COMPUTING
SCIENCE

Proceedings: Workshop on the experience of and advances in developing dependable systems in Event-B

Fuyuki Ishikawa and Alexander Romanovsky (Eds.)
Proceedings: Workshop on the experience of and advances in developing dependable systems in Event-B

F. Ishikawa, A. Romanovsky (Eds.)

Abstract

These proceedings include papers to be presented at the Workshop on "The experience of and advances in developing dependable systems in Event-B" to be held on November 13, 2012 as part of the ICFEM 2012 (Kyoto, Japan).
Bibliographical details

ISHIKAWA, F., ROMANOVSKY, A. (EDS.)
Proceedings: Workshop on the experience of and advances in developing dependable systems in Event-B
[By] F. Ishikawa, A. Romanovsky (Eds.)
(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1355)

Added entries
NEWCASTLE UNIVERSITY

Abstract
These proceedings include papers to be presented at the Workshop on “The experience of and advances in developing dependable systems in Event-B” to be held on November 13, 2012 as part of the ICFEM 2012 (Kyoto, Japan).

About the editors
Fuyuki Ishikawa is an associate professor at National Institute of Informatics (NII), Japan. His Ph.D. degree is in Information Science and Technology, from The University of Tokyo, Japan (2007). His research interests include Software Engineering and Services Computing. He has also served for dissemination and education of formal methods to the Japanese industry.

Alexander (Sascha) Romanovsky is a Professor in the Centre for Software and Reliability, Newcastle University. His main research interests are system dependability, fault tolerance, software architectures, exception handling, error recovery, system structuring and verification of fault tolerance. He received a PhD degree in Computer Science from St. Petersburg State Technical University and has worked as a visiting researcher at ABB Ltd Computer Architecture Lab Research Center, Switzerland and at Istituto di Elaborazione della Informazione, CNR, Pisa, Italy. In 1993 he became a postdoctoral fellow in Newcastle University, and worked on the ESPRIT projects on Predictable Dependable Computing Systems (PDCS), Design for Validation (DeVa) and on UK-funded projects on the Diversity, both in Safety Critical Software using Off-the-Shelf components. He was a member of the executive board of EU Dependable Systems of Systems (DSoS) Project, and between 2004 and 2012 headed projects on the development of a Rigorous Open Development Environment for Complex Systems (RODIN), and latterly was coordinator of the major FP7 Integrated Project on Industrial Deployment of System Engineering Methods Providing High Dependability and Productivity (DEPLOY). He now leads work on fault tolerance in Systems of Systems within the COMPASS project and is Principal Investigator of Newcastle’s Platform Grant on Trustworthy Ambient Systems.

Suggested keywords
FORMAL METHODS
EVENT-B
RODIN
DEPLOY
DEPENDABILITY
Proceedings:

Workshop on the experience of and advances in developing dependable systems in Event-B

Fuyuki Ishikawa, Alexander Romanovsky (Co-Chairs)

These proceedings include papers to be presented at the Workshop on “The experience of and advances in developing dependable systems in Event-B” to be held on November 13, 2012 as part of the ICFEM 2012 (Kyoto, Japan).
Proceedings: Workshop on the experience of and advances in developing dependable systems in Event-B

These proceedings include papers to be presented at the Workshop on “The experience of and advances in developing dependable systems in Event-B” to be held on November 13, 2012 as part of the ICFEM 2012 (Kyoto, Japan).

Event-B is a formal method for the system level modelling and analysis of dependable applications. It is supported by an open and extendable Eclipse-based toolset called Rodin (http://www.event-b.org/), which has been developed in a series of European projects (notably DEPLOY - http://www.deploy-project.eu/). A significant recent rise in interest in these methods and tools has now led to a shaping of a broad community of users, developers, researchers and educators.

The workshop aims to bring this community together to discuss new and emerging issues in applying and advancing both the Event-B method and the Rodin platform as well as address challenges that industrial takers are facing while deploying them.

One of the aims of this workshop is to expand the community of researchers and practitioners working with Event-B/Rodin. This is achieved in two ways: by addressing general scientific challenges and by providing information valuable for the newcomers.

