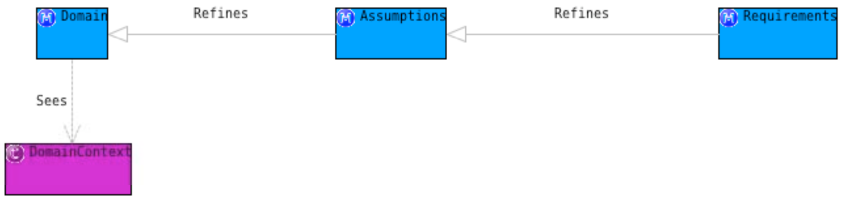


Fig. 1. The components in the model: three refinement steps and a context.

This model is deliberately at a high level of abstraction. It does not describe the functionality of an interlocking system. The intention is to describe just enough detail in order to express the safety hazards that must be avoided. This is done using some abstract state variables and some events that alter those variables. These events are then given guards (conditions which are necessary before the event can occur). In this way, constraints on the behaviour are modelled in a way that is expected to satisfy the safety requirements. The safety requirements are expressed as invariants (properties of the state variables that should hold at all times). It is then automatically proven[2], using the Rodin verification tools, that no sequence of events can reach a state that violates the invariants.

2.1 Domain Concepts

At the first level we model the domain in (only just) sufficient detail to be able to express the hazards. In this level we just introduce the domain concepts (state-space) and do not constrain the system to behave safely.

For H1, we need a set of trains, a set of track segments and a relationship occupies between them. Since implicitly, trains move about on the track, we allow trains to change their occupies value to the next track (next is introduced in H2 below).

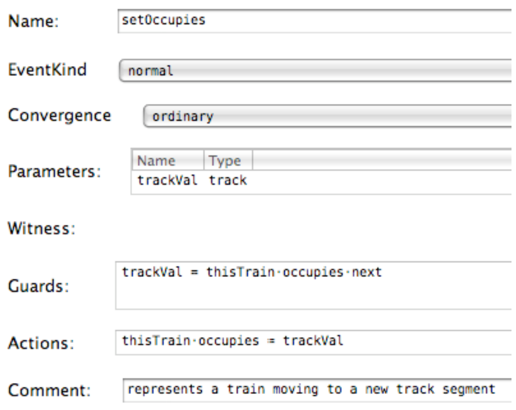
For H2, we need the concept of changing points. We model this by giving track a ‘next’ relationship to another track and allowing it to change. I.e. all track segments are potentially points although, in reality, most never change.

For H3, we give track a Boolean attribute, ‘obstructed’ and allow this to be altered non-deterministically.

For H4, we give ‘train’ a Natural attribute, ‘speed’ and allow this to change non-deterministically. Although not made explicit in the hazard, the concept ‘safe speed’

[2] One proof obligation of the feasibility of an initialisation required a small amount of interactive assistance.

depends on the occupied track segment, so we also introduced a constant attribute ‘safeSpeed’ for the track. This is done in the associated context.



Name:	setOccupies
EventKind	normal
Convergence	ordinary
Parameters:	Name: trackVal, Type: track
Witness:	
Guards:	trackVal = thisTrain·occupies·next
Actions:	thisTrain·occupies ≔ trackVal
Comment:	represents a train moving to a new track segment

Fig. 2. Specification for the event 'setOccupies'. The value of the parameter trackVal is non-deterministically chosen from its type (here ‘track’) but this choice is further constrained by the guards to be the ‘next’ track from the one the train currently occupies. [n.b. EventKind is discussed in a later example. Witness and Convergence concern refinement of parameters and new events, neither of which are used in this document].

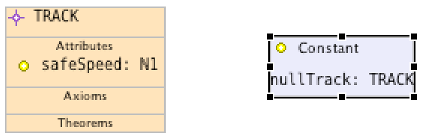


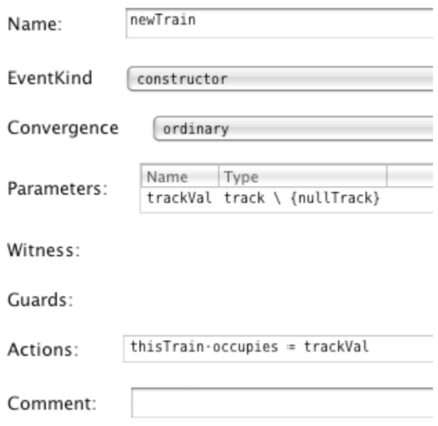
Fig. 3. Domain Context. In UML-B, any explicitly defined sets and constants are specified in a separate component called a context. Here, we just need to specify that there is a set of track sections called ‘TRACK’ and each one has a constant which represents the safe speed for that track section. For modelling convenience we also define special instance of TRACK called ‘nullTrack’ to represent non-existent track.

For H5, we give track a Boolean attribute ‘failed’ and allow this to change non-deterministically.

