Project DEPLOY
Grant Agreement 214158

*"Industrial deployment of advanced system engineering methods for high productivity and dependability"*

# First Deploy Technical Workshop
**Aix-en-Provence, France, October 21-23, 2009**

http://www.deploy-project.eu

# Preface

The second DEPLOY annual plenary meeting was held from 21st to 23rd of October in Aix-en-Provence, France. The main difference with respect to the first meeting has been a significant amount of time dedicated to technical presentations of papers. We invited all the academic and industrial partners to submit papers about the work they were carrying on inside the DEPLOY project. The accepted submissions have been then organized in five different sessions, each regarding a DEPLOY relevant topic, plus one for short papers. The structure of this document reflects exactly the structure of the workshop, each of the parts represents a workshop session:

1. Event-B and Extensions
2. Code Generation
3. Event-B Metrics and Tools
4. Model Checking
5. Business Information Systems
6. Short Papers

The objective of bringing together both academic and industrial partners to discuss open problems and technical solutions has been successfully reached in a sincerely amicable, open and productive atmosphere. I have seen all the speakers being comfortable during their talks and the audience always attentive and supportive. The offline discussions have been very productive as well. The location was simply fantastic, inspiring and beneficial for everybody's health.

I would like to genuinely thank all the DTW organization for reviewing and helping and I also want to say a warm "thank you" to all the people who participated in the workshop and made it possible. There is no paper that can be great without the curiosity and the genuine enthusiasm of its readers.

January 2010

Manuel Mazzara
Organizing Committee Member
DTW'09

# Organization

## Organizing Committee and Editors

|  |  |
|---|---|
| Michael Jastram | Heinrich-Heine University Düsseldorf, Germany |
| Linas Laibinis | Åbo Akademi University, Finland |
| Felix Lösch | Robert Bosch GmbH, Germany |
| Manuel Mazzara | University of Newcastle, UK |

## Contributors

|  |  |  |
|---|---|---|
| Z. Andrews | A. Iliasov | A. Romanovsky |
| J. Bryans | V. Kozyura | A. Roth |
| J. Bendisposto | L. Laibinis | K. Sere |
| M. Butler | M. Leuschel | R. Silva |
| R. De Landtsheer | M. Mazzara | C. Snook |
| N. Dragoni | A. Michot | A. Tarasyuk |
| F. L. Dotti | C. Pascal | E. Troubitsyna |
| A. Edmunds | M. Pląska |  |
| J. S. Fitzgerald | C. Ponsard |  |

## Reviewers

|  |  |  |
|---|---|---|
| Z. Andrews | F. Degerl | K. Pierce |
| B. Arief | R. De Landtsheer | M. Plaska |
| A. Bhattacharyya | R. Gmehlich | S. Riddle |
| P. Boström | S. Hallerstede | S. Saadaoui |
| J. Bryans | S. Hoang | M. Schmalz |
| J. Coleman | A. Iliasov | A. Tarasyuk |

## Proofreaders

|  |  |  |
|---|---|---|
| Z. Andrews | M. Jastram | M. Mazzucco |
| N. Dragoni | M. Mazzara | C. Succi |

# Table of Contents

# Part I

# Event-B and Extensions

# Towards a Stochastic Event-B for Designing Dependable Systems

Zoe Andrews

School of Computing Science, Newcastle University, UK
Z.H.Andrews@ncl.ac.uk

**Abstract.** Designiging dependable systems is complex and, whilst the state of the art goes some way towards assisting in the design process, existing approaches have some limitations. The benefits and limitations of existing approaches are summarised. Based on these observations, stochastic extensions to the Event-B notation are proposed that build on the strengths of the existing approaches and aim to address some of their limitations. A simple case study is used to demonstrate the use of this new language. The paper concludes with some suggestions for further work in this area.

## 1 Introduction

The range and complexity of systems in which computers play a major part is ever increasing. As complexity increases it becomes harder to design a computer-based system that functions predictably. At the same time, greater reliance is being placed on their correct functioning, in particular in those systems that are responsible for preserving lives or livelihood. Providing the required increases in dependability is only possible if the techniques for obtaining and *assuring* such dependability are continually improved and updated.

There are already many techniques and tools used for modelling and analysing dependable computer systems. However, the existing approaches tend to fall into two distinct research areas, which we shall term: *formal methods* and *dependability*. Formal methods use “mathematically-based languages, techniques, and tools for specifying and verifying computer-based systems” [6]. Formal methods concentrate on the functionality of a program, with the goal being to *prove* that a program is correct with respect to a specification of its functionality. Dependability is defined by Avizienis et. al [4] as “the ability to deliver service that can justifiably be trusted”. As such, dependability research concentrates on evaluating the quality of service aspects of a computing system, such as availability (readiness for correct service) and reliability (continuity of correct service). Such evaluation usually takes place quite late in the development process.

There is currently a gap between these two areas, with a limited body of research that attempts to combine the two ways of thinking. This paper focusses on a promising bridge between formal methods and dependability; that of using *quantitative* formal methods to assist in the design of a dependable system.

In order to analyse the stochastic behaviour of a system a formal language is first needed to express such behaviour. An ideal language would:

- be easy to understand and analyse, especially providing extra insight during the design phase
- allow analysis of a variety of stochastic systems, particularly those with continuous behaviour
- be supported by a useful toolkit
- allow separation of concerns: functional and stochastic behaviour can be analysed separately where appropriate

The contribution of this paper is twofold: first it provides an overview of the state of the art in modelling dependable systems; second it proposes a new approach for modelling dependable systems that addresses limitations of current solutions.

The paper is structured as follows: in Section 2 an overview of the existing approaches for designing dependable stochastic systems is provided; the benefits and limitations of these approaches are discussed. In Section 3 the case study used to demonstrate the stochastic extensions to Event-B is introduced. In Section 4 Event-B is introduced and, based on the findings in Section 2, suitable stochastic extensions to this language are proposed before being applied to the case study. In Section 5 further work to develop the approach (as outlined in Section 4) is considered. Finally, conclusions are presented in Section 6.

## 2 Related Work

There are essentially two main approaches to modelling the stochastic behaviour of dependable systems. These are: Continuous Time Markov Chain (CTMC) based languages; and refinement calculus (proof-based) languages that have been extended with probabilistic choice. These two approaches are introduced below; their benefits and limitations are summarised. Note that a more detailed discussion of the existing approaches is presented in [3].

Based on our observations of the existing approaches, a new approach is proposed to address some of the uncovered limitations. This new approach involves extending Event-B [2, 15] with suitable stochastic behaviour.

The CTMC based formalisms use standard Markov Chain analysis techniques and tend to follow a model checking approach to show that a property holds in a given model. A number of properties of a system can be analysed from a CTMC. Some of these are simply based on the steady state probability distribution – as time approaches infinity a Markov Chain may enter a steady state, i.e. the probability of observing the system in each of its possible states remains constant. Others also rely on *rewards* – a value is assigned to each state and/or state transition and the *expected* value of the reward is calculated.

There are many languages and tools that base their analysis on CTMCs. Some examples include: PRISM [12]; Generalised Stochastic Petri Nets [11]; Stochastic Activity Networks [16]; and PEPA [9]. These approaches tend to be petri net or process algebra based and often use a model checking approach for analysis. The quality of tool support varies widely between the different notations.

PRISM was chosen for further exploration in [3], as an example of a CTMC based approach. The main benefits of PRISM were found to be: the ability to model continuous probability distributions as well as probabilistic choice; good tool support; and the ability to structure models through the use of modules and rewards. The main limitation of PRISM is that all parameters need initialisation, no algebraic analysis is possible – this isn't particularly useful in the early stages of design when values of parameters may still need to be determined. Also, non-determinism (useful for abstraction) cannot be combined with continuous probability distributions. Overall PRISM is a useful tool, but perhaps more beneficial during the later stages of the software lifecycle.

The proof based formalisms use the refinement calculus, extended to include probabilistic choice [13], as their foundation. These notations use a theorem-proving approach to show that certain probabilistic properties hold. Such proofs either use a *refinement* approach, or utilise probabilistic invariants (known as *expectations* – see Section 4.2) to prove the properties of interest.

There are a number of different formalisms that have been extended with probabilistic choice. These include: B (pB) [10]; Action Systems [17]; Hoare logic (pL) [7]; Z [18]; and Event-B [8] (although only for the purposes of qualitative reasoning). Some work has also been done on algebraic transformations of probabilistic specifications [14] to enable simpler reasoning.

Probabilistic B (pB) was explored in more detail in [3]. The main benefit of pB was found to be: the ability to obtain algebraic solutions from the analysis of expectations – allowing the designer to determine the relationship between the different parameters of the model and of the design requirements. The main limitation of pB is that it is not possible to model continuous probability distributions. Attempting to model systems that exhibit continuous behaviour using only discrete probabilistic choice is complex and/or leads to a limited set of properties that can be analysed. Overall probabilistic B is a promising approach to designing dependable systems, but the language needs to include continuous stochastic behaviour to enhance its usefulness.

## 3 Case Study Overview

The case study is a simplified scenario derived from the Deploy project[1]. A vehicle has an emergency brake (EB) that can either be *applied* (the brake is on and the vehicle is stopping) or *not applied* (the brake is off and has no effect on the speed of the car). Some external system (either a person or another computer system) can *command* the brake to be *applied* at any time.

The emergency brake system can fail in two possible ways:

- An *unsafe failure* occurs when the emergency brake has been *commanded*, but not *applied*
- A *safe failure* occurs when the emergency brake is *applied*, even though it has not been *commanded*

The purpose of the case study is to illustrate the use of the proposed stochastic extensions to Event-B, not to provide an accurate representa-

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

tion of a realistic scenario. Thus, for simplicity, recovery from emergency brake requests and failures is ignored.

There are two types of events that can occur in this system. Those that occur in time according to some average transition rate and those that occur instantaneously, triggered by some other state change. An example of the former is the rate at which the emergency brake is requested, events like this are most naturally modelled by the exponential distribution. An example of an instantaneous transition is whether, after an emergency brake request has occurred, the emergency brake is applied or it has an unsafe failure. This type of event is most naturally modelled as a probabilistic choice between the two options, occurring immediately after the state change triggering the choice. Therefore, an intuitive model of this system should include state transitions according to the exponential distribution as well as instantaneous state updates with probabilistic choice.

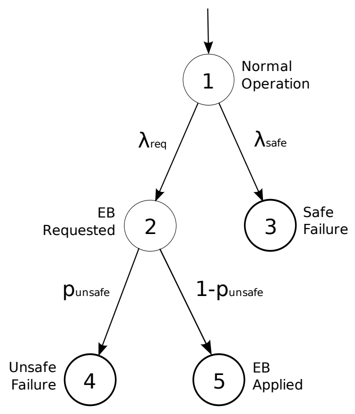


**Fig. 1.** States and transitions for the EB system

Figure 1 shows the transitions that can occur in the emergency brake system considered. State 1 is the initial state in which the EB is not applied and has not been commanded. From the initial state there are two possibilities. An emergency brake request can occur taking the system to state 2 where *EB_command* is set – this modelled according to the exponential distribution with rate $\lambda_{req}$. State 2 is considered to be a transient state from which one of two options will occur instantaneously. The first possible transition from state 2 is that of an unsafe failure (state 4), where the emergency brake is not set – this occurs with some probability, $p_{unsafe}$. The other transition from state 2 is a normal application of the emergency brake (state 5), i.e. *EB_applied* is set – this occurs with probability $1 - p_{unsafe}$. The final transition that should be mentioned is the safe failure transition, which occurs from state 1 and takes the system to the safe failure state (state 3) in which *EB_applied* is set, but *EB_command* is not. This transition is considered to occur according to the exponential distribution with rate $\lambda_{safe}$. There is a safety objective[2] required of the system that "$unsafe\ situation \leq \lambda_{max}/hour$". In this paper the safety objective is interpreted to mean that (on average) less than $\lambda_{max}$ *transitions* into an unsafe situation occur per hour, where an unsafe situation is represented by an unsafe failure.

## 4 Stochastic Event-B

This section describes Event-B and the proposed stochastic extensions to it for modelling dependable systems. An overview of "standard" Event B is given, before describing the proposed stochastic extensions. The emergency brake scenario is used to illustrate the proposed approach and some comments are made on the use of stochastic Event-B.

The proposed stochastic extensions to Event-B aim to build on, and combine, the strengths found in probabilistic B and PRISM. The proposed approach is similar to that of probabilistic B, but also with the inclusion of the exponential distribution for modelling continuous time behaviour. The method of *expectation* analysis (see Section 4.2) is used to prove probabilistic properties of interest for the emergency brake scenario. A key advantage that Event-B has over B is the Rodin platform[3], an open source toolkit that supports the language with various features – including automated and interactive provers. Event-B also seems to be a more natural language for modelling the rate of occurrence of failures, by modelling these as *events*.

### 4.1 Event-B overview

The Event-B formalism is derived from classical B [1], but also incorporates concepts from Action Systems [5]. The semantics of an Event-B

[2] according to railways standard documents such as EN50126, EN50128 and EN50129
[3] http://www.event-b.org/platform.html

model is given as a set of *proof obligations*. Reasoning in Event-B is based on demonic non-determinism.

An Event-B model consists of two types of components: *Machines* and *Contexts*. A *Machine* models the dynamic behaviour of the system such as the *variables* and the *events*. The *Context* provides details of the static information – constant identifiers, values and properties over such values.

The *Machine* description contains the bulk of the model and may include *variables*, *invariants* and *events*. Variables store the state of the machine. Invariants are used to constrain the types of the variables as well as state other logical properties over the variables that must hold at all times. Events define the behaviour of the system, i.e. the state transitions that may occur. Each event may include a *guard*, which defines the states from which the transition can occur, and will include a set *actions* which define the updates to the variables. For more detailed information about the contents of Event-B models the reader is referred to Abrial's forthcoming book [2].

A number of proof obligations are automatically generated for an Event-B model. These proof obligations state the requirements for the model to be internally consistent. For example, events must not invalidate the invariants (known as *invariant preservation*), the proof obligation that states this is as follows:

$$\begin{array}{ll} & I(v) \\ & G(t,v) \\ & S(t,v,v') \\ \vdash & \\ & I(v') \end{array} \tag{1}$$

- $v$ represents the variables of the Machine *before* the event has occurred
- $v'$ represents the variables of the Machine *after* the event has occurred
- $t$ represents the parameters of the event
- $I(v)$ states the invariants hold for the assignment of variables $v$
- $G(t,v)$ states the guard of the event holds for the assignment of variables $v$ and parameters $t$
- $S(t,v,v')$ represents the state transition modelled in the actions of the event

For further details on the proof obligations generated from an Event-B model see Abrial's forthcoming book [2].

## 4.2 Stochastic extensions

As with classical B, standard reasoning in Event-B is based on demonic nondeterminism. Such reasoning is not sufficient for analysing quantitative properties such as reliability and safety. Therefore some extensions to standard Event-B are proposed to support stochastic reasoning for the analysis of such properties.

There has already been a little research on how to add probabilistic choice to Event-B [8]. However, that research focuses on *qualitative* probabilistic reasoning; it is not possible to analyse numerical properties such as the emergency brake's safety requirement using such techniques. To reason about the safety requirement, *quantitative* reasoning is essential. Therefore it is proposed to extend Event-B actions with the probabilistic choice operator, e.g. an action of $x := 1 \;_{p}\!\oplus\; x := 2$ would assign a value of 1 to $x$ with probability $p$ and 2 otherwise (with probability $1-p$).

The ability to model continuous probability distributions is also of importance, in particular the exponential distribution for timing of events. In order to do this it is proposed that an event may have an associated *rate* parameter, which represents the rate of the occurrence of the event with respect to the exponential probability distribution. The stochastic behaviour of an event is assumed to be independent of that occurring in other events and also of previous (historical) behaviours.

For some analysis a designer may be interested in the amount of time that has passed as well as the rate of occurrence of events. Therefore it is also proposed to extend Event-B actions to include an operator for assigning a value to a variable from a continuous probability distribution. For example, an action of $time := time \,+\, \mathbf{exp}(\lambda)$ would increment time by a randomly assigned observation from the exponential distribution with parameter $\lambda$.

In this study *expectations* (analogous to those used in pB [10]) are used to analyse the safety property of the emergency brake system. An expectations clause will be added to Event-B Machines to model these. Expectations are essentially probabilistic versions of invariants. For an invariant $I$, the initialisation would be expected to establish $I$ and all of the events required to maintain $I$. An *expectation*, $E$, works in a similar way except that it is an expression over the real numbers instead of the booleans. An initial value, $e$, for $E$ is also required when working with expectations, this can either be a constant or some expression over the values from the *Context*. The initial value $e$ must be established by the initialisation. The notion of establishing and maintaining an expectation is slightly different from that of invariants due to the use of real numbers. It is required that $e \Rrightarrow [Init]E$, i.e. that at least $e$ is established by the initialisation event, and that $E \Rrightarrow [Op]E$, i.e. that no event can decrease the expected value of $E$. For example consider a system that observes the number of heads ($h$) and tails occurring when a tossing a fair coin $n$ times. An expectation of the system may be $0 \Rrightarrow h - \frac{n}{2}$, meaning that heads account for *at least* half of the total number of observations. Note that a similar expectation about the number of tails observed would also be required to ensure that the coin is *fair*.

## 4.3 Emergency brake models and analysis

This section summarises how Event-B was used to model and analyse the case study. Three different models of the emergency brake are discussed. In a first model the limitations of analysing the scenario in standard Event-B (i.e. without any stochastic extensions) are demonstrated. Afterwards, two different options for analysing the stochastic behaviour of the emergency brake are explored. In option one time is modelled implicitly using a rate parameter. Option two makes use of the statement $time := time + \mathbf{exp}(\lambda)$ to update time explicitly. Note that these two options for modelling stochastic behaviour are semantically equivalent (and as will be seen give the same result on analysis of the expectation). The main differences are in the event and expectation notations, and the amount of flexibility each option provides, these issues are discussed towards the end of this section. Full descriptions of all the Event-B models can be found in [3].

In standard Event-B the closest approximation to the emergency brake scenario involves a non-deterministic choice between the possible events: $EB\_Normal$, $Safe\_Failure$ and $Unsafe\_Failure$. There is no way of stating how often each of these events occur. Similarly, the best approach available for including the safety property is to model it as an invariant $EB\_command = TRUE \Rightarrow EB\_applied = TRUE$. The $Unsafe\_Failure$

event violates this invariant as it sets $EB\_command$ to $TRUE$, but $EB\_applied$ remains $FALSE$. Therefore, it can be concluded from the standard Event-B model that the safety property is not preserved by the $Unsafe\_Failure$ event. However, it is not possible to build an implementation of the emergency brake system in which it can be guaranteed 100% that an unsafe failure will *never* occur. Thus stochastic modelling is needed to establish and minimise the chances of an unsafe situation (and guarantee the stochastic version of the safety property).

For both of the stochastic Event-B options, the standard model above is used as a basis and the $EB\_Normal$ and $Unsafe\_Failure$ events are combined into a single $EB\_Request$ event. This event includes a probabilistic choice statement that results in either the unsafe failure situation (with probability $p$) or the normal application of the emergency brake. Both the $EB\_Request$ event and the $Safe\_Failure$ event are assigned a *rate* value ($\lambda_{req}$ and $\lambda_{safe}$ respectively). These rate values are taken to be parameters of the exponential distribution parameter and model the rate at which the events occur. Note that this representation of the emergency brake scenario is analogous to the natural way of modelling the system, described in Section 3.

```
Event  $EB\_Request\_Option1 \mathrel{\widehat{=}}$
    when
        grd1 : $EB\_applied = FALSE \land EB\_command = FALSE$
    then
        rate : $\lambda_{req}$
        act1 : $(EB\_command, c := TRUE, c+1)\ {}_p\!\oplus$
               $(EB\_applied, EB\_command := TRUE, TRUE)$
        act2 : $n := n+1$
    end

Event  $EB\_Request\_Option2 \mathrel{\widehat{=}}$
    when
        grd1 : $EB\_applied = FALSE \land EB\_command = FALSE$
    then
        rate : $\lambda_{req}$
        act1 : $(EB\_command, c := TRUE, c+1)\ {}_p\!\oplus$
               $(EB\_applied, EB\_command := TRUE, TRUE)$
        act2 : $time := time + \mathbf{exp}(\lambda_{req})$
    end
```

**Fig. 2.** Stochastic Event-B *EB_Request* event for options 1 and 2

For the first stochastic Event-B option considered (option 1), time is treated implicitly through the use of the *rate* parameter (see Figure 2). In order to analyse the stochastic behaviour of the model, *fresh variables* are needed to track the history of the probabilistic choice statement. The total number of times the probabilistic choice is exercised is represented by $n$, and the number of times it resulted in an unsafe failure by $c$. The safety property can then be translated into the following expectation to be analysed:

$$0 \Rrightarrow n \times \lambda_{max} - c \times \lambda_{req} \tag{2}$$

This is interpreted as $n \times \lambda_{max} - c \times \lambda_{req}$ is always at least 0, i.e. that $\frac{c}{n} \times \lambda_{req}$ (the frequency of unsafe failures) always occurs at some maximum rate $\lambda_{max}$. The result of the expectation analysis is presented for the *EB_Request* event, as this event exhibits the most interesting behaviour. The analysis provides a relationship between the parameters of the model and the safety property as follows:

$$p \times \lambda_{req} \leq \lambda_{max} \tag{3}$$

The expectation analysis for the $Safe\_Failure$ event is satisfied trivially as the event does not update any of the variables included in the expectation. Full details of the expectation analysis can be found in [3].

In the second stochastic Event-B option considered (option 2), time is treated explicitly in event actions that update time according to the exponential distribution (see Figure 2). A *fresh variable* is still required for analysis, but this time only $c$ (the number of unsafe failures) is needed as time is being recorded explicitly. The safety property is translated into the following expectation to be analysed:

$$0 \Rrightarrow time \times \lambda_{max} - c \tag{4}$$

This can be read as $time \times \lambda_{max} - c$ is always at least 0, i.e. that $\frac{c}{time}$ (the frequency of unsafe failures) always occurs at some maximum rate $\lambda_{max}$. Analysis of this expectation also gives the relationship found above (Equation 3); this is as expected because the two options are semantically equivalent. As before, the expectation analysis for the $Safe\_Failure$ event is omitted, but is satisfied trivially as this event only increments $time$ (not $c$) and therefore never decreases the value of the expectation. Full details of the expectation analysis can be found in [3].

Option two for modelling stochastic behaviour in Event-B requires additional syntax and semantics to be defined. However, it results in a cleaner formulation of the expectation. The explicit approach could also be used in other situations, for example the error of some measurement may follow a normal distribution.

## 4.4 Experiences with stochastic Event-B

In this section we reflect on the use of stochastic Event-B for modelling dependable systems, particularly in comparison to the existing approaches described in Section 2.

Both options for stochastic Event-B, illustrated by the emergency brake case study above, provide a clean and useful method for combining stochastic and logical reasoning. The use of rates and the exponential distribution allow an intuitive model of the case study, where-as using probabilistic choice alone (as with pB) is rather more complicated. As a consequence the stochastic Event-B models are smaller than those possible in pB, with fewer proof steps required to analyse the expectations. With a simpler notation and shorter proofs, more complex problems should be easier to analyse. Finally, using the expectation approach allows an algebraic analysis of the model. Algebraic solutions reveal the precise relationship between parameters of the model and the stochastic requirements, thus making the impact of design decisions more transparent.

Some valuable lessons were also learnt whilst analysing the emergency brake scenario in the proposed stochastic version of Event-B. Interestingly, it would seem that the way an expectation is formulated has an impact on the algebraic solution obtained. For example, for option 1, an alternative (semantically equivalent) expectation was initially analysed, $0 \Rrightarrow \lambda_{max} - \frac{c}{n} \times \lambda_{req}$, and gave the result $p \leq \frac{c}{n}$. Whilst the two solutions are not contradictory (different variables of the model are referred to in each), the solution presented in Section 4.3 is clearly more useful for finding a suitable design for the system. Therefore the way in which expectations are formulated seems to impact on the usefulness of the results obtained.

Option 2 should also be used with some caution, as a couple of subtle issues were noticed when analysing/implementing such a model. There are rounding issues to consider when obtaining an observation from a continuous probability distribution, i.e. when implementing a statement such as $time := time + \mathbf{exp}(\lambda)$. Also the statement

$$(x := 1 \ {}_p\oplus \ x := 2) \ || \ time := time + \mathbf{exp}(\lambda)$$

would not be equivalent to

$$(x := 1 \ || \ time := time + \mathbf{exp}(\lambda)) \ {}_p\oplus \ (x := 2 \ || \ time := time + \mathbf{exp}(\lambda))$$

in general. This parallel substitution rule is valid for probabilistic programs. However, if there is a stochastic assignment on the right-hand side of $||$ there needs to be some constraint in place to prevent different values being allocated to each instance of the stochastic assignment in the resulting statement. Note that this is not an issue if the analysis only depends on the *expected* value of the observation.

Overall, stochastic Event-B looks like a promising approach to modelling and analysing stochastic systems. Some suggested steps to improve this approach further are discussed in the next section.

## 5 Further Work

The proposed extensions to Event-B for stochastic behaviour are still very early ideas – there are many ways in which this work can be improved and extended.

The reported extensions to Event-B for stochastic reasoning were essentially a feasibility study, therefore the only proof obligation considered was that of *expectation preservation*. In order for the stochastic extensions to be fully integrated into Event-B, further investigation is required to determine fully which proof obligations need to be added or amended. In particular those on refinement, as refining stochastic behaviour is an interesting problem.

An intriguing problem with adding probabilistic choice to a modelling language such as B, is that of the interaction between demonic and probabilistic choice. This issue is discussed for pB in [10]. The author is interested in exploring how demonic choice would interact with continuous probability distributions. What results would be obtained if the model had some events occurring according to the exponential distribution, as well as some events that can occur at any time (controlled by an external (demonic) entity)?

Another possible extension to the proposed approach would be to include other probability distributions, for example the Normal distribution could be useful for modelling measurement errors. Reward structures would also be useful for identifying the costs and benefits of specific states, thus eliminating the need for messy *fresh variables*.

Comprehensive tool support encourages industrial uptake of new methods; therefore an important extension to this work would involve developing a plug-in to the Rodin platform that enables modelling and analysis of stochastic Event-B models.

Finally, more complex case studies would allow further investigation into this promising approach and could lead to more interesting problems to explore.

## 6 Conclusions

The need for providing *quantitative* probabilistic modelling has been demonstrated and existing approaches for doing so explored. The case study highlighted the benefits of the two languages examined in detail: PRISM and pB. However, it also exposed a number of limitations in each. Based on these experiences a new approach has been proposed that aims to address some of the limitations. This approach looks promising, but needs further work as outlined in the previous section. The main findings are summarised below.

The CTMC based approach benefits from: a good range of tools to support modelling and analysis of properties; and the possibility to incorporate both continuous probability distributions and probabilistic choice. However, this approach is aimed at the analysis of systems, not design, so algebraic solutions to a design problem may be impossible to find. The CTMC approach also often suffers from state space explosion problems making it costly (in terms of time and memory usage) to model anything too complex.

The proof-based approach has the benefit of being able to obtain algebraic solutions, making design decisions clearer. However, there is currently no way to model or reason about continuous probability distributions with this approach, which makes it hard to model systems that naturally follow continuous distributions without losing a significant amount of information about their behaviour. Tool support is also very limited for this approach.

The proposed approach of a "stochastic Event-B" builds on the benefits of the proof-based approach, but extends this approach to include continuous probability distributions (in particular the exponential distribution). Applying this approach to the case study gave promising results.

Whilst there are still many issues that need to be addressed, the proposed approach potentially has a lot to offer a designer of dependable systems.

## Acknowledgements

The author is grateful to John Fitzgerald, Thai Son Hoang, Larissa Meinicke and Jeremy Bryans for their valuable discussions on this work. Many useful collaborations would not have been possible without the support of the IST FP7 DEPLOY project. The research was funded by an EPSRC PhD grant.

## References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, New York, 1996.
2. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, to appear 2009.
3. Z. H. Andrews. Towards a stochastic event-b for designing dependable systems. Technical Report CS-TR-1154, Newcastle University, School of Computing Science, July 2009.
4. A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.
5. R. J. 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 *LNCS*, pages 67–93, Mook, The Netherlands, May 1989. Springer-Verlag.
6. E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28:626–643, 1996.
7. R. J. Corin and J. I. den Hartog. A probabilistic Hoare-style logic for game-based cryptographic proofs. In M. Bugliesi, B. Preneel, and V. Sassone, editors, *ICALP 2006 track C, Venice, Italy*, volume 4052 of *Lecture Notes in Computer Science*, pages 252–263, Berlin, July 2006. Springer-Verlag.
8. S. Hallerstede and T. S. Hoang. Qualitative probabilistic modelling in Event-B. In *Proc. 6th International Conference on Integrated Formal Methods (IFM 2007)*, volume 4591 of *LNCS*, pages 293–312, 2007.
9. J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, New York, NY, USA, 1996.
10. T. S. Hoang. *The Development of a Probabilistic B-Method and a Supporting Toolkit*. PhD thesis, School of Computer Science and Engineering, The University of New South Wales, 2005.
11. D. Kartson, G. Balbo, S. Donatelli, G. Franceschinis, and G. Conte. *Modelling with Generalized Stochastic Petri Nets*. John Wiley & Sons, Inc., New York, NY, USA, 1994.

12. M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In T. Field, P. Harrison, J. Bradley, and U. Harder, editors, *Proc. 12th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, volume 2324 of *LNCS*, pages 200–204. Springer, April 2002.
13. A. McIver and C. Morgan. *Abstraction, Refinement And Proof For Probabilistic Systems (Monographs in Computer Science)*. Springer, 2004.
14. L. Meinicke and I.J. Hayes. Algebraic reasoning for probabilistic action systems and while-loops. *Acta Informatica*, 45(5):321–382, 2008.
15. C. Métayer, J.-R. Abrial, and L. Voisin. Event-B Language, RODIN Deliverable D7, 2005.
16. W. H. Sanders and J. F. Meyer. Stochastic activity networks: Formal definitions and concepts. In *Lectures on Formal Methods and Performance Analysis: first EEF/Euro summer school on trends in computer science*, pages 315–343, New York, NY, USA, 2002. Springer-Verlag New York, Inc.
17. E. Troubitsyna. *Stepwise Development of Dependable Systems*. PhD thesis, Åbo Akademi, 2000.
18. N. White. Probabilistic specification and refinement. Master's thesis, Keble College, Oxford, 1996.

# On Event-B and Control Flow

A. Iliasov

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

## 1 Introduction

Event-B [1, 2, 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, 6, 7, 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 [9]. 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: $'stop = \textbf{when}\ \neg(G_1 \vee \cdots \vee G_n)\ \textbf{then skip end}$. 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:* $a \neq b \Rightarrow \neg rdcfl(a,b) \wedge \neg wrcfl(a,b)$

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.

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

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:

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

$p\overline{\|}_E q$ 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 [10] and [11].

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 $T : Event \leftrightarrow S \leftrightarrow S : T \overset{\text{df}}{=} \exists p_e \cdot (G_e(p_e, v) \wedge S_e(p_e, v, v'))$. 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: $seq(\langle\rangle) = id(S)$ and $seq(\langle e \rangle t) = T(e); seq(t)$. From these definitions, the traces of a machine are formulated as all possible traces reachable from the initial machine state *Init*:

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

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,*

$a = \textbf{any}\ p\ \textbf{where}\ G(p,d)\ \textbf{then}\ S(p,d,w')\ \textbf{end}$
$b = \textbf{any}\ q\ \textbf{where}\ H(q,g)\ \textbf{then}\ R(q,g,u')\ \textbf{end}$

*a merged event takes the following general form:*

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

*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* [12].

**Definition 5.** Operator $s;_m t$. *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*

$;: M \times \mathbb{P}(\mathbb{P}(Event)) \times \mathbb{P}(\mathbb{P}(Event)) \rightarrow BOOL$

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

*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):*

$e_1 = \textbf{any}\ p\ \textbf{where}\ G(p,v)\ \textbf{then}\ S(p,v,v')\ \textbf{end}$

$e_2 = \textbf{any}\ q\ \textbf{where}\ H(q,v)\ \textbf{then}\ R(q,v,v')\ \textbf{end}$

*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* $e_1$*:*

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

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

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

*Now we are ready to define the meaning of* $;_m$ *when, as a special case, it is applied to a pair of events:* $e_1 \hat{;}_m e_2 = \mathsf{FIS}("e_1;e_2")$*, 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 ; 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:* $s;_m t = \forall d,f \cdot (d,f) \in s \times t \Rightarrow mergeall(d)\hat{;}_m mergeall(f)$*, where* $mergeall(x)$ *is a following generalisation of merge:*

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

Finally, we are ready to approach the problem of checking flow/machine consistency. Using the ; 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 ; 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* $cons : F \times M \rightarrow BOOL$ *and the definition is as follows:*

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

*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

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

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 $'start;*(e_1|\dots|e_k)$. 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:* $\forall s,t \cdot \{s,t\} \in EN(q) \wedge s \neq t \Rightarrow \neg(EX(p);_m s \wedge EX(p);_m t)$

*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: $EX(fp);_m \{a\} \wedge EX(fp);_m \{b\}$. Let us consider the theorem condition where let $p = fp$ and $\{a,b\} \in EX(q)$. Then $\neg(EX(fp);_m \{a\} \wedge EX(fp);_m \{b\})$. 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[13].

## 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.

axm1 : $I(v) \wedge G(p,v) \vdash \exists v' \cdot S(p,v,v')$
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:

$$\begin{aligned}
open1 &= \textbf{when } d1 = CL \land pr = LOW \textbf{ then } d1 := OP \textbf{ end}\\
close1 &= \textbf{when } d1 = OP \textbf{ then } d1 := CL \textbf{ end}\\
open2 &= \textbf{when } d2 = CL \land pr = HIGH \textbf{ then } d2 := OP \textbf{ end}\\
close2 &= \textbf{when } d2 = OP \textbf{ then } d2 := CL \textbf{ end}\\
pr\_low &= \textbf{when } d1 = CL \land d2 = CL \land pr = HIGH \textbf{ then } pr := LOW \textbf{ end}\\
pr\_high &= \textbf{when } d1 = CL \land d2 = CL \land pr = LOW \textbf{ then } pr := HIGH \textbf{ end}
\end{aligned}$$

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 ($pr\_low; open1$, $open1; close1$ and so on) it is needed to show that condition (**??**) holds. For example, for $pr\_low; open1$ it is:

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

The condition is trivially true. Another proof obligation, generated from $open1; close1$, also trivially holds: $d1 = CL \land pr = LOW \land pr' = pr \land d1' = OP \land d2' = d2 \vdash d1' = OP$

The next case presents some difficulties. When trying to demonstrate that event $close1$ always enables $pr\_high$ we find that there is not enough information to discharge the proof obligation:

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

The problem here is that the guard of event $close1$, although strong enough to satisfy safety properties, is too weak for the flow. By strengthening the guard with the additional clauses $d2 = CL \land pr = LOW$ 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 $close1$ is always preceded by $pr\_low$ and thus may only be enabled when $pr = LOW$. Likewise, since $close1$ always follows $open1$ and the second door is always closed in the after-states of $open1$ (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 $close1$ 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 $H_i(v_1, \ldots, v_n, v)$ 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
$H_{i+1} = H_i(v_1, \ldots, v_n, v_{n+1}) \land G(v_{n+1}) \land S(v_{n+1}, v)$
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 [9] 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:

| *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

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

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 [11].

## 3.7 Reasoning about Flows

A flow expression may be seen as a directed graph. Its vertices represent model events and edges correspond to 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 [13].

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 [14] and refinement calculus [15]. 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 [13]. 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 [11] failure-divergence semantics for event-based systems discussing the integration of state-based and process-based formalisms [16, 17, 18, 8, 19]. 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 [17] while the CSP style message passing was used to compose Event-B machines[12].

## 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. C. A. R. Hoare, "Communicating Sequential Processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.
10. M. Butler, "A CSP Approach to Action Systems. phd thesis." 1992.
11. C. Morgan, "Of wp and CSP," pp. 319–326, 1990.
12. M. Butler, "Decomposition Structures for Event-B," in *Integrated Formal Methods iFM2009, Springer, LNCS 5423*, vol. LNCS, no. 5423. Springer, February 2009.
13. "Event-B and RODIN Platform," http://www.event-b.org, 2004.
14. C. B. Jones, *Systematic software development using VDM*. Prentice Hall International (UK) Ltd., 1986.
15. R.-J. J. Back and J. V. Wright, *Refinement Calculus: A Systematic Introduction*. Springer-Verlag New York, Inc., 1998.
16. 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.
17. 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.
18. S. Schneider and H. Treharne, "Verifying Controlled Components," in *In Proc. IFM*. Springer, 2004, pp. 87–107.
19. 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.

# Structuring Specifications with Modes

Alexei Iliasov, Fernando Luís Dotti, Alexander Romanovsky

Centre for Software Reliability, Newcastle University, UK

**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.

## 1 Introduction

Systems are dependable if they deliver service that can be justifiably trusted [15]. 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 [9], 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 [21].

Several authors have investigated fault-tolerance modelling using different specification formalisms and verification approaches (e.g. [20, 12]). 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 is 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 tolerance mechanisms; and (iii) system properties: the statement of system properties under different working conditions (addressing fault assumptions) should be supported.

In this paper the concept of ‘operation mode’ [13] is revi-sited. We use modes to structure system specification to facilitate rigorous design and to integrate fault tolerance. Since modes appear in different types of systems, such as real-time [13], avionic and space [8, 18], the approach is useful for building wide classes of dependable systems.

We use term mode in the same sense as [13]: both as partitions of the state space, representing different working conditions of the system, and as a way to define control information, structuring system operation. In Section 2, modes are defined to allow the modeller to state the property that must be respected, called guarantee, in each working system condition, called assumption. In Section 3, 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 5. Mode refinement is performed hand in hand with the refinement of the respective formal model and allow layered definition and reasoning about properties. This helps to trace properties to requirements. Refinement also offers a strategy to obtain a correct implementation from the formal model. Theorem proving strategies and tools sometimes offer an attractive option to model-checking as they avoid the state-space problem. Section 4 discusses the use of modes in the design of fault-tolerant systems. Section 6 exemplifies the ideas with the model of a cruise control system. Related work and conclusions are presented in Sections 7 and 8.

## 2 Operation Modes

Operation modes help to reason about system behaviour by focusing on the 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 thus 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. 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 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

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

and thus transition $\top \rightsquigarrow \bot$ is not possible. There can be any number of initiating and terminating mode transitions.

There are restrictions on the way mode assumptions and guarantees are formulated. The states described by a guarantee must be wholly included into valid model states:

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

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:

$$\begin{array}{ccc} A_1/G_1, & \ldots & A_n/G_n \\ i \rightsquigarrow j, & \ldots & k \rightsquigarrow l \end{array} \tag{4}$$

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:

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

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.

## 3 Mode Refinement

Refinement is formal technique for transitioning from an abstract model to a concrete one [7]. 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].

[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.

Refinement is a partial order relation on model universe. This relation is denoted as $\sqsubseteq$ and 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 formal in a sense that a refined model may be used in place of its abstraction. A number of refinement techniques can be used.

*Data Refinement* With data refinement, data types are changed and data structures are introduced. 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 use of variables $v$ in new invariant $J$ allows modeller to expresses a linking relation between the state of concrete and abstract models.