We are grateful to all workshop participants for their interest and to the two invited speakers, Andreas Roth (SAP, Germany) and Yuusuke Hashimoto (NEC and Dependable Software Forum, Japan), for accepting our invitations. Our special thanks go to the PC members who helped us with building the exciting programme: Michael Butler, Stefan Hallerstede, Thai Son Hoang, Alexei Iliasov, Hironobu Kuruma, Linas Laibinis, Regine Laleau, Thierry Lecomte, Michael Leuschel, Felix Loesch, Dominique Mery, Shin Nakajima, Jose Reis, Andreas Roth, Aryldo G Russo, Elena Troubitsyna, Laurent Voisin and Stephen Wright.

Fuyuki Ishikawa (National Institute of Informatics, Japan)
Alexander Romanovsky (Newcastle University, UK)

Workshop Co-Chairs
Table of Content

**Andreas Fürst, Thai Son Hoang, David Basin, Kunihiko Miyazaki and Naoto Sato:** Abstract Data Types in Event-B – An Application of Generic Instantiation

**Inna Pereverzeva, Elena Troubitsyna and Linas Laibinis:** Development of Fault Tolerant MAS with Cooperative Error Recovery by Refinement in Event-B

**Andrew Edmunds, Michael Butler and John Colley:** Building on the DEPLOY Legacy: Code Generation and Simulation

**Renato Alexandre Silva:** Lessons Learned/Sharing the Experience of Developing a Metro System Case Study

**Alexei Iliasov:** Event-B/SLP

**Thai Son Hoang:** Proof Hints for Event-B

**Tsutomu Kobayashi and Shinichi Honiden:** Towards Refinement Strategy Planning for Event-B

**Thierry Lecomte, Lilian Burdy and Michael Leuschel:** Formally Checking Large Data Sets in the Railways

**Alexei Iliasov and Jeremy W. Bryans:** A recipe for timed Event-B specifications

**Elena Troubitsyna:** Dependability-explicit engineering with Event-B: overview of recent achievements

**Frédéric Badeau and Marielle Doche-Petit:** Formal Data Validation with Event-B
Abstract Data Types in Event-B – An Application of Generic Instantiation

David Basin\textsuperscript{1}, Andreas Fürst\textsuperscript{1}, Thai Son Hoang\textsuperscript{1}, Kunihiko Miyazaki\textsuperscript{2}, and Naoto Sato\textsuperscript{2}

\textsuperscript{1} Institute of Information Security, ETH Zurich
\textsuperscript{2} Yokohama Research Lab, Hitachi

Abstract. Integrating formal methods into industrial practice is a challenging task. Often, different kinds of expertise are required within the same development. On the one hand, there are domain engineers who have specific knowledge of the system under development. On the other hand, there are formal methods experts who have experience in rigorously specifying and reasoning about formal systems. Coordination between these groups is important for taking advantage of their expertise. In this paper, we describe our approach of using generic instantiation to facilitate this coordination. In particular, generic instantiation enables a separation of concerns between the different parties involved in developing formal systems.

1 Introduction

Event-B is a formal method for modelling safe and reliable systems. Industrial awareness of Event-B has been enhanced by recent collaboration projects (e.g., DEPLOY [4]). These projects acted as a bridge for deploying research results in various industrial contexts with considerable success. Moreover, they also highlighted several challenges in integrating formal methods into industrial development processes. In particular, questions about interactions between developers with different kinds of expertise often arise during the deployment. On the one hand, engineers have domain knowledge including how the systems should work and why they work, but often find it challenging to formalise their reasoning. On the other hand, formal method experts, which do not have inside knowledge about the specific systems, have experience in reasoning formally about systems in general.

In this paper, we propose adapting the concept of abstract data types to Event-B to enable the interaction between the domain and formal methods experts. Abstract data types allow developers to hide implementation details that are initially irrelevant to the development of a system. As a result, systems developed with abstract data types are more intuitive and easier to verify. The realisation of the abstract data types can be done via generic instantiation by Event-B experts. In particular, the choice of which (concrete) data structure to use to represent the abstract data type can be done independently of the actual
system under development. Later, generic instantiation in Event-B enables the
Event-B expert to prove that the chosen data structure is a valid realisation of
the abstract data type.