For H6, we give track a Boolean attribute ‘incorrect’ and allow this to change non-deterministically. Note that we take a slightly different interpretation of ‘incorrect’ than in the hazard analysis. The hazard analysis seems to treat incorrect as a general term including track that is occupied, obstructed or is in use by people or cars. The first two meanings are already covered by H1 and H3 respectively. Since people or cars using the track are either treated as obstructions (H3) or are using a track device that is controlled by the interlocking system. Therefore, we reduce incorrect to mean track sections that have an associated device that is in a state where it cannot be safely occupied. This may be an incorrect interpretation but through discussion with domain experts this could be clarified and fixed if necessary. An important outcome of formalising the safety requirements is that it forces us to confront these ambiguities and encourages them to be clarified.

Note that, for modelling convenience, we treat track as a fixed set of instances with a configuration whereas we treat trains as a variable set of instances so that the train's attributes can be easily initialised to a valid and consistent state, which is refined in later steps.

For this domain model, seventeen proof obligations were generated by the Rodin tools. Of these sixteen were discharged automatically and one required some interactive assistance.



Name:	newTrain
EventKind	constructor
Convergence	ordinary
Parameters:	Name: trackVal, Type: track \ {nullTrack}
Witness:	
Guards:	
Actions:	thisTrain·occupies ≔ trackVal
Comment:	

Fig. 4. Specification of the newTrain event which adds a train to the system. The EventKind is set to 'constructor' to represent this behaviour. A parameter trackVal picks any track segment (other than nullTrack) and this is used to initialise the train's track position.

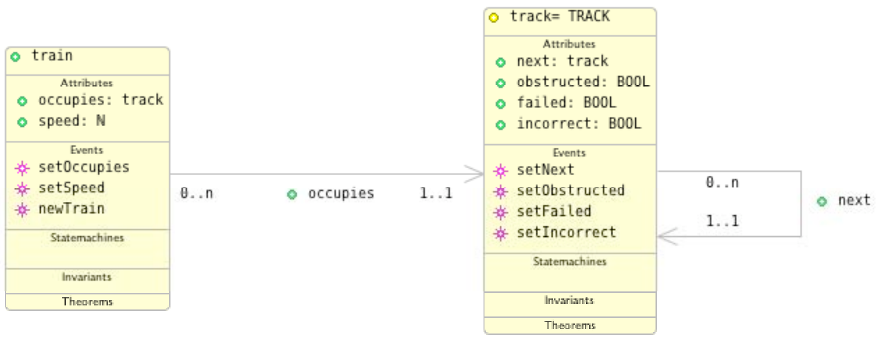


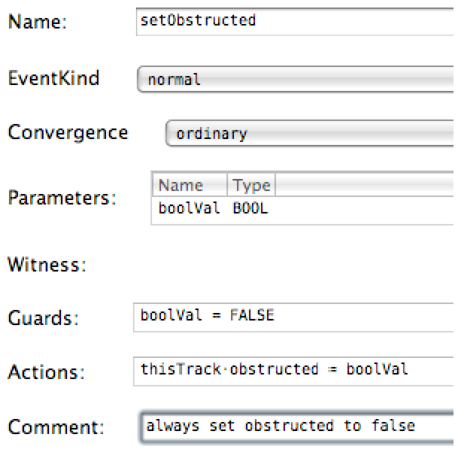
Fig. 5. The Domain model shows the main features needed to consider the given hazards. This consists of the two classes, train and track and their attributes, associations and events. Note that the instances of class track are linked to the set TRACK that was defined in the context.

2.2 Excluded from Scope of Interlocking System

In the next level we introduce assumptions about the things that are not covered by our interlocking system. This includes things that are outside of the scope of interlocking and cannot be influenced by it, as well as limitations of the interlocking system.

2.2.1 Assumptions

For H3, obstacles are not detected by the interlocking system. Therefore we assume that no obstacles will ever be present by introducing a guard into the ‘setObstructed’ event so that ‘obstructed’ is only ever set to FALSE. (In the previous level, boolVal was the non-deterministically selected Boolean parameter representing the new value of obstructed).



Name:	setObstructed
EventKind	normal
Convergence	ordinary
Parameters:	Name: boolVal, Type: BOOL
Witness:	
Guards:	boolVal = FALSE
Actions:	thisTrack·obstructed ≔ boolVal
Comment:	always set obstructed to false

Fig. 6. Specification for the event setObstructed. The guards force the new boolean value of obstructed to always be false.