*Behavioural Refinement* Behaviour refinement details the mode view on a system. System behaviour becomes more deterministic and also described in a finer level of details. 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 it is understood as a contract of a mode with the rest of a system and its environment. Weakening a mode guarantee could violate expectations of another system part. The following condition summarises this refinement rule:

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

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:

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

*Superposition Refinement* Sometimes it is needed to add new modes without splitting an existing abstract mode. Through superposition refinement it is possible to refine an implicit skip mode *false*/*true*. This is the weakest form of a mode and it can be refined into any other mode.

*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.

*Visual Notation* To assist in application of the approach, a visual notation loosely based on Modechards [13] is proposed. A mode is represented by a box with name; a mode transition is an arrow connecting the previous and next modes. Special modes $\top$ and $\perp$ are omitted so that initiating and terminating transitions appear to be connected with a single mode. Refinement is expressed by nesting boxes. Figure 2 exemplifies this. A transition from an abstract mode is equivalent to having transitions from each of the concrete modes, e.g. transition $ccOff$ from abstract mode Cruise Control in diagram (C) of Figure 2.

## 4 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.

In general, 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 mode system may assert that the recovery procedure: (i) successfully recovers the state and thus switches back to normal mode to proceed execution (Figure 1(B) or (C)); (ii) provides degraded service in cases where full functionality is not recoverable (Figure 1(D)); (iii) fails to recover, in which case measures to stop safely may be taken (Figure 1(A) and part of (D)).

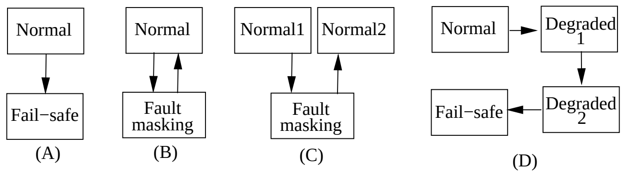


**Fig. 1.** Modes for fault tolerance.

# 5 Operation Modes for Event-B

The operation modes method is not intended to be used as a modelling method on its own. The schematic nature of the approach makes it it well suited to integration with an existing formalism. In this section we discuss how modes can be used with a well known formalism: Event-B. 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 [2] and Action Systems [5]. 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 [1]. 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:

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

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. 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 [10]. 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: 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:

$$\begin{array}{ll} A_1/G_1 \mapsto E_1, \quad \dots & A_n/G_n \mapsto E_n \\ (i \rightsquigarrow j) \mapsto E_p \quad \dots & (k \rightsquigarrow l) \mapsto E_q \end{array} \tag{10}$$

Event sets $E_1, \dots, E_n$ may overlap but may not be identical. In the latter case they specify the same mode. 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 $(i \rightsquigarrow j) \mapsto E_p$ it is required to demonstrate the following:

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

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 ($e \in E_i$):

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

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 2. The rule above gives a rise to a number of conditions on Event-B. Firstly, all the events of a mode must satisfy its guarantee provided the 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 Event-B development method offers a gradual, refinement-based, model detailing. To refine model $M$ one constructs a new model $M'$ such 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. 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:

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

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

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

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.

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

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

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

Conditions 17 and 18 state how mode refinement is related to Event-B 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* 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. 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.

## 6 Cruise Control Case Study

A simplified version of one of the DEPLOY case studies [4] 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 a predefined speed. Due to the nature of the system, 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. Figure 2 presents the diagrams with the sequence of refinements.

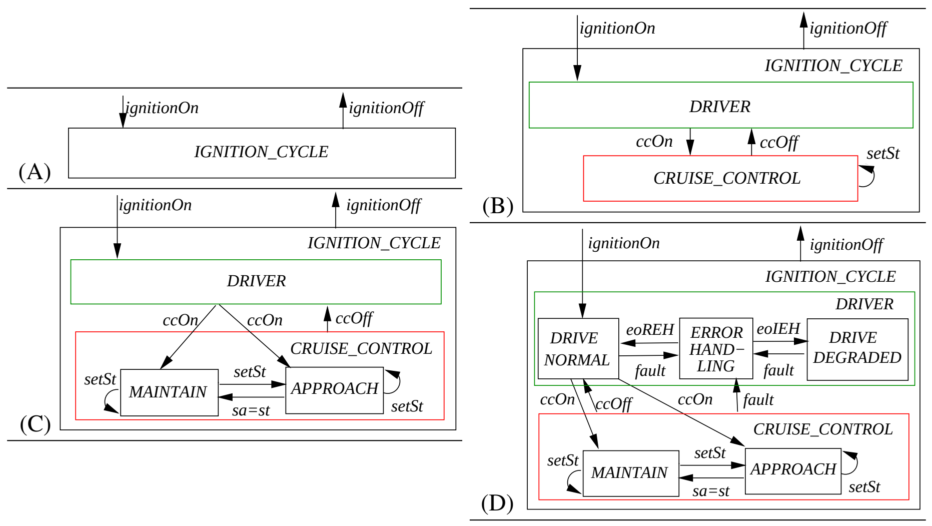


**Fig. 2.** Mode refinement sequence for the Cruise Control System.

*(A) IGNITION_CYCLE* Figure 2(A) presents the diagram of the most abstract model for the system. It is composed only by the *IGNITION_CYCLE* mode and represents the activity from the instant the ignition is turned on, event *ignitionOn* establishes the assumption for that mode, to the instant it is turned off, event $ignitionOff$ changes the conditions of the system and falsify the assumptions for this mode. The 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 *maxSpeedV*. No memory is retained about the states in the previous ignition cycle. Initially, the current speed is zero and ignition is off: $sa \in 0 \land ig \in FALSE$. 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 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 |
| | $ig = true$ | $(sa < speedLimit) \land$ $(\ \lvert sa' - sa\rvert < maxSpeedV\ )$ |

*(B) DRIVER and CRUISE_CONTROL* 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 2(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: $st \in \mathbb{N} \land cc \in FALSE$. The description of the modes:

| mode | assumption | guarantee |
|---|---|---|
| *DRIVER* | ignition cycle assumptions and cruise control off | ignition cycle guarantees |
| | $ig = true \land$ $cc = false$ | $(sa < speedLimit) \land$ $(\lvert sa' - sa\rvert < maxSpeedV)$ |
| *CRUISE_ CONTROL* | ignition cycle assumptions and cruise control on | ignition cycle guarantees and maintain target speed or approach target speed |
| | $ig = true \land$ $cc = true$ | $(sa < speedLimit) \land$ $(\lvert sa' - sa\rvert < maxSpeedV)$ $\land(\ \lvert sa' - st'\rvert \le isp\ \lor$ $\lvert sa' - st'\rvert < \lvert sa - st\rvert\ )$ |

*(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*. Respective assumptions and guarantees are described in the table below. Figure 2(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 modes the cruise control can be switched off and control returned to the driver.

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

| mode | assumption | guarantee |
|---|---|---|
| *MAINTAIN* | cruise control assumptions and speed close to target | cruise control guarantees and maintain target speed |
| | $ig = true \wedge$ $cc = true \wedge$ $\lvert sa' - st' \rvert \leq isp$ | $(sa < speedLimit) \wedge$ $( \lvert sa' - sa \rvert <$ $maxSpeedV\ ) \wedge$ $(\lvert sa' - st' \rvert \leq isp)$ |

***(D) Error handling*** at any time failures of the surrounding components (e.g. airbag activated, low energy in battery) may affect the cruise control system. These faults are signaled as erroneous conditions and can be either reversible or irreversible: the reversible errors result 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.

| mode | assumption | guarantee |
|---|---|---|
| *DRIVE_ NORMAL* | driver assumptions and no error | driver guarantees and cruise control available |
| | $ig = true \wedge$ $cc = false \wedge$ $error = false$ | $(sa < speedLimit) \wedge$ $( \lvert sa' - sa \rvert <$ $maxSpeedV\ )$ |
| *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 |
| | $ig = true \wedge$ $cc = false \wedge$ $error = true \wedge$ $eHand = true$ | $(sa < speedLimit) \wedge$ $( \lvert sa' - sa \rvert <$ $maxSpeedV\ )$ |
| *DRIVE_ DEGRA- DED* | driver assumptions and error and handling finished | driver guarantees and cruise control not available |
| | $ig = true \wedge$ $cc = false \wedge$ $error = true \wedge$ $eHand = false$ | $(sa < speedLimit) \wedge$ $( \lvert sa' - sa \rvert <$ $maxSpeedV\ ) \wedge$ |

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 1. Figure 2(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.

## 7 Related Work

Several applications, structured in modes, can be found in the literature. Papers [8] and [18] show how to formally model and analyse modal space and avionic systems. In [17] an Automated Highway System is extended to tolerate several kinds of faults, modes are used to characterize degraded operation. A classic case study on formal methods, the Steam Boiler Control [3], is based on operation modes. More recent examples on the use of modes for the specification of airspace, transportation and automotive systems can be found in [4]. Such contributions focus on specific applications and not on general means to model and reason using modes.

In [22] the authors discuss characteristics of mode-driven distributed applications and an infrastructure is proposed to support mode-driven fault tolerance in run time. In [19], 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 [13] a specification language for real-time systems, called Modechart, is presented. In [11] 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 [6], 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, the flexibility of allowing different modes and mode switching, is not offered.

Finally, Jones, Hayes and Jackson, in [14], discuss a method 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.

## 8 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 [16], 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. 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).
2. J. R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 2005.
3. Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors. *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control*, volume 1165 of *Lecture Notes in Computer Science*. Springer, 1996.
4. Jean-Raymond Abrial, Jeremy Bryans, Michael Butler, Jerome Falampin, Thai Son Hoang, Dubavka Ilic, Timo Latvala, Christine Rossa, Andreas Roth, and Kimmo Varpaaniemi. Report on knowledge transfer - deploy deliverable d5, February 2009.
5. Ralph-Johan Back and Kaisa Sere. Stepwise Refinement of Action Systems. In Jan L. A. van de Snepscheut, editor, *Proceedings of the International Conference on Mathematics of Program Construction, 375th Anniv. of the Groningen Univ.*, pages 115–138, London, UK, 1989. Springer-Verlag.
6. Ralph-Johan Back and Joakim von Wright. Compositional action system refinement. *Formal Asp. Comput.*, 15(2-3):103–117, 2003.
7. Ralph-Johan J. Back and J. Von Wright. *Refinement Calculus: A Systematic Introduction*. Springer NY, Inc., 1998.
8. Ricky W. Butler. An introduction to requirements capture using pvs: Specification of a simple autopilot. Technical Report 110225, NASA, 1996.
9. Flaviu Cristian. *Exception handling*. Blackwell Scientific Publications, 1989.
10. E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
11. Gerhard 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*, pages 287–300. Springer Verlag, 1992.
12. F. C. Gärtner. Transformational approaches to the specification and verification of fault-tolerant systems: formal background and classification. *Journal of Univ. Computer Science*, 5(10):668–692, 1999.
13. F. Jahanian and A.K. Mok. Modechart: A specification language for real-time systems. *IEEE Transactions on Software Engineering*, 20(12):933–947, 1994.
14. Cliff B. Jones, Ian J. Hayes, and Michael A. Jackson. Deriving specifications for systems that are connected to the physical world. In *Formal Methods and Hybrid Real-Time Systems*, pages 364–390, 2007.
15. Jean-Claude Laprie, Brian Randell, Algirdas Avizienis, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, 2004.
16. P. A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1990.
17. John Lygeros, Datta N. Godbole, and Mireille E. Broucke. Design of an extended architecture for degraded modes of operation of ivhs. In *In American Control Conf.*, pages 3592–3596, 1995.

18. Steven 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*, pages 44–53, New York, USA, 1998. ACM.
19. Sadaf Mustafiz, Jörg Kienzle, and Andrey 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*, pages 19–28, New York, NY, USA, 2008. ACM.
20. Jan Peleska. Formal methods and the development of dependable systems - habilitationsschrift. Technical Report 9612, Institut für Informatik und Praktische Mathematik der Christian-Albrechts-Universität su Kiel, 1996.
21. Alexander Romanovsky. A looming fault tolerance software crisis? *SIGSOFT Softw. Eng. Notes*, 32(2):1–4, 2007.
22. Deepti Srivastava and Priya Narasimhan. Architectural support for mode-driven fault tolerance in distributed applications. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, 2005.

# Part II

# Code Generation

# A Code Generation Example for Event-B: A Shared Channel with Concurrent Read/Writers

Andrew Edmunds and Michael Butler

Southampton University, UK

**Abstract.** The Event-B method is a formal approach to modelling systems, using refinement. Initial specification is done at a high level of abstraction; detail is added in refinement steps as the development proceeds toward implementation. In previous work we developed an approach to bridge the gap between abstract specifications and implementations using an implementation level specification notation that we call OCB. We present here an example abstract development of processes reading and writing via a shared buffer. We show how, with the help of a diagrammatic representation, we transition between the abstract model and the implementation specification. The implementation specification has Event-B semantics and is translated to an Event-B model and Java source code.

## 1 Introduction

The Event-B method [2] is a formal approach to modelling systems, with tool support [3]. The modelling approach uses an event based view of how a system evolves atomically, from one state to another. The Event-B approach has evolved from classical-B [1] which was targeted more specifically at modelling software systems. In Event-B a system's state is modelled using sets, constants and variables; and updates to state are described in the bodies of guarded events. System properties are specified in invariants and proof obligations are generated, which should be discharged in order to prove that the event actions do not violate the invariants. The Event-B approach uses refinement to link an abstract model with successively more concrete models, and a linking invariant relates the state of the concrete model with its abstract counterpart. When modelling a software system, an Event-B model will be refined to a point where we are ready to provide information about the implementation. Consideration is given to how tasks may be performed by executing processes, and how the processes may interleave. In our paper [8] we introduce an intermediate specification language, Object-oriented, Concurrent-B (OCB), which we use to link Event-B models and object-oriented implementations. The new notation sits at the interface between the two technologies, and we incorporate aspects of both. We use Java [10] as the target implementation language since it is often used to implement concurrent systems; however our work is not limited to this target in principle.

The example that we present here involves processes reading and writing to a shared channel. A channel may have at most one reader reading, and at most one writer writing at any one time; however a number of processes may be waiting to read from, or write to, the channel. In our most abstract model data is transferred as a block in a single atomic step. A write event constitutes moving a block from a writing process to a channel buffer; and a read event constitutes moving a block from a channel buffer to a reader. The atomicity of the read and write activity is altered in the refinement - we introduce blocks that are made up of packets, and each packet is written to the channel individually. This allows the reader to begin reading as soon as there is data in the channel - without the writer having to complete the data transfer. We continue with an overview of our implementation notation (OCB), and show the implementation level specification for the concurrent read/writers example. We then present details of the implementation level refinement that results from the translation of the OCB model into Event-B, and also show details of the source code generated by the Java translator.

## 2 The Abstract Event-B Development

In our first model we introduce processes, channels and data. We define carrier sets for the set of processes *PROCESS*, the set of channels *CHANNEL*, and set of data blocks represented by *Block*. A block of data is a function of packet identifiers to data, $Block = PKTID \rightarrow\!\!\!\!\!+ DATA$. Process objects are represented by a variable *proc*, and channels are represented by a variable *chan*. Each process has a local buffer called $buff$, and channels hold data in a buffer called $data$.

```
INVARIANTS
    $proc \subseteq PROCESS$
    $chan \subseteq CHANNEL$
    $data \in chan \rightarrow Block$
    $buff \in proc \rightarrow Block$
```

The *Write* event models the writing of a block of data $b$ to a channel. The parameters include the writing process $p$, and the target channel $c$. In addition to these parameters an additional local parameter $b$ is introduced to keep track of the block of data to be written from the local buffer, where $b = buff(p)$.

```
$Write \triangleq$
  ANY $p,\ c,\ b$
  WHERE $p \in proc\ \wedge\ c \in chan\ \wedge b = buff(p)\ \wedge$
    $buff(p) \neq \varnothing\ \wedge\ data(c) = \varnothing$
  THEN $data(c) := b\ \|\ buff(p) := \varnothing$
  END
```

The block $b$ is copied to the data buffer $data(c)$ and the local buffer $buff(p)$ is emptied.

The event to read a block from the channel is parameterized by a reading process $p$, and channel $c$.

```
$Read \triangleq$
  ANY $p,\ c,\ b$
  WHERE $p \in proc\ \wedge\ c \in chan\ \wedge\ b = data(c)$
    $buff(p) = \varnothing\ \wedge\ data(c) \neq \varnothing$
  THEN $data(c) := \varnothing\ \|\ buff(p) := b$
  END
```

The block in the channel buffer $b$ is copied to the local buffer $buff(p)$ and the channel buffer $data(c)$ is emptied.

## 3 Refinement with Packetized Data

In the first refinement we introduce writing behaviour which is performed in three steps by the $StartWrite$, $WritePacket$, and $EndWrite$ events. Similarly $StartRead$, $ReadPacket$ and $EndRead$ perform reading. We can make use of a graphical representation, introduced in [4], called Event Refinement Diagrams. These diagrams are based on Jackson Structure Diagrams, and are used to clarify the relationships between events of the abstraction and refinement. It should be noted though that the diagrams are an informal representation of the relationship between abstract events and events of refinements. The most abstract specification appears uppermost in the diagram with more concrete representations below. At each level the order of events is read from left to right, indicating the sequence in which the events are required to occur. The solid lines connecting events represent event refinement. Dashed lines represent events that refine skip, and add behaviour related to the abstract event. The '*' annotation indicates that iteration of a particular event is possible. We indicate the parameter names of each event in the form of an event signature. In Figures 1 and 2, $k^*$ indicates that the number of iterations is determined by a guard involving parameter $k$. The diagrams show abstract $Read$ and $Write$ events being refined by a number of events.

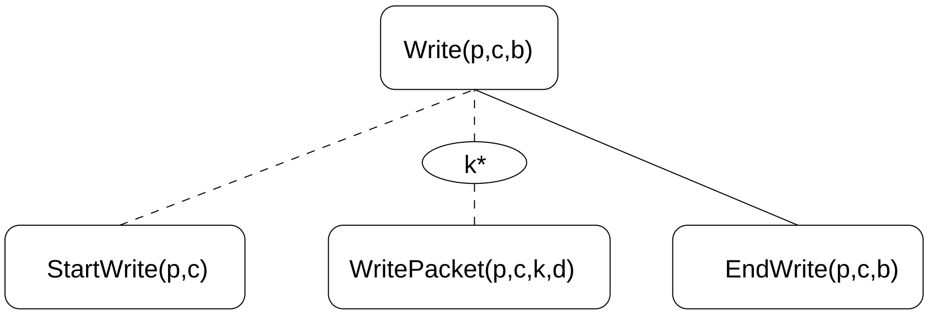


**Fig. 1.** Decomposing the Write Event

We introduce $writing$ ($reading$) to keep track of the process/channel pairs that are involved in the writing (reading) activity. Each channel may have, at most, one writing (reading) process (defined by partial injection). We also introduce $buff2$ and $data2$. $buff2$ represents a local buffer, as-

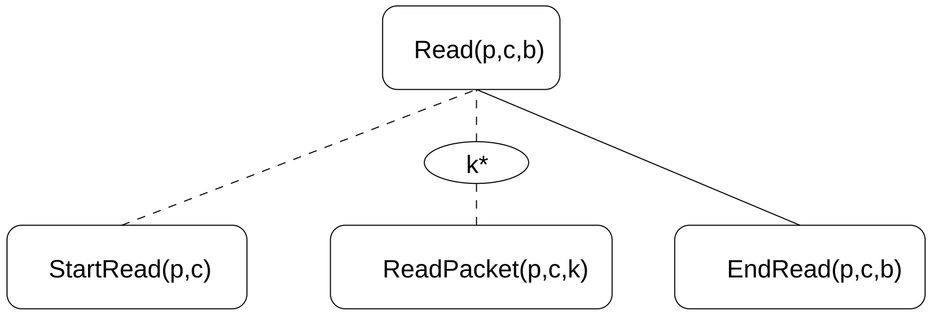


**Fig. 2.** Decomposing the Read Event

sociated with a particular process, which replaces $buff$ of the abstraction. The data in $buff2$ can be added or removed one packet at a time. $data2$ represents a channel buffer, which replaces $data$ of the abstraction. The data in $data2$ can be added or removed one packet at a time.

**Invariants**

$writing \in proc \rightarrowtail\mapsto chan$

$reading \in proc \rightarrowtail\mapsto chan$

$buff2 \in proc \rightarrow Block$

$data2 \in chan \rightarrow Block$

We ensure processes cannot be reading and writing at the same time with the invariant,

$$dom(writing) \cap dom(reading) = \varnothing$$

However we allow channels to be in the range of both $reading$ and $writing$ simultaneously.

The new event $StartWrite$ refines skip.

```
$StartWrite \triangleq$
  **ANY** $p,\ c$
  **WHERE** $p \in proc\ \wedge\ c \in chan\ \wedge\ p \notin dom(writing)\ \wedge$
    $c \notin ran(writing)\ \wedge\ buff2(p) \neq \varnothing\ \wedge\ data2(c) = \varnothing \wedge$
    $p \notin dom(reading)$
  **THEN** $writing := writing \cup \{p \mapsto c\}$
  **END**
```

The process and channel $p \mapsto c$ are added to the set of writing pairs. Once a process-channel pair is added to the set of writing pairs we transfer individual packets of data $k \mapsto d$, from one to the other. $k$ is the packet identifier and $d$ represents the data.

The $WritePacket$ event:

```
$WritePacket \triangleq$
  **ANY** $p,\ c,\ k,\ d$
  **WHERE** $p \mapsto c \in writing\ \wedge\ k \in dom(buff2(p))\ \wedge$
    $d = buff2(p)(k)\ \wedge\ k \notin dom(data2(c))$
  **THEN** $data2(c) := data2(c) \cup \{k \mapsto d\}$
  **END**
```

In the $WritePacket$ event a packet $k \mapsto d$ is added to the channel buffer $data2(c)$.

The $EndWrite$ event refines $Write$.

```
$EndWrite \triangleq$
  **REFINES** $Write$
  **ANY** $p,\ c$
  **WHERE** $p \mapsto c \in writing\ \wedge\ c \in chan\ \wedge\ data2(c) = buff2(p)$
  **WITH** $b = buff2(p)$
  **THEN** $writing := \{p\} \lhd writing \parallel buff2(p) := \varnothing$
  **END**
```

The process $p$ is removed from the set of writers $writing := \{p\} \lhd writing$ and the local buffer is cleared, $buff2(p) := \varnothing$. The witness $b = buff2(p)$ links local variable $b$ of the abstraction to $buff2(p)$ of the refinement. Local variable $b$ does not appear in the refinement and is, instead, replaced by buff2(p). Both $b$ and $buff2(p)$ represent the process' block of data.

*StartRead* refines skip:

```
$StartRead \triangleq$
  ANY $p,\ c$
  WHERE $p \in proc \ \wedge\ c \in chan \ \wedge\ p \notin dom(reading) \ \wedge$
    $c \notin ran(reading) \ \wedge\ p \notin dom(writing) \ \wedge\ data2(c) \neq \varnothing \ \wedge\ buff2(p) = \varnothing$
  THEN $reading \cup \{p \mapsto c\}$
  END
```

The process and channel pair $p \mapsto c$ are added to the set of reading pairs.

*ReadPacket* refines skip:

```
$ReadPacket \triangleq$
  ANY $p,\ c,\ k$
  WHERE $p \mapsto c \in reading \ \wedge\ k \in dom(data2(c)) \ \wedge\ k \notin dom(buff2(p))$
  THEN $buff2(p) := buff2(p) \cup \{k \mapsto data2(c)(k)\}$
  END
```

A packet from the channel buffer, represented by $k \mapsto data2(c)(k)$, is added to the local buffer $buff2(p)$.

*EndRead* refines *Read*:

```
$EndRead \triangleq$
  REFINES $Read$
  ANY $p,\ c$
  WHERE $p \in proc \ \wedge\ c \in chan \ \wedge p \mapsto c \in reading \ \wedge$
    $c \notin ran(writing) \ \wedge\ buff2(p) = data2(c)$
  WITH $b = data2(c)$
  THEN $data2(c) := \varnothing \ \|\ reading := reading \setminus \{p \mapsto c\}$
  END
```

The channel buffer is emptied in the event action, $data2(c) := \varnothing$, the data is considered to have been consumed by the reading process. The process-channel pair is removed from the set of reading pairs, $reading := reading \setminus \{p \mapsto c\}$.

We added the following gluing invariant to ensure that the channel data block $data$ is equal to the packetized data $data2$, except when the process is writing.

$$\forall c \cdot c \in chan \land c \notin ran(writing) \Rightarrow data(c) = data2(c)$$

We also require that the process block buffer $buff$ must be the same as the packetized buffer $buff2$, except when the process is reading.

$$\forall p \cdot p \in proc \land p \notin dom(reading) \Rightarrow buff(p) = buff2(p)$$

All proofs of the abstract development have been discharged using the Rodin tool.

## 4 A Brief Introduction to OCB

Object-oriented Concurrent-B (OCB) [8] is an intermediate specification notation used to link Event-B and an object-oriented programming language for implementation, see Figure 3. OCB is used to specify implementation details for an Event-B Development. The intermediate specification is translated to Java, and also to another Event-B model which is shown to refine the abstract development. Our system may consist of a number of processes which may perform some tasks, and some objects which may be shared; with mutually exclusive access provided by atomic procedure calls. A specification consists of process and monitor classes. Process classes allow specification of interleaving behaviour, using non-atomic constructs, where atomic regions are defined by labelled atomic clauses. Monitor classes may be shared between the processes, and contain atomic procedure definitions, these may optionally incorporate conditional waiting. We specify sequences of atomic clauses using a semi-colon operator. A simple example of a non-atomic clause, with two labelled atomic clauses which update variables $x$ and $y$, follows:

$$label1 : x := 0; label2 : y := 0$$

Processes are able to interleave in a non-atomic clause where a semi-colon is specified. Each labelled atomic clause maps to an event guarded by a program counter which is derived from the label. This allows us

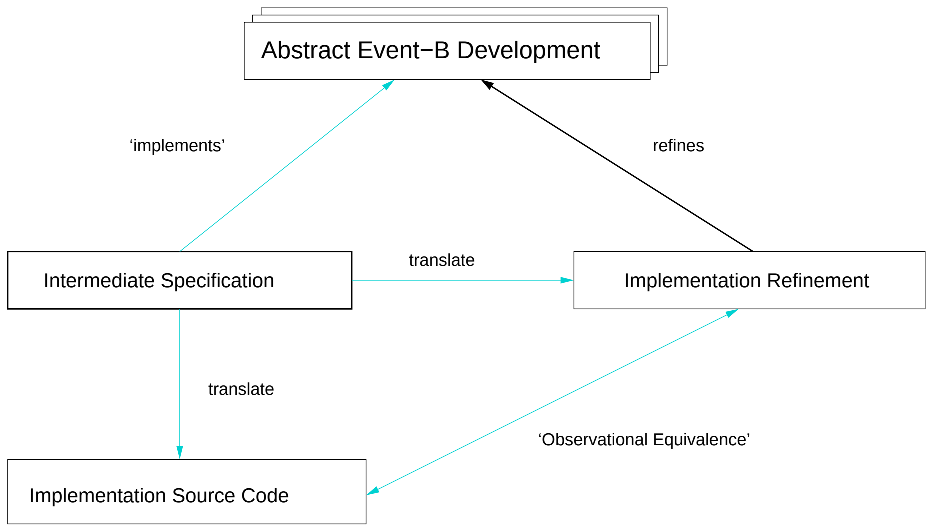

Abstract Event−B Development
'implements'
refines
Intermediate Specification
translate
Implementation Refinement
translate
'Observational Equivalence'
Implementation Source Code

**Fig. 3.** Overview of a Development

to model ordered the execution of an implementation. In addition to the semi-colon operator we have a branching construct,

$$\textbf{if}(g)\ \textbf{then}\ a\ [\ \textbf{andthen}\ na\ ]\ \textbf{endif}$$

where $g$ is a guard, $a$ is an assignment action or procedure call, and $na$ is an optional non-atomic clause. In a branching clause, $g$ and $a$ form an atomic guarded action. This may optionally be followed by a non-atomic clause or additional branches. Each conditional branch maps to a guarded event, and includes a default 'else' branch for the construct shown. The looping construct follows,

$$\textbf{while}(g)\ \textbf{do}\ a\ [\ \textbf{andthen}\ na\ ]\ \textbf{endwhile}$$

As in the branching clause, $g$ and $a$ form an atomic guarded action, and processes may interleave after evaluation of the guarded action at each loop iteration, and optionally in the non-atomic construct $na$, if one is present. Once again, each atomic clause maps to a guarded event, and include a branch for the false guard. Conditional waiting is specified in a procedure using a conditional waiting construct of the following form,

$$\textbf{when}(\ g\ )\{\ a\ \}$$

where $g$ is a guard and $a$ is an action. A clause corresponds to the guarded action, $g \rightarrow a$, where $g$ maps to an event guard, and $a$ is mapped to an event action.

## 5 The Implementation-level Specification

In this section we describe the intermediate level specification. The abstract development has been refined to the point where it is well understood and the process of specifying implementation specific detail can begin. We make choices about which elements are suited to implementation as a processes, and which are the shared data. We also make design decisions such as determining loop implementation, and use of control data to represent set membership. The implementation level refinements are illustrated in Figures 4 and 5. We see that $StartWrite$ is implemented

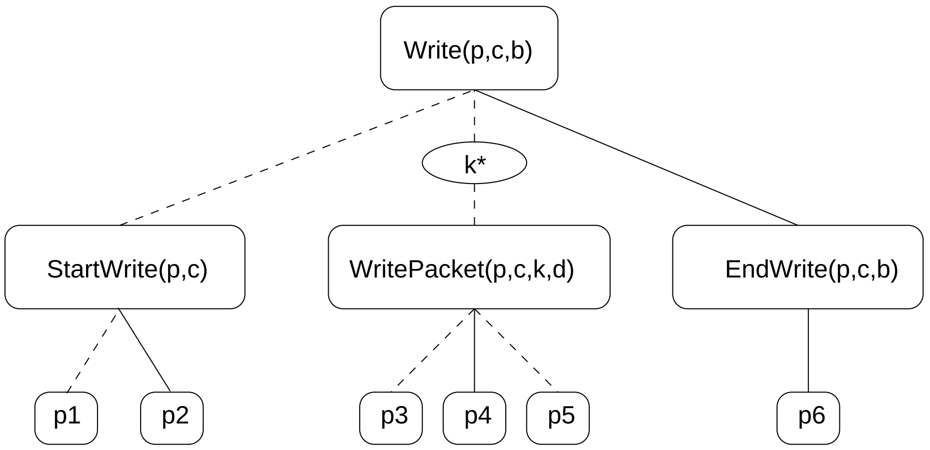


**Fig. 4.** Implementing the Write Event

by clauses labelled $p1$ and $p2$; the iterating $WritePacket$ event is implemented by clauses $p3 \ldots p5$; and $p6$ implements $EndWrite$. A similar arrangement exists for the reading process. A brief description of the clauses follows,

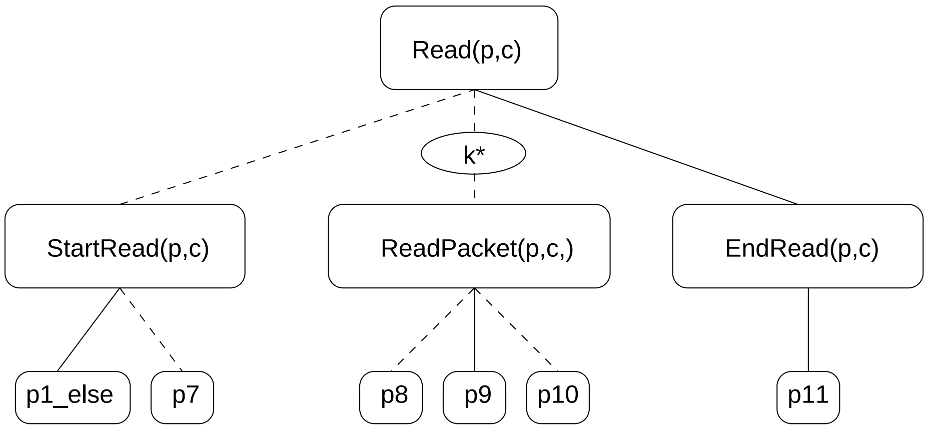


**Fig. 5.** Implementing the Read Event

| Label | Description |
|---|---|
| p1 | If the process is a writer then get the size of the local buffer. |
| p2 | Obtain the write channel if it is free else block. |
| | The process ID and number of packets to send are parameters |
| p3 | While there is a packet to send from the local buffer |
| | remove the packet assigning to the temporary attribute. |
| p4 | Add the data to the channel buffer. |
| p5 | Decrement the count of packets. |
| p6 | Release the channel for other writers. |
| p1_else | If the process is a reader obtain a read channel |
| | if it is free, else block. |
| p7 | Obtain the number of packets to read from the channel. |
| p8 | While there are packets remove a packet from the channel |
| | buffer and assign to the temporary attribute. |
| p9 | Add the packet to the local buffer. |
| p10 | Decrement the buffer counter. |
| p11 | Free the read channel for another reader. |

The clauses added at the implementation level add flow control information, and manipulate data, which also involves additional steps to store shared data in local attributes.

Our design decision is to specify a single process to perform both reading and writing tasks. The specific behaviour of the process is determined by the boolean parameter *isWriter* supplied at instantiation. The process class *Proc* specification is shown here.

```
ProcessClass Proc{
 Buffer buff, Boolean isWriter, Channel ch, Integer id,
 Integer tmpBuffSz, Integer tmpDat
 // The constructor procedure
 Procedure create(Integer pid, Buffer bff,
                   Boolean isWritr, Channel chn){
  id:=pid || buff:=bff || isWriter:=isWritr || ch:= chn ||
  tmpBuffSz:=-1 || tmpDat:=-1
 }
 // The process behaviour
 Operation run(){
  p1: if(isWriter=TRUE) then
      tmpBuffSz:=buff.getSize() andthen
      p2: ch.getWChan(id, tmpBuffSz);  // refines StartWrite
      p3: while(tmpBuffSz>0) do tmpDat:=buff.remove() andthen
          p4: ch.add(tmpDat);  // refines WritePacket
          p5: tmpBuffSz:=tmpBuffSz-1 endwhile ;
      p6: ch.freeWChan() endif  // refines EndWrite
  else ch.getRChan(id) andthen  // refines StartRead
      p7: tmpBuffSz:=ch.getWriteSize();
      p8: while(tmpBuffSz>0) do tmpDat:=ch.remove() andthen
          p9: buff.add(tmpDat);  // refines ReadPacket
          p10: tmpBuffSz:=tmpBuffSz-1 endwhile ;
      p11: ch.freeRChan() endelse  // refines EndRead
 }
}
```

We now look at the monitor class specification of the channel. *Channel* has a cyclic buffer *buff* of where integer data elements are added to the tail, and removed from the head of, the buffer. We now provide an informal description of the monitor procedures,

| | |
|---|---|
| *add* | Add a packet to the buffer tail, block the caller if there is no spare capacity. |
| *remove* | Remove a packet from the buffer head and return it, block the caller if there is nothing to remove. |
| *getWChannel* | Obtain a channel for writing if it is available and there is no reader, else block the caller. |
| *freeWChan* | Release a write channel by removing the process ID. |
| *getRChannel* | Obtain a channel for reading if it is available and there is data to read, else block the caller. |
| *freeRChan* | Release a read channel by removing the process ID. |
| *getWriteSize* | Returns the size of the data block. |

The monitor procedures described above are specified in the Channel MonitorClass. The MonitorClass serves to encapsulate its attributes, access to data is only permissible through atomic procedure calls and is shown in Appendix 10.

A *MainClass* is used as the entry point for execution and is considered to be a special kind of process. The processes may equally be initiated by a GUI or scheduler but details are omitted here.

## 6 The OCB Refinement

In this section we give details of the Event-B model which results from translation of the OCB specification. Each labelled atomic clause gives rise to an event with the appropriate guards and actions, but we present only one of the translated events here. The *Proc* class' clause labelled $p2$ refines $StartWrite$ and gives rise to the event $Proc\_p2$. $Proc\_p2$ is shown here with some minor changes to improve readability. The $StartWrite$ process $p$ is related to $self$ of $Proc\_p2$ using a predicate $p = self$ in the event's $WITH$ clause. The $StartWrite$ channel $c$ is related to $target$ of

$Proc\_p2$ with $c = target$.

```
$Proc\_p2 \triangleq$
  **REFINES** $StartWrite$
  **ANY** $self, target$
  **WHERE** $self \in Proc \wedge state(self) = p2 \wedge$
    $target = ch(self) \wedge wPID(target) = -1 \wedge$
    $writeSize(target) \leq 0$
  **WITH** $p = self \wedge c = target$
  **THEN** $wPID(target) := id(self) \wedge$
    $writeSize(target) := tmpBuffSz(self)$
    $state(self) := p3$
  **END**
```

When proving the refinement of the implementation model we use gluing invariants to relate the abstraction with the implementation. An example of such an invariant follows,

$$\begin{aligned}&\forall pr, cn \cdot pr \in proc \wedge cn \in chan \wedge\\ &pr \in dom(ch) \wedge ch(pr) = cn \wedge\\ &id(pr) = wPID(cn)\\ &\Rightarrow pr \mapsto cn \in writing\end{aligned}$$

This states that if a channel implementation has a writing process identifier value as its *wPID* attribute (given by id(pr) = wPID(cn)) then this implies that the process-channel pair should be in the writing set in the abstraction. A similar invariant exists for the readers,

$$\begin{aligned}&\forall pr, cn \cdot pr \in proc \wedge cn \in chan \wedge\\ &pr \in dom(ch) \wedge ch(pr) = cn \wedge\\ &id(pr) = rPID(cn)\\ &\Rightarrow pr \mapsto cn \in reading\end{aligned}$$

We also wish to ensure that no processes have the identifier -1, which is reserved for indicating that the channel has no reader/writer, We specify the following,

$$\forall pr \cdot pr \in Proc \wedge pr \in dom(id) => id(pr) \neq -1$$

## 7 The Java Implementation

The mapping to Java is mostly self evident since it is very similar to the OCB specification, we therefore present it without extensive explanation. In the OCB translation to Java, processes map to threads implementing the Java *Runnable* interface. The Java source for the reader/writer example is shown in Appendix 11. The Channel class is implemented and encapsulated by Java constructs. Atomic procedures are implemented by synchronized methods, and the conditional wait construct is implemented by a $try - wait - catch$ block. The Java code for the *Channel* class is shown in Appendix 12.

## 8 Related Work

Our presentation is an example of code-generation from a formal specification. To the best of our knowledge this is the only published code-generation approach for Event-B. In this approach we use an intermediate specification notation to describe object-oriented, concurrent implementations. In classical-B the implementation level refinement specification was facilitated by the $B0$ language [6]. Tools for translation to various languages, including C and Ada, is described in [5]. A feature of this approach is that it mapped only to sequential programs; no concurrency constructs were provided. There is a code-generation approach handling concurrency, aimed at Classical-B, called JCSProB [23]. It makes use of the JCSP libraries. JCSP [21, 22] establishes a link between CSP [11, 14] and Java. The JCSP libraries provide an implementation of the Occam concurrency framework, it uses a message passing, rendezvous style, as a basis for communication between concurrent Java threads. Using JCSProB the ProB [13] tool can be used to construct and model check a combined CSP specification and B machine, which can then be translated to Java code. Our work is an alternative to this style and uses a shared memory approach, where processes share data in memory and accesses are protected using synchronized method calls. We also tailor our approach to the new Event-B tool rather than classical B.