Generic instantiation in Event-B was introduced in [3] and further elaborated
in [8]. These works show how generic instantiation works with other standard
techniques in Event-B such as refinement and composition. This paper illustrates
how abstract data types can be modelled and realised using generic instantiation.
Similar to our work is the recently developed Theory Plug-in [6]. The primary
usage of the Theory Plug-in is to extend the mathematical language to include
new data types. A theory module also provides an encapsulation of datatypes
and enables the separation of concerns between the data types and the models
that make use of them. The main difference between our work and the Theory
Plug-in is that data types are usually developed together with their properties
within the same theory module. As a results, the data types developed using
the Theory Plug-in are usually already concrete. There is no clear separation
between actual representation of data types and their abstract properties. More
information on related work is in Section 5.

Structure In Section 2 we will give a brief overview of Event-B and generic
instantiation in Event-B. We describe our approach in Section 3. In Section 4 we
demonstrate our methodology of splitting the modelling effort on an example.
In Section 5 we compare our approach with other existing approaches and in
Section 6 we draw conclusions.

2 Background

2.1 The Event-B Modelling Method

Event-B [1] is a modelling method for formalising and developing systems whose
components can be modeled as discrete transition systems. Event-B is centered
around the general notion of events and its semantics is based on transition
systems and simulation between such systems, as described in [1]. We will not
describe in detail the semantics of Event-B here. Instead we just give a brief
description of Event-B models, which are important for generic instantiation.

Event-B models are organised in terms of two basic constructs: contexts and
machines. Contexts specify the static part of a model whereas machines specify
the dynamic part. Contexts may contain carrier sets, constants, axioms, and
theorems. Carrier sets are similar to types. Axioms constrain carrier sets and
constants, whereas theorems are additional properties derived from axioms. The
role of a context is to isolate the parameters of a formal model (carrier sets and
constants) and their properties, which are intended to hold for all instances.

Machines specify behavioural properties of Event-B models. Machines may
contain variables, invariants (and theorems), and events. Variables v define the
state of a machine and are constrained by invariants I(v). Theorems are addi-
tional properties of v derivable from I(v). Possible state changes are described
by events. An event \( \text{evt} \) can be represented by the term

\[
\text{evt} \doteq \text{any } t \text{ where } G(t, v) \text{ then } S(t, v) \text{ end}
\]

where \( t \) stands for the event’s parameters, \( G(t, v) \) is the guard (the conjunction of one or more predicates) and \( S(t, v) \) is the action. The guard states the necessary condition under which an event may occur, and the action describes how the state variables evolve when the event occurs. We use the short form \( \text{evt} \doteq \text{when } G(v) \text{ then } S(v) \text{ end} \) when the event does not have any parameters, and we write \( \text{evt} \doteq \text{begin } S(v) \text{ end} \) when, in addition, the event’s guard equals true. A dedicated event without parameters and guard is used for the initialisation event (usually represented as \( \text{init} \)).

A machine can see multiple contexts. During the development, a context extends one or more contexts by declaring additional carrier sets, constants, axioms or theorems. An abstract machine can be refined by another concrete machine. The variables of the abstract and concrete machines are related by some gluing invariants. The existing events are refined accordingly to this relationship. Moreover, new events can be added to the concrete machine. The new events must refine a special \( \text{skip} \) event, which does not change the abstract variables.

### 2.2 Generic Instantiation in Event-B

Generic instantiation is a technique for reusing models by giving concrete values for abstract parameters of the models. Generic instantiation for Event-B is first mentioned in [3] and is further elaborated in [8]. We summarise the approach as follows. Suppose we have an abstract development with machines \( M_1 \ldots M_n \) and their corresponding contexts \( C_1 \ldots C_n \) as shown in Fig. 1. The development is generic, with the carrier sets \( s \) and constants \( c \) from the contexts \( C_1 \ldots C_n \) acting as its parameters. Assume that \( s \) and \( c \) are constrained by axioms \( A(s, c) \).

![Fig. 1. Generic instantiation in Event-B](image)

The abstract generic model can be instantiated within another development containing contexts \( D_1 \ldots D_m \). Assume that the concrete contexts \( D_1 \ldots D_m \) contain concrete carrier sets \( t \) and constants \( d \), constrained by axioms \( B(t, d) \). The instantiation is done by giving values for the abstract carrier sets \( s \) and
constants $c$ in terms of concrete $t$ and $d$. Let the concrete expressions $E(t, d)$ and $F(t, d)$ be the instantiated values for $s$ and $c$ respectively. Soundness for generic instantiation requires us to prove that the instantiated abstract axioms are derivable from the concrete axioms, i.e.,

$$B(t, d) \Rightarrow A(E(t, d), F(t, d)).$$

In this paper, we further restrict the instantiation for the abstract carrier sets $s$ so that they can only be instantiated by type-expressions, i.e., $E(t, d)$ must be some type-expressions. This is because a carrier set $S$ in Event-B is assumed to satisfy two additional constraints (i.e., beside the stated axioms).