For H4, speed is outside of the scope of the interlocking system. There is nothing in the interlocking system that can influence the speed of the train and we therefore make the assumption that the driver only selects safe speeds. We implement this assumption by introducing a guard into the setSpeed event that restricts the new speed (natVal) to be less than the safe speed of the currently occupied track.

```
natVal < thisTrain·occupies·safeSpeed
```

We also assume the driver will slow down before moving the train into a new track that has a safe speed limit that is lower than the current train speed. We implement this assumption by introducing a guard into the setOccupies event that prevents the new track segment being occupied unless the train’s current speed is lower than the new track segment’s safe speed.

```
thisTrain·speed < thisTrain·occupies·next·safeSpeed
```

2.2.1 Limitations

For H5, although the interlocking system can prevent a train from moving to a track section that is failed, it cannot prevent failures from occurring. Therefore, it is a limitation of the system that it cannot prevent a track from failing when it is already occupied. It is assumed that this is so unlikely that it can be assumed not to occur. (If necessary, it could be made more unlikely by providing redundant systems). To represent this assumption we introduce a guard into the setFailed event so that track sections cannot fail while they are occupied. Note that the guard only prevents occupied track from failing, it does not prevent failures from being fixed (boolVal = FALSE)[3].

```
thisTrack ∉  ran(occupies) ∨ boolVal = FALSE
```

2.2.2 Invariants

We can now introduce the safety conditions for these hazards as invariants on the properties of a train. (Since they are placed within the class train, they apply to all instances of trains, i.e. universal quantification is implicit for all thisTrain belonging to train).

For H3, we introduce an invariant to say that an occupied track is never obstructed.

```
thisTrain·occupies·obstructed = FALSE
```

For H4, we introduce an invariant to say that a train's speed is always less than the safe speed of the track it occupies.

```
thisTrain·speed < thisTrain·occupies·safeSpeed
```

Initially the Rodin tools generated twelve proof obligations for this stage of the model. At the first attempt, the automatic prover, failed to prove two proof obligations. Examining these two proof obligations (e.g. `obstructed(next(occupies(thisTrain)))=FALSE`) we observed that they both concern the invariant associated with H3. We are fairly confident that this invariant is satisfied since we added a guard to ensure that all track segments are always unobstructed. However, the prover doesn't know this information. We could attempt to assist the prover interactively but in this case it is easier to add an invariant to the model to give the prover the additional information it needs so that it can break the proof in to smaller steps. The prover will first prove that this new invariant is true and then use it to prove H3. After adding this invariant fifteen proof obligations were generated by the Rodin tools but all were discharged automatically.

```
thisTrack·obstructed = FALSE
```

[3] Range (ran) of a function is the set of all target instances that are currently mapped to by the function. Hence ran(occupies) is the set of track segments that are occupied by trains.

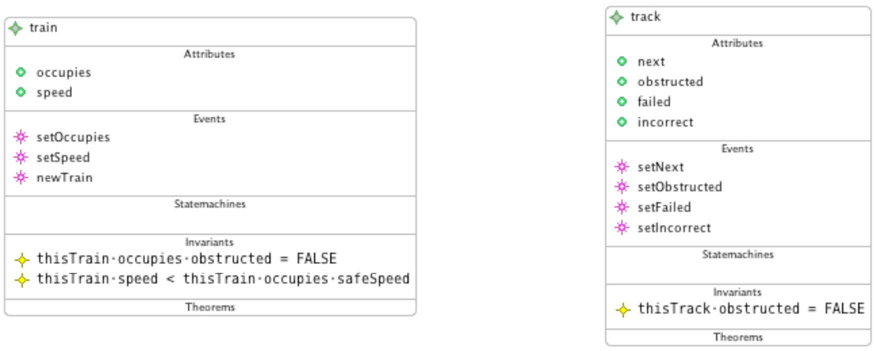


Fig. 7. The refined model consists of the two refined classes with all their previous attributes and associations inherited (i.e. retained) and some new invariants added.

By omission we assume that the remaining events are fully controlled by the interlocking system that is yet to be introduced. More explicitly, the interlocking system has full control over when trains can change their occupancy of track sections, when points can change (the value of next) and when a track section becomes ‘incorrect’.

2.3 Safety Requirements

In this level we introduce the safety requirements for the interlocking system as invariants and then constrain our model with additional guards needed to satisfy them.

For H1, we introduce an invariant to say that occupies is injective. (An injective function is one where at most one element of the domain maps to each element in the range. I.e. only one train occupies a particular track element).

$$\boxed{\texttt{occupies} \in \texttt{train} \rightarrowtail \texttt{track}}$$

For H2, we introduce an invariant to say that the track occupied by a train has the same value for next as it did when the train first occupied the track. Note that, since this is a temporal property, we need to introduce some extra history data, nextWhenOccupies in order to express this as an invariant.

$$\boxed{\texttt{thisTrain} \cdot \texttt{nextWhenOccupies} = \texttt{thisTrain} \cdot \texttt{occupies} \cdot \texttt{next}}$$

For H5, we introduce an invariant to say that the track occupied by a train is not failed.

$$\boxed{\texttt{thisTrain} \cdot \texttt{occupies} \cdot \texttt{failed} = \texttt{FALSE}}$$

For H6, we introduce an invariant to say that the track occupied by a train is not incorrect.

$$\boxed{\texttt{thisTrain} \cdot \texttt{occupies} \cdot \texttt{incorrect} = \texttt{FALSE}}$$

We now need to constrain the behaviour of our model so that it satisfies these invariants.