We use some of the concepts of UML-B, [16, 17, 18, 19], to model objects and instances, but our notation introduces process classes that give rise to concurrently executing processes, with interleaving operations. The sequential operator used within a non-atomic operation defines points where interleaving may take place in addition to points we define in the looping and branching clauses. We define monitor classes that are shared between processes, and also define a mapping to Java code which is absent from UML-B. The OCB syntax incorporates features such as the non-atomic looping and branching clauses which are not part of UML-B.

Another formal approach incorporating code generation is Object-Z [15], it is a specification language which is an extension of the Z notation, incorporating the notion of classes. A class schema encapsulates the state and behaviour of a class, and variables can take the type of a class. Inheritance mechanisms are used to clarify the structure of the systems and aid refinement and verification. Object-Z differs from OCB in a number of ways, for example we do not incorporate the notion of inheritance and we do not refine an OCB specification. OCB forms a link in the development process between the Event-B modelling language and the implementation, Object-Z is used for system specification. VDM++ [7] is an object oriented approach which is an extension of VDM-SL [12], UML diagrams are used to specify an object oriented development which are mapped to an underlying VDM++ model. VDM++ can be translated to Java but is not able to model features involving concurrency. Circus combines CSP and Z [20]. The JCircus [9] translation tool gives rise to Java code which is intended to serve as an animator for circus. JCircus makes use of the JCSP libraries and gives rise to Java code that is based on the message passing approach, in this respect it is similar to JCSProB.

## 9 Conclusion

In this example we developed an Event-B model of reading and writing processes sharing a channel. At the highest level of abstraction we modelled reads and writes of blocks of data, this was refined so that reads and writes of individual packets of data were modelled. We gave an overview of the OCB notation, which we use to specify implementations for concurrent processes sharing data. We also gave an overview of JSDs and showed how they can be beneficial in understanding the relationship between abstract and refined events, and event ordering, as a development proceeds. The OCB notation was used to provide an implementation level specification for our abstract development, which was subsequently mapped to Java code. We also map to an Event-B model which models the implementation, and are required to show that this implementation model is a refinement of the abstract development (on-going work).

## References

1. J.R. Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. J.R. Abrial. Event based sequential program development: Application to constructing a pointer program. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 51–74. Springer, 2003.
3. J.R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An open extensible tool environment for Event-B. In Z. Liu and J. He, editors, *ICFEM*, volume 4260 of *Lecture Notes in Computer Science*, pages 588–605. Springer, 2006.
4. Michael Butler. Decomposition Structures for Event-B. In *Integrated Formal Methods iFM2009, Springer, LNCS 5423*, volume LNCS. Springer, February 2009.
5. ClearSy System Engineering. *Atelier B Translators*, version 4.6 edition.
6. ClearSy System Engineering. *The B Language Reference Manual*, version 4.6 edition.
7. CSK Systems Corporation. The vdm++ language manual.
8. A. Edmunds and M. Butler. Linking Event-B and Concurrent Object-Oriented Programs. In *Refine 2008 - International Refinement Workshop*, May 2008.
9. A. Freitas and A. Cavalcanti. Automatic translation from Circus to java. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2006.
10. J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification - Third Edition*. Addison-Wesley, 2004.
11. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
12. C.B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1990.
13. M. Leuschel and M. Butler. ProB: A Model Checker for B. In *Proceedings of Formal Methods Europe 2003*, 2003.
14. A. W. Roscoe, C. A. R. Hoare, and R. Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
15. G. Smith. *The Object-Z specification language*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.
16. C. Snook and M. Butler. *U2B - A tool for translating UML-B models into B*, volume UML-B Specification for Proven Embedded Systems Design. Springer, 2004.
17. C. Snook and M. Butler. UML-B: Formal modelling and design aided by UML. *ACM Transactions on Software Engineering and Methodology*, 2006.
18. C. Snook and M. Butler. UML-B and Event-B: an integration of languages and tools. In *The IASTED International Conference on Software Engineering - SE2008*, February 2008.
19. C. Snook, M. Butler, and I. Oliver. Towards a UML profile for UML-B. Technical report, Electronics and Computer Science, University of Southampton, 2003.
20. J. M. Spivey. Understanding Z: A specification language and its formal semantics. *Cambridge Tracts in Theoretical Computer Science*, 3, 1988.
21. P.H. Welch and J.M.R. Martin. A CSP model for Java multithreading. In *Software Engineering for Parallel and Distributed Systems*, 2000.
22. P.H. Welch and J.M.R. Martin. Formal Analysis of Concurrent Java Systems. In P.H. Welch and A.W.P. Bakkers, editors, *Communicating Process Architectures 2000*, pages 275–301, sep 2000.

23. L. Yang and M. Poppleton. Automatic translation from combined and csp specification to java programs. In J. Julliand and O. Kouchnarenko, editors, *B*, volume 4355 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2007.

# Appendix

## 10 The Channel Specification

```
MonitorClass Channel{
 // Attributes
 Integer capacity, Integer[5] buff, Integer head, Integer tail,
 Integer size, Integer rPID, Integer wPID, Integer writeSize

 // The Constructor
 Procedure create(){
  head:= 0 || tail:= 0 || size:= 0 || capacity:= 5 ||
  rPID:= -1 || wPID:= -1 || writeSize:= -1
 }

 // 'Refines' WritePacket - in a call from clause p4
 Procedure add(Integer val){
  when(size<capacity & capacity /= 0 & tail>=0 & tail<= 4){
       buff[tail]:= val || tail:= (tail+1) mod capacity ||
       size:= size+1}
 }

 // The value is stored in a temporary buffer in a
 // call from clause p8 - implementing ReadPacket
 //as part of the reading activity.
 Procedure remove(){
  when(size>0  & capacity /= 0 & head>=0 & head<=4){
       return:= buff[head] || size:= size-1 ||
       head:= (head+1) mod capacity}
 }: Integer

 // Called in p1_else clause - refines StartRead.
 // Set the channel for reading, by the process
 // with identifier pid.
 // Block if it is already owned or has nothing to read.
 Procedure getRChan(Integer pid){
  when(rPID=-1 & writeSize>0){rPID:= pid}
 }

 // Called in p11 clause - refines EndRead.
 // Free the channel for reading.
 Procedure freeRChan(){
  rPID:= -1 || writeSize:= -1
 }

 // Called in p1 clause - implementing StartWrite.
 // Set the channel for writing writesze bytes, by
 // the process pid.
 // Block if the channel is already owned for writing or
 // has bytes still to write.
 Procedure getWChan(Integer pid,Integer writeSze){
  when(wPID=-1 & writeSize<=0){
        wPID:= pid || writeSize:= writeSze}
 }

 // Called in p6 clause - refines EndWrite.
 // Free the channel for writing.
 Procedure freeWChan(){ wPID:= -1 }

 // Return the number of bytes to write.
 Procedure getWriteSize(){ return:= writeSize }: Integer
}
```

## 11 The Process Java Code

```
public class Proc implements Runnable {

 private Buffer buff = null; private boolean isWriter;
 private  Channel ch = null; private int id;
 private int tmpBuffSz; private int tmpDat;

 public Proc(int pid, Buffer bff, boolean isWritr, Channel chn) {
  id = pid; buff = bff; isWriter = isWritr; ch = chn;
  tmpBuffSz = -1; tmpDat = -1;
 }

 public void run() {
  if (isWriter == true) {
   tmpBuffSz = buff.getSize(); // p1
   ch.getWChan(id, tmpBuffSz); // p2
   while (tmpBuffSz > 0) {
     tmpDat = buff.remove(); // p3
     ch.add(tmpDat); // p4
     tmpBuffSz = tmpBuffSz - 1; // p5
   }
   ch.freeWChan(); // p6
  }else {
   ch.getRChan(id);
   tmpBuffSz = ch.getWriteSize(); // p7
   while (tmpBuffSz > 0) {
     tmpDat = ch.remove(); // p8
     buff.add(tmpDat); // p9
     tmpBuffSz = tmpBuffSz - 1; // p10
   }
   ch.freeRChan(); // p11
  }
 }
}
```

## 12 The Channel Java Code

```
public class Channel {
 private int capacity;private int[] buff = new int[5];
 private int head;private int tail;private int size;
 private int rPID;private int wPID;private int writeSize;

 public Channel() {
  head = 0; tail = 0; size = 0; capacity = 5; rPID = -1;
  wPID = -1; writeSize = -1;
 }
```

```
public synchronized void add(int val) {
  try {
    while (!(size < capacity && capacity != 0 &&
                         tail >= 0 && tail <= 4)) {
      wait();
  }} catch (InterruptedException e) { e.printStackTrace(); }
  buff[tail] = val;
  tail = (tail + 1) % capacity;
  size = size + 1;
  notifyAll();
}

public synchronized int remove() {
  int initial_head = head;
  try {
    while (!(size > 0 && capacity != 0 && head >= 0 && head <= 4)) {
      wait();
      initial_head = head;
    } catch (InterruptedException e) { e.printStackTrace();}
  size = size - 1;
  head = (initial_head + 1) % capacity;
  notifyAll();
  return buff[initial_head];
}

public synchronized void getRChan(int pid) {
  try {
  while (!(rPID == -1 && writeSize > 0)) {
    wait();
  }} catch (InterruptedException e) { e.printStackTrace(); }
  rPID = pid;
  notifyAll();
}

public synchronized void freeRChan() {
  rPID = -1;
  writeSize = -1;
  notifyAll();
 }

 public synchronized void getWChan(int pid, int writeSze) {
  try {
    while (!(wPID == -1 && writeSize <= 0)) {
      wait();
    }} catch (InterruptedException e) { e.printStackTrace(); }
  wPID = pid;
  writeSize = writeSze;
  notifyAll();
  }

 public synchronized void freeWChan() {
  wPID = -1;
  notifyAll();
 }

 public synchronized int getWriteSize() {
  return writeSize;
 }
}
```

# Part III

# Event-B Metrics and Tools

# Towards Event-B Metric Support in RODIN

Christophe Ponsard, Renaud De Landtsheer and Arnaud Michot

CETIC Research Centre, Charleroi (Belgium)
{cp,rdl,arm}@cetic.be

## Abstract

Metrics are a widely used instrument at code level to provide useful information on its maturity and maintainability. However they are less widely used at earlier development stages. Some metrics address semi-formal modelling languages such as UML but little work focuses on formal models metrics for languages such as Event-B.

In this paper, we consider metrics for Event-B specification in the light of the needs expressed by some industrial partners of the project. We identify some useful product and process metrics and discuss the way to compute them from the specification structure and building process. We present how to support their computation in an extensible way on the RODIN toolset and show the on-going work of prototype implementation.

## 1 Introduction

A software metric is "a measure of some property of a piece of software or its specifications. The purpose of making these measures is to improve the overall quality of what is measured. One needs to measure something in order to improve over it" [2]. Metrics can be directly observable quantities or can be derived from them. Some examples of raw metrics are: number of source lines of code, number of requirements, number of tests,etc. A metric can quantify either a characteristic of product (such as code, test or model) or the effort to build it.

Metrics are widely used at code level to assess a number of quality attributes of the code such as maintainability, evolvability, security, reliability. Over time, a number of metrics have been developed to assess such qualities for a variety of language families (such as procedural, object-oriented) and specific instances (such as Cobol, C, C++, Java,

.Net). Over time, a number of meaningful metrics have emerged. For example, maintainability can be assessed by examining the distribution of the documentation (line of comments) versus the complexity of the procedure/methods (McCabe metrics [9]) and evaluating the importance of complex under-documented methods. This information is useful for the developer but more importantly to the project manager in order to run the project by monitoring metric evolution over time.

Similar methods can also be applied to models rather than code. They are not widely used, still they are gaining interest and usage, especially in a model driven context where the code is derived from higher level models (typically: object-oriented code derived from UML models [11]). For such generated code, it is more meaningful to make the quality assessment at model level. A number of common metrics have proved to be reusable for the object-oriented case while other have been defined to take more into account the design-level dimension.

Concerning more formal models, with strong underlying mathematics, there is however little published work on the use of such technique. However characteristics like understandability can be significant problem for such languages: for example special attention is required both for the experts having to maintain a formal specification or the non-expert having to validate it. For this, the presence of proper inline documentation or traceability information to originating or explanatory documents should be assessed, preferably in a quantitative way. Part of the reason for this could be the fact that such models enable powerful reasoning and processings, reducing less importance of more empirical measurement techniques. Indeed metrics extracted from a project have to be analysed in the project context.

In the scope of the DEPLOY project, the Event-B formal language is used for modelling complex systems. An interest was confirmed by some DEPLOY industrial partners to be able to monitor some specific characteristics of the model and of its development process (with activities such as modelling, proving and model-checking) and their evolution over time.

In this paper we explore how metrics can be collected to support these activities, based on the infrastructure provided by the RODIN toolset and the underlying Eclipse platform [13]. We propose an extensible metrics plug-in for RODIN in which new metrics can easily be registered and illustrate it on some simple metrics as well as more complex metrics such as the Halstead metrics presented in the related paper [12].

This paper is structured as follows: section 2 summarises the main requirements for the plugin. Section 3 reviews some available model metrics (including semi-formal and formal models). Section 4 reviews the design of representative existing tools for model metrics. Section 5 presents the implementation of the plug-in and illustrates it on a few metrics. Finally section 6 discusses future work.

## 2 Requirements

From the user point of view, the purpose of the metric tools is to provide the Event-B specifier with information both about the model and about the model construction process. More precisely, it should:

- *provide feedback to the user about characteristics of the Event-B model* being built to highlight potential problems. Regarding maintainability, the analyst can be reminded that he is not providing enough documentation. Regarding easiness to prove, the analyst could question if his approach with the tool is adequate. Basic metrics such as size were also requested for project reporting.
- *automate data collection about the evolution of those characteristics over time*, ideally at successive model versions. For example, some industrial partner need a high level of proof automation, monitoring the related metric allows to see if methods and tools are improving to meet the required level.
- *be user-friendly*: by providing easy to access and understand reporting, for example by using meaningful graphics.
- *be efficient*: metrics evaluation will not significantly slow down the interactivity with the tool
- *be open*: by allowing data export for external reporting.

At a more technical level, the requirements are:

- to integrate in the last version of the RODIN toolset both for the collection and reporting
- to facilitate addition and adaptation of new metrics

## 3 Survey of Model Metrics

### 3.1 Object-Oriented Designs

A large set of metrics is available for assessing the quality of object-oriented design (typically expressed as UML diagrams). Those were elaborated based on an initial suite of metrics[1]. It is interesting to examine them from a classification point of view:

- *Good design* is based on principles such as strong cohesion (H, LCOM metrics), weak coupling between packages (Ca and Ce: afferent and efferent coupling metrics), stable abstractions (D metric), use of inheritance hierarchies (DIT metric)[6].
- *Complexity*. This can be the complexity related to methods, based on weighted methods per class (WMC)[6]. It is equal to the sum of the complexities of each method defined in a class which can be supposed equal, estimated from other diagrams such as the activity, sequence, communication diagrams or state machine diagrams. Another type of complexity is related to the relationships between classes, based on the number of associated elements either in the same package as the class or not. A more generic complexity measure was developed by Halstead and based on the analysis of the vocabulary [4]. It is also described in details in the related paper on metrics [12].
- *Understandability* is based on metrics such as the presence of attached documentation to model element (which is generally a tool feature) and the associated complexity.

### 3.2 Formal Methods Metrics

There is little existing literature on metrics over formal models and they mainly target complexity and understandability.

Metrics for Z specifications [15] are proposed to measure the complexity of Z models [18]. They are based on count and weighted counts over the structure of schema inclusion and referencing and on count of the number of methods per schema. It is quite similar to the classical metrics applied on the object-oriented software.

Another, more general, approach is Alpha-metric[5]. The underlying principle expressed by Morowitz that complex systems share certain features like having a large number of elements, possessing high dimensionality and representing an extended space of possibilities [10].

There has been some indication that a metric might be proposed for measuring the understandability of Z specifications. It is based on the assessment that structuring a Z specification into schemas of about 20 lines long significantly improved understandability over a monolithic specification. However, there seems to be no perceived advantage in breaking down the schemas into much smaller components[3].

## 4 Survey of Model Measurement Tools

This section provides here a survey of some tools having a metric feature and presents their implementation in order to employ them in designing our own tool.

A number of commercial tools for UML modelling have some form of metric feature, for example: MagicDraw UML[7] or Enterprise Architect [14]. Some metrics plug-ins have also been developed for open-source tools such as [17], those are less frequent and less mature. Finally tools such as SD Metrics are also available, such as SD Metrics [19] which simply analyses the standard XMI export of a model. For UML, an interesting and generic way to compute metrics, is to use Object Constraint Language (OCL) as powerful query language over the model.

In another kind of modelling activity, Matlab/Simulink also proposes a "Modeling Metric Tool" [8]. This tool addresses process metrics in order to cope with the fact that the productivity figures will be different when using model-based tools with code generation.

At requirements level, models are also used to capture the structure and traceability information. Such tools have a dedicated database and a corresponding (script-based) query language that can be used to compute the desired metrics (e.g. requirements to test coverage).

## 5 Design of a Metric Plug-in

### 5.1 Evolvable Tool

Regarding Event-B, there is no known work about specific metrics for measuring specific aspects of a model such as complexity, maintainability. To address this need some measuring elements have been identified, but were not consolidated in such a higher level metric. Some classes of metrics are:

– Size: based on number of machines/contexts, number of events
– Complexity: could be based on size of contexts, number of refinements, invariants
– Maintainability/understandability: based on complexity, comment ratio, proof ratio, structure of assertions
– Effort to prove: based on the ratio of manual/automated proofs

Developing a metric for providing a good ranking of useful characteristics is not easy and relies on the understanding, analysis and observation of the application of the Event-B methodology itself. A key methodological element is the refinement strategy which should introduce smart and progressive refinement. This suggests a metric for determining the "distance" between two refinements. At this stage such metrics are not yet available but the purpose of the tool is to easily allow the addition of such metrics and the validation of their reliability.

### 5.2 Computing Product Metrics

In order to compute metrics on Event-B model, the model must be available for interrogation. This is best done using a query language. Currently RODIN provides a number of different ways to access the language: directly through an XML file, using a specific RODIN API (in JAVA) or using an EMF meta-model of Event-B. The worse option is to use the file itself because it works at XML level and not at Event-B level and there is no guarantee of it. The RODIN API is stable and efficient but does not provide a powerful query language and therefore some procedural encoding is required. Finally the EMF API allows to use a declarative style in OCL but is less efficient.

### 5.3 Computing Process Metrics

Eclipse provides build-in activity support through the standard Mylyn plugin [16]. Mylyn enables the definition of user specific task contexts on a project which can greatly enhance the productivity by removing all the information that is not relevant for completing that task. It also provides a monitor interface to collect information about a user's activity in Eclipse. Two kind of clients can connect to the monitor: first a "Context UI" which can transform interaction into a degree-of-interest model, and second, a user study plug-ins which can report on Eclipse usage trends (with the user's consent).

Specific process metrics can be implemented using this Monitor API. For example, a task context can be defined for Event-B proving and a client can monitor the proof related events and times for manual proving. Some assumption/convention may be necessary about the actual activity of the user for the data to be meaningful.

## 6 Implementation

The current prototype is limited to the product metrics and relies on the RODIN API. The plug-in is composed of two parts:

- the plug-in framework, providing the display at various levels of structure (such as project, machine and event). It also provides an interface to register new metrics.
- a metric repository, composed of a set of standard metrics. The initial set is composed of simple metrics for volumetry, provability and complexity.

The standard interface for describing a set of metrics is composed of a method returning the name of the whole set, the names of the metrics and the way to compute them from a model element (possibly at various level of granularity).

```
public interface IMetrics {
  public String getName();
  public String[] getMetricNames();
  public Object[] getMetricValues(IModelElement element);
}
```

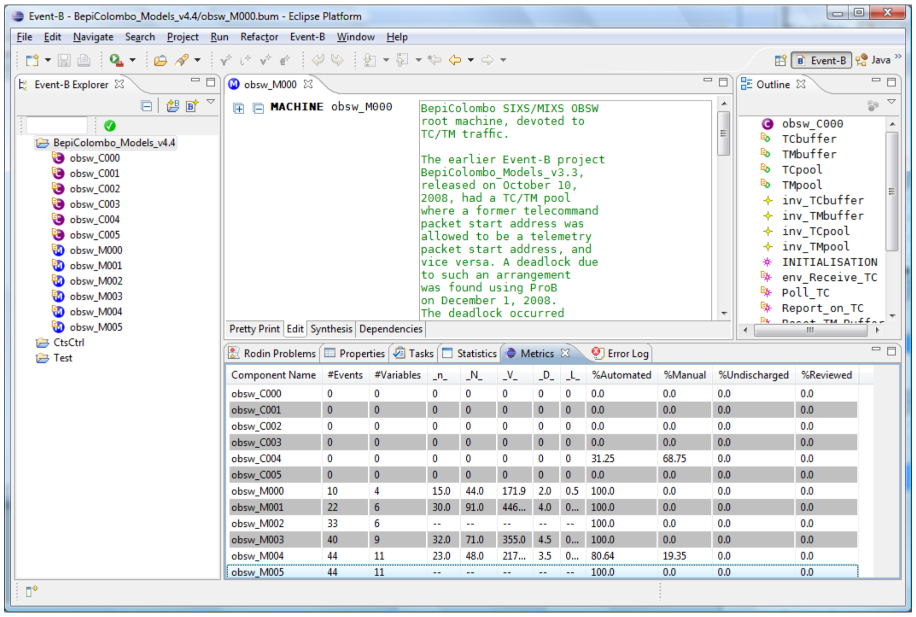
**Fig. 1.** IMetrics Interface

The following code shows the implementation of simple volumetry metrics: it counts the number of events and variables. The information is provided at machine and project level.

The Halstead metrics adapted for Event-B in [12] was implemented in a similar way. The total code size is about 200 lines. Figure 3 shows the plug-in in action on the similar BepiColumbo specification used in this paper.

```
public Object[] getMetricValues(IModelElement element) {
  if (element instanceof ModelProject) {
    int n_evt = 0;
    ModelProject project = (ModelProject) element;
    for (final ModelMachine machine : project.getRootMachines()) {
      n_evt += getEventCount(machine);
      n_var += getVarCount(machine);
    }
    return new Integer[] { n_evt };
  }
  if (element instanceof ModelContext)
    return new Integer[] { 0 };
  if (element instanceof ModelMachine) {
    ModelMachine mach = (ModelMachine)element;
    return new Integer[] { getEventCount(mach) };
  }
}
private int getEventCount(ModelMachine machine) throws RodinDBException {
  return machine.getInternalMachine().getEvents().length;
}
```

**Fig. 2.** Implementation of Volumetry Metrics



```
MACHINE obsw_M000
```

```
BepiColombo SIXS/MIXS OBSW
root machine, devoted to
TC/TM traffic.

The earlier Event-B project
BepiColombo_Models_v3.3,
released on October 10,
2008, had a TC/TM pool
where a former telecommand
packet start address was
allowed to be a telemetry
packet start address, and
vice versa. A deadlock due
to such an arrangement
was found using ProB
on December 1, 2008.
The deadlock occurred
```

| Component Name | #Events | #Variables | _n_ | _N_ | _V_ | _D_ | _L_ | %Automated | %Manual | %Undischarged | %Reviewed |
|---|---|---|---|---|---|---|---|---|---|---|---|
| obsw_C000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0.0 | 0.0 | 0.0 | 0.0 |
| obsw_C001 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0.0 | 0.0 | 0.0 | 0.0 |
| obsw_C002 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0.0 | 0.0 | 0.0 | 0.0 |
| obsw_C003 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0.0 | 0.0 | 0.0 | 0.0 |
| obsw_C004 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 31.25 | 68.75 | 0.0 | 0.0 |
| obsw_C005 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0.0 | 0.0 | 0.0 | 0.0 |
| obsw_M000 | 10 | 4 | 15.0 | 44.0 | 171.9 | 2.0 | 0.5 | 100.0 | 0.0 | 0.0 | 0.0 |
| obsw_M001 | 22 | 6 | 30.0 | 91.0 | 446... | 4.0 | 0... | 100.0 | 0.0 | 0.0 | 0.0 |
| obsw_M002 | 33 | 6 | -- | -- | -- | -- | -- | 100.0 | 0.0 | 0.0 | 0.0 |
| obsw_M003 | 40 | 9 | 32.0 | 71.0 | 355.0 | 4.5 | 0... | 100.0 | 0.0 | 0.0 | 0.0 |
| obsw_M004 | 44 | 11 | 23.0 | 48.0 | 217... | 3.5 | 0... | 80.64 | 19.35 | 0.0 | 0.0 |
| obsw_M005 | 44 | 11 | -- | -- | -- | -- | -- | 100.0 | 0.0 | 0.0 | 0.0 |

**Fig. 3.** Metrics plug-in in action on the BepiColombo Model

## 7 Future Work

The plan is to move to the EMF API to allow users to directly encode metrics using OCL strings without any need to recompile. The Mylyn monitor API will also be deployed to collect activity information that will be available for the metrics computation. The process information can be attached to specific extension points of the Event-B meta-model, for example to those related to the proofs.

Currently the evolution of metrics over time is not supported. Only computation of the current state is possible. Recording the evolution over time is planned for a meaningful set of metrics which will be stored in a project file. In the longer term, when model versioning becomes available, this feature should be associated with it to directly allow to query the repository for computing the evolution of a metric over time.

## Acknowledgement

This project was financially supported by the DEPLOY FP7 project (project reference number 214158).

## References

1. S. R. Chidamber and C. F. Kemerer, *A metrics suite for object oriented design*, IEEE Trans. Softw. Eng. **20** (1994), no. 6, 476–493.
2. Tom DeMarco, *Controlling Software Projects: Management, Measurement, and Estimates*, Prentice Hall, 1986.
3. Kate Finney, Keith Rennolls, and Alex Fedorec, *Measuring the comprehensibility of z specifications*, J. Syst. Softw. **42** (1998), no. 1, 3–15.
4. M.H. Halstead, *Elements of software science*, Elsevier North Holland, 1977.
5. Peter Kokol and Vili Podgorelec, *Ranking the complexity of niam conceptual schemas by alpha metric*, SIGPLAN Not. **35** (2000), no. 3, 59–64.
6. Michele Lanza and Radu Marinescu, *Object-oriented metrics in practice*, Springer, 2006.
7. No Magic, *MagicDraw UML*, `http://www.magicdraw.com`.
8. Mathworks, *Modeling Metrics Tool*, `http://www.mathworks.com/matlabcentral/fileexchange/5574`.
9. Thomas J. McCabe, *A complexity measure*, ICSE '76: Proceedings of the 2nd international conference on Software engineering (Los Alamitos, CA, USA), IEEE Computer Society Press, 1976, p. 407.
10. H. Morowitz, *The Emergence of Complexity*, Complexity 1(1):4 (1995).
11. Object Management Group, *Unified Modeling Language*, `http://www.uml.org`.
12. Marta Plaşka and Kaisa Sere, *Towards event-b specification metrics*, Proceedings of Deploy Technical Workshop 2009, Technical Report of the School of Computing Science, University of Newcastle (M. Mazzara et al. Editors, ed.), 2010.
13. RODIN Community, *RODIN toolset*, `http://sourceforge.net/projects/rodin-b-sharp`.
14. Sparx Systems, *Enterprise Architect*, `http://www.sparxsystems.com.au`.
15. J. M. Spivey, *The z notation: a reference manual*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

16. Tasktop Technologies, *Eclipse Mylyn Project*, `http://www.eclipse.org/mylyn`.
17. Tigris Technologies, *ArgoUML*, `http://http://argouml.tigris.org`.
18. Fangjun Wu and Tong Yi, *Measuring z specifications*, SIGSOFT Softw. Eng. Notes **29** (2004), no. 5, 1–5.
19. Jürgen Wüst, *SD Metrics*, `http://www.sdmetrics.com`.

# Towards Event-B Specification Metrics

Marta (Pląska) Olszewska and Kaisa Sere

Åbo Akademi University, Finland
{Marta.Plaska, Kaisa.Sere}@abo.fi

**Abstract.** In this paper we describe our ongoing research on Event-B specification metrics. We focus on the physical features of specification, such as its vocabulary or length for the Event-B machines. We base our metrics on the syntactic properties of the Event-B language, namely operators and operands that we consider meaningful for our measurement model. Presented metrics are applied for a number of Event-B machines. Obtained results can be analysed in a perspective of an abstract machine and its refinements.

**Keywords:** Event-B, specification metrics, size, direct measurements, quantitative measurements, Halstead model, complexity.

## 1 Introduction

Measurements for software systems and their development process are nowadays a good practice in the computer world [1] [2]. They are a part of software projects in order to assure certain quality [3] or even for standardisation purposes. They have been evolved for many years, extended for particular development methods like Object-Oriented programming [4] [5] or narrowed down to meet the needs of specific programming languages, like Java Programming Language [6]. Quality measurements are done not only at the end-product stage, but much earlier, for the requirements [7] or (formal) model of the system [8]. Early quality assessment has major influence on the final product, as there is a thorough control over the whole development process.

Research on metrics for formal specifications has already been done for the Z language. In [9] authors perform static analysis of Z specification notation, whereas in [10] the focus is on the linguistic properties of the notation and predicting erroneous parts of specifications created in Z. There has also been done an assessment for B language [11], where existing metrics concerned mostly traceability and safety analyses, proof related metrics and direct statistics, like number of LOC (lines of code), variables in a component or imported components. However, it has been observed that there is still a need for metrics in the early phase of the development [12]. The primary users of our metrics are anticipated to be managers, who would be able to analyse the project in its initial stage. Obtained information would then serve as a benchmark of how the current approach influences the specification statistics and, ideally, allow gathering experience on process of developing such specification. The development team could also benefit from metrics, which could be present already when creating a specification and assist developers with specification's analysis later on. To the best of our knowledge only direct statistics, like number of LOC, automatic and interactive proofs, invariants, theorems, refinement steps and the like, are gathered. It would be valuable to provide more elaborated metrics that would help specification analysis and assessment and in the future possibly deal with more involved characteristics, e.g. complexity of the specification or effort prediction.

Event-B is a formal method and a specification language, which is used for system-level modelling and analysis [13]. It is supported by a tool called Rodin Platform, which is an Integrated Development Environment based on Eclipse framework. An Event-B specification consists of a *machine* and its *context* that depict the dynamic and the static part of the specification, respectively. The dynamic model defines the state variables, as well as the operations on these. The context, on the other hand, contains the sets and constants of the model with their properties and is accessed by the machine through the SEES relationship [14]. At this point of our research on metrics we concentrate on the machine only, because we consider machine measurements as a first step towards establishing global Event-B specification metrics.

Since Rodin Platform enables users to obtain $\LaTeX$ version of the Event-B specifications, herein machines, we can process each of such files in an automated way. We have implemented a script [15], which parses each of the machine $\LaTeX$ file and gathers the statistics about its contents. We perform data collection from the *variables* and *events* sections of the machine, as these are crucial for obtaining the information for our metrics. When the metrics are fully developed, they will be incorporated in a Rodin metrics plugin.

In our work we focus on the syntactical properties of the specification at the source code level. We benefit from the existing code metrics and incorporate them to our early stage development measurements. We derive from Halstead's metrics [16], which describe a program as a collection of tokens that can be classified as either operators or operands. These metrics are used to determine a quantitative measure, e.g. program length, vocabulary size, program volume or complexity directly from the source code [17]. Empirical studies show that standard Halstead metric is a good estimate for program length, provided that the data needed for the equations are available. This means that the program should be (almost) completed, which seems to be a downside of this metric that we are aware of. However, it can be useful in gathering the information about a size of a program after the coding process as well. It should also be mentioned that there have been strong debates and criticism on the methodology and the derivations of equations [18]. We think that with certain degree of modification this metric is worth conducting experiments and could demonstrate to be useful in the domain of (formal) specifications. To our knowledge, no experimentation with Halstead metrics for formal specifications has been done yet.

We use the objectives of the Halstead model and carefully adjust them to Event-B language specifics. Our motivation is that the Event-B syntax is appropriate to be experimented with in terms of Halstead metrics, as it contains all primitives needed for further computations. We believe that experimenting with syntactical metrics originating from a programming language will occur to be successful in Event-B case and the results of this investigation will be meaningful.

Our paper is structured as follows. In Section 2, we describe our concept of metrics for Event-B machines. Then we illustrate it with an example in Section 3. We conclude and present plans for the continuation our work in Section 4.

## 2 Metrics for Event-B specification

We derive our metric for Event-B machines from the Halstead model [16], which we now shortly depict. The model is constructed based on a collection of tokens: operators and operands, where four primitive measures can be distinguished:

- $n_1$ - number of distinct operators in a program,
- $n_2$ - number of distinct operands in a program,
- $N_1$ - number of operator occurrences,

– $N_2$ - number of operand occurrences.

These primitive measures are a foundation of Halstead model, which consists of equations expressing the vocabulary, overall program length, potential minimum volume for an algorithm and the difficulty level, which indicates software complexity. Moreover, program difficulty, as well as other features like development effort can be specified with given primitive measures. One needs to take into account that accuracy and behaviour of this model varied between its application domains.

At this point of our research we consider only the machine part of the Event-B specification, focusing on variables and events blocks. Although we presume that refinement has an impact on the qualitative and quantitative aspects of the specification, currently, for the simplicity reasons, we do not take it into consideration. It is planned as a part of our future work.

In order to be able to adjust Halstead model to Event-B environment we have to make several assumptions. Firstly, we decide upon meaningfulness of operators [19] [20]. As an operator we consider unary operators, binary operators except functions, range operator (symbol ..), forward composition (symbol ;), parallel product (symbol ||) and direct product (symbol $\otimes$). We also consider quantifiers, except separators for set comprehension and bounded qualification (symbols | and . respectively), to be operators in our model. The full list of language operators can be found in the language description [21]. We gather the data about the number of distinct operators ($n_1$), as well as the number of their occurrences ($N_1$).

Secondly, we determine the operands, which we chose to be witnesses and variables [21]. We count the number of unique operands ($n_2$) and the number of their appearances ($N_2$) respecting their visibility rules. If a witness name appears in several events in a single machine, each of such occurrences is distinct due to the scope. Witnesses are visible only inside a single event, whereas variables are global for the machine. Having defined primitive measures, we identify several metrics for specification's machine measurements and list them after their description. It is worth mentioning that these metrics do not depend on text formatting, like in a case of using LOC as a machine size metric. They are more credible, as the primitive measures are clearly defined. For comparison in case of LOC it might not be obvious whether to include comments or data definitions to the size computations [22] [23], unless well defined.

We find the metrics of Halstead [16] suitable for our case. The size of a machine's *vocabulary* ($n$) is defined as a sum of distinct operators and operands (1). A machine's *size* ($N$) is a sum of operator and operand occurrences (2). Next metric, a machine's *volume* ($V$), represents the information contents of the program, i.e. the size of the code of a machine. The calculation of $V$ is based on the number of operations and operands present in the machine (3). Another metric, *difficulty level D*, representing the difficulty experienced during writing a specification, is proportional to the number of distinct operators $n_1$ and occurrences of operands $N_2$, and inversely proportional to the number of distinct operands $n_2$ (4). Other Halstead metrics are anticipated in our future work.

$$n = n_1 + n_2 \tag{1}$$

$$N = N_1 + N_2 \tag{2}$$

$$V = N \log_2 n \tag{3}$$

$$D = \frac{n_1 N_2}{2 n_2} \tag{4}$$

## 3 Experimental setup and preliminary results

We apply presented metrics to an abstract machine and its subsequent refinements in order to perform comparative study and evaluation of the obtained results. This enables us to quantitatively assess the machine development from its physical characteristics point of view.

Our methodology was applied to a number of specifications created by Space Systems Finland within the European Project DEPLOY. We performed a single domain experiment, with the same type of the development and the same staff. This made the investigation more credible, as we reduced the number of experiment variables that could skew the overall results. Table 1 presents the data obtained from BepiColombo specification Version 5.0, uploaded by SSF to the BSCW repository on the 2nd May, 2009. The analysis of gathered statistics considers also the number of LOC, which we defined as non empty and non-comment lines of code. The hyphen in the $D$ row indicates that there were only *skip*

**Table 1.** SSF BepiColombo Specification, Version 5.0 measurements

| Metric | M00 | M01 | M02 | M03 | M04 | M05 | M06 | M07 |
|---|---|---|---|---|---|---|---|---|
| $n_1$ | 2 | 5 | 0 | 3 | 4 | 0 | 5 | 9 |
| $N_1$ | 7 | 69 | 0 | 7 | 11 | 0 | 44 | 22 |
| $n_2$ | 19 | 48 | 13 | 21 | 21 | 18 | 85 | 31 |
| $N_2$ | 89 | 360 | 30 | 104 | 61 | 28 | 419 | 116 |
| $n$ | 21 | 53 | 13 | 24 | 25 | 18 | 90 | 40 |
| $N$ | 96 | 429 | 30 | 111 | 72 | 28 | 463 | 138 |
| $V$ | 422 | 2457 | 111 | 509 | 334 | 117 | 3006 | 734 |
| $D$ | 5 | 19 | - | 7 | 6 | - | 12 | 17 |
| *LOC* | 231 | 758 | 455 | 599 | 538 | 477 | 1132 | 783 |

events in the corresponding machine, which arithmetically means that the result of the computation was not a number.

There is a strong correlation between the vocabulary size $n$ and length $N$ of each machine, where $n$ is always less or equal than $N$. This is a direct consequence of the $n$ and $N$ definitions. Moreover, a positive relationship of $n$ and $N$ concerning other statistics, like number of events (also actions) can be observed. There is a strong correlation between *volume* and formerly mentioned machine *size* metrics ($N$, $n$, LOC), which is particularly visible when logarithmic scale is used in the diagrams. Therefore $V$ could be used as a meaningful size metric.

The *difficulty level* characteristic is proportional to the number of unique operators in the program, so also to *vocabulary size*. There seems to be a correlation between $D$ and Interactive Proof Obligations, however it is not strong enough to be a complete basis for a quality prediction model. Therefore, it needs to be further investigated.

The Halstead metrics for Event-B specifications have to be verified on more examples. We will pursue the search for possible relations with other indicators, e.g. number of requirements covered.

## 4 Conclusions and future work directions

Our metrics for Event-B should provide better understanding of specifications' physical features. They could facilitate early-stage development analysis of an abstract specification and its consecutive refinements. Such metrics could be used as a basis for further, indirect measurements, like predictions of time and human resources necessary for a development.

In our paper we proposed a quantitative approach to assess Event-B specifications. We described several metrics for the machine part of a specification adjusted to the specifics of the Event-B language. Moreover, we have implemented a script for automatic data collection and ran it against existing specification.

As a continuation of our research we plan to check the validity of presented metrics on a bigger number of machines (specifications) and discuss the obtained results with the designers. We want to modify the models of the machine metrics if needed, i.e. when not fully expressing the results or not entirely reflecting the developer's intuition. We also need to create metrics for the context part of the specification. This would enable us to create a set of global metrics, which take under consideration complete Event-B specification, i.e. both context and machine, so that it can be analysed as a whole. We also need to incorporate refinement into our model in order to analyse the accumulated Event-B developments, which can be done e.g. using the Breiman's random forest theory [24].

Since software is becoming larger and more complex nowadays, the metrics we proposed could be used for building a complexity model for Event-B specification. Handling complexity already at the beginning of the project would benefit not only the design process as such, but also later on, e.g. in programming or maintenance phases. It would also serve as early development data to managers.

## Acknowledgments

This work has been done within the EC FP7 Integrated Project Deploy (grant no. 214158).