- **non-empty**: $S$ is non-empty, i.e., $S \neq \emptyset$.
- **maximal**: $S$ is maximal, i.e., $\forall x : x \in S$.

The **maximal** condition is due to the fact that the Event-B models are typed. As a result, expressions used for instantiating carrier sets must be also some type-expressions, i.e., satisfying the above two conditions.

Applying generic instantiation, machines $N_1 \ldots N_n$ are instances of $M_1 \ldots M_n$ by syntactically replacing $s$ and $c$ by $E(t, d)$ and $F(t, d)$. The advantage here is that the instantiated machines are correct by construction. The resulting model can be used in conjunction with other techniques such as refinement [3] and composition [8].

### 3 Abstract Data Types in Event-B

An abstract data type is a mathematical model of a class of data structures. An abstract data type is typically defined in terms of the operations that may be performed on the data type with some mathematical constraints on the effects of such operations. The advantage of using an abstract data type is that the reasoning can be done purely based on the properties of the operations, regardless of the implementation. We want to use this idea in our developments. In particular the separation between the abstraction and the implementation enables us to split the work between domain experts and formal methods experts.

An abstract data type and its operations can be captured straightforwardly using contexts in Event-B. Generic instantiation can then be used to “implement” the abstract data type and prove that the actual implementation satisfies the constraints on the effects of the operations. Our approach can be summarised as follows.

**Domain experts**: The domain experts make use of some abstract data types and operations defined within some context to model the system in Event-B.

**Formal methods experts**: The formal methods experts use generic instantiation to include the details on how the abstract data types are represented and prove that the representations satisfy the assumptions of the abstract data types stated earlier.
We illustrate the use of generic instantiation by a model of the standard stack data type. A stack is a last in, first out (LIFO) data type that contains a collection of elements. A stack is characterised by two fundamental operations: push and pop. The push operation adds a new item to the top of the stack. The pop operation removes the stack’s top element. A special constant empty_stack denotes the empty stack. The stack abstract data type can be modelled using a context as follows. Notice that we have defined the “type” STACK_TYPE as a carrier set and the set of possible stacks STACK as a constant.

\[ \text{sets} : \text{STACK\_TYPE, ELEM} \]

\[ \text{constants} : \text{STACK, empty\_stack, push, pop} \]

\[ \text{axioms} : \]

axm0_1 : STACK \subseteq STACK\_TYPE
axm0_2 : empty\_stack \in STACK
axm0_3 : push \in STACK \times ELEM \rightarrow STACK
axm0_4 : pop \in STACK \rightarrow STACK
axm0_5 : \text{dom}(\text{pop}) = \text{STACK} \setminus \{\text{empty\_stack}\}
axm0_6 : \forall s, e \cdot s \in \text{STACK} \Rightarrow \text{push}(s, e) \neq \text{empty\_stack}
axm0_7 : \forall s, e \cdot s \in \text{STACK} \Rightarrow \text{pop}((\text{push}(s, e))) = s

In the representation of stack data type, each stack is represented by a pair \( f \mapsto n \), where \( f \) represents the content of the stack and \( n \) represents the size of the stack. Other operations of the stack data type are defined accordingly. The concrete context used for instantiation is as follows. Note that we use set comprehension to define the constants accordingly.

\[ \text{sets} : \text{ELEM} \quad \text{constants} : \text{STACK, empty\_stack, push, pop} \]

\[ \text{axioms} : \]

axm1_1 : \text{STACK} = \{f \mapsto n \mid n \in \mathbb{N} \land f \in 1..n \rightarrow \text{ELEM}\}
axm1_2 : \text{empty\_stack} = \varnothing \mapsto 0
axm1_3 : \text{push} = \{f, n, e \cdot f \mapsto n \in \text{STACK} \land e \in \text{ELEM} \mid ((f \mapsto n) \mapsto e) \mapsto ((f \mapsto (\{(n + 1) \mapsto e\}) \mapsto n + 1))\}
axm1_4 : \text{pop} = \{f, n \cdot f \mapsto n \in \text{STACK} \land n \neq 0 \mid (f \mapsto n) \mapsto ((\{n \mapsto f\}) \mapsto n - 1)\}