For H1, we introduce a guard to setOccupies that represents the requirement that the interlocking system will prevent a train from moving to a track section that is already occupied.

```
thisTrain·occupies·next ∉ ran(occupies)
```

For H2, we introduce a guard to setNext that represents the requirement that a point cannot change while it is occupied.

```
thisTrack ∉ ran(occupies)
```

For H5, we introduce a guard to setOccupies that represents the requirement that the interlocking system will prevent a train from moving to a track section that is failed.

```
thisTrain·occupies·next·failed = FALSE
```

For H6, we introduce a guard to setOccupies that represents the requirement that the interlocking system will prevent a train from moving to a track section that is ‘incorrect’.

```
thisTrain·occupies·next·incorrect = FALSE
```

For H6, we also introduce a guard to setIncorrect that represents the requirement that the interlocking system will prevent a track section from becoming incorrect while it is occupied.

```
thisTrack ∉ ran(occupies) ∨ boolVal = FALSE
```

For this stage of the model the Rodin tools generated twenty eight proof obligations, all of which were discharged automatically.

The hazards make no mention of trains failing to stop at the end of a line. Hence we assumed in our model that this is not a safety requirement. Of course, this could be an omission from the hazards but because our model faithfully recreates the given hazards in this respect, none of the verification tools detect that there is a problem. Although the verification tools can detect many internal consistency problems, they cannot decide whether the model is what the users want. Validation of the model could be addressed by animating the model in the presence of domain experts using the Rodin animation tools. (There are two animation plug-ins for Rodin, ProB and AnimB). If the animation reveals that trains are supposed to stop when there is no next track to go to, suitable guards and invariants could be added to prevent this hazard.

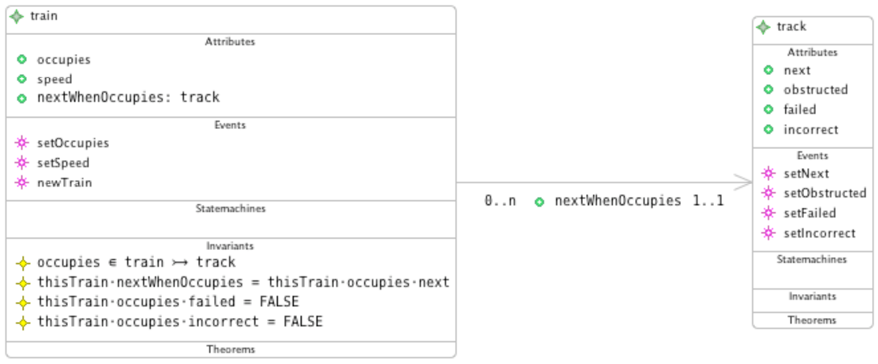


Fig. 8. The final refinement consists of the two refined classes with an additional association to represent the value of next when a track is first occupied by a train. The four safety invariants have been added to the class train.

3 Summary

We briefly summarise what has been achieved with this model.

The model is intended as an illustration and may require revision to make it more realistic. For example the model does not consider the possibility of track being bi-directional. The model illustrates how some example hazards can be used to derive safety requirements. To derive safety requirements it was first necessary to introduce some concepts that are present in the domain. We also made explicit which hazards are not covered by an interlocking system. The four invariants given in Figure 8 are a formalisation of the relevant safety requirements in terms of these domain concepts.

In addition, we introduced some very abstract behaviour and proved that this behaviour satisfies the formalized safety requirements. By modelling the safety requirements and a behaviour that is safe, and proving this to be true, we gain confidence that the safety requirements are consistent. By keeping the model simple and abstract we gain confidence that the safety requirements are correct. We should also validate the model by animation.

Using refinement (according to the Rodin tools upon which UML-B is based) it would be possible to make this abstract model more detailed until it describes an interlocking system. The Rodin tools force us to prove that the more detailed model is a proper refinement of this abstract one and hence that the interlocking system also satisfies these safety requirements.

Acknowledgments. We would like to thank our colleagues in the INESS project who provided valuable feedback on the models and overall approach during its development.

References

1. Snook, C. and Butler, M. : UML-B and Event-B: an integration of languages and tools. In: The IASTED International Conference on Software Engineering - SE2008, 12-14th February 2008, innsbruck, Austria.(2008)
2. Event-B and the Rodin Platform, http://www.event-b.org/index.html Date Last Accessed: 22/09/2009
3. Schacher, M. (2008). Mini-interlocking system - hazard's model and safety requirements. UIC extranet.