## References

1. P. Goodman. *Software Metrics: Best Practices for Successful IT Management.* Rothstein Associates Inc., 2004
2. A. Gopal, M.S. Krishnan, T. Mukhopadhyay, D. Goldenson. *Measurement Programs in Software Development: Determinants of Success.* IEEE Transactions on Software Engineering, Vol. 28, No. 9, 2002
3. R. Kandt. *Software Engineering Quality Practices.* Auerbach Publications, 2005
4. M. Lanza, R. Marinescu, S. Ducasse. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems.* Springer, 2006

5. R.C. Martin. *OO Design Quality Metrics. An Analysis of Dependencies.*. 1994
6. Metrics *1.3.6*, *http://metrics.sourceforge.net/*
7. J. Robertson, S. Robertson. *Requirements: Made to Measure.* American Programmer, X, 1997
8. A. Tang, M.H. Tran, J. Han, H. van Vliet. *Design Reasoning Improves Software Design Quality, Quality of Software Architectures. Models and Architectures.*. Springer, Heidelberg, 2008
9. I.J. Hayes, B.E. Mahony. *Using Units of Measurement in Formal Specifications.* Formal Aspects of Computing, 7, pp.329-347, 1995
10. R. Vinter, M. Loomes, D. Kornbrot. *Applying Software Metrics to Formal Specifications: A Cognitive Approach.* IEEE International Symposium on Software Metrics, pp.216, 1998
11. E.M. El Koursi, G. Mariano. *Assessment and certification of safety critical software.* Proceedings of the 5th Biannual World Automation Congress, IEEE, Orlando, 2002
12. R.W. Whitty. *Research in Specification Methods.* IEEE Colloquium on Software Metrics, 2002
13. Event-B.org. *http://www.event-b.org/index.html*
14. J.R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996
15. M. Olszewska, M. Olszewski. *http://valhalla.cs.abo.fi/ mplaska/btexcount.rb*
16. M.H. Halstead. *Elements of Software Science.* Elsevier North Holland, pp.128., 1977
17. S. Kan. textslMetrics and Models in Software Quality Engineering. Addison–Wesley, 2003
18. P.G. Hamer, G. Frewin. *M. H. Halstead's Software Science – A Critical Examination.* IEEE Proceedings of 6th International Conference on Software Engineering, ICSE., Tokyo, 1982
19. M. Jorgensen. *Software Quality Measurement.* Advances in Engineering Software, 30, pp.907-912., 1999
20. *User Manual of the RODIN Platform, Version 2.3.* 2007
21. C. Metayer, J.R. Abrial, L. Voisin. *Event-B Language, RODIN Deliverable 3.2 (D7).* 2005
22. N. Fenton, S. Pflegger. *Software Metrics. A Rigorous and Practical Approach.* PWS Publishing Company, 1997
23. N. Fenton, M. Neil. *Software metrics: roadmap.* Conference on the Future of Software Engineering , Limerick, Ireland, 2000
24. L. Breiman. *Random Forests*, Machine Learning, Volume 45, Number 1, pp.5-32., October 2001

# Part IV

# Model Checking

# Proof Assisted Model Checking for B[⋆]

Jens Bendisposto and Michael Leuschel

Institut für Informatik, Heinrich-Heine Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
{bendisposto, leuschel}@cs.uni-duesseldorf.de

**Abstract.** With the aid of the ProB Plugin, the Rodin Platform provides an integrated environment for editing, proving, animating and model checking Event-B models. This is of considerable benefit to the modeler, as it allows him to switch between the various tools to validate, debug and improve his or her models. The crucial idea of this paper is that the integrated platform also provides benefits to the tool developer, i.e., it allows easy access to information from other tools. Indeed, there has been considerable interest in combining model checking, proving and testing. In previous work we have already shown how a model checker can be used to complement the Event-B proving environment, by acting as a disprover. In this paper we show how the prover can help improve the efficiency of the animator and model checker.

**Keywords:** Model Checking, B-Method, Theorem Proving, Experiment, Tool Integration.

## 1 Introduction

There has been considerable interest in combining model checking, proving and testing (e.g., [16, 17, 7, 19, 4, 8]). The Rodin platform for the formal Event-B notation provides an ideal framework for integrating these techniques. Indeed, Rodin is based on the extensible Eclipse platform and as such it is easy for provers, model checkers and other arbitrary tools to interact. In this paper we make use of this feature of Rodin to improve the ProB [12, 13] model checking algorithm by using information provided by the various Rodin provers.

More concretely, in this paper we show how we can optimize the *consistency checking* of Event-B and B models, i.e., checking whether the invariants of the model hold in all reachable states. The key insight is that from the proof information we can deduce that certain events are guaranteed to preserve the correctness of specific parts of the invariant.

⋆ This research is being carried out as part of the DFG funded research project GEPAVAS and the EU funded FP7 research project 214158: DEPLOY (Industrial deployment of advanced system engineering methods for high productivity and dependability).

By keeping track of which events lead to which states, we can avoid having to check a (sometimes considerable) amount of invariants.

The paper is structured as follows. In Section 2 we introduce the Event-B formal method and the Rodin platform, while in Section 3 we provide background about consistency checking and the PROB model checker, which itself already employs a combination of model checking and constraint solving techniques. In Section 4 we explain our approach to using proof information for optimizing the process of checking invariants in the PROB model checker, and present an improved model checking algorithm. Section 5 introduces a fully proven formal model of our approach. In Section 6 we evaluate our approach on a series of case studies, drawn from the Deploy project. The experiments show that there can be considerable benefit from exploiting proof information during model checking. In Section 7 we discuss how our method can be used in the context of classical B without easy access to proof information. We conclude with related work and discussions in Section 8.

## 2 Event-B and Rodin

Event-B is a formal method for state-based system modeling and analysis evolved from the B-method [1]. The B-method itself is derived from Z and based upon predicate logic combined with set theory and arithmetic, and provides several sophisticated data structures (sets, sequences, relations, higher-order functions) and operations on them (set union, intersection, relational composition, relational image, to name but a few).

An Event-B development consists of two types of artifacts: contexts and machines. The static properties are expressed in contexts, the dynamic properties of a system are specified in machines. A context contains definitions of carrier sets, constants as well as a set of axioms. A machine basically consists of finite sets of variables $v$ and a finite set of events. The variables form the state of the machine, they are restricted and given a type by an invariant. The events describe transitions from one state into another state. An event has the form:

$$\text{event} \mathrel{\widehat{=}} \text{ANY } t \text{ WHERE } G(v,t) \text{ THEN } S(v,t) \text{ END}$$

It consists of a set of local variables $t$, a predicate $G(v,t)$, called the guard and a substitution $S(v,t)$. The guard restricts possible values for $t$ and $v$.

If the guard of an event is false, the event cannot occur and it is called disabled. The substitution $S$ modifies some of the variables in $v$, it can use the old values of $v$ and the local variables $t$. For instance, an event that chooses two natural numbers $a, b$ and adds their product $ab$ to the state variable $x \in v$ could be written as

$$\text{evt1} \mathrel{\widehat{=}} \text{ANY } a, b \text{ WHERE } a \in \mathbb{N} \wedge b \in \mathbb{N} \text{ THEN } x := x + ab \text{ END}$$

The Rodin tool [2] was developed within the EU funded project RODIN and is an open platform for Event-B. The Rodin core puts emphasis on mathematical proof of models, while other plug-ins allow, for instance, UML-like editing, animation or model checking. The platform interactively checks a model, generates and discharges proof obligations for Event-B. These proof obligations deal with different aspects of the correctness of a model. In this paper we only deal with proofs that are related to invariant preservation, i.e., if the invariant holds in a state and we observe an event, the invariant still holds in the successor state:

$$I(v) \wedge G(v,t) \wedge S_{BA}(v,t,v') \Longrightarrow I(v')$$

By $S_{BA}(v,t,v')$ we mean the substitution S expressed as a Before-After predicate. The primed variables refer to the state after the event happened, the unprimed variables to the state before the event happened. In our small example, $S_{BA}(v,t,v')$ is the predicate $x' = x + ab$. If we want to express, that $x$ is a positive integer, i.e. $x \in \mathbb{N}_1$, we need to prove:

$$x \in \mathbb{N}_1 \wedge a \in \mathbb{N} \wedge b \in \mathbb{N} \wedge x' = x + ab \Longrightarrow x' \in \mathbb{N}_1$$

This implication is obviously very easy to prove, in particular, it is possible to automatically discharge this obligation using the Rodin tool.

For each pair of invariant and event the Rodin Proof Obligation Generator, generates a proof obligation (PO) that needs to be discharged in order to prove correctness of a model as mentioned before. A reasonable number of these POs are discharged fully automatically by the tool. If an obligation is discharged, we know that if we observe an event and the invariant was valid before, then it will be valid afterwards. Before generating proof obligations, Rodin statically checks the model. Because this also includes type checking, the platform can eliminate a number of proof obligations that deal with typing only. For instance the invariant $x \in \mathbb{Z}$ does not give rise to any proof obligation, its correctness is guaranteed by the type checker.

The propagation and exploitation of this kind of proof information to help the model checker is the key concept of the combination of proving and model checking presented in this paper.

## 3 Consistency checking and ProB

PROB [12, 13] is an animator for B and Event-B built in Prolog using constraint-solving technology. It incorporates optimizations such as symmetry reduction (see, e.g., [20]) and has been successfully applied to several industrial case studies such as a cruise control system [12], parts of the Nokia Mobile Internet Technical Architecture (MITA) and the most recent one: the application of PROB to verify the properties of the San Juan Metro System deployment [14].

One core application of PROB is the *consistency checking* of a B model, i.e., checking whether the invariant of a B machine is satisfied in all initial states and whether the invariant is preserved by the operations of the machine. PROB achieves this by computing the *state space* of a B model, by

– computing all possible initializations of a model and
– by computing for every state all possible ways to enable events and computing the effects of these events (i.e., computing all possible successor states).

Graphically, the state space of a B model looks like in Figure 1. Note that the initial states are represented as successor states of a special root node.

PROB then checks the invariant for every state in the state space. (Note that PROB can also check assertions, deadlock absence and full LTL properties [15].)

Another interesting aspect is that PROB uses a mixture of depth-first and breadth-first evaluation of the state space, which can lead to considerable performance improvements in practice [11].

## 4 Proof-Supported Consistency Checking

The status of a proof obligation carries valuable information for other tools, such as a model checker. As described, PROB does an exhaustive

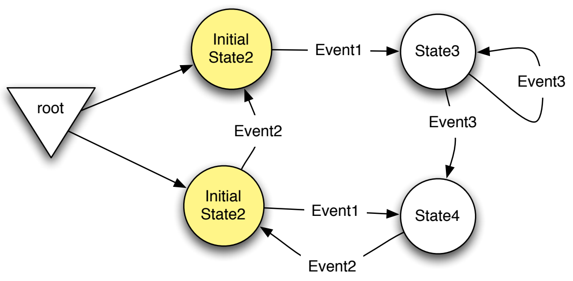


**Fig. 1.** A simple state space with four states

search, i.e. it traverses the state space and verifies that the invariant is preserved in each state. This section describes how we incorporate proof information from Rodin into the ProB core.

Assuming we have a model, that contains the invariant $[I_1, I_2, I_3]$[1] and we follow an event *evt* to a new state. If we would, for instance, know that *evt* preserves $I_1$ and $I_3$, there would be no need to check these invariants. This kind of knowledge, which is precisely what we get from a prover, can potentially reduce the cost of invariant verification during the model checking.

The ProB plug-in translates a Rodin development, consisting of the model itself, its abstractions and all necessary contexts into a representation used by ProB. We evolved this translation process to also incorporate proof information, i.e., our representation contains a list of tuples $(E_i, I_j)$ of all discharged POs, that is event $E_i$ preserves invariant $I_j$.

Using all this information, we determine an individual invariant for each event that is defined in the machine. Because we only remove proven conjuncts, this specialized invariant is a subset of the model's invariant. When encountering a new state, we can evaluate the specialized invariant rather than the machine's full invariant.

As an example we can use the Event-B model shown in Figure 2. The full state space of this model and the proof status delivered by the automatic provers of the Rodin tool are shown in Figure 3.

The proof status at the right shows, that Rodin is able to discharge the proof obligations $a/inv1$ and $b/inv2$ but not $a/inv2$ and $b/inv1$. This

[1] Sometimes it is handier to use a list of predicates rather than a single predicate, we use both notations equivalently. If we write $[P_1, P_2, \ldots, P_n]$, we mean the predicate $P_1 \wedge P2 \wedge \ldots \wedge P_n$.

**VARIABLES**
    $\mathtt{f,x}$
**INVARIANTS**
    inv1 : $\mathtt{f} \in \mathbb{N} \mathrel{+\!\!\!\rightarrow} \mathbb{N}$
    inv2 : $\mathtt{x} > 3$
**EVENTS**
**Initialisation**
$\mathtt{f} := \{1 \mapsto 100\} || \mathtt{x} := 10$
**Event** $a \mathrel{\widehat{=}}$
$\mathtt{f} := \{1 \mapsto 100\} || \mathtt{x} := \mathtt{f}(1)$
**Event** $b \mathrel{\widehat{=}}$
$\mathtt{f} := \mathtt{f} \cup \{1 \mapsto 100\} || \mathtt{x} := 100$

**Fig. 2.** Example for intersection of invariants

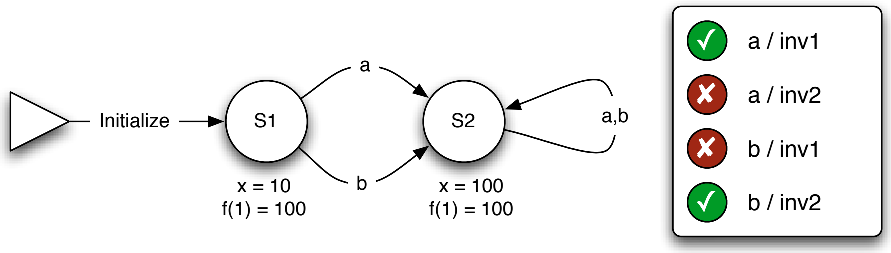


**Fig. 3.** State space of the model in figure 2

means, if $a$ occurs, we can be sure that $f \in \mathbb{N} \mathrel{+\!\!\!\rightarrow} \mathbb{N}$ holds in the successor state if it holds in the predecessor state. Analogously, we know, that if $b$ occurs, we are sure, that $x > 3$ holds in the successor state if it holds in the predecessor state.

Consider a situation, where we already verified that all invariants hold for $S1$ and we are about to check $S2$ is consistent. We discovered two incoming transitions corresponding to the events $a$ and $b$. From $a$, we can deduct that $f \in \mathbb{N} \mathrel{+\!\!\!\rightarrow} \mathbb{N}$ holds. From $b$, we know that $x > 3$ holds. To verify $S2$, we need to check the intersection of unproven invariants, i.e., $\{f \in \mathbb{N} \mathrel{+\!\!\!\rightarrow} \mathbb{N}\} \cap \{x > 3\} = \varnothing$, thus we already know that all invariants hold for $S2$.

This is of course only a tiny example but it demonstrates, that using proof information we are able to reduce the number of invariants for each event significantly, and sometimes by combining proof information from different events, we are able to get rid of the whole invariant. We actually have evidence that this is not only a theoretical possibility, but happens in real world specifications (see Section 6).

**Algorithm 4.1**[*Proof-Supported Consistency Checking* ]

```
Input: An Event-B model with invariant $I = inv_1 \wedge \ldots \wedge inv_n$
$Queue$ := $\{root\}$ ; $Visited$ := $\{\}$; $Graph$ := $\{\}$
$\underline{\textbf{for all} \text{ events } evt \textbf{ do } Unproven(evt) := \{inv_i \mid inv_i \text{ not proven for } evt\}; \textbf{ end do}}$
while $Queue$ is not empty do
 if random(1) < $\alpha$ then
  $state$ := pop_from_front($Queue$); /* depth-first */
 else
  $state$ := pop_from_end($Queue$); /* breadth-first */
 end if
 if $\underline{\exists inv_i \in Inv(state) \text{ s.t. } inv_i \text{ is false}}$ then
  return counter-example trace in $Graph$
       from $root$ to $state$
 else
  for all $succ$,$evt$ such that $state \rightarrow_{evt} succ$ do
   $Graph$ := $Graph \cup \{state \rightarrow_{evt} succ\}$
    if $succ \notin Visited$ then
      push_to_front($succ$, $Queue$);
      $Visited := Visited \cup \{succ\}$
      $\underline{Inv(succ) := Unproven(evt)}$
    else
      $\underline{Inv(succ) := Inv(succ) \cap Unproven(evt)}$
    end if
   end if
  end for
od
return ok
```

Algorithm 4.1 describes PROB's consistency checking algorithm, we will justify it formally in section 5. The algorithm employs a standard queue data structure to store the unexplored nodes. The key operations are:

- Computing the successor states, i.e., "$state \rightarrow_{evt} succ$".
- Verification of the invariant "$\exists inv_i \in Inv(state)$ *s.t.* $inv_i$ *is false*"
- Determining whether "$succ \notin$ Visited"

The algorithm terminates when there are no further queued states to explore or when an error state is discovered. The underlined parts highlight the important differences with the algorithm in [13].

In contrast to the algorithm, the actual implementation does the calculation of the intersection ($Inv(succ) := Inv(succ) \cap Unproven(op)$) in a lazy manner, i.e., for each $state \notin Visited$, we store the event names as a list. As soon as we evaluate the invariant of a state, we calculate and evaluate the intersection on the fly. The reason is, that storing the invariant's predicate for each state is typically more expensive than storing the event names.

## 5 Verification

To show, that our approach is indeed correct, we developed a formal model of an abstraction of algorithm 4.1. We omitted few technical details, such as the way the state space is traversed by the actual implementation and also we omitted the fact, that our implementation always uses all available information. Instead, we have proven correctness for any traversal and any subset of the available information. Our model was developed using Event-B and fully proven in Rodin. The model is available as a Rodin 1.0 archive from
`http://www.stups.uni-duesseldorf.de/models/pomc_paper.zip`.
In this paper we present only some parts of the model and some lemmas, without their proofs. All the proofs can be found in the file, we thus refer the reader to the Rodin model.

We used three carrier sets *STATES*, *INVARIANTS* and *EVENTS*. We assume, that these sets are finite. For invariants and events this is true by definition in Event-B, but the state space can in general be unbounded. However, the assumption of only dealing with finite state spaces is reasonable in the context of our particular model, because we can interpret the *STATES* set as the subset of all states that can be traversed by the model checker within some finite number of steps.[2] The following definitions are used to prove some properties of Event-B:

$truth \subseteq STATES \times INVARIANTS$
$trans \subseteq STATES \times STATES$
$preserve = \{s \mid \{s\} \times INVARIANTS \subseteq truth\}$
$violate = STATES \setminus preserve$
$label \subseteq trans \times EVENTS$

[2] Alternatively, we can remove this assumption from our Rodin models. This only means that we are not be able to prove termination of our algorithm; all other invariants and proofs remain unchanged.

$discharged \subseteq EVENTS \times INVARIANTS$

The model also contains a set *truth*: pair of a state $s$ and an invariant $i$ is in *truth* if and only if $i$ holds in $s$. The set *preserve* is defined as the set of states where each invariant holds, the relations *trans* and *label* describe, how two states are related, i.e. a triple $(s \mapsto t) \mapsto e$ is in *label* (and therefore $s \mapsto t \in trans$) if and only if $t$ can be reached from $s$ by executing $e$. The observation that is the foundation of all theorems we proved and is the following assumption:

$$\forall i,t \cdot (\exists s,e \cdot s \in preserve \land (s \mapsto t) \in trans \land$$
$$(s \mapsto t) \mapsto e \in label \land (e \mapsto i) \in discharged)$$
$$\Rightarrow (t \mapsto i) \in truth$$

The assumption is, that if we reach a state $t$ from a state $s$ where all invariants hold by executing an event $e$ and we know, that the invariant $i$ is preserved by $e$, we an be sure, that $i$ holds in $t$. This statement is what we prove by discharging an invariant proof obligation in Event-B, thus it is reasonable to assume that it holds.

We are now able to prove a lemma, that will capture the essence of our proposal; it is enough to find for each invariant $i$ one event that preserves this invariant leading from a consistent state into a state $t$ to prove, that all invariants hold in $t$.

**Lemma 1.** $\forall t \cdot t \in STATES \land (\forall i \cdot i \in INVARIANTS \land (\exists s,e \cdot s \in preserve \land e \in EVENTS \land (s \mapsto t) \in trans \land (s \mapsto t) \mapsto e \in label \land e \mapsto i \in discharged)) \Rightarrow t \in preserve$

*Proof.* All proofs have been done using Rodin and can be found in the model archive. $\square$

We used five refinement steps to prove correctness of our algorithm. We will describe the first three steps, the last two steps are introduced to prove termination of new events. The first refinement step *mc0* contains two events *check_state_ok* and *check_state_broken*. The events take a yet unprocessed state and move it either into a set containing consistent or inconsistent states. Algorithm 5.1 shows the *check_state_ok* event, *check_state_broken* is defined analogously, except that it has the guard $s \notin preserve$ and it puts the state into the set *inv_broken*.

**Algorithm 5.1**[Event *check_state_ok* from *mc0*]

```
event check_state_ok
  any s
  where
    $s \in open$
    $s \in preserve$
  then
    $inv\_ok := inv\_ok \cup \{s\}$
    $open := open \setminus \{s\}$
end
```

At this very abstract level this machine specifies that our algorithm separates the states into two sets. If they belong to *preserve*, the states are moved into the set *inv_ok*. Otherwise, they are moved into *inv_broken*. Lemma 2 guarantees, that our model always generate correct results.

**Lemma 2.** *mc0 satisfies the invariants*

1. $inv\_ok \cup inv\_broken = STATES \setminus open$
2. $open = \varnothing \Rightarrow inv\_ok = preserve \wedge inv\_broken = violate$

The next refinement strengthens the guard and removes the explicit knowledge of the sets *preserve* and *violate*, the resulting proof obligation leads to lemma 3.

**Lemma 3.** *For all* $s \in open$

$$\{s\} \times INVARIANTS \setminus discharged[label[inv\_ok \lhd trans \rhd \{s\}]]) \subseteq truth$$
$$\Leftrightarrow s \in preserve$$

The third refinement introduces the algorithm. We introduce a new relation *invs_to_verify* in this refinement. The relation keeps track of those invariants, that need to be checked, in the initialization, we set $invs\_to\_verify := STATES \times INVARIANTS$.

The algorithm has three different phases. It first selects a state that has not been processed yet then it checks if the invariant holds and moves the state into either *inv_ok* or *inv_broken*. Finally, it uses the information about discharged proofs to remove some elements from *invs_to_verify* as shown in algorithm 5.2.

**Algorithm 5.2**[Event *mark_successor* from *mc2*]

```
event mark_successor
  any p s e
  where
    $p \in inv\_ok$
    $s \in trans[\{p\}]$
    $(p \mapsto s) \mapsto e \in label$
    $(p \mapsto s) \mapsto e \notin marked$
    $ctrl = mark$
  then
    $invs\_to\_verify := invs\_to\_verify \Lleftarrow (\{s\} \times (invs\_to\_verify[\{s\}] \cap unproven[\{e\}]))$
    $marked := marked \cup \{(p \mapsto s) \mapsto e\}$
end
```

We take some state $s$ and event $e$, where we know that $s$ is reachable via $e$ from a state $p$, where all invariants hold. Then we remove all invariants but those, that are not proven to be preserved by $e$. This corresponds to the calculation of the intersection in algorithm 4.1.

The main differences between the formal model and our implementation are, that the model does not explicitly describe how the states are chosen and the algorithm uses all available proof information while the formal model can use any subset. In addition, the model does not stop if it detects an invariant violation. We did not specify these details because it causes technical difficulties (e.g., we need the transitive closure of the *trans* relation) but does not seem to provide enough extra benefit.

Correctness of algorithm 4.1 is established by the fact that the outgoing edges of a *state* are added to the *Graph* only *after* the invariants have been checked for *state*. Hence, the removal of a preserved invariant only occurs *after* it has been established that the invariant is true before applying the event. This corresponds to the guard $p \in inv\_ok$. However, the proven proof obligations for an event only guarantee *preservation* of a particular invariant, not that this invariant is established by the event. Hence, if the invariant is false before applying the event, it could be false after the event, *even* if the corresponding proof obligation is proven and true. If one is not careful, one could easily set up cyclic dependencies and our algorithm would incorrectly infer that an incorrect model is correct.

## 6 Experimental results

To verify that the combination of proving and model checking results in a considerable reduction of model checking effort, we prepared an experiment consisting of specifications we got from academia and industry. In addition we prepared a constructed example as one case, where the prover has a very high impact on the performance of the model checker. The rest of this section describes how we carried out the measurement. We will also briefly introduce the models and discuss the result for each of them. The experiment contains models where we expected to have a reasonable reduction and models where we expected to have only a minor or no impact.

## 6.1 Measurement

The latest development versions of ProB can do consistency checking of a refinement chain. Previous versions of ProB checked a specific refinement level only and removed all gluing invariants. We carried out both, single refinement level and multiple refinement level checks. The results have been gathered using a Mac Book Pro, 2.4 GHz Intel Core 2 Duo Computer with 4 GB RAM running Mac OS X 10.5. For the single level animation, we collected 40 samples for each model and calculated the average and standard deviation of the times measured in milliseconds. For the multi level animation, we collected 5 samples for each model. The result of the experiment is shown in tables 1, 2 and 3. The absolute values of tables 1 and 2 are very difficult to compare, because we used different versions of ProB.

Except for the case of the Siemens specification, we removed all interactive proofs from the models and used only those proof information, that Rodin was able to automatically generate using default settings. In the case of the Siemens model, we used both, a version with automatic proofs only and a development version with few additional interactive proofs; the development version was not fully proven.

## 6.2 Mondex

The mechanical verification of the Mondex Electronic Purse was proposed for the repository of the verification grand challenge in 2006. We use an Event-B model developed at the University of Southampton. We have chosen two refinements from the model, m2 and m3. The refinement m2 is a rather big development step while the second refinement m3 was used to prove convergence of some events introduced in m2, in particular, m3 only contains gluing invariants.

In case of single refinement level checking, it is obvious that it is not possible to further simplify the invariant of m3 but we noticed, that we do not even lose performance caused by the additional specialization of the invariants. This is important because it is evidence, that our implementation's performance is in the order of the standard deviation in our measurement. For the case of m2, where we have machine invariants, we measured a reduction of about 12%.

In case of multiple refinement level checking, we have the only case, where we lost a bit of performance for m2. However, the absolute value is in the order of the standard deviation. For m3 we also did not get significant improvements of performance, most likely because the gluing invariant is very simple, actually it only contains simple equalities.

## 6.3 Siemens Deploy Mini Pilot

The Siemens Mini Pilot was developed within the Deploy Project. It is a specification of a fault-tolerant automatic train protection system, that ensures that only one train is allowed on a part of a track at a time. The Siemens model shows a very good reduction, as the invariants are rather complex. This model does contain a single machine, thus multi level refinement checking does not affect the speedup.

## 6.4 Scheduler

This model is an Event-B translation of the scheduler from [10]. The model describes a typical scheduler that allows a number of processes to enter a critical section. The experiment has shown, that the improvement using proof information is rather small, which was no surprise. The model has a state space that grows exponential when increasing the number of processes. It is rather cheap to check the invariant

$$ready \cap waiting = \varnothing \wedge active \cap (ready \cup waiting) = \varnothing \wedge active = \varnothing \Rightarrow ready = \varnothing$$

because the number of processes is small compared to the number of states. But nevertheless, we save a small amount of time in each state and these savings can sum up to a reasonable speedup. The scheduler also contains a single level of refinement.

### 6.5 Earley Parser

The model of the Earley parsing algorithm was developed and proven by Abrial. Like in the mondex example, we used two refinement steps that have different purposes. The second refinement step m2 contains a lot of invariants, while the m3 contains only very few of them. This is reflected in the savings we gained from using the proof information in the case of single refinement level checking. While m3 showed practically no improvement, in the m2 model the savings sum up to a reasonable amount of time. In the case of multiple refinement level checking the result are very different, while m2 is not affected, the m3 model benefits a lot. The reason is, that it contains several automatically proven gluing invariants.

### 6.6 SAP Deploy Mini Pilot

Like the Siemens model this is a Deploy pilot project. It is a model of system that coordinates transactions between seller and buyer agents. In the case of single refinement level case, we gain a very good speedup from using proof information, i.e., model checking takes less than half of the time. Like in the Siemens example, the model contains rather complicated invariants. In case of the multi refinement level checking the speedup is still good, but not as impressive as in single refinement level checking.

### 6.7 SSF Deploy Mini Pilot

The Space Systems Finland example is a model of a subsystem used for the ESA BepiColombo mission. The BepiColombo spacecraft will start in 2013 on its journey to Mercury. The model is a specification of parts of the BepiColombo On-Board software, that contains a core software and two subsystems used for tele command and telemetry of the scientific experiments, the Solar Intensity X-ray and particle Spectrometer (SIXS) and the Mercury Imaging X-ray Spectrometer (MIXS). The time for model checking could be reduced by 7% for a single refinement level and by 16% for multiple refinement checking.

### 6.8 Cooperative Crosslayer Congestion Control CXCC

CXCC [18] is a cross-layer approach to prevent congestion in wireless networks. The key concept is that, for each end-to-end connection, an intermediate node may only forward a packet towards the destination after its successor along the route has forwarded the previous one. The information that the successor node has successfully retrieved a package is gained by active listening. The model is described in [**?**]. The invariants used in the model are rather complex and thus we get a good improvement by using the proof information in both cases.

### 6.9 Constructed Example

The constructed example is mainly to show a case, where we get a huge saving from using the proofs. It basically contains an event, that increments a number $x$ and an invariant $\forall a,b,c \cdot a \in \mathbb{N} \wedge b \in \mathbb{N} \wedge c \in \mathbb{N} \Rightarrow (a = a \wedge b = b \wedge c = c \wedge x = x)$. Because the invariant contains the variable modified by the event, we cannot simply remove it. But Rodin can automatically prove that the event preserves the invariant, thus our tool is able to remove the whole invariant. Without proof information, ProB needs to enumerate all possible values for $a$,$b$ and $c$ which results in an expensive calculation.

## 7 Proof-Assisted Consistency Checking for Classical-B

In the setting of Event-B and the Rodin platform, ProB can rely on the other tools for providing type inference and as we have seen the proof information.

In the context of classical B, we are working on a tighter integration with Atelier B [21]. However, at the moment ProB does not have access to the proof information of classical B models.

ProB does perform some additional analyses of the model and annotates the AST (Abstract Syntax Tree) with additional information. For instance for each event we calculate a set of variables that are possibly modified. For instance if we analyze the operation[3]

$$Operation1 = BEGIN\ x := z \mid\mid y := y \wedge \{x \mapsto z\}\ END$$

[3] Operations are the equivalent of events in classical B.

| | **w/o proof information [ms]** | **using proof information [ms]** | **Speedup-Factor** |
|---|---|---|---|
| Mondex m3 | $1454 \pm 5$ | $1453 \pm 5$ | 1.00 |
| Earley Parser m3 | $2803 \pm 8$ | $2776 \pm 7$ | 1.01 |
| Earley Parser m2 | $140310 \pm 93$ | $131045 \pm 86$ | 1.07 |
| SSF | $31242 \pm 64$ | $29304 \pm 44$ | 1.07 |
| Scheduler | $9039 \pm 15$ | $8341 \pm 14$ | 1.08 |
| Mondex m2 | $1863 \pm 7$ | $1665 \pm 6$ | 1.12 |
| Siemens (auto proof) | $54153 \pm 50$ | $25243 \pm 22$ | 2.15 |
| Siemens | $56541 \pm 57$ | $26230 \pm 28$ | 2.16 |
| SAP | $18126 \pm 18$ | $8280 \pm 14$ | 2.19 |
| CXCC | $18198 \pm 21$ | $6874 \pm 12$ | 2.65 |
| Constructed Example | $18396 \pm 26$ | $923 \pm 8$ | 19.93 |

**Table 1.** Experimental results (single refinement level check)

| | **w/o proof information [ms]** | **using proof information [ms]** | **Speedup-Factor** |
|---|---|---|---|
| Mondex m2 | $1747 \pm 21$ | $1767 \pm 38$ | 0.99 |
| Mondex m3 | $1910 \pm 20$ | $1893 \pm 6$ | 1.01 |
| Earley Parser m2 | $309810 \pm 938$ | $292093 \pm 1076$ | 1.06 |
| Scheduler | $9387 \pm 124$ | $8167 \pm 45$ | 1.15 |
| SSF | $35447 \pm 285$ | $30590 \pm 110$ | 1.16 |
| SAP | $50783 \pm 232$ | $34927 \pm 114$ | 1.45 |
| Earley Parser m3 | $7713 \pm 40$ | $5047 \pm 15$ | 1.53 |
| Siemens (auto proof) | $51560 \pm 254$ | $24127 \pm 93$ | 2.14 |
| Siemens | $51533 \pm 297$ | $23677 \pm 117$ | 2.18 |
| CXCC | $18470 \pm 151$ | $6700 \pm 36$ | 2.76 |
| Constructed Example | $18963 \pm 31$ | $967 \pm 6$ | 19.61 |

**Table 2.** Experimental results (multiple refinement level check)

| | **w/o Proof [#]** | **w Proof [#]** | **Savings [%]** |
|---|---|---|---|
| Earley Parser m2 | – | – | - |
| Mondex m3 | 440 | 440 | 0 |
| Earley Parser m3 | 540 | 271 | 50 |
| Constructed Example | 42 | 22 | 50 |
| SAP | 48672 | 16392 | 66 |
| Scheduler | 20924 | 5231 | 75 |
| Mondex m2 | 6600 | 1560 | 76 |
| SSF | 24985 | 5009 | 80 |
| CXCC | 88480 | 15368 | 83 |
| Siemens | 280000 | 10000 | 96 |
| Siemens (auto proof) | 280000 | 10000 | 96 |

**Table 3.** Number of invariants evaluated (single refinement level check).

the analysis will discover that the set of variables that could potentially influence the truth value of the invariant is $\{x,y\}$.

This analysis was originally used to verify the correct usage of `SEES` in the classical B-Method. The `SEES` construct was used in the predecessor of Event-B, so-called classical B, to structure different models. In classical B a machine can see another machine, i.e., it is allowed to call operations that do not modify the state of the other machine. To support this behavior, it was necessary to know if an operation has effect on state variables, that is the set of modified variables is the empty set. It turned out, that the information is more valuable than originally thought, as it is equivalent to some proof obligation:

If $u$ and $v$ are disjoint sets of state variables, and the substitution of an operation is $S_{BA}(v,t,v')$ we know that $u = u'$ and thus a simplified proof obligation for the preservation of an invariant $I(u)$ over the variables $u$ is

$$I(u) \wedge G(u \cup v,t) \wedge S_{BA}(v,t,v') \Rightarrow I(u)$$

which is obviously true. These kind of proof obligations are not generated by any of the proving environments for B we are aware of. In particular Rodin does not generate them. For a proving environment, this is a good idea as they do not contain valuable information for the user and they can be filtered out by simple syntax analysis. But for the model checker these proofs are very valuable; in most cases they allow us to reduce the number of invariants we need to check. As this type of proof information can be created from the syntax, we can use them even if we do not get proof information from Rodin, i.e., when working on classical B machines. As such, we were able to use Algorithm 4.1 also for classical B models and also obtain improvements of the model checking performance (although less impressive than for Event-B).

## 8 Conclusion and Future Work

First of all, we never found a model where using proof information significantly reduced the performance, i.e., the additional costs for calculating individual invariants for each state are rather low. Using proof information is the new default setting in PROB.

We got a number of models, in particular those coming from industry, where using the proof information has a high impact on the model checking time. In other cases, we gained only a bit or no improvement. This typically happens if the invariant is rather cheap to evaluate compared to the costs of calculating the guards of the events. We used an out-of-the-box version of Rodin[4] to produce our experimental results. Obviously, it is possible to further improve them by adding manual proof effort. In particular, it gives the user a chance to influence the speed of the model checker by proving invariant preservation for those parts that are difficult to evaluate, i.e., those predicates that need some kind of enumeration.

**Related Work** A similar kind of integration of theorem proving into a model checker was previously described in [**?**]. In their work Pnueli and Shahar introduced a system to verfify CTL and LTL properties. This system works as a layer on top of CMU SMV and was sucessfully applied to fragments of the Futurebus+ system. SAL is a framework and tool to combine different symbolic analysis [19], and can also be viewed as an integration of theorem proving and model checking. Mocha [3] is another work where a model checker is complemented by proof, mostly for assume-guarantee reasoning. Some more works using theorem proving and model checking together are [7, 4, 8, 9].

In the context of B, the idea of using a model checker to assist a prover has already been exploited in practice. For example, in previous work [5] we have already shown how a model checker can be used to complement the proving environment, by acting as a disprover. In [5] it was also shown that sometimes the model checker can be used as a prover, namely when the underlying sets of the proof obligation are finite. This is for example the case for the vehicle function mentioned in [12]. Another example is the Hamming encoder in [6], where Dominique Cansell has used PROB to prove certain theorems which are difficult to prove with a classical prover (due to the large number of cases).

**Future Work** We have done but a first step towards exploiting the full potential for integrating proving and model checking. For instance, we may feed the theorem prover with proof obligations generated by the model checker in order to speed up the model checking. A reasonable amount of time is spent evaluating the guards. If the model checker can use the theorem prover to prove that an event $e$ is guaranteed to be disabled after an event $f$ occurs, we can reduce the effort of checking

[4] For legal reasons, it is necessary to install the provers separately

guards. We may need to develop heuristics to find out when the model checker should try to get help from the provers.

Also we might feed information from the model checker back into the proving environment. If the state space is finite and we traverse all states, we can use this as a proof for invariant preservation. PROB restricts all sets to finite sets [13] to overcome the undecidability of B, so this needs to be handled with care. We need to ensure, that we do not miss states because PROB restricted some sets. Also we need to ensure that all states are reachable by the model checker, thus we may need some additional analysis of the model.

We also think of integrating a prover for classical B, to exploit proof information. The integration is most likely not as seamless as in Rodin and the costs of getting proof information is higher.

Although the cost of calculating the intersections of the invariants for each state is too low to measure it, the stored invariants take some memory. It might be possible to find a more efficient way to represent the intersections of invariants.

## References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.
2. J.-R. Abrial, M. Butler, and S. Hallerstede. An open extensible tool environment for Event-B. In *ICFEM06*, LNCS 4260, pages 588–605. Springer, 2006.
3. R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. Mocha: Modularity in model checking. In A. J. Hu and M. Y. Vardi, editors, *CAV*, volume 1427 of *Lecture Notes in Computer Science*, pages 521–525. Springer, 1998.
4. K. Arkoudas, S. Khurshid, D. Marinov, and M. C. Rinard. Integrating model checking and theorem proving for relational reasoning. In R. Berghammer, B. Möller, and G. Struth, editors, *RelMiCS*, volume 3051 of *Lecture Notes in Computer Science*, pages 21–33. Springer, 2003.
5. J. Bendisposto, M. Leuschel, O. Ligot, and M. Samia. La validation de modèles Event-B avec le plug-in ProB pour RODIN. *Technique et Science Informatiques*, 27(8):1065–1084, 2008.
6. D. Cansell, S. Hallerstede, and I. Oliver. UML-B specification and hardware implementation of a hamming coder/decoder. In J. Mermet, editor, *UML-B Specification for Proven Embedded Systems Design*. Kluwer Academic Publishers, Nov 2004. Chapter 16.
7. D. Dams, D. Hutter, and N. Sidorova. Using the inka prover to automate safety proofs in abstract interpretation - a case study. In F. Bellegarde and O. Kouchnarenko, editors, *Workshop on Modelling and Verification*, C.I.S., Besançon, France, 1999. Alternative title: Combining Theorem Proving and Model Checking - A Case Study.
8. P. Dybjer, Q. Haiyan, and M. Takeyama. Verifying haskell programs by combining testing, model checking and interactive theorem proving. *Information & Software Technology*, 46(15):1011–1025, 2004.