Formal Development and Assessment of Dependable Control Systems

Elena Troubitsyna[1]

Åbo Akademi University, Turku, Finland

1 Introduction

Dependability is degree of reliance that can be justifiably placed on a computer-based system [1]. It is a multi-facet system characteristic that encompasses safety, reliability, availability, maintainability and security. Dependability is impaired by faults that might propagate to a system level and result in a system failure. If a failure occurs then the system might cease to provide its services or provide them incorrectly. A set of techniques known as *means for dependability* aims at mitigating consequences of fault occurrence as well as avoiding and removing faults during the system design. Means for dependability include fault avoidance, fault tolerance, fault removal and fault forecasting. We argue that by interfacing refinement process with these techniques we would significantly enhance system development process. Next we present several examples to support our argument.

2 Fault Avoidance and Fault Tolerance via Formal Modelling

Fault avoidance is achieved via an application of rigorous design and verification methods usually called formal methods. Formal methods provide us with powerful mathematical foundations for establishing functional correctness of complex systems. The advances in expressiveness, usability and automation of these techniques enable their use in the design of wide range of complex dependable systems. For instance, the B Method [2] and its extension Event-B [3] have proven their worth in several large industrial projects [4, 5].

Application of formal methods helps us to gain confidence in building correct systems. On the other hand, to guarantee dependability we need to build not only correct but also *fault tolerant* systems, i.e., systems that also are able to cope with faults of their components. Obviously, this goal is attainable only if fault tolerance mechanisms constitute an intrinsic part of system behaviour. In [6] we have proposed a refinement-based approach to development of fault tolerant control systems.

Essentially our approach allows us to express and prove two important facts about system behaviour. We ensure that controlling software, called a controller, 1) does not contribute to failure of the system by forcing it from a safe to an unsafe state and 2) prevents a critical system failure by forcing the system from an unsafe to a safe (but probably non-operational) state. The first goal essentially means that the controller should ensure safety of a fault-free system and not introduce failures into it. The second goal states that the controller should be able to detect failures of other components and cope with them by returning the system to a safe state.

We demonstrated that the development of systems by stepwise refinement in B facilitates achieving these goals. Indeed, since the development starts at a high level of abstraction, the reasoning about safety at such an abstract level allows us to formulate safety properties in a clear and succinct way. The formal development by refinement ensures that the final implementation adheres to the initial abstract specification. This gives us means to guarantee that safety is also preserved at the implementation level. To ensure that safety is not jeopardised in the presence of faults we need to ensure correctness of fault tolerance mechanisms. Although fault tolerance mechanisms usually constitute a significant part of controller (sometimes up to 60% of program code), fault tolerance mechanisms are often introduced only at the implementation stage and in a rather ad-hoc fashion. However, ad-hoc approaches do not guarantee that the system would detect all possible errors and implement adequate error recovery. Hence behaviour in present of faults, error detection and recovery should be considered already at the abstract specification level.

In [7, 8] we demonstrated how to use safety analysis techniques to extract the requirements for error detection and recovery as well as to formulate safety invariant. We proposed an integral approach for incorporating results of Fault Tree Analysis (FTA) and Failure Mode and Effect Analysis (FMEA) into the formal specification. In our approach statecharts facilitate construction of the control system and serve as a basis for structuring and integrating results of FTA and FMEA. The use of statecharts as a communication media between safety and software engineers assists the process of requirements discovery and supports requirements traceability.

Refinement can be seen as a formalized model-driven development approach that allows us to build systems correct and dependable by construction. It can be described as a process of model evolution as follows:

1. Abstract specification of entire system: the initial specification captures requirements for routine control, models failure occurrence and defines safety property as a part of its invariant
2. Specification with refined error detection mechanism: the abstract specification is augmented with the representation of failures of the components, more elaborated description of plant's dynamics and detailed description of error detection.
3. Specification of the system supplemented with redundancy: the specification is refined to describe behaviour of redundant components and control over them. The error detection mechanism is enhanced to distinguish between criticality of failures.
4. Decomposition: the specification of overall system is split into specifications of the controller and the plant.

5. Implementation: executable code of controller is produced.

Every evolution step of our development is a refinement of the initial formal specification in B. We argue that the system developed by an application of the proposed approach have a high degree of dependability because of the underlying complete formal verification within a logically sound framework.

3 Reliability Assessment for Fault Forecasting

While developing system by refinement, we start from a specification that abstracts away from low-level design decisions yet defines the most essential behaviour and properties of the system. While refining the abstract specification, we gradually introduce the desired implementation decisions that initially were modelled non-deterministically. In general, there are might be several ways to resolve non-determinism, i.e., there are might be several possible implementation decisions that adhere to the abstract specification. These alternatives are equivalent from the correctness point of view, i.e., they faithfully implement functional requirements. Yet they might be different from the point of view of non-functional requirements, e.g., reliability, performance etc. Early quantitative assessment of various design alternatives is certainly useful and desirable. However, within the current refinement frameworks there are no scalable solutions for that [9].