9. E. L. Gunter and D. Peled. Model checking, testing and verification working together. *Formal Asp. Comput.*, 17(2):201–221, 2005.
10. B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from Z and B. In *Proceedings FME'02*, LNCS 2391, pages 21–40. Springer-Verlag, 2002.
11. M. Leuschel. The high road to formal validation. In E. Börger, M. Butler, J. P. Bowen, and P. Boca, editors, *ABZ*, volume 5238 of *Lecture Notes in Computer Science*, pages 4–23. Springer, 2008.
12. M. Leuschel and M. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.
13. M. Leuschel and M. J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.
14. M. Leuschel, J. Falampin, F. Fritz, and D. Plagge. Automated property verification for large scale b models. In A. Cavalcanti and D. Dams, editors, *Proceedings FM 2009*, LNCS 5850, pages 708–723. Springer, 2009.
15. M. Leuschel and D. Plagge. Seven at a stroke: LTL model checking for high-level specifications in B, Z, CSP, and more. In Y. A. Ameur, F. Boniol, and V. Wiels, editors, *Proceedings Isola 2007*, volume RNTI-SM-1 of *Revue des Nouvelles Technologies de l'Information*, pages 73–84. Cépaduès-Éditions, 2007.
16. O. Müller and T. Nipkow. Combining model checking and deduction for i/o-automata. In E. Brinksma, R. Cleaveland, K. G. Larsen, T. Margaria, and B. Steffen, editors, *TACAS*, LNCS 1019, pages 1–16. Springer, 1995.
17. S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, LNCS 1102, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.
18. B. Scheuermann, C. Lochert, and M. Mauve. Implicit hop-by-hop congestion control in wireless multihop networks. *Ad Hoc Networks*, 2007. doi: 10.1016/j.adhoc.2007.01.001.
19. N. Shankar. Combining theorem proving and model checking through symbolic analysis. In C. Palamidessi, editor, *CONCUR*, LNCS 1877, pages 1–16. Springer, 2000.
20. C. Spermann and M. Leuschel. ProB gets nauty: Effective symmetry reduction for B and Z models. In *Proceedings Symposium TASE 2008*, pages 15–22, Nanjing, China, June 2008. IEEE.
21. F. Steria, Aix-en-Provence. *Atelier B, User and Reference Manuals*, 1996. Available at `http://www.atelierb.societe.com`.

# Integrating Reliability Assessment into Formal Development by Refinement

Anton Tarasyuk, Elena Troubitsyna, and Linas Laibinis

Åbo Akademi University, Turku, Finland
{Anton.Tarasyuk, Elena.Troubitsyna, Linas.Laibinis}@abo.fi

**Abstract.** Formal methods are indispensable for ensuring dependability of complex software-intensive systems. In particular, the B Method and its recent extension Event-B have been successfully used in the development of several complex safety-critical systems. However, they are currently not supporting quantitative assessment of dependability attributes that is often required for certifying safety-critical systems. In this paper we demonstrate how to integrate reliability assessment into Event-B development. This work shows how to conduct probabilistic assessment of system reliability at the development stage rather than at the implementation level. This allows the developers to chose the design alternative that offers the most optimal solution from the reliability point of view.

## 1 Introduction

Formal verification techniques provide us with rigorous and powerful methods for establishing correctness of complex systems. The advances in expressiveness, usability and automation of these techniques enable their use in the design of a wide range of complex dependable systems. For instance, the B Method [2] and its extension Event-B [1] provide us with a powerful framework for developing systems correct by construction. The top-down development paradigm based on stepwise refinement adopted by these frameworks has proven its worth in several industrial projects [16, 4].

While developing system by refinement, we start from an abstract system specification and, in a number of refinement steps, introduce the desired implementation decisions. While approaching the final implementation, we decrease the abstraction level and reduce non-determinism inherently present in the abstract specifications. In general, an abstract specification can be refined in several different ways because usually there are several ways to resolve its non-determinism. These refinement 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 we cannot perform it. In this paper we propose an approach to overcoming this problem.

We propose to integrate stepwise development in Event-B with probabilistic model checking [11] 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 [19, 21, 14]. Obviously, to assess reliability of various design alternatives, we need to model their behaviour stochastically. In this paper we demonstrate 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 that we verify by probabilistic model checking. To illustrate our approach, we assess reliability of refinement alternatives that model different fault tolerance mechanisms.

We believe that our approach can facilitate the process of developing dependable systems by enabling evaluation of design alternatives at early development stages. Moreover, it 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.

The remainder of the paper is structured as follows. In Section 2 we give a brief overview of our modelling formalism – the Event-B framework. In Section 3 we give an example of refinement in Event-B. In Section 4 we demonstrate how to augment Event-B specifications with probabilistic information and convert them into specifications of the PRISM model checker [15]. In Section 5 we define how to assess reliability via probabilistic verification and compare the results obtained by model checking with algebraic solutions. Finally, in Section 6 we discuss the obtained results, overview the related work and propose some directions for the future work.

## 2 Modelling and Refinement in Event-B

The B Method is an approach for the industrial development of highly dependable software. The method has been successfully used in the development of several complex real-life applications [16, 4]. Event-B [1] is an extension of the B Method [2] to model parallel, distributed and reactive systems. The Rodin platform [18] provides automated tool support for modelling and verification (by theorem proving) in Event-B. Currently Event-B is used in the EU project Deploy [6] to model several industrial systems from automotive, railway, space and business domains.

Event-B uses the Abstract Machine Notation [17] for constructing and verifying system models. An abstract machine encapsulates the state (the variables) of a model and defines operations on its state. A simple abstract machine has the following general form:

```
Machine AM
Variables v
Invariants I
Events
    init
    evt_1
    ...
    evt_N
```

The machine is uniquely identified by its name *AM*. The state variables of the machine, *v*, are declared in the **Variables** clause and initialised in *init* event. The variables are strongly typed by constraining predicates of invariants *I* given in the **Invariants** clause. The **Invariants** clause might also contain other predicates defining properties that should be preserved during system execution.

The dynamic behaviour of the system is defined by the set of atomic events specified in the **Events** clause. An event is defined as follows:

$$evt \ \widehat{=}\ \textbf{when}\ g\ \textbf{then}\ S\ \textbf{end}$$

where the guard *g* is conjunction of predicates over the state variables *v*, and the action *S* is an assignment to the state variables.

The guard defines the conditions under which the action can be executed, i.e., when the event is *enabled*. If several events are enabled then any of them can be chosen for execution non-deterministically. If none of the events is enabled then the system deadlocks.

In general, the action of an event is a composition of variable assignments executed simultaneously (simultaneous execution is denoted as $\|$). Variable assignments can be either deterministic or non-deterministic. The deterministic assignment is denoted as $x := E(v)$, where $x$ is a state variable and $E(v)$ expression over the state variables $v$. The non-deterministic assignment can be denoted as $x :\in S$ or $x :\mid Q(v,x')$, where $S$ is a set of values and $Q(v,x')$ is a predicate. As a result of non-deterministic assignment, $x$ gets any value from $S$ or it obtains such a value $x'$ that $Q(v,x')$ is satisfied.

The semantics of Event-B events is defined using so called before-after predicates [17]. It is a variation of the weakest precondition semantics [5]. A before-after predicate describes a relationship between the system states before and after execution of an event. The formal semantics provides us with a foundation for establishing correctness of Event-B specifications. To verify correctness of a specification we need to prove that its initialization and all events preserve the invariant.

The formal semantics also establishes a basis for system refinement – the process of developing systems correct by construction. The basic idea underlying formal stepwise development by refinement is to design the system implementation gradually, by a number of correctness preserving steps, called *refinements*. The refinement process starts from creating an abstract, albeit unimplementable, specification and finishes with generating executable code. The intermediate stages yield the specifications containing a mixture of abstract mathematical constructs and executable programming artifacts.

Assume that the refinement machine $AM'$ is a result of refinement of the abstract machine $AM$:

$$\boxed{\begin{array}{l} \textbf{Machine } AM \\ \textbf{Variables } v \\ \textbf{Invariants } I \\ \textbf{Events} \\ \quad init \\ \quad evt_1 \\ \quad \cdots \\ \quad evt_N \end{array}} \quad \sqsubseteq \quad \boxed{\begin{array}{l} \textbf{Machine } AM' \\ \textbf{Variables } v' \\ \textbf{Invariants } I' \\ \textbf{Events} \\ \quad init' \\ \quad evt'_1 \\ \quad \cdots \\ \quad evt'_K \end{array}}$$

The machine $AM'$ might contain new variables and events as well as replace the abstract data structures of $AM$ with the concrete ones. The invariants of $AM'$ – $I'$ – define not only the invariant properties of the refined model, but also the connection between the state spaces of $AM$ and $AM'$. For a refinement step to be valid, every possible execution of the refined machine must correspond (via $I'$) to some execution of the abstract machine. To demonstrate this, we should prove that $init'$ is a valid refinement of $init$, each event of $AM'$ is a valid refinement of its counterpart in $AM$ and that the refined specification does not introduce additional deadlocks.

In the next section we illustrate modelling and refinement in Event-B by an example.

## 3 Example of Refinement in Event-B

Control and monitoring systems constitute a large class of dependable systems. Essentially, the behaviour of these systems is periodic. Indeed, a control system periodically executes a control cycle that consists of reading sensors and setting actuators. The monitoring systems periodically perform certain measurements. Due to faults (e.g., caused by random hardware failures) inevitably present in any system, the system can fail to perform its functions. In this paper we focus on modelling fail-safe systems, i.e., the systems that shut down upon occurrence of failure. In general, the behaviour of such system can be represented as shown in the specification below.

For the sake of simplicity, we omit the detailed modelling of the system functionality. The variable *res* abstractly models success or failure to perform the required functions at each iteration. Each iteration of the system corresponds to the execution of the event $output$. If no failure has occurred then, as a result of the non-deterministic assignment, the variable *res* obtains the value $TRUE$. In this case the next iteration can be executed. However, if a failure has occurred then *res* obtains the value *FALSE* and the system deadlocks.

```
**Machine** *System*
**Variables** *res*
**Invariants**
  $inv_1 : res \in BOOL$
**Events**
  $init \widehat{=}$
      **begin**
          $res := TRUE$
      **end**
  $output \widehat{=}$
      **when**
          $res = TRUE$
      **then**
          $res :\in BOOL$
      **end**
```

In the initial specification we have deliberately abstracted away from modelling system components and their failures. In the next refinement step we introduce explicit representation of system components and introduce fault tolerance mechanisms. These mechanisms allow the system to perform its functions even in the presence of certain faults [19]. Fault tolerance is usually achieved by introducing redundancy into the system design. The redundancy can be either static or dynamic. When static redundancy is used, the redundant components work in parallel with the main ones. In dynamic redundancy activation of the redundant components occurs only after the main ones have failed.

Refining a system by introducing the fault tolerance mechanisms is a rather standard model transformation frequently performed in the development of dependable systems. Next we show by examples how to introduce various fault tolerance mechanisms by refinement.

Triple Modular Redundancy (TMR) [19] is a well-known mechanism based on static redundancy. The general principle is to triplicate a system module and introduce the majority voting to obtain a single result of the module, as shown in Figure 1. Such an arrangement allows us to mask failures of a single module. TMR can be introduced into a system specification by refinement as explained below. We introduce variables $m_1$, $m_2$ and $m_3$ to model the results produced by the redundant modules. The variable *phase* models the phases of TMR execution – first reading the results produced by the modules and then voting.

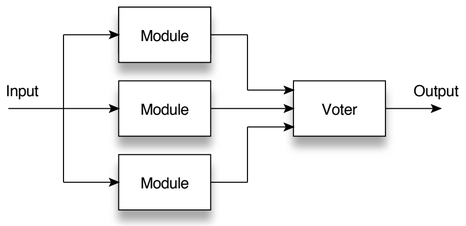


**Fig. 1.** A Triple Modular Redundancy Arrangement

```
**Machine** $System_{TMR}$
**Variables**
  $res, m_1, m_2, m_3, phase$
  $flag_1, flag_2, flag_3$
**Invariants**
  $inv_{1..3} : m_1, m_2, m_3 \in \{0,1\}$
  $inv_4 : phase \in \{reading, voting\}$
  $inv_{5..7} : flag_1, flag_2, flag_3 \in \{0,1\}$
  $inv_6 : \sum m_i > 1 \Rightarrow res = TRUE$
**Events**
    $\dots$
  $module_{ok_1} \widehat{=}$
    **when**
      $m_1 = 1 \wedge flag_1 = 1 \wedge$
      $phase = reading$
    **then**
      $m_1 :\in \{0,1\} \parallel flag_1 := 0$
    **end**

$module_{failed_1} \widehat{=}$
  **when**
    $m_1 = 0 \wedge flag_1 = 1 \wedge$
    $phase = reading$
  **then**
    $flag_1 := 0$
  **end**
    $\dots$
$synchr \widehat{=}$
  **when**
    $flag_1 = 0 \wedge flag_1 = 0 \wedge$
    $flag_3 = 0 \wedge phase = reading$
  **then**
    $phase := voting$
  **end**
```

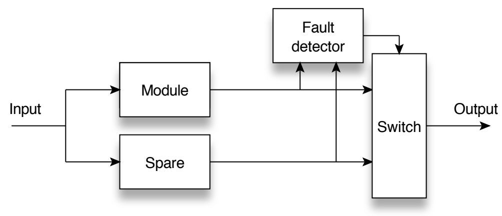


**Fig. 2.** A Standby Spare Arrangement

The modules work in parallel. In our specification it is reflected by the fact that all the events modelling module behaviour are enabled simultaneously. Each event disables itself after being executed once. When all the modules complete their execution, the event *synchr* enables the events modelling voting. Let us observe that the invariant $m_1 + m_2 + m_3 > 1 \Rightarrow res = TRUE$ relates the abstract and refined systems, i.e, it requires that the correct output can be produced only if no more than one module has failed.

In general, we can introduce any fault tolerance mechanism by refinement. Below we show other alternatives. For instance, instead of the TMR arrangement we can introduce a standby spare mechanism shown in Figure 2. In this mechanism, every result produced by an active (main) module is checked by a fault detector. If an error is detected then the result produced by the failed module is ignored and the system switches to accepting the results produced by the spare. The spare can be *hot* meaning that the main module and spare work in parallel. In this case the switch to spare happens almost instantly. The spare also can be *cold*, i.e., the spare is in the standby mode and is activated only after the main module fails.

Below we present an excerpt from the specification that refines the *System* specification to model dynamic redundancy. The values *in* and *out* of the variable *phase* correspond to the values *reading* and *voting* in the TMR specification. The additional execution phase *det* is introduced to model failure detection. The presented events specify the detection phase for the hot spare arrangement.

The output can be produced successfully if at least one module functions correctly. If an error is detected then the system switches the failed module off.

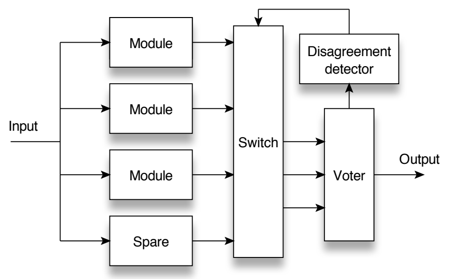


**Fig. 3.** TMR with a Spare Arrangement

Finally, we can also introduce a hybrid arrangement, which combines static and dynamic redundancy, as shown in Figure 3. The system works as TMR until a failure of a module occurs. Then the system activates the spare to "replace" the failed module. The full Event-B specifications of this and the previous arrangements can be found in [20].

Let us observe that any specification described above is a valid refinement of our abstract specification *System*. However, even though the fault tolerance mechanisms were introduced to increase system reliability, we cannot evaluate which of the specifications is more optimal from the point of view of reliability. This problem is caused by the non-deterministic modelling of the failure occurrence – the only possible modelling currently available in Event-B. To evaluate reliability, we need to replace the non-deterministic modelling of failure occurrence by the probabilistic ones and use the suitable techniques for reliability evaluation. Next we present our approach for achieving this.

```
$detection_{ok_1} \widehat{=}$
  when
    $m_1 = 1 \wedge phase = det$
  then
    $phase := out \parallel m := m_1$
    $flag_1 := 1 \parallel flag_2 := 1$
  end
$detection_{ok_2} \widehat{=}$
  when
    $m_1 = 0 \wedge m_2 = 1 \wedge phase = det$
  then
    $phase := out \parallel m := m_2 \parallel flag_2 := 1$
  end
$detection_{nok} \widehat{=}$
  when
    $m_1 = 0 \wedge m_2 = 0 \wedge phase = det$
  then
    $phase := out \parallel m := 0$
  end
```

## 4 From Event-B Modelling to Probabilistic Model Checking

To enable formal, probabilistic analysis of reliability in Event-B we can choose several options. The first and the most powerful is to rely on probabilistic weakest precondition semantics [12] and use probabilistic refinement technique [13] to evaluate reliability. This technique allows us to express algebraically the reliability of the system as a function of reliabilities of its components. However, for complex industrial-size systems finding this function might be very complex or even analytically intractable. A simpler and more scalable solution is to use probabilistic model checking to obtain numeric solution. To achieve this we need to augment Event-B models with probabilities in such way that they would become amenable for probabilistic verification. Then we need to establish connection between probabilistic verification and reliability assessment.

To tackle the first problem let us observe that Event-B is a state-based formalism. The state space of the system specified in Event-B is formed by the values of the state variables. The transitions between states are determined by the actions of the system events. The states that can be reached as a result of event execution are defined by the current state. If we augment Event-B specification with the probabilities of reaching the next system state from the current one then we obtain a probabilistic automaton [3]. In case the events are mutually exclusive, i.e., only one event is enabled at each system state then the specification can be represented by a Markov chain. Otherwise, it corresponds to a Markov Decision process [7, 10, 22]. More specifically, it is a discrete time Markov process since we can use it to describe the states at certain instances of time.

The probabilistic model checking framework developed by Kwiatkowska et al. [11] supports verification of Discrete-Time Markov Chains (DTMC) and Markov Decision Processes (MDP). The framework has a mature tool support – the PRISM model checker [15]

The PRISM modelling language is a high-level state-based language. It relies on the Reactive Modules formalism of Alur and Henzinger [3]. PRISM supports the use of constants and variables that can be integers, doubles (real numbers) and Booleans. Constants are used, for instance, to define the probabilities associated with variable updates. The variables in PRISM are finite-ranged and strongly typed. They can be either local or global. The definition of an initial value of a variable is usually attached to its declaration. The state space of a PRISM model is defined by the set of all variables, both global and local.

In general, a PRISM specification looks as follows:

```
dtmc
const double $p_{11} = \ldots;$
  ...
global $v : Type$ init ...;
  ...
module $M_1$
  $v_1 : Type$ init ...;
  $[]\ g_{11} \rightarrow p_{11} : act_{11} + \cdots + p_{1n} : act_{1n};$
  $[sync]\ g_{12} \rightarrow q_{11} : act'_{11} + \cdots + q_{1m} : act'_{1m};$
  ...
endmodule
module $M_2$
  $v_2 : Type$ init ...;
  $[sync]\ g_{21} \rightarrow p_{21} : act_{21} + \cdots + p_{2k} : act_{2k};$
  $[]\ g_{22} \rightarrow q_{21} : act'_{21} + \cdots + q_{2l} : act'_{2l};$
  ...
endmodule
  ....
```

A system specification in PRISM is constructed as a parallel composition of modules. Modules work in parallel. They can be independent of each other or interact with each other. Each module has a number of local variables $v_1,\ v_2$ and a set of guarded commands that determine its dynamic behaviour. The guarded commands can have names. Similarly to the events of Event-B, a guarded command can be executed if its guard evaluates to $TRUE$. If several guarded commands are enabled then the choice between them can be non-deterministic in case of MDP or probabilistic (according to the uniform distribution) in case of DTMC. In general, the body of a guarded command is a probabilistic choice between deterministic assignments.

The guarded commands define not only the dynamic behaviour of a stand-alone module but can also be used to define synchronisation between modules. If several modules synchronise then each of them should contain a command defining the synchronisation condition. These commands should have identical names. For instance, in our general PRISM specification shown above, the modules $M_1$ and $M_2$ synchronise. They contain the corresponding guarded commands labelled with the name *sync*. The guarded commands that provide synchronisation with other modules cannot modify the global variables. This allows to avoid read-write and write-write conflicts on the global variables.

Converting Event-B model into a PRISM model is rather straightforward. When converting Event-B model into its counterpart, we need to restrict the types of variables and constants to the types supported by PRISM. The invariants that describe system properties can be represented as a number of temporal logic formulas in a list of properties of the model and then can be verified by PRISM if needed. While converting events into the PRISM guarded commands, we identify four classes of events: initilisation events, events with parallel deterministic assignment, non-deterministic assignment and parallel non-deterministic assignment. The conversion of an Event-B event to a PRISM guarded command depends on its class:

- The initialisation events are used to form the initialisation part of the corresponding variable declaration. Hence the initialisation does not have a corresponding guarded command in PRISM;
- An event with a parallel deterministic assignment

$$evt \mathrel{\widehat{=}} \textbf{when}\ g\ \textbf{then}\ x := x_1 \parallel y := y_1 \parallel z := z_1\ \textbf{end}$$

can be represented by the following guarded command in PRISM:

$$[]\; g \rightarrow (x' = x_1)\&(y' = y_1)\&(z' = z_1)$$

Here & denotes the parallel composition;

– An event with a non-deterministic assignment

$$evt \mathrel{\widehat{=}} \textbf{when}\ g\ \textbf{then}\ x :\in \{x_1, \ldots x_n\}\ \textbf{end}$$

can be represented as

$$[]\; g \rightarrow p_1 : (x' = x_1) + \cdots + p_n : (x' = x_n)$$

where $p_1, ..., p_n$ are defined according to a certain probability distribution;

– An event with a parallel non-deterministic assignment

$$evt \mathrel{\widehat{=}} \textbf{when}\ g\ \textbf{then}\ x :\in \{x_1, \ldots x_n\} \parallel \\ y :\in \{y_1, \ldots y_m\} \parallel z :\in \{z_1, \ldots z_k\}\ \textbf{end}$$

can be represented using the PRISM synchronisation mechanism. It corresponds to a set of the guarded commands modelling synchronisation. These commands have the identical guards. Their bodies are formed from the assignments used in the parallel composition of the Event-B action.

```
module X
  x : Type init ...;
  [name] g → p1 : (x' = x1) + ··· + pn : (x' = xn);
endmodule
module Y
  y : Type init ...;
  [name] g → q1 : (y' = y1) + ··· + qm : (y' = ym);
endmodule
module Z
  z : Type init ...;
  [name] g → r1 : (z' = z1) + ··· + rk : (z' = zk);
endmodule.
```

To demonstrate the conversion of an Event-B specification into a PRISM specification, below we present an excerpt from the PRISM counterpart of the TMR specification. Here we assume that at each iteration step a module successfully produces a result with a constant probability *p*.

```
$System_{TMR}$
**module** $module_1$
   $m_1 : [0..1]$ **init** $1$;
   $f : [0..1]$ **init** $0$;

   $[m]\ (phase = 0)\&(m_1 = 1)\&(f = 0) \rightarrow$
            $p : (m_1' = 1)\&(f' = 1) + (1-p) : (m_1' = 0)\&(f' = 1);$
   $[m]\ (phase = 0)\&(m_1 = 0)\&(f = 0) \rightarrow (f' = 1);$
   $[]\ (phase = 0)\&(f = 1) \rightarrow (phase' = 1)\&(f' = 0);$
**endmodule**
**module** $module_2$   ...
**module** $module_3$   ...
**module** $voter$
   $res$ : ***bool*** **init** $true$;
   $[]\ (phase = 1)\&(m_1 + m_2 + m_3 > 1) \rightarrow (phase' = 0);$
   $[]\ (phase = 1)\&(m_1 + m_2 + m_3 \leq 1) \rightarrow (res' = false);$
**endmodule**
```

While converting an Event-B model into PRISM we could have modelled the parallel work of the system modules in the same way as we have done it in the Event-B specifications, i.e., using non-determinism to represent parallel behaviour and explicitly modelling the phases of system execution. However, we can also directly use the synchronisation mechanism of PRISM because all the modules update only their local variables and no read-write conflict can occur. This solution is presented in the excerpt above. In the $System_{TMR}$ specification, the guarded commands of the modules $module_1$, $module_2$ and $module_3$ are synchronised (as designated by the *m* label). In the $module_1$ we additionally update the global variable *phase* to model transition of the system to the voting phase.

## 5 Reliability Assessment via Probabilistic Model Checking

In engineering, reliability [21, 14] is generally measured by the probability that an entity $\mathcal{E}$ can perform a required function under given conditions for the time interval $[0,t]$:

$$R(t) = \mathbf{P}[\mathcal{E} \textit{ not failed over time } [0,t]].$$

The analysis of the abstract and refined specification shows that we can clearly distinguish between two classes of systems states: operating and failed. In our case the operating states are the states where the variable *res* has the value $TRUE$. Correspondingly, the failed states are the states where the variable *res* has the value *FALSE*. While the system is in an operating state, it continues to iterate. When the system fails it deadlocks. Therefore, we define *reliability of the system as a probability of staying operational for a given number of iterations*.

Let $\mathcal{T}$ be the random variable measuring the number of iterations before the deadlock is reached and $F(t)$ its cumulative distribution function. Then clearly $R(t)$ and $F(t)$ are related as follows:

$$R(t) = \mathbf{P}[\mathcal{T} > t] = 1 - \mathbf{P}[\mathcal{T} \leq t] = 1 - F(t).$$

It is straightforward to see that our definition corresponds to the standard definition of reliability given above. Now let us discuss how to employ PRISM model checking to assess system reliability.

While analysing a PRISM model we define a number of temporal logic properties and systematically check the model to verify them. Properties of discrete-time PRISM models, i.e, DTMC and MDP, are expressed formally in the probabilistic computational tree logic [9]. The PRISM property specification language supports a number of different types of properties. For example, the **P** operator is used to refer to the probability of a certain event occurrence.

Since we are interested in assessment of system reliability, we have to verify *invariant* properties, i.e., properties maintained by the system globally. In the PRISM property specification language, the operator **G** is used inside the operator **P** to express properties of such type. In general, the property

$$\mathbf{P}_{=?}[\mathbf{G} \leq t \; prop]$$

returns a probability that the predicate *prop* remains $TRUE$ in all states within the period of time $t$.

To evaluate reliability of a system we have to assess a probability of system staying operational within time $t$. We define a predicate $\mathcal{OP}$ that defines a subset of all system states where the system is operational. Then, the PRISM property

$$\mathbf{P}_{=?}[\mathbf{G} \leq T\ \mathcal{OP}] \tag{1}$$

gives us the probability that the system will stay operational during the first $T$ iterations, i.e, it is a probability that any state in which the system will be during this time belongs to the subset of operational states. In other words, the property (1) defines the reliability function of the system.

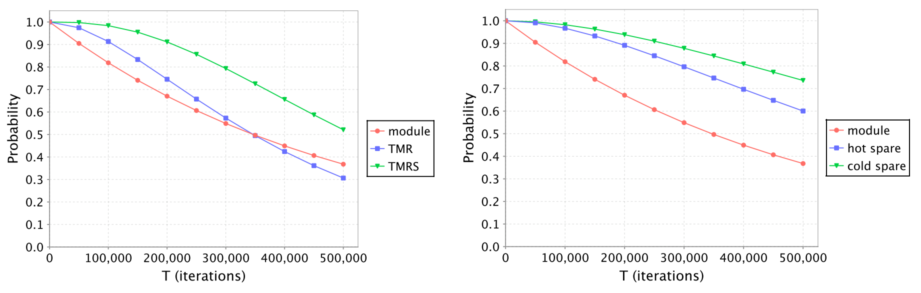
Let us return to our examples. As we discussed previously, the operational states of our systems are defined by the predicate $res = true$, i.e., $\mathcal{OP} \mathrel{\widehat{=}} res = true$. Then the PRISM property

$$\mathbf{P}_{=?}[\mathbf{G} \leq T\ (res = true)] \tag{2}$$

denotes the reliability of our systems within time $T$.

To evaluate reliability of our refinement system, let us assume that a module produces a result successfully with the probability $p$ equal to 0.999998. In Figure 4 we present the results of analysis of reliability up to 500000 iterations. Figure 4 (a) shows the comparative results between single-module and both TMR systems. The results show that the triple modular redundant system with a spare always gives better reliability. Note that using the simple TMR arrangement is better comparing to a single module only up to approximately 350000 iterations. In Figure 4 (b) we compare single-module and standby spare arrangements. The results clearly indicate that the better reliability is provided by the dynamic redundancy systems and that using of the cold spare arrangement is always more reliable.

It would be interesting to evaluate precision of the results obtained by the model checking with PRISM. For our case study it is possible to derive analytical representations of reliability functions, which then can be used for comparison with verification results of property (2). It is well-known that the reliability of a single module system is $R_M(t) = p^t$ and



(a) (b)

**Fig. 4.** Resulting Reliabilities

it is easy to show that the reliability of a TMR system, consists of three identical modules, is

$$R_{TMR}(t) = R_M^3(t) + 3R_M^2(t)(1 - R_M(t)) = 3R_M^2(t) - 2R_M^3(t) = 3p^{2t} - 2p^{3t}.$$

Indeed, we can also calculate that the standby spare arrangement with a faulty detector has the resulting reliability

$$R_{HSS} = 1 - (1 - p^t)^2$$

for the hot spare, and the resulting reliabilty

$$R_{CSS} = p^t(1 + t(1 - p))$$

for the cold spare module. Finally, for the TMR arrangement with a spare, the resulting reliability is given by the expression

$$R_{TMRS} = (6t - 8)p^{3t} - 6tp^{3t-1} + 9p^{2t}.$$

It is easy to verify that the results obtained by the model checking are identical to those can be calculated from the formulas presented above. This fact demonstrates the feasibility of using the PRISM model checker for reliability assessment.

## 6 Conclusion

In this paper we have proposed an approach to integrating reliability assessment into the refinement process. The proposed approach enables reliability assessment at early design phases that allows the designers to evaluate reliability of different design alternatives already at the development phase.

Our approach integrates two frameworks: refinement in Event-B and probabilistic model checking. Event-B supported by the RODIN tool platform provides us with a suitable framework for development of complex industrial-size systems. By integrating probabilistic verification supported by PRISM model checker we open a possibility to reason about non-functional system requirements in the refinement process.

The Event-B framework has been extended by Hallerstede and Hoang [8] to take into account probabilistic behaviour. They introduce qualitative probabilistic choice operator to reason about almost certain termination. This operator attempts to bound demonic non-determinism that, for instance, allows to demonstrate convergence of certain protocols. However, this approach is not suitable for reliability assessment since explicit quantitative representation of reliability would not be supported.

Kwiatkowska et al. [11] proposed an approach to assessing dependability of control systems using continuous time Markov chains. The general idea is similar to ours – to formulate reliability as a system property to be verified. However, this approach aims at assessing reliability of already developed system. In our approach reliability assessment proceeds hand-in-hand with system development.

The similar topic in the context of refinement calculus has been explored previously by Morgan et al. [13, 12]. In this approach the probabilistic refinement was used to assess system dependability. However, this work does not have the corresponding tool support, so the use of this approach in industrial practice might be cumbersome. In our approach we see a great benefit in integrating frameworks that have mature tool support [18, 15].

When using model checking we need to validate whether the analysed model represents the behaviour of the real system accurately enough. For example, the validation can be done if we demonstrate that model checking provides a good approximation of the corresponding algebraic solutions. In this paper we have deliberately chosen the examples for which algebraic solutions can be provided. The experiments have demonstrated that the results obtained by model checking accurately match the algebraic solutions.

In our future work it would be interesting to further explore the connection between Event-B modeling and dependability assessment. In particular, an additional study is required to establish a formal basis for converting all types of non-deterministic assignments into the probabilistic ones. Furthermore, it would be interesting to explore the topic of probabilistic data refinement in connection with dependability assessment.

## References

1. J.-R. Abrial. Extending B without changing it (for developing distributed systems). In H. Habiras, editor, *First Conference on the B method*, pages 169–190. IRIN Institut de recherche en informatique de Nantes, 1996.
2. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 2005.
3. R. Alur and T. Henzinger. Reactive modules. In *Formal Methods in System Design*, pages 7–48, 1999.
4. D. Craigen, S. Gerhart, and T.Ralson. Case study: Paris metro signaling system. In *IEEE Software*, pages 32–35, 1994.
5. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
6. EU-project DEPLOY. online at http://www.deploy-project.eu/.
7. W. Feller. *An Introduction to Probability Theory and its Applications*, volume 1. John Wiley & Sons, 1967.
8. S. Hallerstede and T. S. Hoang. Qualitative probabilistic modelling in Event-B. In J. Davies and J. Gibbons, editors, *IFM 2007, LNCS 4591*, pages 293–312, 2007.
9. H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. In *Formal Aspects of Computing*, pages 512–535, 1994.
10. J. G. Kemeny and J. L. Snell. *Finite Markov Chains*. D. Van Nostrand Company, 1960.
11. M. Kwiatkowska, G. Norman, and D. Parker. Controller dependability analysis by probabilistic model checking. In *Control Engineering Practice*, pages 1427–1434, 2007.
12. A. K. McIver and C. C. Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer, 2005.
13. 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*. Springer-Verlag, 1998.
14. P. D. T. O'Connor. *Practical Reliability Engineering, 3rd ed*. John Wiley & Sons, 1995.
15. PRISM. Probabilistic symbolic model checker.
online at http://www.prismmodelchecker.org/.
16. Rigorous Open Development Environment for Complex Systems (RODIN). IST FP6 STREP project, online at http://rodin.cs.ncl.ac.uk/.
17. Rigorous Open Development Environment for Complex Systems (RODIN). Deliverable D7, Event-B Language, online at http://rodin.cs.ncl.ac.uk/.
18. RODIN. Event-B platform. online at http://www.event-b.org/.
19. N. Storey. *Safety-Critical Computer Systems*. Addison-Wesley, 1996.
20. A. Tarasyuk, E. Troubitsyna, and L. Laibinis. Reliability assessment in Event-B. Technical Report 932, Turku Centre for Computer Science, 2009.
21. A. Villemeur. *Reliability, Availability, Maintainability and Safety Assessment*. John Wiley & Sons, 1995.
22. D. J. White. *Markov Decision Processes*. John Wiley & Sons, 1993.

# Part V

# Business Information Systems

# A Formal Semantics for the WS-BPEL Recovery Framework

## The π-Calculus Way

Nicola Dragoni[1] and Manuel Mazzara[2]

[1] DTU Informatics, Technical University of Denmark, Denmark
ndra@imm.dtu.dk
[2] School of Computing Science, Newcastle University, UK
manuel.mazzara@newcastle.ac.uk

**Abstract.** While current studies on Web services composition are mostly focused — from the technical viewpoint — on standards and protocols, this work investigates the adoption of formal methods for dependable composition. The Web Services Business Process Execution Language (WS-BPEL) — an OASIS standard widely adopted both in academic and industrial environments — is considered as a touchstone for concrete composition languages and an analysis of its ambiguous Recovery Framework specification is offered. In order to show the use of formal methods, a precise and unambiguous description of its (simplified) mechanisms is provided by means of a conservative extension of the π-calculus. This has to be intended as a well known case study providing methodological arguments for the adoption of formal methods in software specification. The aspect of verification is not the main topic of the paper but some hints are given.

## 1 Introduction

Service Oriented Architectures and the related paradigm are modern attempts to cope with old problems connected to Business-to-Business (B2B) and information interchange. Many implementations of this paradigm are possible and the so called Web services look to be the most prominent, mainly because the underlying architecture is already there; it is simply the web which has been extensively used in the last 15 years and where we can easily exploit HTTP [21], XML [5], SOAP [8] and WSDL [3]. The World Wide Web provides a basic platform for the interconnection on a point-to-point basis of different companies and customers but one of the B2B complications is the management of causal interactions between different services and the way in which the messages between them need to be handled (e.g., not always in a sequential way). This area of investigation is called composition, *i.e.,* the way to build complex services out of simpler ones [4]. These days, the need for workflow technology is becoming quite evident and the positive aspect is that we had investigated this technology for decades and we also have excellent modeling tools providing verification features that are grounded in the very active field of concurrency theory research.

## 1.1 BPEL and its Ambiguous Specification

Several organizations worked on composition proposals. The most important in the past have been IBM's WSFL [1] and Microsoft's XLANG [2]. These two have then converged into Web Services Business Process Execution Language [18] (BPEL for short) which is presently an OASIS standard and, given its wide adoption, it will be used as a touchstone for composition languages in this paper. BPEL allows workflow-based composition of services. In the committee members' words the aim is *"enabling users to describe business process activities as Web services and define how they can be connected to accomplish specific tasks"*. The problem with BPEL was that the earlier versions of the language were not very clear, the specification was huge and many points confusing, especially in relation to the Recovery Framework (RF) and the interactions between different mechanisms (fault handlers and compensation handlers). BPEL indeed represents a business tradeoff where not necessarily all the single technical choices have been made considering all the available options. Although in the final version of the specification (which is lighter and cleaner) fault handling during compensation has been simplified, we strongly believe that the sophisticated mechanism of recovery still needs a clarification.

## 1.2 Contribution of the Paper

In this paper we aim to reduce this ambiguity providing an *easily readable formal semantics* of the BPEL Recovery Framework (BPEL RF for short). This goal requires at least two different contributions:

1. a *formal semantics* of the framework, focusing on its essential mechanisms
2. an *easily readable* specification of these mechanisms

We provide both contributions following a "$\pi$-calculus way", that is using the $\pi$-calculus as formal specification language. It is worth noting that here the actual challenge is to provide not only a formal semantics for the BPEL RF but also an *easily readable* specification. Indeed, other attempts might be found in literature providing the first contribution only. For instance, in [12] such encoding has been proposed by one of the authors. However, one of the unsatisfactory aspects about that encoding is that it is hardly readable and complex. The actual challenge here is to reduce such complexity while keeping a formal and rigorous approach. As a result, in this paper we contribute with a better understanding of how the BPEL RF works. Moreover, the case study allows us to show the real power of the $\mathtt{web}\pi_\infty$ calculus (*i.e.,* the π-calculus based formal language exploited for the mentioned encoding) not only in terms of simplicity of the resulting BPEL specification, but also sketching how $\mathtt{web}\pi_\infty$ can contribute to the implementation of real orchestration engines.

Finally, we would like to stress that different formal models might be chosen for this goal. As discussed in the next section, our choice is primarily motivated by the "foundational feature" of the π-calculus, namely *mobility*, i.e. the possibility of transmitting channel names that will be, in turn, used by any receiving process. It is worth noting that in the specific contribution of this paper this feature is not really exploited or totally necessary since the modeled mechanisms requested us to pay more attention to process synchronization and concurrency than to full mobility. Anyway, in the general case, we have the strong opinion that mobility is an essential feature that composition languages should exhibit [13]. This aspect will be better discussed in section 2.

**Outline.** The paper is organized as follows. Section 2 will discuss the rationale behind our "π-calculus way" choice, briefly motivating why the π-calculus could be considered a formal foundation for dependable Web services composition. Section 3 will present $\mathtt{web}\pi_\infty$ discussing its syntax and semantics. Sections 4 and 5 will contribute with a clarification of the BPEL RF semantics. In particular, Section 4 will show how it appears in the original (ambiguous) specification, and Section 5 will propose the actual simplification and formal specification. Section 6 will add some conclusive remarks.

## 2 The π-Calculus Way to Dependable Composition

The need for formal foundation has been discussed widely in the last years, although many attempts to use formal methods in this setting have been speculative. Some communities, for example, criticized the process algebra options [19] promoting the Petri nets choice. The question here is whether we need a formal foundation and, if that is the case, which kind of formalism we need. While sequential computation has well established foundations in the λ-calculus and Turing machines, when it comes to concurrency things are far from being settled. The π-calculus ([16] and [17]) emerged during the eighties as a theory of mobile systems providing a conceptual framework for expressing them and reasoning about their behavior. It introduces mobility generalizing the channel-based communication of CCS by allowing channels to be passed as data through rendezvous over other channels. In other words, it is a model for prescribing (specification) and describing (analysis) concurrent systems consisting of agents which mutually interact and in which the communication structure can dynamically evolve during the execution of processes. Here, a communication topology is intended as the linkage between processes which indicates who can communicate with whom. Thus, changing the communication links means, for a process, moving inside this abstract space of linked entities.

A symmetry between λ-calculus and π-calculus could be suggested and the option to build concurrent languages (and so workflow languages as well) on a formal basis could actually make sense. It has indeed been investigated in many works, even in the BPEL context. But, while formal methods are expected to bring mathematical precision to the development of computer systems (providing precise notations for specification and verification), so far BPEL — despite having been subject of a number of formalizations (for example [10], [7] and [22]) — has not yet been proved to be built on an exact and specific mathematical model, including process algebras (this argument has been carefully developed in [13]). Thus, we do not have any conceptual and software tools for analysis, reasoning and software verification. If we are not able to provide this kind of tools, any hype about mathematical rigor becomes pointless.

It is also worth noting that, although many papers use the term π-calculus and process algebra interchangeably, there is a difference between them. Algebra is a mathematical structure with a set of values and a set of operations on the values. These operations enjoy algebraic properties such as commutativity, associativity, idempotency, and distributivity. In a typical process algebra, processes are values and parallel composition is defined to be a commutative and associative operation on processes. The π-calculus is an algebra but it differs from previous models for concurrency precisely for the fact that it includes a notion of mobility, i.e. the possibility of transmitting channel names that will be, in turn, used by receiving processes. This allows a sort of dynamic reconfiguration with the possibility of creating (and deleting) processes through the alteration of the process topology (although it can be argued that, even if the link to a process disappears, the process itself disappears only from "an external point of view").

The π-calculus looks interesting because of its treatment of component bindings as first class objects, which enables this dynamic reconfiguration to be expressed simply. So, the question now is: do we need this additional feature of the π-calculus or should we restrict our choice to models, like CCS, without this notion of mobility? Why all this hype over the π-calculus and such a rare focus on its crucial characteristic? We have the strong opinion that mobility is an essential feature that composition languages should exhibit. Indeed, while in some scenarios services can be selected already at design-time, in others some services might only be selected at runtime and this selection has then to be propagated to different parties. This phenomenon is called link passing mobility and it is properly approached in [6].

It is worth noting that in the specific contribution of this paper this feature is not really exploited or totally necessary since the modeled mechanisms requested we pay more attention to process synchronization and concurrency than to full mobility. This aspect has been instead essential in the full formalization of BPEL. In [13] it has been shown how it plays an important role in the encoding of interactions of the kind request-response. Indeed, in that case the invoker must send a channel name to be used then to return the response. This is a typical case of the so called output capability of the π-calculus, i.e. a received name is used as the subject of outputs only. The full input capability of the π-calculus — i.e. when a received name is used also as the subjects of inputs — has been not exploited in the BPEL encoding (and neither it is in this work). Indeed in [13] a specific well-formedness constraint imposes that *"received names cannot be used as subjects of inputs or of replicated* *inputs"*. Thus, at the present moment we remain agnostic regarding the need of the π-calculus input capability in the description of BPEL mechanism. We realize that this admission could be an argument for discussing again the choice of the original model.

## 2.1 Our Approach

WS-standards for dependability only concerns SOAP when employed as an XML messaging protocol (e.g. OASIS WS-Reliability and WS-Security), *i.e.,* at the message level. However, things are more complicated than this since loosely coupled components like Web services, being autonomous in their decisions, may refuse requests or suspend their functionality without notice, thus making their behavior unreliable to other activities. Henceforth, most of the web languages also include the notion of loosely coupled transaction – called *web transaction* [11] in the following – as a unit of work involving loosely coupled activities that may last long periods of time. These transactions, being orthogonal to administrative domains, have the typical atomicity and isolation properties relaxed, and instead of assuming a perfect roll-back in case of failure, support the explicit programming of compensation activities. Web transactions usually contain the description of three processes: *body*, *failure handler*, and *compensation*. The failure handler is responsible for reacting to events that occur during the execution of the body; when these events occur, the body is blocked and the failure handler is activated. The compensation, on the contrary, is installed when the body commits; it remains available for outer transactions to require some undo of previously performed actions. BPEL also uses this approach.

Our approach to recovery is instead described in [13], where it has been shown that different mechanisms for error handling are not necessary and the BPEL semantics has been presented in terms of $\mathtt{web}\pi_\infty$, which is based on the idea of event notification as the unique error handling mechanism. This result allows us to extend any semantic considerations about $\mathtt{web}\pi_\infty$ to BPEL. $\mathtt{web}\pi_\infty$ (originally in [14]) has been introduced to investigate how process algebras can be used as a foundation in this context. It is a simple and conservative extension of the π-calculus where the original algebra is augmented with an operator for asynchronous events raising and catching in order to enable the programming of widely accepted error handling techniques (such as long running transactions and compensations) with reasonable simplicity. We addressed the problem of composing services starting directly from the π-calculus and considering this proposal as a foundational model for composition simply to verify statements regarding any mathematical foundations of composition languages and not to say that the π-calculus is more suitable than other models (such as Petri nets) for these purposes. The calculus is presented in detail in section 3 while in section 4 and 5 it is showed how it can be useful to clarify the BPEL RF semantics.

## 3 The Composition Calculus

In this section we present a proposal to cope with the issues presented in section 2. Although `web`$\pi_\infty$ is ambitious, for sure we do not pretend to solve all the problems and to give the ultimate answer to all the questions. Giving all the details about the language and its theory is beyond the scope of this paper which is giving a brief account about how `web`$\pi_\infty$ can be considered in the overall scenario of formal methods for dependable Web services. You can find all the relevant details in some previous work, especially in [12], [13] and [15].

### 3.1 Syntax

The syntax of `web`$\pi_\infty$ *processes* relies on a countable set of *names*, ranged over by $x, y, z, u, \cdots$. Tuples of names are written $\widetilde{u}$. We intend $i \in I$ with $I$ a finite non-empty set of indexes.

$$
\begin{array}{lll}
P ::= & & \\
 & \mathbf{0} & (\textbf{nil}) \\
 & \mid \overline{x}\widetilde{u} & (\textbf{output}) \\
 & \mid \sum_{i \in I} x_i(\widetilde{u_i}).P_i & (\textbf{alternative composition}) \\
 & \mid (x)P & (\textbf{restriction}) \\
 & \mid P \,|\, P & (\textbf{parallel composition}) \\
 & \mid\ !x(\widetilde{u}).P & (\textbf{guarded replication}) \\
 & \mid \langle\!| P \,;\, P |\!\rangle_x & (\textbf{workunit})
\end{array}
$$

A process can be the inert process $\mathbf{0}$, an output $\overline{x}\widetilde{u}$ sent on a name $x$ that carries a tuple of names $\widetilde{u}$, an alternative composition consisting of input guarded processes that consumes a message $\overline{x_i}\widetilde{w_i}$ and behaves like $P_i\left\{\widetilde{w_i}/\widetilde{u_i}\right\}$, a restriction $(x)P$ that behaves as $P$ except that inputs and messages on $x$ are prohibited, a parallel composition of processes, a replicated input $!x(\widetilde{u}).P$ that consumes a message $\overline{x}\widetilde{w}$ and behaves like

$P\{\widetilde{w}/\widetilde{u}\} \mid !x(\widetilde{u}).P$, or a workunit $\langle\!| P \,;\, Q |\!\rangle_x$ that behaves as the *body* $P$ until an abort $\overline{x}$ is received and then behaves as the *event handler* $Q$.

Names $x$ in outputs, inputs, and replicated inputs are called *subjects* of outputs, inputs, and replicated inputs, respectively. It is worth to notice that the syntax of `web`$\pi_\infty$ processes simply augments the asynchronous $\pi$-calculus with workunit process. The input $x(\widetilde{u}).P$, restriction $(x)P$ and replicated input $!x(\widetilde{u}).P$ are binders of names $\widetilde{u}$, $x$ and $\widetilde{u}$ respectively. The scope of these binders is the process $P$. We use the standard notions of $\alpha$-equivalence, *free* and *bound names* of processes, noted $\mathrm{fn}(P)$, $\mathrm{bn}(P)$ respectively.

## 3.2 Semantics

We give the semantics for the language in two steps, following the approach of Milner [17], separating the laws that govern the static relations between processes from the laws that rule their interactions. The first step is defining a static structural congruence relation over syntactic processes. A structural congruence relation for processes equates all agents we do not want to distinguish. It is introduced as a small collection of axioms that allow minor manipulation on the processes' structure. This relation is intended to express some intrinsic meanings of the operators, for example the fact that parallel is commutative. The second step is defining the way in which processes evolve dynamically by means of an operational semantics. This way we simplify the statement of the semantics just closing with respect to $\equiv$, *i.e.,* closing under process order manipulation induced by structural congruence.

**Definition 1.** *The* structural congruence $\equiv$ *is the least congruence satisfying the Abelian Monoid laws for parallel and summation (associativity, commutativity and* $\mathbf{0}$ *as identity) closed with respect to* $\alpha$*-renaming and the following axioms:*

1. *Scope laws:*

$$(u)\mathbf{0} \equiv \mathbf{0}, \qquad (u)(v)P \equiv (v)(u)P,$$
$$P \mid (u)Q \equiv (u)(P \mid Q)\,, \quad \textit{if } u \notin \mathrm{fn}(P)$$
$$\langle\!| (z)P \,;\, Q |\!\rangle_x \equiv (z)\langle\!| P \,;\, Q |\!\rangle_x\,, \quad \textit{if } z \notin \{x\} \cup \mathrm{fn}(Q)$$

2. *Workunit laws:*

$$\langle\!| \mathbf{0} \,;\, Q |\!\rangle_x \equiv \mathbf{0}$$
$$\langle\!| \langle\!| P \,;\, Q |\!\rangle_y \mid R \,;\, R' |\!\rangle_x \equiv \langle\!| P \,;\, Q |\!\rangle_y \mid \langle\!| R \,;\, R' |\!\rangle_x$$

3. *Floating law:*

$$\langle\!| \overline{z}\widetilde{u} \mid P \,;\, Q |\!\rangle_x \equiv \overline{z}\widetilde{u} \mid \langle\!| P \,;\, Q |\!\rangle_x$$

The scope laws are standard while novelties regard workunit and floating laws. The law $\langle\!| \mathbf{0} \,;\, Q |\!\rangle_x \equiv \mathbf{0}$ defines committed workunit, namely workunit with $\mathbf{0}$ as body. These ones, being committed, are equivalent to $\mathbf{0}$ and, therefore, cannot fail anymore. The law $\langle\!| \langle\!| P \,;\, Q |\!\rangle_y \,|\, R \,;\, R' |\!\rangle_x \equiv \langle\!| P \,;\, Q |\!\rangle_y \,|\, \langle\!| R \,;\, R' |\!\rangle_x$ moves workunit outside parents, thus flattening the nesting. Notwithstanding this flattening, parent workunits may still affect the children ones by means of names. The law $\langle\!| \overline{z}\,\widetilde{u} \,|\, P \,;\, Q |\!\rangle_x \equiv \overline{z}\,\widetilde{u} \,|\, \langle\!| P \,;\, Q |\!\rangle_x$ floats messages outside workunit boundaries. By this law, messages are particles that independently move towards their inputs. The intended semantics is the following: if a process emits a message, this message traverses the surrounding workunit boundaries until it reaches the corresponding input. In case an outer workunit fails, recoveries for this message may be detailed inside the handler processes.

The dynamic behavior of processes is defined by the reduction relation where we use the shortcut:

$$\langle\!| P \,;\, Q |\!\rangle \stackrel{\text{def}}{=} (z)\langle\!| P \,;\, Q |\!\rangle_z \text{ where } z \notin \text{fn}(P) \cup \text{fn}(Q)$$

**Definition 2.** *The* reduction relation $\rightarrow$ *is the least relation satisfying the following axioms and rules, and closed with respect to* $\equiv$, $(x)\_$, $\_ \,|\, \_$, *and* $\langle\!| \_ \,;\, Q |\!\rangle_z$*:*

(COM)

$$\overline{x_i}\,\widetilde{v} \,|\, \textstyle\sum_{i\in I} x_i(\widetilde{u_i}).P_i \;\rightarrow\; P_i\{\widetilde{v}/\widetilde{u_i}\}$$

(REP)

$$\overline{x}\,\widetilde{v} \,|\, !x(\widetilde{u}).P \;\rightarrow\; P\{\widetilde{v}/\widetilde{u}\} \,|\, !x(\widetilde{u}).P$$

(FAIL)

$$\overline{x} \,|\, \langle\!| \textstyle\prod_{i\in I} \sum_{s\in S} x_{is}(\widetilde{u_{is}}).P_{is} \,|\, \prod_{j\in J} !x_j(\widetilde{u_j}).P_j \,;\, Q |\!\rangle_x \;\rightarrow\; \langle\!| Q \,;\, \mathbf{0} |\!\rangle$$

*whereJ* $\neq \varnothing \vee (I \neq \varnothing \wedge S \neq \varnothing)$

Rules (COM) and (REP) are standard in process calculi and models input-output interaction and lazy replication. Rule (FAIL) models workunit failures: when a unit abort (a message on a unit name) is emitted, the corresponding body is terminated and the handler activated. On the contrary, aborts are not possible if the transaction is already terminated (namely every thread in the body has completed its own work), for this reason we close the workunit restricting its name.

Interested readers may find all the definitions and proofs with an extensive explanation for the extensional semantics, the notions of barb, process contexts and barbed bisimulation in [13]. Definitions for Labelled Semantics, asynchronous bisimulation, labelled bisimilarity and the proof that it is a congruence are also present. Finally, results relating barbed bisimulation and asynchronous labeled bisimulation as well as many examples are discussed. A core BPEL is encoded in $\mathtt{web}\pi_\infty$ and a few properties connected to this encoding are proved for it.

## 4 A Case Study: the BPEL RF

One of the unsatisfactory things about the encoding of the BPEL RF we presented in [12] is that it was hardly readable for humans. The goal was to capture in that encoding all the hidden details of the BPEL semantics and working out the full theory also for verification purpose. But surely we lost something in readability since the target for that encoding were not humans but machines. Many people who approached our work justified their problems in understanding the encoding claiming that was exactly the proof of the BPEL recovery framework complexity. This is definitely true but, in order to be really useful, that work needs to be understandable also to non-specialists (and humans in general). With the goal of better understanding how the BPEL RF works, in this section we analyze a case study where $\mathtt{web}\pi_\infty$ shows its power. We will firstly report the description of the mechanisms following the original BPEL specification, then we will consider a simplification of the actual mechanisms giving a simplified semantics and a simplified explanation. In this way some details will be lost but we will improve readability. The first simplification is considering only the case in which a single handler exists for each of the three different type (fault, compensation and event). Furthermore, we do not consider interdependencies between the mechanisms: default handlers with automatic compensation of inner scope. This study is an integration of what done before in [12] and [15]. The semantics provided is not the one implemented by the engines supporting BPEL, we have already given a formalization for the Oracle BPEL Manager in [13]. While in [12] you can find a complete description, here we want to focus only on the essence of the single mechanisms to understand at which stage of the execution they play their role and in which way.

## 4.1 Details from the BPEL Specification

Instead of assuming a perfect roll-back in case of failure, BPEL supports in its RF the notion of the so-called *loosely coupled transactions* and the explicit programming of compensation activities. This kind of transactions lasts long periods (atomicity needs to be relaxed wrt ACIDity), crosses administrative domains (isolation needs to be relaxed) and possibly fails because of services unavailability etc... They usually contain the description of three processes:

- body
- fault handler
- compensation handler

BPEL also adds the possibility to have a third kind of handler called the event handler. The whole set of activities is included in a construct called `scope` introduced as follows in the specification:

> "A `scope` provides the context which influences the *execution behavior* of its enclosed activities. This behavioral context includes variables, partner links, message exchanges, correlation sets, *event handlers*, *fault handlers*, a *compensation handler*, and a termination handler [...]
> Each `scope` has a required primary activity that defines its normal behavior. The primary activity can be a complex structured activity, with many nested activities to arbitrary depth. All other syntactic constructs of a `scope` activity are optional, and some of them have default semantics. The context provided by a `scope` is shared by all its nested activities."

In the following, we report the way in which the concepts of the Recovery Framework and the need for it are motivated in [18].

### Compensation Handler

> "Business processes are often of long duration. They can manipulate business data in back-end databases and line-of-business applications. Error handling in this environment is both difficult and business critical. The use of ACID transactions is usually limited to local updates because of trust issues and because locks and

isolation cannot be maintained for the long periods during which fault conditions and technical and business errors can occur in a business process instance. As a result, the overall business transaction can fail or be cancelled after many ACID transactions have been committed. The partial work done must be undone as best as possible. Error handling in BPEL processes therefore leverages the concept of compensation, that is, *application-specific activities that attempt to reverse the effects of a previous activity that was carried out as part of a larger unit of work that is being abandoned*. There is a history of work in this area regarding the use of Sagas and open nested transactions. BPEL provides a variant of such a compensation mechanism by providing the ability for flexible control of the reversal. BPEL achieves this by providing the ability to define fault handling and compensation in an application-specific manner, in support of Long-Running Transactions (LRT's) [...] *BPEL allows scopes to delineate that part of the behavior that is meant to be reversible* in an application-defined way by specifying a compensation handler. Scopes with compensation and fault handlers can be nested without constraint to arbitrary depth.[...]
A compensation handler can be invoked by using the `compensateScope` or `compensate` (together referred to as the "compensation activities"). A compensation handler for a scope MUST be made *available for invocation only when the scope completes successfully. Any attempt to compensate a scope, for which the compensation handler either has not been installed or has been installed and executed, MUST be treated as executing an* `empty` *activity.* [...]"

## Fault Handler

"Fault handling in a business process can be thought of as *a mode switch from the normal processing in a scope*. Fault handling in BPEL is designed to be treated as "reverse work" in that its aim is *to undo the partial and unsuccessful work of a scope in which a fault has occurred*. The completion of the activity of a fault handler, even when it does not rethrow the handled fault, is not considered successful completion of the attached scope. *Compen-*

> *sation is not enabled for a scope that has had an associated fault handler invoked.*
> Explicit fault handlers, if used, attached to a scope provide a way to define a set of custom fault-handling activities, defined by catch and catchAll constructs. Each catch construct is defined to intercept a specific kind of fault, defined by a fault QName. An optional variable can be provided to hold the data associated with the fault. If the fault name is missing, then the catch will intercept all faults with the same type of fault data. The fault variable is specified using the faultVariable attribute in a catch fault handler. The variable is deemed to be implicitly declared by virtue of being used as the value of this attribute and is local to the fault handler. It is not visible or usable outside the fault handler in which it is declared. A catchAll clause can be added to catch any fault not caught by a more specific fault handler."

**Event Handler**

> "Each scope, including the process scope, can have a set of event handlers. *These event handlers can run concurrently and are invoked when the corresponding event occurs* [...] There are two types of events. First, events can be inbound messages that correspond to a WSDL operation. Second, events can be alarms, that go off after user-set times."

## 5 Formal Semantics of a (Simplified) BPEL RF

The plain text description of these mechanisms taken from the specification should give an idea of the complexity of this framework. The main difficulty we have found at the beginning of this investigation was to clarify the basic difference between failure and compensation handlers, since many words have been spent on this but the true essence of these mechanisms has never been given in a concise and simple way. In the past we also promoted a complete explanation of the mechanisms focusing on inessential minor details. Here we want to give the basic idea explaining that failure and compensation handlers differ mainly because they play their role at different stages of computation: failure handler is responsible for reacting to signals that occur during the normal execution of the body; when these occur, the body is interrupted and the failure handler is activated. On the contrary, compensation handler is installed only when the body successfully terminates. It remains available if another activity requires some undo of the committed activity. In some sense, failures regard "living" (not terminated) processes, while compensation is only for "successfully terminated process". The key point regarding event handlers is instead bound to the sentence reported above: they are *invoked concurrently* to the body of a scope that meanwhile continues running. This is very different from what happens for failures that interrupt the main execution and compensations which run only after the completion of the relative body.

The difficulty of the encoding we gave in [12] lies in the nontrivial interactions between the different mechanisms and it is due to the sophisticated implicit mechanism of recovery activated when designer-defined fault or compensation handlers are absent. Indeed, in this case, BPEL provides backward compensation of nested activities on a causal dependency basis relying on two rules:

- control dependency: links and sequence define causality
- peer-scope dependency: the basic control dependency causality is reflected over peer scopes

These two rules resemble some kind of structural inductive definition, as is usually done in process algebra. It is exactly our goal to skip these details here and to clarify the semantics.

## 5.1 Syntax

Let $(\mathtt{A};\, \mathtt{H})_s$ be a scope named $s$ where `A` is the main activity (body) and `H` a handler. Both `A` and `H` have to be intended as BPEL activities coming from a subset of the ones defined in [12]. Practically, that work was limited to basic activities, structured activities and error handling. The idea now is to represent a simplified BPEL scope called $s$ having a single handler `H`, so we are providing a semantics for the error handling mechanisms alternative to the previous one. For the sake of simplicity, we start considering a single handler at a time. Afterward we will consider the full scope construct. In the following subsection the formal semantics derived from $\mathtt{web}\pi_\infty$ will be presented, here we just define the syntax giving an informal explanation.

**Definition 3 (Compensation Handler).** *We define the compensation handler as follows:*

$$(\mathtt{A}; \mathtt{COMP}\ s \rightarrow \mathtt{C})_s$$

*If s is invoked after the successful termination of* A*, then run the allocated compensation* C*.*

**Definition 4 (Fault Handler).** *We define the fault handler as follows:*

$$(\mathtt{A}; \mathtt{FAULT}\ f \rightarrow \mathtt{F})_s$$

*If f is invoked in* A*, then abort immediately the body* A *and run* F*.*

**Definition 5 (Event Handler).** *We define the event handler as follows:*

$$(\mathtt{A}; \mathtt{EVENT}\ e \rightarrow \mathtt{E})_s$$

*If e is invoked in* A *then run* E *in parallel while the body* A *continues running still listening for another event e.*

## 5.2 Semantics

The formal semantics of the three mechanisms is defined here in terms of $\mathtt{web}\pi_\infty$. These constructs are encoded in $\mathtt{web}\pi_\infty$ which has a formal semantics, as a consequence the semantic of the constructs themselves is given. The continuation passing style technique is used like in [12]. Briefly, $[\![\![A]\!]\!]_y$ means that the encoding of the BPEL activity $A$ completes with a message sent over the channel $y$. More details can be found also in [13]. In that work the function $[\![\![A]\!]\!]_y : A_{BPEL} \rightarrow \mathit{Process}$ has been used to map BPEL activities into $\mathtt{web}\pi_\infty$ processes flagging out $y$ to signal termination.

## 5.3 Compensation Handler

**Definition 6 (Compensation Handler).** *The semantics of the single Compensation Handler scope is defined in terms of* $\mathtt{web}\pi_\infty$ *as follows:*

$$(\mathtt{A}; \mathtt{COMP}\ \ s \rightarrow \mathtt{C})_s \stackrel{\text{def}}{=} (y)(y')(\langle\!| [\![\![A]\!]\!]_y \ ; \ s().[\![\![C]\!]\!]_{y'} |\!\rangle_y)$$

The reader will realize that there are two new names $y$ and $y'$ defined at the outer level. This means that all the interactions related to this name are local to this process, *i.e.,* interferences from the outside are not allowed (they are restricted names). Then you have a workunit containing the main process and the compensation handler. Both these processes are, in turn, contained by the double brackets, which means that their encodings need to be put here. As you can see the compensation is blocked until a message on $s$ (the name of the scope) is received and $C$ will be available only after the successful termination of $A$ signaled on the local channel $y$. This expresses exactly the fact that the compensation is available only after the successful termination of the body as required in the BPEL specification. The reason for which $C$ is activated after the termination of $A$ stands in the web$\pi_\infty$ rule (FAIL) which activates the workunit handler $s().[\![\![C]\!]\!]_{y'}$ when the signal $y$ (the workunit name) is received. This name is precisely sent by $A$ when it terminates (because of the continuation passing style encoding).

## 5.4 Fault Handler

**Definition 7 (Fault Handler).** *The semantics of the single Fault Handler scope is defined in terms of* web$\pi_\infty$ *as follows:*

$$(\mathtt{A};\mathtt{FAULT}\ \ f \rightarrow \mathtt{F})_s \stackrel{\text{def}}{=} (f)(y)(y')(\langle\!|\, [\![\![A]\!]\!]_y \,;\, [\![\![F]\!]\!]_{y'} \,|\!\rangle_f)$$

The fault handler has a semantic very close to the web$\pi_\infty$ workunit. For this reason the encoding here is basically an isomorphism. The handler is triggered when receiving the signal $f$ which interrupts the normal execution of the body. Since the activation of the fault handler is internal to the scope itself, the scope name is not relevant in the right hand side.

## 5.5 Event Handler

**Definition 8 (Event Handler).** *The semantics of the single Event Handler scope is defined in terms of* web$\pi_\infty$ *as follows:*

$$(\mathtt{A};\mathtt{EVENT}\ \ e \rightarrow \mathtt{E})_s \stackrel{\text{def}}{=} (e)(y)(y')(\langle\!|\, [\![\![A]\!]\!]_y \,;\, \mathbf{0} \,|\!\rangle_y \mid !e().[\![\![E]\!]\!]_{y'})$$

The event handler is interesting. The main point here is that the body execution is not interrupted when $e$ is received. Consider indeed that $E$ is outside the workunit and it is triggered only by $e$. The handler, receiving $e$ and activating $E$, will run in parallel with $A$ without interrupting it. It is worth noting also that the presence of the replication allows $e$ to be received many times during the execution of $A$, each time running a new handler. The event handler will stay active without any risk of being stopped by other scopes since all the names inside the handler are local to $E$ (bound names) due to the way in which BPEL activities are encoded by the function $[\![\![A]\!]\!]_y$. This is a simplification to clarify the mechanism, it actually represents a deviation from the BPEL standard where the events are not restricted in this way.

## 5.6 BPEL Scope

Now that we have understood each mechanism let us put all together. We define a scope construct including all the three handlers. Again, we consider single handlers of each type with no interactions, no default handler and no automatic compensation of inner scopes.

**Definition 9 (Full Scope Construct).** *The semantics of the full scope construct is defined in terms of* $\texttt{web}\pi_\infty$ *as follows:*

$$\begin{aligned}&(A; \mathtt{FAULT}\ \ f \longrightarrow \mathtt{F}; \mathtt{EVENT}\ \ e \longrightarrow \mathtt{E}; \mathtt{COMP}\ \ s \longrightarrow \mathtt{C})_s \overset{\text{def}}{=}\\ &(e)(f)(y)(y')(y'')(y''')(\langle\!|\, [\![\![A]\!]\!]_y \ ;\ [\![\![F]\!]\!]_{y'} |\!\rangle_f\\ &\mid\ !e().[\![\![E]\!]\!]_{y''} \mid \langle\!| (x)x() \ ;\ s().[\![\![C]\!]\!]_{y'''} |\!\rangle_y)\end{aligned}$$

It is worth noting that here the name $s$ is a free global name (undefined) available to all the scopes which possibly run in parallel. The technical problem is that, in this way, the encoding is not compositional. Actually, this problem is easily fixed when the encoding is extended to the complete set of BPEL constructs, including the top level process where all the scopes are defined since there you can restrict all the names of the inner scopes. This has been done previously in [12]. The purpose of this work is just to explain in a clearer way the differences between the mechanisms of the recovery framework without presenting again the whole encoding. A synergy between this result and what we have done in [12] is left as future work.

## 5.7 Example

Let us now show an example of how this mechanism works in practice. To do this we will run a process description on the "reduction semantics machine" of $\mathtt{web}\pi_\infty$. This example serves as a clarification for all the concepts presented in this paper, especially for those readers who are not very familiar with the mathematical tools exploited in our investigation. Let us consider the following process where, for simplicity, the body and the handlers are already presented in terms of $\mathtt{web}\pi_\infty$:

$$((z)(\overline{f}\,|\,z().\mathbf{0});\mathtt{FAULT}\ \ f \rightarrow \overline{warning}\,;\mathtt{EVENT}\ \ e \rightarrow \mathbf{0};\mathtt{COMP}\ \ s \rightarrow \mathbf{0})_s$$

Looking at the previous encoding it results in the following full $\mathtt{web}\pi_\infty$ process:

$$(e)(f)(y)(y')(y'')(y''')(\langle\!|(z)(\overline{f}\,|\,z().\mathbf{0})\,|\,\overline{y}\ ;\ \overline{warning}\,|\,\overline{y'}\,|\!\rangle_f \\ |\,!e().\overline{y''}\,|\,\langle\!|(x)x()\ ;\ s().\overline{y'''}\,|\!\rangle_y)$$

where *warning* is some global channel handling the actual warning (for example displaying a message on the screen). This is a specific instance of the Full Scope Construct as defined above where event and compensation handlers are empty while the fault handler sends an empty message on the warning channel. The process $z().\mathbf{0}$ expresses the fact that we want the process to fail without allocating the compensation handler and it has to be the standard encoding when raising a failure signal to indicate that there is no successful termination. Now, applying the (FAIL) rule and the floating law, we have:

$$(e)(f)(y)(y')(y'')(y''')(\langle\!|\overline{warning}\,|\,\overline{y'}\ ;\ \mathbf{0}|\!\rangle \\ |\,!e().\overline{y''}\,|\,\langle\!|(x)x()\ ;\ s().\overline{y'''}\,|\!\rangle_y)$$

which will lead to a warning on the appropriate channel *without* activating the compensation (which would need a message on $y$) since the scope did not successfully complete. It is worth noting that the event handler remains ready to accept events but it never activates in this scenario. This happens because the channel on which the event handler listens is restricted, and this is consistent with the expected behaviour.

## 5.8 Is It Really Simpler?

The intention of this work is to demonstrate, in real life scenarios, the added value of formal methods. We believe that what has been introduced so far can been really useful in the clarification of the BPEL RF semantic. Just to stress better this point, let us recall only the complete Event Handler compilation presented in [12]:

$$\left(\begin{array}{l} EH(S_e, y_{eh}) = (y')(\{e_x \mid x \in h_e(S_e)\}) \\ \qquad\qquad\qquad en_{eh}().(\langle\!| \prod_{(x,\widetilde{u},A)\in S_e} \ !\, x(\widetilde{u}).\overline{e_x}\,\widetilde{u} \ ; \ \overline{y_{eh}} \,|\!\rangle_{dis_{eh}} \\ \qquad\qquad\qquad \mid \prod_{(x,\widetilde{u},A_x)\in S_e} \ !\, e_x(\widetilde{u}).[\![\![A_x]\!]\!]_{\overline{y'}}\,) \end{array}\right)$$

while the new one is:

$$(\texttt{A;EVENT}\ \ e \rightarrow \texttt{E})_s \overset{\text{def}}{=} (e)(y)(y')(\langle\!|[\![\![A]\!]\!]_y \ ; \ \mathbf{0}|\!\rangle_y \,|\, !e().[\![\![E]\!]\!]_{y'})$$

For the proper background please refer to [13] where you can find a detailed explanation of the encodings and all the theory. Here the idea is just to give a flavor of how this work contributes (in terms of simplification) to the improvement of the BPEL specification.

## 5.9 Design of BPEL Orchestration Engines

Although this paper has to be intended as investigating a well known case study and providing methodological arguments for the adoption of formal methods in software specification, the aspect of verification is not alien to our work and here we intend to give some hints in this regard. The most common formalization of behavioral equivalence is through barbed congruence, which guarantees that equated processes are indistinguishable by external observers, even when put in arbitrary contexts. For instance, equivalent Web services remain indistinguishable also when composed to form complex business transactions. The barbed congruence in this scenario has been presented in [15]. The proposed encoding, based on the function $[\![\![A]\!]\!]_y : A_{BPEL} \rightarrow Process$, can be used to test the equivalence of BPEL processes on the basis of the barbed congruence developed in the theory. The idea is to inherit the equivalence notion from web$\pi_\infty$ to decide BPEL processes equivalence. Here, as a further contribution, we want to show that, despite its simplicity, there are many ways in which BPEL can benefit from this work exploiting this idea of behavioral equivalence. For example, our proposal can contribute to the implementation of real orchestration engines. The application example comes from one of the theorems proved in [13]:

$$\langle\!| !z(u).P \,|\, Q \ ; \ \overline{v} \,|\!\rangle_x \approx_a (y)(\langle\!| !z(u).P \ ; \ \overline{y} \,|\!\rangle_x \,|\, \langle\!| Q \,|\, (w)w(u) \ ; \ \overline{v} \,|\!\rangle_y)$$

where the symbol $\approx_a$ has to be intended as barbed congruence, i.e. the process on its left and the one on its right exhibit the same behavior. It is worth noting that the process $(w)w(u)$ is necessary to prevent $v$ from disappearing in the case the workunit on the right would terminate successfully. This theorem suggests a transformation where it is always possible to separate the body and the recovery logics of the workunit expressing, for example, the event handler behavior. This is possible not only when the recovery logic is a simple output (as in this case) but in all the other cases, on the basis of another theorem showed in the same work:

$$\langle\!| P \,;\, Q |\!\rangle_x \approx_a (x')(\langle\!| P \,;\, \overline{x'} |\!\rangle_x \mid \langle\!| x'().Q \,;\, \mathbf{0} |\!\rangle)$$

Now we have the design option to compile the mechanism in an alternative way allowing a logical separation of code which can lead to an actual physical separation. For example, different workunits could be loaded on different machines. Although BPEL typically allows a centralized control and a local compilation, this result gives us further insights in the direction of distribution. Consider, for example, the case in which different scopes can share instances of the same handler loaded on a specific dedicated machine. This result can also be interpreted in a choreographic perspective.

## 6 Summary, related works and criticisms

The goal of this paper was to show how a variant of the π-calculus can be of some use in the context of dependable Web services composition. The specific case study presented aimed at reducing the ambiguity of the BPEL RF providing a (simplified) formal semantics opposed to the complete one already given in [12]. This is what we have called the "π-calculus way", *i.e.,* using the π-calculus as formal specification language. As we have already underlined, several different formal notations might have been chosen for this purpose. Our choice depended on the "foundational feature" of *mobility*. It has been noted that in the specific contribution the mobility feature has not been fully exploited since the modeled mechanisms required us to pay more attention to process synchronization and concurrency than to full mobility. Anyway, we have realized that, in the general case, mobility is an essential feature of composition languages and this point is discussed more in detail in [13].

Although before this work [15] and [12] have been earlier attempts at defining a formal semantics for WS-BPEL and unifying and simplifying its recovery mechanisms, those papers are far from being complete and from providing the ultimate BPEL formal semantics. Many other works have been presented recently that significantly improved what has been done there. For example, Blite [10] is a "lightweight BPEL" with formal semantics taking into account also dynamic aspects (e.g. dynamic compensations) that have not been directly part of our investigation. Another relevant work adding dynamic compensation features is [20]. In this paper the interested reader can find a comparison between different compensation mechanisms presented in the recent literature. The criticism in this work is that in $\mathtt{web}\pi_\infty$ completed transactions cannot be compensated. This is of course true but, as shown in this paper, this aspect can be easily modeled (look for example at the encoding of the BPEL compensation handler). The basic idea behind $\mathtt{web}\pi_\infty$ is indeed to provide a unifying theory for Web services composition as discussed in [15] where different mechanisms can be easily mapped without being directly supported. A good analysis of fault, compensation and termination (FCT) in WS-BPEL is also discussed in [7]. Here the BPEL approach to FCT with related formal semantics is given, thus covering termination handler that has not been part of our work. Furthermore, the authors in [22] recognize that in [12] the lack of support for control links has to be seen as a major drawback. And this is a criticism that we do not hide and we find relevant. The same paper proposes an alternative formalization of WS-BPEL 2.0 based on the $\pi$-calculus and then compares different approaches (including the one in [12]) from the complexity point of view for verification purposes. The authors found out that their approach presents a smaller number of states deriving from the neglect of internal activity states. Indeed, while the encoding in [12] requires every activity to signal (at least) its termination (due to the continuation passing style technique used), in [22] the activity lifecycle is not modeled. Apart from the criticisms presented in the recent literature (the list included here is not exhaustive anyway), other interesting questions have been asked regarding this approach to the BPEL RF, for example if we intend to capture fault tolerance behavior depending on external factors, for example timeout. This topic indeed has not been central to our investigation. Other authors worked on these aspects, in particular [9] discusses timed transactions.

Although we know that much needs to be done yet, we are confident that the issues we have identified are worth investigating. We have to admit that sometimes we have doubts regarding what we are doing and the solution we are adopting, so we usually look for some reassurance in the famous words of Descartes: "Dubium Sapientiae initium", i.e. "Doubt is the origin of wisdom".

**Acknowledgments.** The paper has been improved during the useful conversations with Cliff Jones, Alexander Romanovsky and Anirban Bhattacharyya. For some of the ideas discussed here we have to thank Cosimo Laneve, Roberto Lucchi, Claudio Guidi and Gianluigi Zavattaro. Very useful comments came also by Joey Coleman, Felix Loesch and Michael Jastram that kindly provided other written reviews for this work (Ani Bhattacharyya also provided a written review). Finally, we have also to thank the anonymous WSFM reviewers for their contribution. This work has been partially funded by the EU FP7 DEPLOY Project (Industrial deployment of system engineering methods providing high dependability and productivity). More details at http://www.deploy-project.eu/.

## References

1. Web services flow language (wsfl 1.0). www.ebpml.org/wsfl.htm.
2. Xlang: Web services for business process design. http://www.ebpml.org/xlang.htm.
3. R. Chinnici, J.J. Moreau, A. Ryman, and S. Weerawarana. Web services description language (wsdl 1.1), W3C Recommendation 26 June 2007. http://www.w3.org/TR/wsdl20/.
4. P. Chris. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.
5. World Wide Web Consortium. Extensible markup language (xml) 1.0. W3C Recommendation: http://www.w3.org/XML/.
6. G. Decker, F. Leymann, and M. Weske. Bpel4chor: Extending bpel for modeling choreographies. In *Proceedings International Conference on Web Services (ICWS)*, 2007.
7. Christian Eisentraut and David Spieler. Fault, compensation and termination in ws-bpel 2.0 – a comparative analysis. pages 107–126, 2009.
8. M. Gudgin, M. Hadley, N. Mendelsohn, J.J Moreau, H.F. Nielsen, A. Karmarkar, and Y. Lafon. Simple object access protocol (soap) 1.1, W3C Recommendation 27 April 2007. http://www.w3.org/TR/soap12-part1/.
9. Cosimo Laneve and Gianluigi Zavattaro. Foundations of web transactions. pages 282–298. Springer, 2005.
10. A. Lapadula, R. Pugliese, and F. Tiezzi. A formal account of WS-BPEL. In *Proc. 10th international conference on Coordination Models and Languages (COORDINATION'08)*, volume 5052 of *Lecture Notes in Computer Science*, pages 199–215. Springer, 2008.
11. M. Little. Web services transactions: Past, present and future. www.jboss.org/jbosstm/resources/presentations/XML2003.pdf.
12. R. Lucchi and M. Mazzara. A pi-calculus based semantics for ws-bpel. *Journal of Logic and Algebraic Programming*, 70(1):96–118, 2007.