In [10] we proposed to integrate probabilistic model checking [11] into stepwise development in Event-B to enable reliability assessment already at the development stage. Reliability is a probability of system to function correctly over a given period of time under a given set of operating conditions [12–14]. Obviously, to assess reliability of various design alternatives, we need to model their behaviour stochastically. We demonstrated how to augment (non-deterministic) Event-B models with probabilistic information and then convert them into the form amenable to probabilistic verification. Reliability is expressed as a property of converted specification that we verify by probabilistic model checking. For instance, reliability assessment can guides a refinement step that aims at introducing a fault tolerance mechanism into the system specification.

To demonstrate validity of our approach, we performed several experiments that aimed at comparing results obtained via model checking with the analytical solutions. Since the obtained results matched, we argue that the proposed approach to assessing reliability can be safely applied in practice.

Our approach can also be used to demonstrate that the system adheres to the desired dependability levels, for instance, by proving statistically that the probability of a catastrophic failure is acceptably low. This application is especially useful for certifying safety-critical systems. Furthermore, reliability assessment allows us to predict time when the system should be automatically or manually reconfigured to ensure that the predefined reliability level is maintained.

Essentially our approach to development and assessment of periodic dependable system can be described by the following guidelines:

1. Abstractly specify system behaviour. To achieve this create an abstract model of system behaviour during a single iteration and then define logical *criterion* that distinguishes correct system behaviour from incorrect one. Strengthen the guards of events to ensure that then *criterion* is not satisfied then system deadlocks
2. Refine system to introduce the required implementation details. If a refinement step introduces representation of unreliable component into the system specification then explicitly model fault-free behaviour as well as faults. At each refinement step reformulate *criterion* in terms of newly introduced variables and functionality. In the invariant explicitly define connection between more abstract and more concrete representation of *criterion* and prove refinement to ensure that refined system does not introduce additional deadlocks.
3. To evaluate various refinement alternatives or impact of unreliable components on the system reliability convert Event-B specification into its PRISM counterpart. Explicitly define synchronising events. Replace non-deterministic representation of behaviour of unreliable components by their probabilistic counterpart. Use component reliability to define the corresponding probability. Evaluate property

   $$\mathbf{P}_{=?}[\mathbf{G} \leq T \ \mathit{criterion}]$$

   to evaluate reliability.
4. Continue refinement process until the desired implementation level is achieved.

4 Conclusions

We see the main benefit of integrating refinement process with the means for dependability in merging reasoning about functional correctness with the explicit reasoning about dependability. It is widely accepted that only dependability-explicit development process can ensure high degree of system dependability and its robustness. We believe that refinement constitutes a suitable basis for conducting system construction hand-in-hand with dependability analysis.

References

1. J.-C. Laprie. "Dependability: Basic Concepts and Terminology," Springer-Verlag, Vienna, 1991.
2. J.-R. Abrial, *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 2005.
3. J.-R. Abrial, "Extending B without changing it (for developing distributed systems)," in *First Conference on the B method*, H. Habiras, Ed. IRIN Institut de recherche en informatique de Nantes, 1996, pp. 169–190.
4. Rigorous Open Development Environment for Complex Systems (RODIN), iST FP6 STREP project, online at http://rodin.cs.ncl.ac.uk/.
5. D. Craigen, S. Gerhart, and T.Ralson, "Case study: Paris metro signaling system," in *IEEE Software*, 1994, pp. 32–35.

6. L. Laibinis and E. Troubitsyna. “Refinement of fault tolerant control systems in B. , ” in *In Proc. of SAFECOMP’2004 – International conference on Computer Safety, Reliability, and Security, LNCS 3219*, M. Heisel, P. Liggesmeyer, S. Wittmann Eds., Springer-Verlag, pp.254-268, Potsdam, Germany, September 2004.
7. E. Troubitsyna. “From System Safety Analysis to Software Specification.,” in *In Proc. of International Workshop on Requirements for High Assurance Systems RHAS’04*. Kyoto, Japan, IEEE Computer Society, pp.41-49, September 2004.
8. E. Troubitsyna. “Elicitation and Specification of Safety Requirements., ” in *Proc. of The Third International Conference on Systems - ICONS’08*. IEEE Computer press. April 2008.
9. A. K. McIver, C. C. Morgan, and E. Troubitsyna, “The probabilistic steam boiler: a case study in probabilistic data refinement,” in *Proc. International Refinement Workshop, ANU, Canberra*, J. Grundy, M. Schwenke, and T. Vickers, Eds. Springer-Verlag, 1998.
10. A. Tarasyuk, E. Troubitsyna, and L. Laibinis, “Reliability assessment in Event-B,” Turku Centre for Computer Science, Tech. Rep. 932, 2009.
11. M. Kwiatkowska, G. Norman, and D. Parker, “Controller dependability analysis by probabilistic model checking,” in *Control Engineering Practice*, 2007, pp. 1427–1434.
12. N. Storey, *Safety-Critical Computer Systems.* Addison-Wesley, 1996.
13. A. Villemeur, *Reliability, Availability, Maintainability and Safety Assessment.* John Wiley & Sons, 1995.
14. P. D. T. O’Connor, *Practical Reliability Engineering, 3rd ed.* John Wiley & Sons, 1995.