13. M. Mazzara. *Towards Abstractions for Web Services Composition*. PhD thesis, 2006.
14. M. Mazzara and S. Govoni. A case study of web services orchestration. In *COORDINATION*, pages 1–16, 2005.
15. M. Mazzara and I. Lanese. Towards a unifying theory for web services composition. In *WS-FM*, pages 257–272, 2006.
16. R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.
17. Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.
18. OASIS Web Services Business Process Execution Language (WSBPEL) TC. Web services business process execution language version 2.0. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`.
19. W.M.P. van der Aalst. Pi calculus versus Petri nets: Let us eat humble pie rather than further inflate the Pi hype, 2004. http://is.tm.tue.nl/research/patterns/download/pi-hype.pdf.
20. Cátia Vaz, Carla Ferreira, and António Ravara. Dynamic recovering of long running transactions. pages 201–215, 2009.
21. W3C. Http - hypertext transfer protocol. `www.w3.org/protocols`.
22. Matthias Weidlich, Gero Decker, and Mathias Weske. Efficient analysis of bpel 2.0 processes using pi-calculus. In *APSCC '07: Proceedings of the The 2nd IEEE Asia-Pacific Service Computing Conference*, pages 266–274, Washington, DC, USA, 2007. IEEE Computer Society.

# Generation of Gluing Invariants for Checking Local Enforceability of Message Choreographies

Vitaly Kozyura and Andreas Roth

SAP

**Abstract.** The paper describes how to check local enforceability (refinement) property for Message Choreography Models (MCM) using Event-B specification language. We consider the definition of MCM and its translation to Event-B. The challenge is to find (heuristically) a number of gluing invariants between local and global models, which allow one to prove the refinement property. The invariants are constructed partially automatically from the translation of MCM to Event-B. Our experiments have shown that for the most of the considered message choreographies the refinement property could be successfully proved using only the proposed types of invariants.

## 1 Introduction

Introducing formal modeling and verification into the development of business information application is a challenging task. There are specialized established domain-specific languages for the development of this software in place which developers need to use. Switching to non-domain-specific languages and working with mathematical syntax is typically considered not to be feasible.

One approach to make business application developers nevertheless benefit from formal verification techniques is to hide the mathematical formalisms, such as Event-B, behind the languages normally used in business software development and to rely on the high automation of today's verification tools, so that developers do not have to directly interact with the formalism.

In earlier work [6] we have presented a diagrammatic domain-specific language called Message Choreography Modeling (MCM) to precisely model message choreographies in the context of SAP applications. MCMs describe the allowed order of how messages are exchanged between two independent business components. This domain is very important in service-oriented architectures and modeling the choreography can uncover in a very early design phase crucial flaws of the design. MCM consists of two layers: one for the component-neutral global view, and one for modeling the send and receive events of the individual components (local view).

Our approach to hide the mathematical formalism behind domain-specific front-ends imposes a number of challenges. First there must be a semantics preserving translation from the state transition systems of global and local MCM models to Event-B. Second, since formal verification in Event-B heavily relies on the addition of additional invariants enriching the model, we need to cater for the addition of invariants in MCM. Third, one needs to take care that results of the verification process, e.g. failed proof attempts, are adequately presented in the MCM diagram.

We have introduced the translation procedure and its implementation in earlier work [5] and will report on our experiments with the third task in future work. This paper focuses on the second activity, i.e. how to make the process of specifying invariants for users of our approach easy. The invariants are crucial in verifying MCM because the result of the translation from MCM are two Event-B models, one for the local model and one for the global model. Since we would like to show that the local model is a refinement of the global model, gluing invariants are needed which link the two models together.

Our approach relies on templates of gluing invariants which we typically find for MCM. A user of our tool may then select instances of these templates which she/he believes are applicable for the concrete MCM instance. This selection is considered much more feasible for the tool users than creating the invariants from scratch. It moreover exploits the domain-specific information reflected in MCMs. Finding the gluing invariant templates is a heuristic task. We have thus conducted a number of experiments with concrete realistic MCM examples from SAP and report on our experience with the approach.

The paper is structured as follows. In Section 2 we introduce the MCM modeling approach and the translation of MCM into Event-B. Section 3 presents our approach to generating gluing invariants for MCM. How users are interacting with the tool to specify the invariants is described in Section 3.4. Then the process of working with the tool is illustrated in Section 4. In Section 5 the results of our experiments are presented and conclusions are drawn in Section 6.

## 2 MCM Modeling

The service choreography modeling language MCM complements the structural information of the communicating components (e.g. service interface descriptions and message types) with information on the message exchange between them. A detailed discussion of the underlying concepts of MCM and how they support service development can be found in [7, 6]. MCM consists of different model types each defining different aspects of service choreographies.

1. Global Choreography Model. The global choreography model (GCM) is a labeled transition system which specifies a high-level view of the conversation between service components. Its purpose is to define every allowed sequence of observed messages.
2. Local Partner Model. The local partner models (LPMs) specify the communication-relevant behavior for exactly one participating service component. Each LPM must be a structural copy of the GCM with extra constraints on some of the local transitions, usually leading to the affected sending actions being deactivated.
3. Channel Model. The channel model (CM) describes the characteristics of the communication channel on which messages are exchanged between the service components. These characteristics determine for example whether messages sent by one component preserve their order during transmission and are formalized by the WS-RM standard [1].

The following running example (as in [5]) illustrates the approach: Two service components, a buyer and a seller, negotiate a sales order. The buyer starts the communication by sending a *Request* message that will be answered with a *Confirm* by the seller. The buyer afterwards has the choice either to send a *Cancel* that rolls back the previous communication and allows to restart the negotiation or to send an *Order* that successfully concludes the ordering process. We assume a (reliable) communication channel that is not necessarily preserving the message order. Because of this a Cancel can be delivered after a new negotiation process already started.

Figure 1 shows how the example described above can be modeled with MCM. In the GCM at the top of Figure 1, the arrows labeled with an envelope depict the interactions *Request*, *Confirm*, *Cancel*, *Order*, and

*Cancel(deprecated)*[1] which are ordered with the help of the states *Start*, *Request*, *Reserved*, and *Ordered*. The states connected with a filled circle, i.e. *Ordered* and *Start* are so-called target. Only in these states, the communication between the partners is allowed to terminate.

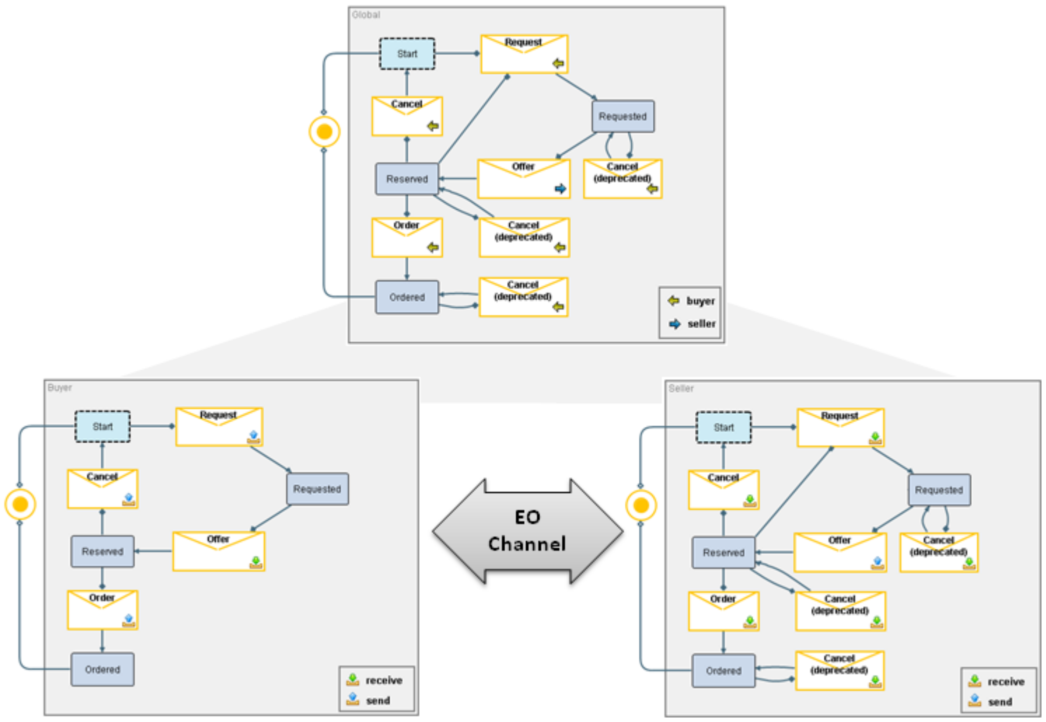


**Fig. 1.** GCM (top) of the choreography and LPMs of the buyer (left) and the seller (right)

The LPM of the buyer partner of our example is depicted in the lower left part of Figure 1. It is a structural copy of the GCM, but the interaction symbols now represent send or receive events of the buyer. Moreover some send-events are "inhibited" by special local constraints. It is for example inhibited that a *Cancel(deprecated)* is ever sent (thus these send-events have been erased) and that a *Request* is sent in the *Reserved* state. However, due to possible message overtaking on a channel that does not guarantee to enforce the message order during transmission, receiving a

[1] Deprecated here means that the message is out-dated and no-longer relevant as the negotiation has been restarted.

deprecated *Cancel* is possible on the seller side. The LPM of the seller is depicted in the lower right part of Figure 1. Event-B fits quite naturally to MCM: interactions can be seamlessly expressed as events and the relationship between GCM and LPMs can be formulated as Event-B refinement. The distinguishing aspect is the tool support in form of the Eclipse-based Rodin tool [2]. Due to the extensible architecture, various plugins for Rodin exist. The tool can be integrated with other Eclipse-based tools such as the MCM editor.

**Abstract Syntax of MCM**. Now we present the abstract syntax of MCM, which is the basis for the translation into Event-B. Although in our tool we consider choreographies having multiple status variables, for a simplified presentation, we assume that each choreography consist of exactly one status variable. In this case a choreography can be considered as based on a finite state machine. A message choreography model $MCM = (GCM, LPM_1, LPM_2, CM)$ consists of a global choreography model ($GCM$), two local partner models ($LPM_1$ and $LPM_2$) and a channel model $CM$. The $GCM$ is a finite state machine $L = (S, I, \rightarrow)$, where $S$ is a finite set of states, $I$ is a finite set of interactions and $\rightarrow \subseteq S \times I \times S$. The system has an initial state $init \in S$ and target states $\{e_1, ..., e_n\}$, where $e_i \in S$. Each interaction $i \in I$ is then assigned to a type $itype(i) \in T$, where $T$ in our context can be considered as being a set of labels. We demand our models to be confluent in the following sense: Starting from the same starting state, the applications of interactions $e_1, e_2$ and $e_2, e_1$ lead to the same resulting state.

$LPM_1$ and $LPM_2$ are obtained from the $GCM$ by duplicating the states, for each of them. Moreover each interaction $i \in I$ is transformed into the corresponding element from
$PI = \{send\_i, receive\_i \mid \text{for all } i \in I\}$. The elements from $PI$ inherit types, states, preconditions and actions from elements from $I$. LPMs can be further extended with an additional inhibitor function $inhib : I \rightarrow \mathcal{P}(S)$ which describes that the partner must not send a message associated with $I$ if it is in one of the states $inhib(i)$.

The channel model $CM$ is a total function from a sequence of messages (of types $T$) to a sequence of messages (of types $T$). With $T' \subseteq T$ and a message sequence $s$, $\pi_{T'}(s)$ denotes the projection of s to sequences of messages of types $T'$. Let $\pi_{T'}$ be canonically extended on the channel model. The channel model $CM$ is then based on assignments of disjoint subsets $T'$ of $T$ to channel reliability guarantees which enforce that

$\pi_{T'}(CM)$ satisfies certain properties. Reliability guarantees such as from WS-RM standard [1] can be modeled:

- Exactly once in order (EOIO) where $\pi_{T'}(CM)$ is the identity function on interaction sequences.
- Exactly once (EO) where $\pi_{T'}(CM)$ is a permutation on an interaction sequence.

**Translation to Event-B**. Here we present a translation of MCM into Event-B as described in detail in [5]. The translation is implemented and can thus be applied completely automatically. For each transition in the GCM we generate exactly one event. For representing the states we define a global variable status with elements from a set type $\{s_1,...,s_k\}$, with constants $s_1,...,s_k$. It is initialized with $init \in S$. The basic translation of an Interaction $i \in I$ with $(s_1,i,s_2) \in\rightarrow$ is as follows:

```
i of type t
  when
    $guard1 : status = s1$
  then
    $act1 : status := s2$
  end
```

For the target state $e \subseteq S$ we define a special event terminate with a guard $status = c_1 \vee ... \vee status = c_n$ (for all $c_i \in e$) and an action $targetstate := true$, where $targetstate$ is a global variable. In each event from the translation of GCM we additionally add an action $targetstate := false$. As a result, targetstate equals true iff the system state is a target state.

In the local model we generate events representing sending and receiving of messages. Depending on the viewpoint either the send or the receive event can be defined to be a refinement of the corresponding interaction in GCM. By definition of LPMs, the status variable is duplicated (one for each partner). In receive events, local variables (parameters) are used in order to obtain some message from a channel. A channel is defined as a global variable of type $\mathcal{P}(T)$ denoting the set of messages on the being exchanged. It is initialized with $\varnothing$. Typically, we have two partners $P_1$ and $P_2$ and two sequencing contexts (EO and EOIO). In that case we obtain four possible channels in the model (two in each direction).

*Example*. Below we show a translation of the interactions Request from the GCM and the LPMs for the partners buyer (B) and seller (S) of the example. The duplicated variables can be distinguished by the corresponding prefixes. The channel from buyer to seller having the sequencing EO is denoted by $channel\_BS\_EO$.

```
Request
  when
    $grd1 : status = Reserved \lor status = Start$
  then
    $act1 : status := Requested$
  end
```

```
send_Request
  when
    $grd1 : B\_status = Reserved \lor B\_status = Start$
  then
    $act1 : B\_status := Requested$
    $act2 : type(msg) := Request$
    $act3 : channel\_BS\_EO := channel\_BS\_EO \cup \{msg\}$
    $act4 : msg := msg + 1$
  end
```

```
receive_Request
  any  $m$  where
    $grd1 : S\_status = Reserved \lor S\_status = Start$
    $grd2 : m \in channel\_BS\_EO$
    $grd3 : type(m) = Request$
  then
    $act1 : S\_status := Requested$
    $act2 : channel\_BS\_EO := channel\_BS\_EO \setminus \{m\}$
  end
```

For inhibitor conditions $inhib(i) = C$ (with $i \in I$) we add a guard $status \notin C$ to the event $send\_i$. In our example, we add the guard $grd2 : B\_status \notin \{Reserved\}$ to $send\_Request$. Target states are treated similar to the translation of GCM except that we additionally demand $channel = \varnothing$ for all of them. Only if all channels are empty the system can enter into a target state. For all other events of the translation from the LPM we add an action $targetstate := false$.

The purpose of the verification procedure is to prove local enforceability property for choreographies. In [7] we have defined a notion of local enforceability as a trace inclusion: Traces of the local model must be a subset of traces of the global model. Depending on viewpoint either the send or the receive event is defined as a refinement of the corresponding interaction in the global model. In order to prove the trace inclusion we show (with the help of the translation to Event-B) that local model is a refinement of the corresponding global one.

## 3 Gluing Invariants

In this section we describe seven types of gluing invariants that we propose to add in the translation of MCM to Event-B in order to prove the refinement property described above. We also present some considerations about rationale and usage of invariants. Following notation is used below: $P_1$ and $P_2$ are two partners, $S = \{s_1, s_2, ..., s_n\}$ is a global status variable, $P_1\_S$ and $P_2\_S$ are the copies of the status variable for both partners, $m(e)$ is a message produced by the event e, and channel is a generalized channel between partners.

In this work we try to obtain a classification of gluing invariants used in order to prove the local enforceability of choreographies. The obtained types of gluing invariants between local and global models are found heuristically and allow one to prove the refinement property in some cases. The list of types is not necessarily complete and can be extended when needed. An extension is required if new types of gluing invariants are needed for some proof obligations. Our experiments have shown that for the typically considered choreographies the refinement property could be successfully proved using only the proposed types of invariants.

### 3.1 Types of Gluing Invariants

The following types of gluing invariants are currently considered:

**Type 1 (Local state implies global state)**:

$$I_1(s_j, P_i) \equiv (P_i\_S = s_j) \Rightarrow (S = s_{k_1} \vee S = s_{k_2} \vee ... \vee S = s_{k_n}).$$

The meaning of the invariants of this type is that given a state of a single partner the possible global states are represented.

**Type 2 (Message in the channel implies state)**: An event $e$ sent by the partner $P_i$ is called a non-cyclic event if it does not produce, together with other events sent by $P_i$, a cycle. Consider an event $e$ such that

1. The event $e$ is a non-cyclic event.

2. There exists a non-cyclic $e'$ of another partner $P_j$, such that $s_l \rightarrow_e s_m$ and $s_m \rightarrow_{e'} s_n$.

The second type of invariants is:

$$I_2(e) \equiv (m(e) \in channel) \Rightarrow ((S = s_{k_1} \vee S = s_{k_2} \vee ... \vee S = s_{k_n}) \wedge (P_i\_S \neq s_l)).$$

Usually we use the form $I_2(e)$, but there exist variants, which can also be useful for proving local enforceability.

**Type 3 (Bounded Event)**: The following invariant holds if the event $e$ is bounded. In a more complex case the event can be bounded by a constant $k \neq 1$ and one can change the invariant $I_3$ correspondingly.

$$I_3(e) \equiv channel(m(e)) \leq 1.$$

**Type 4 and Type 5 (Initial Value without Inputs)**: If an initial state has no input (i.e., there exists no event with an action $S := initial$), then the following types of invariants can be useful:

$$I_4(P_i) \equiv (P_i\_S = initial) \Rightarrow (S = s_{k_1} \vee S = s_{k_2} \vee ... \vee S = s_{k_n}) \ \wedge$$

$$(m(e_1) \notin channel \wedge m(e_2) \notin channel \wedge ... \wedge m(e_k) \notin channel),$$

for all non-cyclic events $\{e_1, e_2, ..., e_k\}$ of the partner $P_i$.

$$I_5(P_i) \equiv (S = initial) \Rightarrow (P_i\_S = initial) \ \wedge$$

$$(m(e_1) \notin channel \wedge m(e_2) \notin channel \wedge ... \wedge m(e_k) \notin channel),$$

for all non-cyclic events $\{e_1, e_2, ..., e_k\}$ of the partner $P_i$ with initial value in the guards.

**Type 6 (Concurrent Events)**: If two events ($e_1$ and $e_2$) of the same partner have the same status value in guards and both events are non-cyclic (with respect to the partner), then the following type of invariants can be needed:

$$I_6(e_1, e_2) \equiv m(e_1) \notin channel \vee m(e_2) \notin channel$$

.

**Type 7 (Variable modified by Partner)**: If a status variable is changed (in send or receive events, corresponding to the point of view) only by the partner $P_i$, then we can write

$$I_7(S,P_i) \equiv (S = P_i\_S)$$

## 3.2 Some Rationale of the Types of Gluing Invariants

In this section we give some rationale for the first three types of invariants. For each event $e$ and the corresponding send or receive (dependent on used send or receive point of view) event $e'$ in order to show the refinement relation we obtain the following proof obligation:

$$Guard(e') \Rightarrow Guard(e) \quad (GRD\ \ PO).$$

For the rest of the section we fix a send point of view.

**Type 1**: Let the partner $P_i$ sends $e$ with $s_1 \rightarrow_e s_2$. In order to prove the GRD PO we consider the invariant $I_1(s_1,P_i)$, which is usually enough.

**Type 2**: Let us consider a receive-event $e_r$ corresponding to the acyclic event $e$ with $s_1 \rightarrow_e s_2$ and an invariant of a first type $I_1(e',P_i)$ corresponding to the acyclic event $e'$ with $s_2 \rightarrow_{e'} s_3$. Before receive-event $e_r$ the invariant holds because the event $e$ is acyclic. After the receive event we have to prove

$$(s_2 = s_2) \Rightarrow (S = s_{k_1} \vee S = s_{k_2} \vee ... \vee S = s_{k_n}).$$

This can be proved only if we have additional information about global states of status during the receive-event $e_r$. Since global states are changed by send events, we need a connection between send and receive events.

The following invariants can be formulated in this case:

$$I_2^0(e) \equiv m(e) \in channel \Rightarrow (S = s_{k_1} \vee S = s_{k_2} \vee ... \vee S = s_{k_n}).$$

Now this invariant cannot be proved by the acyclic send-event $e'_s$ with $s_2 \rightarrow_{e'} s_3$. We need to show that if a message $m(e')$ is sent, then the message $m(e)$ is already received, i.e.:

$$I_2^{add}(e) \equiv m(e') \in channel \Rightarrow m(e) \notin channel.$$

This can be done by extending $I_2^0(e)$ to $I_2(e)$ or by using $I_2^{add}(e)$. The invariant $I_2(e)$ can be usually proved by contradiction using the first types of invariants.

**Type 3**: The invariant of the second type $I_2(e)$ cannot in general be proved in the case of receive-event $e_r$. We have to prove that after deleting a message from the channel the invariant for an event $e$ still holds. In general the right side of the implication in $I_2(e)$ is false and the implication holds only if $m(e) \notin channel$, i.e., we have to prove that after one deletes from the channel a message corresponding to the event $e$, there will be no more such messages in the channel. To prove this we should demand additionally that $channel(m(e)) \leq 1$. The invariant holds if the event $e$ is bounded. In a more complex case the event can be bounded by a constant $k \neq 1$ and one changes the invariant $I_3(e)$ correspondingly.

### 3.3 Automatic Generation of Gluing Invariants

In this section we consider the construction of the invariants on the example of invariants of first type. We show that in the case of the send point of view the construction is automatic and in the case of the receive point of view some user interaction is needed. This is also the case with the second type of invariants. All other types of invariants can be trivially constructed. In the case of the send point of view the following algorithm can be used for the calculation of the invariant $I_1(s_j, P_i)$:

1. Start with $X = s_j$.
2. Repeat until $X$ is fixed:
   Add all $s'$ to $X$ such, that $s \rightarrow_{e'} s'$, where $e'$ is an event sent by another partner $P_j$ and $s \in X$.
3. Write $I_1(s_j, P_i)$ for all $s_j \in X$.

The intuition here is that in the case of the send point of view the global state is either $s_j$ or another partner $P_j$ has sent some messages starting from $s_j$. Note that the choreographies that we consider are confluent (see Section 2).

In the case of receive point of view calculating the possible global states from the partner's state $s_j$ one should consider the situation, where another partner is still not in the state $s_j$. This corresponds to the situation where some messages were not received. Compared to the send point of view, where a forward calculation starting from $s_j$ is needed, in the case of receive point of view one should calculate backward starting from $s_j$. But in this case also the possible channel content should be considered. This makes the later calculation to a hard (if at all decidable) problem. Therefore in the case of receive point of view we give user a possibility to correct the proposed set $\{s_{k_1}, ..., s_{k_n}\}$ manually.

### 3.4 Supporting GUI

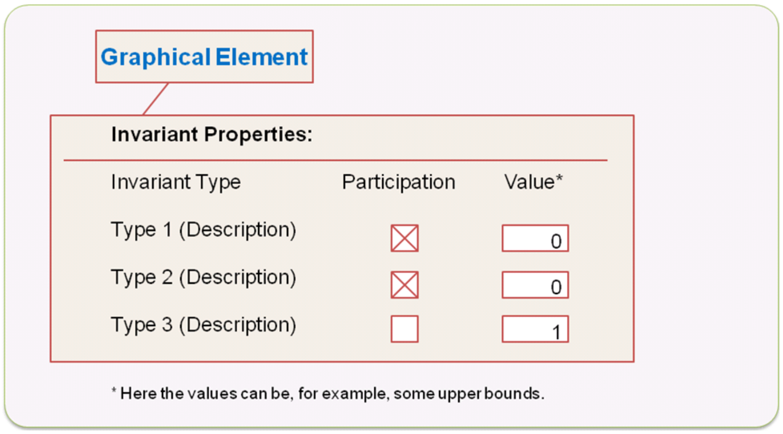
Because the types of invariants and their usage are obtained in the heuristic way, we are not sure that the proposed set of invariants is optimal in each concrete case. In our approach we give users the possibility to correct the set of proposed invariants.

An invariant is in general connected (logically) with the following modeling elements: model itself, status values, status variables (note that in general there is more than one of them), interactions, local partners. Each modeling element has also its graphical representation (graphical element) in the tool. In Fig. 2 we show how the graphical user interface for working with invariant proposal looks like. For example in the case of the second type of invariants the automatic proposal is: Model - always, Variable - always, Value - always, Partner - always, Interaction - only if the interaction is acyclic and has an interaction sent by another partner as a successor.

As we have mentioned already, in the case of receive point of view not all types of invariants can be constructed automatically. It was shown in the previous section, that the status values in invariants of the first and the second types cannot be calculated automatically in general. In this case some graphical interface similar to the one in Fig. 2 can be provided to the user in order to fulfill (correct) the automatically generated versions of invariants. The tool still makes an initial proposal.

## 4 Verification Procedure

In this section we describe the whole verification procedure for MCM using the types of invariants proposed in this paper. Given an MCM model



Graphical Element

**Invariant Properties:**

| Invariant Type | Participation | Value* |
|---|---|---|
| Type 1 (Description) | ☒ | 0 |
| Type 2 (Description) | ☒ | 0 |
| Type 3 (Description) | ☐ | 1 |

* Here the values can be, for example, some upper bounds.

**Fig. 2.** GUI for participation of gluing invariants

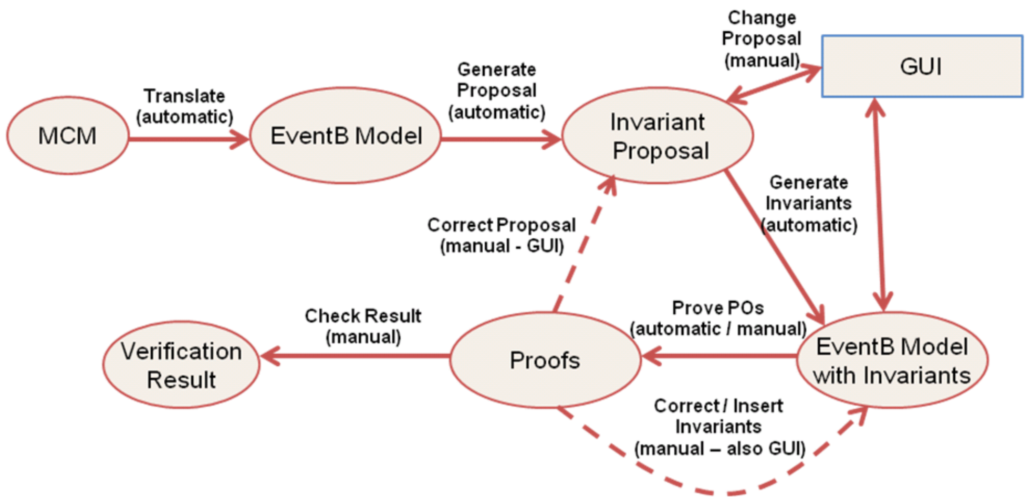

MCM
Translate (automatic)
EventB Model
Generate Proposal (automatic)
Invariant Proposal
Change Proposal (manual)
GUI
Generate Invariants (automatic)
EventB Model with Invariants
Prove POs (automatic / manual)
Proofs
Correct Proposal (manual - GUI)
Check Result (manual)
Verification Result
Correct / Insert Invariants (manual – also GUI)

**Fig. 3.** Verification Procedure

the purpose is to check is the model is local enforceable (by fixed send or receive point of view).

In Fig. 3 the whole verification procedure is schematically depicted. First of all an MCM is translated to Event-B model as described above (both global and local parts). After this the automatic invariant proposal is generated. The proposal can be changed each time using the graphical user interface (GUI) as described above. Having an invariant proposal the Event-B model can be extended with automatically generated invariants.

In the case of receive point of view, where we have no guarantee, that the constructed invariants are correct, the GUI can help one to edit the sets of status values needed for invariant generation. Having an Event-B model with generated invariants, one can start with the proving of the generated prove obligations. In Rodin this can be done in both automatic and manual modes. If during the proofs the necessity of new invariants (or corrected versions of the existing invariants) appears, then the modeler can (through the GUI) help either by changing the invariant proposal or by correcting content of the invariants.

## 5 Results

In the framework of the verification procedure described above we have tried to verify a number of typical and realistically sized choreographies from the SAP context. The following statistics was obtained with Rodin 0.9 and is based on seven considered case studies.

We have used the send point of view allowing automatic generation of proposed invariants and all considered examples could be successfully verified by using only the automatically proposed types of invariants. The numbers of generated proof obligations were between 300 and 600 depending on the example. About 70% of them could be proved automatically (with repeated runs of the auto-provers). Up to 80% of the remaining proof obligations could be successfully solved by manually calling the ML, P0 and P1 provers of Rodin. The fact that these proofs have not been obtained during the runs of auto-provers can be partially explained by the small timeout number set in the auto-provers (which currently cannot be changed from the GUI). The remaining proof obligations were solved manually (about 6% from the whole number of proof obligations).

## 6 Conclusion and Related Work

In this work we have investigated means to integrate modeling in industrial domain-specific languages with formal modeling in Event-B/Rodin. We have built on existing work which automatically translates extended finite state machines like choreography models into Event-B. We have shown that the fact that models are created in a special domain, here message choreographies, can help to even create and propose advanced modeling constructs, such as gluing invariants, here the coupling between global and local views of the choreographies. We have shown that this approach has been useful in practical realistic applications.

As a related work we mention here the project BART [4], where collected libraries of known refinement schemes can be automatically applied to abstract B machines. Another approach to the automatic construction of refined specifications is based on using design patterns [3]. It tries to incorporate a proved and refined mini-model into a larger model and works with a pre-defined number of solutions of small commonly occurring problems.

In future work we would like to extend the approach to cover other kinds of invariants typically found in choreographies which are not gluing. Moreover we will explore how the approach can help to propose invariants in other domains such as business object models and business process models.

## References

1. Web services reliable messaging v1.2, 2007.
2. Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, and Laurent Voisin. A roadmap for the rodin toolset. In Egon Börger, Michael J. Butler, Jonathan P. Bowen, and Paul Boca, editors, *ABZ*, volume 5238 of *Lecture Notes in Computer Science*, page 347. Springer, 2008.
3. Jean-Raymond Abrial and Thai Son Hoang. Using design patterns in formal methods: An event-b approach. In *ICTAC*, pages 1–2, 2008.
4. Antoine Requet. Bart: A tool for automatic refinement. In *ABZ*, page 345, 2008.
5. Sebastian Wieczorek, Vitaly Kozyura, Andreas Roth, Michael Leuschel, Jens Bendisposto, Daniel Plagge, and Ina Schieferdecker. 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. Sebastian Wieczorek, Andreas Roth, Alin Stefanescu, and Anis Charfi. Precise steps for choreography modeling for SOA validation and verification. In *Proceedings of the IEEE 4th International Symposium on Service-Oriented Software Engineering (SOSE'08)*, pages 148–153. IEEE Computer Society, 2008.

7. Sebastian Wieczorek, Andreas Roth, Alin Stefanescu, Vitaly Kozyura, Anis Charfi, Frank Michael Kraft, and Ina Schieferdecker. 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.

# An Adaptation of the Time Constraint Pattern for Modelling Consistency in Business Information Systems

J. W. Bryans[1], J. S Fitzgerald[1], A. Romanovsky[1], and A. Roth[2]

[1] School of Computing Science
Newcastle University
United Kingdom
[2] SAP Research CEC Darmstadt
SAP AG, Bleichstr. 8,
64283 Darmstadt, Germany
{Jeremy.Bryans|John.Fitzgerald|Alexander.Romanovsky}@ncl.ac.uk
Andreas.Roth@sap.com

**Abstract.** Maintaining semantic consistency of data is a significant problem in distributed information systems, particularly those on which a business may depend. Our current work aims to use Event-B and the Rodin tools to support the specification and design of such systems in a way that is well integrated into existing development processes. This paper presents Event-B patterns that may be used to represent time-bounded inconsistency and illustrates their use in a model derived from industrial applications.
KEYWORDS: Real-time, Patterns, Formal Verification, Event-B, Error Recovery

## 1 Introduction

Computer-based business information systems are often critical to the successful functioning of enterprises, but are demanding to develop because of their large scale and distributed character. Our work in Deploy[3] aims to provide formal modelling and analysis technology that helps to reduce development risk by allowing early-stage comparison of design alternatives and identification of potential sources of defects. In order to promote take-up, we aim for a high degree of automation in the analysis of formal models that are derived, where possible automatically, from designs expressed in domain-specific, often diagrammatic, notations [5].

We focus on applications of the kind developed using SAP technology which support companies' business processes. These can be best

[3] www.deploy-project.eu

practice customisable processes like order-to-cash or procure-to-pay. They are assembled from components describing parts of processes, such as buying, selling, planning, site logistics and accounting. On the other hand, business processes can also be very specific to customers and can be modelled with the help of specific business process management (BPM) tools such as SAP NetWeaver BPM [22].

With the help of service-oriented architectures (SOA), business processes can be closely linked to their technical realisation using independent business application components. For a typical business process, dozens of independent business components form a complex network where (mostly asynchronous) messaging is used to satisfy the components' communication needs without giving up their loose coupling. Complexity arises from the large-scale composition of relatively simple protocols, making it a challenge to determine application-level properties such as inconsistency and race conditions.

A typical example is the business process *Sell from Stock*, which describes the fulfilment of a sales order for material from stock. In this scenario, sales order data reflecting a customer's purchase order is handled and transmitted to process components responsible for managing invoicing, accounting, and the supply chain [13]. The consistency of data across the components involved in these processes is of considerable importance. For example, the quantity of a product in the sales order data should be the same in the invoice and the dispatch note. The exact notion of consistency depends on the application, so we refer to this as *semantic consistency*.

Although semantic consistency is important in business information systems, inconsistency is also a fact of life. Consider the *Sell from Stock* example. First, orders or parts of orders can often be changed or cancelled, even though the subsequent process steps have already reached an advanced state. This is problematic if, for example, the delivery of cancelled orders has already started. Second, a sub-process may require a manual approval or other manual processing like moving or packaging goods. Finally, messages which are intended to update other process components may be blocked in transmission through lower layers, causing errors to be propagated to the higher layers. For these reasons, processes are often in temporarily inconsistent states. After a certain delay caused by late changes, manual steps, or recovery from errors, they must however assume a consistent state.

Modelling consistency and error recovery may help the developers to integrate late change or (partial) cancellation of business objects into business logics, and to mechanically analyse earlier in the development cycle the ways the business scenarios are constructed to recover from such errors and to adapt them if necessary before implementing them. Since semantic consistency involves the consideration of delays, handling consistency in models entails understanding and modelling time.

Our objective is to support Event-B modelling and analysis of business processes that can accommodate time-bounded semantic inconsistency. Experience suggests [5] that the choices of abstractions and modelling/refinement patterns have a significant bearing on the level of automation achievable in analysis. We therefore focus first on abstractions and patterns that can form the basis of reasoning about bounded inconsistency. The contribution of this paper is to identify Event-B patterns that can be used to model and analyse time-bounded semantic inconsistency in distributed business information systems. At this stage, we do not deal with the automation of the analysis process based on definitions of workflows in, for example, BPMN-like notations.

We first briefly describe salient aspects of Event-B and patterns (Section 2.1), giving an abstract machine that forms the basis of the patterns presented. We consider the Time Constraint Pattern (TCP) in Section 3.1 and describe an adaptation that separates time and consistency in Section 3.2. We give a simple pattern for modelling error detection and recovery at the level of abstraction of the business information systems models that we deal with (Section 4). This is integrated with the adapted TCP in Section 5. Related work is discussed in Section 6. Conclusions and further work are described in Section 7.

## 2 Background

As indicted in Section 1, inconsistency can arise in normal operation or because of faults in lower layers. In normal operation, each process attempts to maintain semantic consistency by informing other processes of changes that need to be made. The parts of the system related to this change are necessarily inconsistent while these messages are created, transmitted, received and processed. This normal operational inconsistency is time-bounded. Further, the system may include a recovery mechanism to allow it to recover from the occurrence of faults at lower-levels in the system. This will involve the distribution of error recovery messages. In both normal operation and error recovery message transmission, latency will be a significant factor in bounding the resulting inconsistency. We therefore begin to model time-bounded inconsistency by considering message transfer.

## 2.1 Event-B and Event-B Patterns

We use the Event-B modelling formalism [2] and the Rodin tools platform [1], because they have several features that make them appropriate to our application area. The modelling language allows description of both structured data and functionality, and the available abstractions form a promising basis for describing business information systems applications [5]. The tools are open-source and based on the Eclipse framework, allowing the integration of specialised provers, model checkers, editors, pretty printers and other new features. The method and tools have a significant and growing community of practice.

The basic modelling unit in Event-B is the *machine*. Each machine may contain *variables* modelling persistent state data, *invariants* that restrict the valid content of variables, and guarded *events* that describe functionality in terms of actions defined over the state variables. Definitions of the carrier sets and constants may be defined in units called contexts that are visible to machines. A system model typically consists of a chain of Event-B machines, each of which (apart from the first) is linked to its predecessor by a refinement relation expressed in terms of a linking invariant. Machines and refinement steps give rise to proof obligations that ensure internal consistency of machines and behaviour preservation across refinement steps. A typical Event-B model has an extremely simple initial machine, with detail added in a controlled way through refinement steps.

Patterns in Event-B [4, 9, 12, 6] are a means of expressing reusable modelling structures and managing effort by promoting proof re-use. Several forms of Event-B pattern have been proposed, differing in their levels of generality. Iliasov [12] treats a pattern as a general model transformation method. Fürst [9], by contrast, treats a pattern as a fragment of Event-B which may have a number of refinement steps designed to be directly substituted into a development. The first and last machine of the

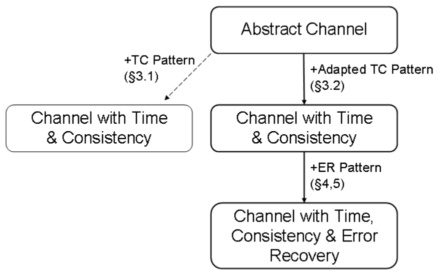

Abstract Channel
+TC Pattern (§3.1)
+Adapted TC Pattern (§3.2)
Channel with Time & Consistency
Channel with Time & Consistency
+ER Pattern (§4,5)
Channel with Time, Consistency & Error Recovery

**Fig. 1.** Structure of the developed models.

pattern are referred to as the abstract and concrete pattern respectively. Cansell et al. [6] regard a pattern as being less general than Iliasov, but still requiring some specialisation before it is applied to a development.

In Fürst's approach [9] the application of a pattern to an Event-B development requires that each variable and event in the abstract pattern must be *matched* with a variable or event in the current machine in the development. Variables match if a subset of the variables in the development have the same type as variables in the abstract pattern. Events match if a subset of the events in the development have the same behaviour as the events in the abstract pattern. The development can then be extended by replacing matched events and variables (and relevant invariants) with the events, variables and invariants of the concrete pattern. The correspondence required to apply the pattern from [6] is less strict, allowing for both data refinement (of variables) and event refinement.