Practical Experiences Constructing Working Virtual Machines

Stephen Wright

Department of Computer Science, University of Bristol, UK

stephen.wright@bris.ac.uk

1. Introduction

The Instruction Set Architecture (ISA) of a computing machine is the definition of the binary instructions, registers, and memory space visible to an executing program. MIDAS (Microprocessor Instruction and Data Abstraction System) is an example ISA, intended for formal construction using the Event-B method [4] through to a working Virtual Machine (VM) implemented in software [6,8].

Construction of an ISA capable of executing compiled C programs necessitates an ISA of sufficient size and complexity to provide a useful test article for the tools and techniques needed for formal construction of practical industrial applications. The MIDAS project can therefore inform future development of the Event-B notation and Rodin toolset [1].

2. Motivation and Objectives

The motivation for the use of Formal Methods for construction of the MIDAS ISA was a desire to systematically derive the exact behavior (i.e. modification of machine state) for all possible operations during program execution. As well as identification of state transitions during normal operation, identification of all possible error conditions is of particular importance, as these are rarely explicitly specified in conventional (i.e. informal) specification documents. Explicit specification of all conditions and their resulting behaviors is needed for deterministic behavior in safety-critical systems, defending against program errors introduced by causes such as coding errors, compiler bugs or corruption of hardware at run time.

The MIDAS ISA was developed specifically to be representative of typical microprocessor ISAs, but using a minimal number of defined instructions in order to make a complete refinement practical. Therefore, a goal of creating an ISA capable of executing C programs generated by a suitable compiler was established.

The ISA was also intended as a test article for a wider objective: the construction of a generic model capable of application to multiple ISAs, and its demonstration by the construction of two variants of MIDAS. A final objective was the refinement to a level capable of auto-generation to C source code, for compilation to a working Virtual Machine.

3. The MIDAS ISA

In order to achieve a simple but representative ISA capable of executing compiled programs, the MIDAS ISA has thirty-four instructions falling into eight categories. No-operation (one instruction) performs no function except incrementing the Program Counter (PC). Fetch (six instructions) moves data from the memory system to the register file. Store (four instructions) moves data from the register file to the memory system. Single-operand operations (two instructions) perform transformations of single data elements within the register file. Dual-operand operations (thirteen instructions) perform combinations of two data elements within the register file. Comparisons (five instructions) perform comparisons of two data elements within the register file. Jumps (two instructions) perform conditional modification of the PC. Halt (one instruction) stops machine execution without error.

ISAs with register files implemented as randomly accessible register array machines are now the most common form of hardware-implemented microprocessors, but stack machines are still used, particularly in the field of VMs not intended for hardware implementation. Therefore, in order to provide ISAs representative of both mechanisms and demonstrate the flexibility of the methodology, two variants of the MIDAS ISA exist. In order to simplify development of decoding mechanisms, both MIDAS variants employ the same instruction codes to identify comparable instructions.

4. The MIDAS Model

The Event-B model may be considered to consist of two parts: the generic description of properties common to most ISAs, and a further refinement to an example ISA. The generic part is incrementally constructed from an initial abstraction, sufficiently simple to be trivially understandable and capable of multiple refinements even at an abstract level. The second part constructs the example ISA from this. In order to demonstrate the principle of multiple refinements of a common abstraction within resource limits, the two variants of MIDAS are constructed within the specific part, rather than entirely different ISAs being constructed from the end of the generic path.

The final objective of automatic generation of VM source code was achieved by the specific development of an extension to the Rodin development environment, B2C [7]. All MIDAS source code is available for download at http://deploy-eprints.ecs.soton.ac.uk/84/. This includes Event-B model and refinements, C-coded prototypes, GCC compilers (and their source code) for both MIDAS variants. B2C source code and additional source code for execution of the VMs is also included, as well as installation and build instructions.

The scale of the MIDAS model is worthy of comment. As described, the MIDAS ISAs have thirty four instructions. The construction of the register variant uses 32 steps to construct 109 events in the final refinement, resulting in 4444 lines of automatically generated C. The stack variant uses 33 steps to construct 113 events in the final refinement, resulting in 4092 lines of C. Construction of the generic and variant-specific parts of the model involves the discharging of 4916 proof obligations, approximately half of which required some manual intervention by the developer.

5. The Development Process

Conventional programming techniques were found to provide greater development productivity at the cost of lower development rigor. Therefore, in order to achieve both rapid and robust development, ISAs and their accompanying compilers were prototyped and tested using conventional C programming techniques before formal construction was attempted. This introduced duplication of effort in the ISA implementation, but gave overall improvements to productivity, due to the greater speed of development and retest in the informal environment.

Host C compilation tools were used to compile three separate items: the target compilation tools, the conventionally coded VM prototype code, and the formally derived VM source code. The target compilation tools were then used to compile the VM test executables. The Rodin environment with the B2C translation extension is used to create the formal model and generate the VM source code. The Eclipse Java development environment was used to create the B2C translation extension.

6. Scaling Issues

Development of an Event-B model of this scale revealed scaling limitations within the Rodin tool, falling into three categories: platform, editing and proving. The Rodin tool is an Eclipse [2] based platform, ultimately relying on the Java Virtual Machine for it execution: this was found to have memory limitations when running on the Windows operating system. The form-based editing tool available within the Rodin version used (0.8.2) became unwieldy for large numbers of events, lacking features such as folding, cut-and-paste, and search-and-replace.

Finally, the automatic proving mechanisms were found to break down as the model became large, due to large numbers of hypotheses becoming visible in the proof context, leading to tool timeout during undirected searches for valid proofs. Some timeout failures can be avoided by crude extension of the tool timeout combined with execution of long batch-runs (i.e. approximately ten hours), although timeout extension also serves to further lengthen total batch time due to unsuccessful proof attempts taking longer to abort. Automatic proof is greatly improved by manual inclusion of appropriate hypotheses as theorems in the machine context.

Thus, towards the end of the development process, more time was spent editing and trivial proving than thinking about the model and the discharge of functionally useful proof obligations.

7. Relative Productivity

The simultaneous development of both conventionally coded and formal-derived VM source code suggests that (using available tools) development in a formal environment takes considerably more effort (approximately ten times). This is due to a combination of a superior product being developed by the inclusion of correctness proof, and the available tools for formal development being less mature. This is not to say that formal development is not cost effective, as a payoff realized when a whole production and deployment cycle considered, which includes costly rework and repair. Thus formal development is particularly suitable for safety-critical and high-value applications, in which the cost of these back-end issues is very high, and may be regarded alongside other quantity assurance methods such as reusable libraries, configuration control, static analysis and test suite development.

8. Recommendations

Construction of Event-B models of the scale described can be greatly enhanced by a number of feasible additions to Event-B and Rodin, many of which have been proposed or are already in development.

Editing of models will be greatly enhanced by the Camille editor [3], which emulates the "look and feel" of a text editor, allowing the efficient manipulation of large models. Higher-level tools such as UML-B [5] will provide superior developer interaction with Event-B models, providing efficient visualization of constructs such as state machines. Development of other features is recommended, such as automated refinement, improved differential refinement techniques, syntactic sugar and decomposition of models.

Improvements to current proof discharge techniques will be needed for large models: possible enhancements being provision of programmable proving tactics and thus automatic tactic generation, and Event-B syntax allowing the inclusion of more proof assistance within models.

Finally, establishing good programming practices is a relevant to Event-B as to conventional programming languages, such as planning of refinement structures before the construction of models, grouping of related events, and inclusion of meaningful comments and tags.

9. Conclusions

The formal construction of the MIDAS ISAs demonstrates that Event-B and Rodin can support the refinement of practically useful software, from a trivially simple high-level abstraction to automatically generated source code. However, the model developed probably represents the upper size limit achievable for current Rodin versions, the main issues being concerned with editing and discharge of proofs. However, the project uncovered no conceptual problems, and the flexible architecture of Event-B/Rodin should support a roadmap of solutions for addressing the construction of large formal models and their proofs.

References

1. Abrial,J-R Butler,M Hallerstede,S Voisin,L "An Open Extensible Tool Environment for Event-B", *Formal Methods and Software Engineering*, SpringerLink, 2006
2. Eclipse. Eclipse platform homepage. http://www.eclipse.org/, 2009
3. Event-B.org "Camille Editor" http://wiki.event-b.org/index.php/Text_Editor, 2009
4. Metayer, C Abrial, J-R, Voisin, L "RODIN Deliverable 3.2 Event-B Language", http//Rodin.cs.ncl.ac.uk, 2005
5. Snook,C Butler,M "UML-B: Formal modeling and design aided by UML", *ACM Transactions on Software Engineering and Methodology*, 2006

6. Wright,S "Using EventB to Create a Virtual Machine Instruction Set Architecture", *Abstract State Machines, B and Z*, SpringerLink, 2008
7. Wright,S "Automatic Generation of C from Event-B", *Workshop on Integration of Model-based Formal Methods and Tools,* 2009
8. Wright,S "Using Event-B to Create Instruction Set Architectures", [to appear in] *Formal Aspects of Computing: Applicable Formal Methods*, SpringerLink, 2010