Beginning with the Time Constraint Pattern [6], we develop a patten to add timing information to a fragment of a business information system. We then develop a pattern for adding error recovery behaviour, and then combine the two patterns developed. The patterns that we develop are in the style of Fürst's approach, since our goal is to automatically apply these patterns.

## 2.2 The Abstract Channel

We have identified the duration of message transmission to be a major source of time-bounded inconsistency. We therefore need to be able to describe real-time requirements, and in particular to articulate and prove properties relating to time-bounded consistency, as well as the means of recovery from inconsistency. In this section we develop an abstract machine of a small fragment of a business information system which contains no representation of time or error recovery. We then investigate patterns for refining this machine by adding time and error recovery. In Section 3.1 we apply the TCP to this abstract channel. In Section 3.2 we apply a modification of the pattern to our example. In Section 4 we propose a pattern to describe error recovery, and in Section 5 we instantiate the modified example with the error recovery pattern. An outline of these developments is given in Fig. 1.

We identify two processes, a sender and a receiver, and consider a channel carrying messages between them. We consider only messages that identify inconsistencies to be resolved. At this level of abstraction, receipt of a message models correction of the inconsistency. We identify a set *cons* of messages. A message is in this set if the inconsistency to which it refers has been resolved. We refer to these messages as consistent.

We model the transmission and reception of messages over the channel (events *snd* and *rcv* in Fig. 2.) The variables *sent*, *chan* and *cons* are all subsets of *MSG* (a carrier set representing the set of all messages identifying inconsistencies). Variables *sent* and *chan* represent messages that have been sent and the contents of the channel respectively. In this abstract machine, and in subsequent developments, we will state *consistency invariants*. These are normal Event-B invariants, and their purpose is to capture what we can prove about consistency in the different machines. The (untimed) consistency invariant is given in Fig. 2. Consistent messages are those that have been removed from the channel.

```
snd
  any  $m$  where
    $m \in MSG \wedge m \notin sent$
  then
    $sent := sent \cup \{m\}$
    $chan := chan \cup \{m\}$
  end
```

```
rcv
  any  $m$  where
    $m \in chan$
  then
    $cons := cons \cup \{m\}$
    $chan := chan \setminus \{m\}$
  end
```

```
invariants
  $cons = sent \setminus chan$
```

**Fig. 2.** The abstract channel.

## 3 An Adaptation of the Time Constraint Pattern

Time is not an explicit part of the Event-B language, so in order to describe time-bounded consistency, we begin by examining an important pattern for representing time.

In this section, we describe the timed behaviour of the channel. We will go on to model recovery from time-bounded inconsistency. At this stage we do not wish to set a hard upper limit on the period on inconsistency, so we do not set an upper bound on the duration of message transmission. Instead, we associate a real-time period $limit$ with each message in the channel. This limit is the time limit of "acceptable" inconsistency. If a message arrives after $limit$ then recovery will be necessary.

### 3.1 The Time Constraint Pattern

Several extensions have been proposed for modelling time in B and Event-B (e.g. [16], [18]). However, we prefer to work entirely within the Event-B language in order to take advantage of the tools provided. Cansell et al. propose a promising Event-B pattern (called the Time Constraint Pattern, TCP) for introducing time constraints into a development [6]. An elaboration is given by Rehm [19].

In the TCP, time progression is measured by the increase of a dedicated variable $time$ of type $NAT$. The passage of time is modelled by a separate event *tick_tock* in [6], see Fig 3. A collection of "active times" is maintained to represent the times in the future when certain events must be performed. (Rehm [19] associates events with these times, whereas the earlier paper of Cansell et al. [6] just records the set of active times in the variable *at*.) The value *tm* is the new value for *time* after the event *tick_tock*. Time must not progress beyond the point at which the next event is scheduled to be performed, as guaranteed by the third guard in *tick_tock*. Time constraints are introduced to and removed from the set of active times by the events *post_time* and *process_time* respectively (Fig. 3).

The TCP is not designed to be applied directly, but must be adapted to a match a specific development. Here, we wish to use the pattern to allow us to record constraints that messages are handled within deadlines: each message should be received within $limit$ time units. The constant *limit* (defined as a natural number in the associated context) is the upper limit on the delivery of messages before recovery is necessary.

```
post_time
  any  $tm$  where
    $tm \in NAT$
    $tm > time$
  then
    $at := at \cup \{tm\}$
  end
```

```
process_time
  when
    $time \in at$
  then
    $at := at \setminus \{time\}$
  end
```

```
tick_tock
  any  $tm$  where
    $tm \in NAT$
    $tm > time$
    $(at \neq \varnothing \Rightarrow tm < \min(at))$
  then
    $time := tm$
  end
```

**Fig. 3.** The TCP events.

In Fig. 4 we show the machine *TCP-channel* resulting from an application of the TCP to our abstract channel of Fig. 2. The variable *now* (representing elapsed time) matches the TCP variable *time*. The correspondence between variable *deadline* and TCP variable *at* is less immediate. We use *deadline* to record the time at which messages must leave the channel. The variable *at* therefore matches $rng(deadline)$. The map structure associating messages and times is similar to the presentation given in [19].

The variable *timesent* is a total function from sent messages to time, and records the time at which messages are placed on the channel. The variable *timercvd* records the time at which messages are received from the channel. All messages in *chan* have an associated deadline.

We refine the *snd* event to record the time at which a message is sent, as suggested by the event *post_time* from the TCP. Event *snd* also adds the message to the channel, and adds timing information in the form of the maplet $\{m \mapsto now + limit\}$ to the variable *deadline*. The time posted in *deadline* is $now + limit$, giving a real-time deadline for receipt of the message.

The concrete *rcv* event corresponds to the *process_time* event. We generalise the timed guard from *process_time* to $now \leq deadline(m)$, to allow messages to be received at any point before their deadline is reached. Event *rcv* removes a message from the channel and the associated maplet from *deadline*, adds the message to *cons*, and records the time at which the message was received. The *tick* event is a refinement of the *tick_tock* event of TCP. When *tick* fires, time progresses by one unit.

```
invariants
  $now \in \mathbb{N}$
  $sent = dom(timesent)$
  $timesent \in dom(timesent) \rightarrow \mathbb{N}$
  $deadline \in dom(timesent) \nrightarrow \mathbb{N}$
  $timercvd \in dom(timesent) \nrightarrow \mathbb{N}$
  $dom(deadline) = chan$
```

```
snd
  any  $m$  where
    $m \in MSG$
    $m \notin dom(timesent)$
  then
    $timesent := timesent \cup \{m \mapsto now\}$
    $chan := chan \cup \{m\}$
    $deadline := deadline \cup \{m \mapsto now + limit\}$
  end
```

```
tick
  when
    $dom(deadline) \neq \varnothing \Rightarrow$
      $now < min(ran(deadline))$
  then
    $now := now + 1$
  end
```

```
rcv
  any  $m$  where
    $deadline \neq \varnothing$
    $m \in chan$
    $now \leq deadline(m)$
  then
    $deadline := deadline \setminus \{m \mapsto deadline(m)\}$
    $chan := chan \setminus \{m\}$
    $cons := cons \cup \{m\}$
    $timercvd := timercvd \cup \{m \mapsto now\}$
  end
```

**Fig. 4.** The invariants and events of the TCP-channel.

For this machine timely receipt of a message represents time-bounded recovery from inconsistency. A timed consistency invariant is given below.

$$\forall m \cdot m \in dom(timesent) \Rightarrow (timesent(m) + limit < now \Rightarrow m \in cons)$$

Any sent messages for which the deadline has passed ($timesent(m) + limit < now$) must be in the consistent set *cons*.

This direct application of TCP produces a machine that excludes the possibility of messages being delayed, since $deadline(m)$ gives an upper bound on the duration of transmission of message $m$. We now adapt the TCP, to model message delay.

## 3.2 Adapting the Time Constraint Pattern for Modelling Bounded Inconsistency

We adapt the TCP by removing the guard on *tick* and the third guard on *rcv*, giving events *ungrd_tick* and *untmd_rcv* (Fig. 5). By allowing time

```
untmd_rcv
  any  $m$  where
    $deadline \neq \varnothing$
    $m \in chan$
  then
    $deadline := deadline \setminus \{m \mapsto deadline(m)\}$
    $chan := chan \setminus \{m\}$
    $cons := cons \cup \{m\}$
    $timercvd := timercvd \cup \{m \mapsto now\}$
  end
```

```
ungrd_tick
  begin
    $now := now + 1$
  end
```

**Fig. 5.** Events *untmd_rcv* and *ungrd_tick*.

to progress at any stage in the execution of a machine, we allow any message to be delayed beyond $deadline(m)$. In the machine that results, with concrete events *snd* (unchanged from *TCP-channel*, Fig. 4), *untmd_rcv* (Fig. 5) and *ungrd_tick* (Fig. 5), all received messages are consistent. A possible consistency invariant is:

$$dom(timercvd) = cons$$

We now need to separate normal and recovery behaviour, so we distinguish the receipt of message *m* before and after $deadline(m)$. We propose an error recovery pattern to deal separately with this error recovery behaviour.

## 4 An Error Recovery Pattern

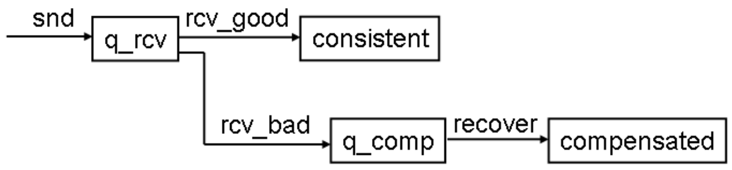


**Fig. 6.** Structure of Concrete Error Recovery Pattern.

Effective modelling and analysis of fault tolerance relies on a clear separation between normal and abnormal states and behaviours. This

```
rcv_good
  refines  $rcv$
  any  $m$  where
    $m \in q\_rcv$
  then
    $consistent := consistent \cup \{m\}$
    $q\_rcv := q\_rcv \setminus \{m\}$
  end
```

```
rcv_bad
  any  $m$  where
    $m \in q\_rcv$
  then
    $q\_comp := q\_comp \cup \{m\}$
    $q\_rcv := q\_rcv \setminus \{m\}$
  end
```

**invariants**
$cons = consistent \cup compensated$
$chan = q\_comp \cup q\_rcv$

```
recover
  refines  $rcv$
  any  $m$  where
    $m \in q\_comp$
  then
    $compensated := compensated \cup \{m\}$
    $q\_comp := q\_comp \setminus \{m\}$
  end
```

**Fig. 7.** Concrete Error Recovery Pattern.

structuring reduces the complexity of system design. It allows developers to reason explicitly about erroneous states and to associate appropriate recovery actions with them. Moreover, it provides a consistent means of expressing the switchover between normal and abnormal modes of execution, the initiation of recovery actions and the management of success and failure of this recovery.

In this section we present a pattern modelling error recovery. We will refer to this as the Error Recovery Pattern (ERP). The ERP has one abstract and one concrete machine. The abstract machine is original description of the channel given in Fig. 2. To model error recovery, we begin by distinguishing the possibilities that the message transmission is either correct or faulty. If the transmission is faulty (the message is corrupted, late, etc.), then some recovery action is required. The concrete part of the pattern is pictured in Fig. 6. Event *rcv* from Fig. 2 is now refined into three events (*rcv_good*, *rcv_bad* and *recover*, Fig.7), and the channel of the abstract machine is split into two queues. *q_rcv* is the queue of messages waiting to be received, and *q_comp* is the queue of messages which have been received and for which compensation is required. The variable *consistent* represents those messages that are immediately consistent on arrival, and *compensated* represents those messages for which the system had to perform recovery.

In the concrete machine of the ERP Fig.7 the event *rcv_good* models the receipt of a good message from *q_rcv*. No further action is required and the part of the system to which it refers is immediately consistent. Event *rcv_bad* models the receipt of a faulty message and its placement in the queue of messages which need to be compensated *q_comp*. Event *recover* performs the recovery action, removing the faulty message from *q_comp* and placing it in the set of compensated messages *compensated*. The event *snd* (not given) is almost unchanged, except that sent messages are placed in the queue for the receiving component *q_rcv* instead of *chan*.

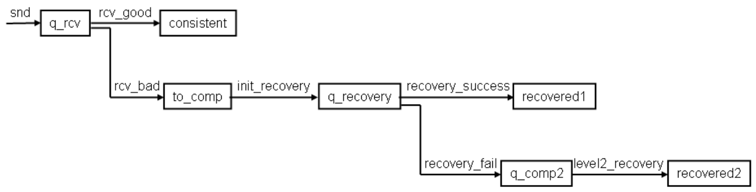


**Fig. 8.** Hierarchical Error Recovery.

As well as the general role that careful structuring plays in the provision of effective fault tolerance, recursive structuring supports the description of the propagation of responsibility for error handling through system levels [17]. An example of this is hierarchical error recovery is Fault Detection, Isolation and Recovery (FDIR) [21]. If, rather than just consider the case where compensation is successful, we consider as well the case where it fails, we can reapply the ERP to itself to model hierarchical error recovery. A single application of ERP to itself results in the machine summarised in Fig. 8, and presented in full in Fig. 9.

## 5 An Instantiation of Adapted TCP and ERP

We apply the ERP to the adapted TCP to generate the concrete part of the time-bounded inconsistency pattern. We distinguish the timely and late arrival of messages. Event *rcv_good* becomes *rcv_ontime* (Fig. 10.) The addition of the timing guard $deadline(m) \geq now$ ensures that this event applies only to messages whose deadline has not yet passed. Event

```
snd
  refines  $snd$
  any  $m$  where
    $m \in MSG$
    $m \notin sent$
  then
    $sent := sent \cup \{m\}$
    $q\_rcv := q\_rcv \cup \{m\}$
  end
```

```
rcv_good
  refines  $rcv\_good$
  any  $m$  where
    $m \in q\_rcv$
  then
    $consistent := consistent \cup \{m\}$
    $q\_rcv := q\_rcv \setminus \{m\}$
  end
```

```
rcv_bad
  refines  $rcv\_bad$
  any  $m$  where
    $m \in q\_rcv$
  then
    $to\_comp := to\_comp \cup \{m\}$
    $q\_rcv := q\_rcv \setminus \{m\}$
  end
```

```
init_recovery
  refines  $init\_recovery$
  any  $m$  where
    $m \in to\_comp$
  then
    $to\_comp := to\_comp \setminus \{m\}$
    $q\_recovery := q\_recovery \cup \{m\}$
  end
```

```
recovery_success
  refines  $term\_recovery$
  any  $m$  where
    $m \in q\_recovery$
  then
    $q\_recovery := q\_recovery \setminus \{m\}$
    $recovered1 := recovered1 \cup \{m\}$
  end
```

```
recovery_fail
  any  $m$  where
    $m \in q\_recovery$
  then
    $q\_comp2 := q\_comp2 \cup \{m\}$
    $q\_recovery := q\_recovery \setminus \{m\}$
  end
```

```
level2_recovery
  any  $m$  where
    $m \in q\_comp2$
  then
    $recovered2 := recovered2 \cup \{m\}$
  end
```

**Fig. 9.** Hierarchical Error Recovery events.

*rcv_late* is derived from the *rcv_bad* event of the ERP, with the addition of the guard $deadline(m) < now$. Event *compensate* deals with a late message which requires compensation. It places the message in the set of messages which have been compensated, and removes the deadline information from *deadline.*

```
tick
  begin
    $now := now + 1$
  end
```

```
rcv_ontime
  refines  $rcv$
  any  $m$  where
    $m \in q\_rcv$
    $deadline(m) \geq now$
  then
    $deadline :=$
        $deadline \setminus \{m \mapsto deadline(m)\}$
    $consistent := consistent \cup \{m\}$
    $q\_rcv := q\_rcv \setminus \{m\}$
  end
```

```
rcv_late
  any  $m$  where
    $m \in q\_rcv$
    $deadline(m) < now$
  then
    $q\_rcv := q\_rcv \setminus \{m\}$
    $q\_comp := q\_comp \cup \{m\}$
  end
```

```
compensate
  refines  $rcv$
  any  $m$  where
    $m \in q\_comp$
  then
    $deadline :=$
        $deadline \setminus \{m \mapsto deadline(m)\}$
    $compensated := compensated \cup \{m\}$
    $q\_comp := q\_comp \setminus \{m\}$
  end
```

```
snd
  refines  $snd$
  any  $m$  where
    $m \in MSG$
    $m \notin dom(timesent)$
  then
    $timesent := timesent \cup$
        $\{m \mapsto now\}$
    $deadline := deadline \cup$
        $\{m \mapsto now + limit\}$
    $q\_rcv := q\_rcv \cup \{m\}$
  end
```

**Fig. 10.** The events of the concrete part of time-bounded inconsistency pattern.

We also record the time at which a message is received (*timercvd.*) With these, we can then prove the bounded time consistency invariants below.

$$\forall m \cdot (m \in dom(timercvd) \wedge timercvd(m) - timesent(m) \leq limit) \Leftrightarrow$$
$$m \in consistent$$
$$\wedge\ \forall m \cdot (m \in dom(timercvd) \wedge timercvd(m) - timesent(m) > limit) \Leftrightarrow$$
$$m \in q\_comp \cup compensated$$

The pattern has been proved using the Rodin provers, largely (but not completely) automatically.

## 6 Related Work

Approaches to the formal modelling and analysis of workflows over service-oriented architectures, particularly for BPEL, use a variety of notations and tools. Many base formal models on transition systems and Petri Nets, allowing the analysis of path and termination properties [10, 23]. Finite automata encoded in Promela have also been used for deadlock detection [15]. Process Calculi such as FSP [8], and LOTOS [20] and Abstract State Machines [7] have also been used to model workflow supporting a range of properties including aspects of fault handling and compensation.

Our approach, based on Event-B, naturally focuses on the use of proof to give semantic analysis of functional and timing aspects of workflows over distributed business systems. The closest work to our own is perhaps that of Aït-Sadoune and Aït-Ameur [3] in which an Event-B model of BPEL workflows, including fault handling, is developed, allowing proof-based validation of properties such as deadlock-freeness. The approach does not yet make use of refinement in the target Event-B models. Ball and Butler [4] identify design patterns relating fault tolerance behaviour in response to problems which agents can encounter when exchanging requests to carry out actions. These are inherent for the family of agent interaction protocols, dealing with agent contracts. The patterns are presented as blueprints, capturing typical elements of the agent models. Our work differs from the approaches above by addressing application-level semantic consistency, focusing on patterns that can be included in the Event-B model derived from workflow and having an explicit model of real time.

We have already discussed the Time Constraint Pattern and related work. Besides time and consistency, our approach makes use of a simple error recovery pattern. There are several approaches to patterns for fault tolerance in B/Event-B. Laibinis and Troubitsyna [14] propose a formal specification pattern (in B) that can be recursively applied to specify exception raising and handling at various architectural layers of component-based systems. The approach is developed for the systems in which components interact by issuing synchronous calls and which mainly face hardware failures and human errors. Iliasov [11, 12] offers a general approach to defining fault tolerance refinement patterns assisting system developers in disciplined application of software fault tolerance mechanisms during rigorous system design. Our work additionally requires modelling of temporal constraints. These patterns are formally defined as model transformations producing correct model refinements. The approach is backed by a tool and a theory of pattern composition.

## 7 Conclusions and Further Work

We have presented a pattern that may be used to describe time-bounded semantic consistency properties for distributed business information systems. Our approach adapts the Timed Consistency pattern of Cansell et al., adding (optionally hierarchical) error recovery.

Although our approach was inspired by business information systems applications, we conjecture that the same approach could be used for a wider class of distributed applications. Our previous work [5] modelled the effect of lower-level faults lower-level faults in communications middleware on the completion of communications protocols that underpin applications. Our work here is at a higher level and deals with the semantics of the application. An important direction for future work is to link the two in order to provide more complete coverage of error propagation and recovery.

We intend to apply the patterns for modelling temporary semantic inconsistency to realistic business processes typically implemented in SAP software and to custom business processes modelled with the help of BPMN. To this end we are investigating ways of representing these processes in a formal language like Event-B, similar to the general approach of Aït-Sadoune and Aït-Ameur [3]. Automating and providing tool support for the translation into a formal language and for the application of the error recovery pattern may give business process designers the most convenient support for their work. It is in our plans for the future work to evaluate the applicability of general patterns approaches such as those of Iliasov [12] to the development of the time constraint and error recovery patterns proposed in this paper. The use of good labour-saving patterns is likely to play a significant part in the achievement of that goal.

*Acknowledgements:* This work has been supported by the EC FP7 Integrated Project *Deploy*. We are grateful to our colleagues in the project for their continuing fruitful collaboration.

## References

1. J.-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An open extensible tool environment for event-b. In Z. Liu and J. He, editors, *Formal Methods and Software Engineering: 8th International Conference on Formal Engineering Methods, ICFEM 2006*, volume 4260 of *Lecture Notes in Computer Science*, pages 588–605. Springer, November 2006.
2. Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2009. To appear. See also http://www.event-b.org.
3. I Aït-Sadoune and Y Aït-Ameur. A proof based approach for modelling and verifying web services compositions. In *Proc. 14th IEEE Intl. Conf. on Engineering of Complex Computer Systems*, June 2009.
4. Elisabeth Ball and Michael Butler. Event-B Patterns for Specifying Fault-Tolerance in Multi-agent Interaction. In Michael Butler, Cliff B. Jones, Alexander Romanovsky, and Elena Troubitsyna, editors, *Methods, Models and Tools for Fault Tolerance*, volume 5454 of *Lecture Notes in Computer Science*, pages 104–129. 2009.
5. J. Bryans, J. Fitzgerald, A. Romanovsky, and A. Roth. Formal Modelling and Analysis of Business Information Applications with Fault Tolerant Middleware. In *Proc. 14th IEEE Intl. Conf. Conference on Engineering of Complex Computer Systems*, pages 68–77, June 2009.
6. Dominique Cansell, Dominique Méry, and Joris Rehm. Time Constraint Patterns for Event B Development. In Jacques Julliand and Olga Kouchnarenko, editors, *B 2007: Formal Specification and Development in B, 7th Intl. Conf. of B Users, Besançon, France, January 17-19, 2007*, volume 4355 of *Lecture Notes in Computer Science*, pages 140–154. Springer, 2007.
7. Roozbeh Farahbod, Uwe Glässer, and Mona Vajihollahi. Specification and validation of the business process execution language for web services. In Wolf Zimmermann and Bernhard Thalheim, editors, *Abstract State Machines 2004. Advances in Theory and Practice, 11th International Workshop, ASM 2004, Lutherstadt Wittenberg, Germany, May 24-28, 2004. Proceedings*, volume 3052 of *Lecture Notes in Computer Science*, pages 78–94. Springer-Verlag, 2004.
8. Howard Foster, Wolfgang Emmerich, Jeff Kramer, Jeff Magee, David S. Rosenblum, and Sebastián Uchitel. Model checking service compositions under resource constraints. In Ivica Crnkovic and Antonia Bertolino, editors, *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pages 225–234. ACM, 2007.
9. Andreas Fürst. Design patterns in event-B and their tool support. Masters thesis, ETH, Eidgenössische Technische Hochschule Zürich, Department of Computer Science, Chair of Information Security, Information Security Group, 2009.
10. Sebastian Hinz, Karsten Schmidt, and Christian Stahl. Transforming BPEL to Petri Nets. In Wil M. P. van der Aalst, Boualem Benatallah, Fabio Casati, and Francisco Curbera, editors, *Business Process Management, 3rd International Conference, BPM 2005, Nancy, France,*

*September 5-8, 2005, Proceedings*, volume 3649 of *Lecture Notes in Computer Science*, pages 220–235. Springer-Verlag, 2005.

11. Alexei Iliasov. Refinement patterns for rapid development of dependable systems. In Nicolas Guelfi, Henry Muccini, Patrizio Pelliccione, and Alexander Romanovsky, editors, *Proc. 2007 Workshop on Engineering Fault Tolerant Systems*. ACM, 2007.
12. Alexei Iliasov. *Design Components*. PhD thesis, School of Computing Science, Newcastle University, Newcastle upon Tyne NE1 7RU, United Kingdom, 2008.
13. Stefan Kätker and Susanne Patig. Model-driven development of serviceoriented business application systems. In Hans Robert Hansen, Dimitris Karagiannis, and Hans-Georg Fill, editors, *Wirtschaftsinformatik (1)*, volume 246 of *books@ocg.at*, pages 171–180. Österreichische Computer Gesellschaft, 2009.
14. Linas Laibinis and Elena Troubitsyna. Fault Tolerance in a Layered Architecture: A General Specification Pattern in B. In *2nd International Conference on Software Engineering and Formal Methods (SEFM 2004), 28-30 September 2004, Beijing, China*, pages 346–355. IEEE Computer Society, 2004.
15. Shin Nakajima. Model-checking behavioral specification of bpel applications. *Electronic Notes in Theoretical Computer Science*, 151(2):89–105, 2006. Proceedings of the International Workshop on Web Languages and Formal Methods (WLFM 2005).
16. Miloud Rached, Jean-Paul Bodeveix, Mamoun Filili, and Odile Nasr. A Timed B method for modelling Real time Reactive Systems. In *Proc. 2nd South-East European Workshop on Formal Methods*, pages 181–195, Nov 2005.
17. Brian Randell. Recursively Structured Distributed Computing Systems. In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–11, 1983.
18. Joris Rehm. A Duration Pattern for Event-B Method. In *Proc. 2nd Junior Workshop on Real-Time computing*, 2008.
19. Joris Rehm. From absolute-time to relative-countdown: Patterns for model-checking. Hyperarticles en Ligne, Article hal-00319104 at `hal.archives-ouvertes.fr`, May 2008.
20. Gwen Salaün, Lucas Bordeaux, and Marco Schaerf. Describing and Reasoning on Web Services using Process Algebra. In *Proceedings of the IEEE International Conference on Web Services (ICWS'04), June 6-9, 2004, San Diego, California, USA*, pages 43–50. IEEE Computer Society, 2004.
21. As'ad Michael Salkham. Fault Detection, Isolation and Recovery (FDIR) in On-Board Software. Master's thesis, Chalmers University of Technology, 2005.
22. Jim Hagemann Snabe, Ann Rosenberg, Charles Moller, and Mark Scavillo. *Business Process Management: The SAP Roadmap*. SAP PRESS, 2008.
23. H.M.W. Verbeek and W.M.P. van der Aalst. Analyzing BPEL Processes using Petri Nets. In D. Marinescu, editor, *Proceedings of the Second International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management, Florida International University, Miami, Florida, USA, 2005*, pages 59–78, 2005.

# Part VI

# Short Papers

# First Models of a Safe System

Colin Snook

Southampton university, UK
cfs@ecs.soton.ac.uk

## 1 Extended Abstract

The most important requirements of a safety-critical system are the safety ones. It is crucial that these requirements are complete and correct and that the system is developed to satisfy them. Hence it makes sense to start with the safety requirements before adding more detailed functionality. Abstraction allows us to specify these requirements in the clearest possible way so that we are less likely to make mistakes. Refinement can then be used to ensure the system meets them. Starting from a list of identified hazards, we show by example how to develop safety requirements in the formal notation UML-B. We start by modelling an unsafe system, then identify hazards that are outside the scope of the proposed system before finally specifying those safety properties that are required to be implemented. 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. 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 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 notation which is a graphical front-end for the Event-B language and tools. The Rodin tool set 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 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. 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 invariants 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 [1].

[1] Due to the stringent approval process of the INESS project we are only allowed to report here an extended abstract of this contribution

# Event-B Model Decomposition

Carine Pascal[1] and Renato Silva[2]

[1] Systerel, France
carine.pascal@systerel.fr
[2] University of Southampton,UK
ras07r@ecs.soton.ac.uk

**Abstract.** Two methods have been identified in the DEPLOY[3] project for Event-B model decomposition: the shared variable decomposition (called A-style decomposition), and the shared event decomposition (or B-style decomposition). Both allow the decomposition of a (concrete) model into several independent sub-models which may then be refined separately. The purpose of this paper is to introduce the Event-B model decomposition, from theory (A-style vs. B-style, differences and similarities) to practice (decomposition plug-in of the Rodin [10] platform).

## 1 Introduction

The "top-down" style of development used in Event-B[2] allows the introduction of new events and data-refinement of variables during refinement steps. A consequence of this development style is an increasing complexity of the refinement process when having to deal with many events and many state variables. The main purpose of the model decomposition is precisely to address such difficulty by cutting a large model into smaller components. Decomposition cuts a model $MC$ into sub-models $MC_1, ..., MC_n$, which can be refined independently and more comfortably than the whole. This solution gives an interesting perspective for the team development of a model: the possibility for several developers to share parts of a model and to work independently and possibly in parallel, on them.

[3] DEPLOY - Industrial deployment of system engineering methods providing high dependability and productivity - FP7 Integrated Project supported by the European Commission (Grant 214158), under Strategic Objective IST-2007.1.2.
See `http://www.deploy-project.eu`.

## 1.1 Definition and constraints

A model contains contexts, machines or both. The notion of model decomposition covers machine and context decomposition.

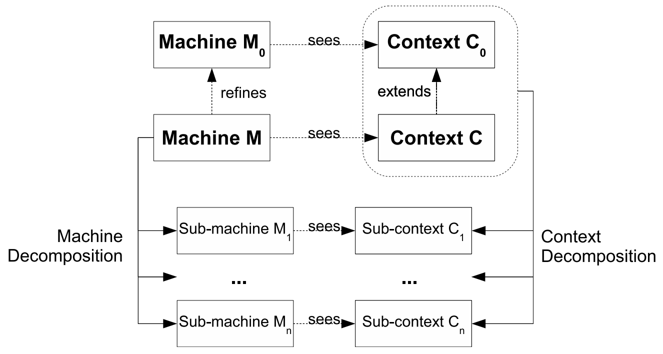


**Fig. 1.** Event model decomposition

The entry point for the decomposition of a model is a machine *M* and its whole hierarchy of seen contexts as illustrated in Fig. 1. The resulting sub-models are independent of the non-decomposed model and the partition of the models includes the splitting of variables, invariants, events and even context elements like sets, constants and axioms. The main constraint is that if these refined models are recomposed into a model *MR*, it is guaranteed that *MR* refines *MC* as depicted in Fig. 2 (monotonicity). Decomposition does not generate new proof obligations (POs).

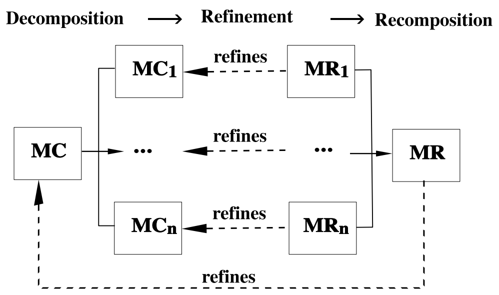


**Fig. 2.** Model decomposition, refinement, and re-composition

## 2 Decomposition Styles

This section concretely explains how to decompose a given machine $M$ in sub-machines $M_1, ..., M_n$, depending on the chosen decomposition style: *shared variable* or *shared event*. The shared event approach seems particularly suitable for message-passing distributed programs, while the shared variable approach seems more suitable when designing concurrent programs [4].

### 2.1 Shared Variable (A-style) Decomposition

Figure 3 illustrates the shared variable decomposition proposed by Abrial [1], Metayer et. al [8] and Abrial and Hallerstede [3]. Machine $M$ has events *e1* to *e4* and variables *v1* to *v3*. The solid lines connect variables used by the events. Events of the non-decomposed component (machine $M$) are partition among the sub-components (Machine *M1*:*e1* and *e2*; Machine *M2*:*e3* and *e4*). After the event partition it is necessary to split the variables. The sub-components can be refined independently respecting some restrictions and the re-composition of the sub-components should always be a refinement of the non-decomposed component.

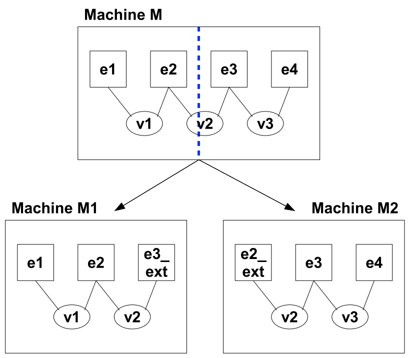


**Fig. 3.** Shared Variable Decomposition of machine *M* into machines *M1* and *M2* with shared variable *v2*

The following sequence must be observed to decompose $M$ [9]:

1. The events of $M$ are first partitioned over $M_1, ..., M_n$.

2. The variables of $M$ are distributed according to the event partition. Variables accessed (i.e. read or written) by events of distinct sub-machines, are called *shared variables* like *v2* (in opposition to *private variables* like *v1* and *v3*).
3. The invariants of $M$ are distributed according to the variable distribution. An invariant based on a predicate $P(v_1,...,v_m)$ is copied to $M_i$ if and only if $M_i$ contains all the $v_1,...,v_m$ variables.
4. *External* events are built in $M_1,...,M_n$. If $e2$ is an event of $M_i$ that modifies a shared variable $v2$, then an external event is built from $e2$ in each sub-machine $M_j$ where $v2$ is accessed. An external event of $M_i$ must always be present and be strictly identical in any refinement of $M_i$.

## 2.2 Shared Event (B-style) Decomposition

The B-style decomposition is achieved with the partition of synchronized events among the sub-models. Figure 4 depicts the shared event decomposition proposed by Butler [5]. The state variables are refined so they may be partitioned amongst the subsystems, introducing internal events representing interaction between subsystems. More details about this approach can be found in [6, 7].

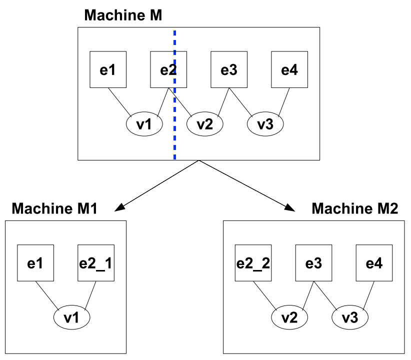


**Fig. 4.** Shared Event Decomposition of Machine *M* in Machines *M1* and *M2* with shared event *e2*

The several steps for the shared event decomposition are given below:

1. The variables of $M$ are first partitioned among $M_1,...,M_n$.

2. The events of $M$ are distributed among $M_1, ..., M_n$, according to the variable partition. There is no notion of shared variables. Events using variables allocated to different sub-components (*e2* shares *v1* and *v2*) must be split. The part corresponding to each variable (*e2_1* and *e2_2*) is used to create partial versions of the non-decomposed event. The sub-components can be refined independently without constraints
3. The invariants of $M$ are distributed similarly to A-style.

## 3 Tool Specification

Both decomposition styles are implemented in the same Rodin platform plug-in. The decomposition input is a machine of a given Rodin project selected by the end-user. The next step of the decomposition (i.e. the event partition for A-style or the variable partition for B-style) requires the intervention of the end-user. Afterwards the sub-components resulting from the decomposition are defined. All other steps are automatically performed by the plug-in. Summarising these are the steps to be followed in order to decompose (we decompose $M$ in Fig. 5(a)):

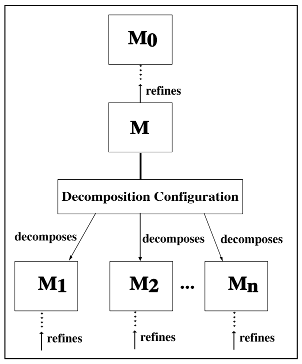


**Fig. 5.** Decomposition tool diagram for machine *M*

1. End-user selects a machine $M$ to decompose.
2. End-user defines sub-components to be generated: $M_1, M_2, \ldots M_n$.
3. End-user selects the decomposition style to use:

**Shared Variable:** end-user selects the events to be allocated to sub-components. The tool automatically decomposes the rest of the model according to the event partition (shared/private variables, external events).

**Shared Event:** the end-user selects the variables to be allocated for each sub-component. The rest is done automatically.

4. The end-user can opt to decompose the seen contexts. The contexts seen by $M_1, ..., M_n$, are built from the hierarchy of contexts associated to $M$ and are built following the sequence:
   (a) A constant in this hierarchy is copied to a sub-context $C_i$ associated to $M_i$ if and only if it appears in a predicate (invariant or guard) or in an assignment (action) of $M_i$.
   (b) A carrier set in this hierarchy is copied to $C_i$ if and only if it appears in a predicate or in an assignment of $M_i$, or if it types a constant previously copied to $C_i$.
   (c) An axiom is copied to $M_i$ if and only if $M_i$ contains the referenced constants and sets.
5. Sub-components are fulfilled according to the decomposition configuration.
6. The decomposition configuration is stored.
7. Sub-components $M_1, M_2, \ldots M_n$ can be further refined.

The generated sub-components can be created in the same project as the non-decomposed model or created as new projects, according to the user's decision. Moreover, the following requirements have been identified for the decomposition plug-in:

1. The configuration (i.e. input machine, decomposition style selection, identification of sub-components to be generated and respective partition) shall be performed through the Graphical User Interface (GUI) of the Rodin platform. It is indeed more suitable for the end-user to visualise the configuration. In the future the option of using GMF (Graphical Modelling Framework) for the decomposition visualization will be explored.
2. The decomposition configuration shall be stored persistently. Since there is no direct relation between the non-decomposed model and the decomposed sub-models, the possibility of saving, editing and replaying a model decomposition seems suitable.

3. A `Decompose` action shall be added. It shall be available from the toolbar and from the popup menu of the Event-B explorer (right-clicking on a machine).
4. The created projects and components (machines and contexts) shall be tagged as “automatically generated”.

## 4 Conclusion

This paper presents a preliminary study about the decomposition of Event-B models. The Event-B model decomposition can advantageously be used if the motivation is to decrease the complexity and increase the modularity of large systems, especially after several layers of refinements. The main benefits are the distribution of POs into several sub-models, which is expected to be easier to discharge, and the further refinement of independent sub-models in parallel. Moreover the possibility of team development seems a very attractive option for the industry. This paper propose the reuse of sub-components in Event-B through the shared variable (A-style) or shared event (B-style) decompositon. The shared event approach seems particularly suitable for message-passing distributed programs, while the shared variable approach seems more suitable when designing concurrent programs. However the end-user will choose a decomposition style depending on the specific system and on their modelling preferences.

## References

1. Jean-Raymond Abrial. Event model decomposition. Technical report, ETH Zurich, 2009, Private communication.
2. Jean-Raymond Abrial. Mathematical models for refinement and decomposition. `http://www.event-b.org/abook.html`, 2009, To be published.
3. Jean-Raymond Abrial and Stefan Hallerstede. Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundam. Inf.*, 77(1-2):1–28, 2007.
4. Michael 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*, pages 221–241, 1997.
5. Michael Butler. Synchronisation-based Decomposition for Event-B. In *RODIN Deliverable D19 Intermediate report on methodology*, 2006.
6. Michael Butler. Incremental Design of Distributed Systems with Event-B. *Marktoberdorf Summer School 2008 Lecture Notes*, November 2008.

7. Michael Butler. Decomposition Structures for Event-B. *Integrated Formal Methods iFM2009*, February 2009.
8. C. Métayer, J.-R. Abrial, and L. Voisin. Event-B Language. Technical report, Deliverable 3.2, EU Project IST-511599 - RODIN, May 2005.
9. Carine Pascal. Event Model Decomposition. `http://wiki.event-b.org/index.php/Event_Model_Decomposition`, 2009.
10. Rodin. RODIN project Homepage. `http://rodin.cs.ncl.ac.uk`, September 2008.