To prove that the representation of the stack data type is consistent with the stack abstract data type, we can use instantiation where the abstract constants are instantiated with concrete constants with the same name. The abstract carrier set STACK\_TYPE is instantiated with \( \mathbb{P}(\mathbb{Z} \times \text{ELEM}) \times \mathbb{Z} \). The abstract axioms (i.e., axm0_1 – axm0_7) must be derived from the concrete axioms (i.e., axm1_1 – axm1_4). This can be done by expanding the definitions of the concrete constants accordingly.

4 Example

We illustrate our approach by modelling a set of trains on a railway network, inspired by the example in [1, Chapter 17].
4.1 Requirements Document

A railway network is divided into sections. An example of such a network is showed in Figure 2, taken from [1, Chapter 17].

![Fig. 2. Layout of a sample network with sections A to N.]

A set of trains are moving within the network. Two important requirements are that trains must not derail or collide. To avoid collision, the system must ensure that each section is occupied by at most one train. Moreover, trains are assumed to move only forward within the network.

SAF 1 For each section, at most one train occupies that section.

SAF 2 Trains are always on the network.

ASM 3 Trains only move forward.

4.2 Informal Discussion

An important part of the model will formalise the trains moving within the network. Intuitively, a train can be seen as the sequence of consecutive sections that it occupies within the network. There are different possible formalisation of the trains, e.g., using functions relating occupied sections as in [1, Chapter 17], or modeling sequences as functions from integers to sections. However, the system should be correct regardless of which modelling style is used to represent the trains. In particular, the formalisation of the trains in Event-B is of little interest to the domain experts. It would be easier for the domain experts to model the trains at the more abstract level, i.e. with a train abstract data type. The decision of which representation for the train data type will be decided by the Event-B experts. In particular, different representations can be used for the train data type via separate instantiation.
4.3 Formal Model

Train Abstract Data Type We first formalise the train abstract data type in a context, focusing on requirement SAF 1. In particular, we consider the following “attributes” of a train: the sections that the train occupies (we refer to them as the train’s area), the section of the train’s head (the end where the train driver is sitting) and the section of the train’s rear (the opposite end). This is illustrated in Figure 3.

![Diagram of train in network occupying sections](image)

**Fig. 3.** Train in the network occupying sections.

Let the set of sections be a carrier set $SECTION$. We abstractly represent the trains state by a constant $TRAIN$, that is a subset of the carrier set $TRAIN\_TYPE$. Three function constants, namely $area$, $head$, and $rear$, are used to get the information about the trains’ area, head position, and rear position, respectively. For an abstract data type describing a train, one can see these constants as operations of the data type.

- $area$: takes a train state, returns a set of sections.
- $head$: takes a train state, returns a section.
- $rear$: takes a train state, returns a section.

In Event-B, we give the typing information for these constants using the following axioms.

$$
\text{area\_Type}: \text{area} \in \text{TRAIN} \rightarrow \mathbb{P}(\text{SECTION}) \\
\text{head\_Type}: \text{head} \in \text{TRAIN} \rightarrow \text{SECTION} \\
\text{rear\_Type}: \text{rear} \in \text{TRAIN} \rightarrow \text{SECTION}
$$

Further constraints on these constants will be given later when they are needed for maintaining the correctness of the machines that use this data type.

When a train moves, the set of sections it occupies changes. When moving forward, ASM 3, the train’s head reaches the end of its head section and moves to the new section ahead. Similarly, when the train’s rear leaves the train’s rear section, the rear is reassigned. The train’s area is updated accordingly: it is extended to include the new head section when the head moves, and the rear section is removed when the rear moves. As a result, we define two additional operations for manipulating the train.
add_head: takes a train state and a section, returns a train state.
front: takes a train state, returns a train state.

In Event-B, we give the type for these constant as follows.

add_head_Type : add_head \in TRAIN \times SECTION \rightarrow TRAIN
front_Type : front \in TRAIN \rightarrow TRAIN

Note that we use partial functions to indicate that there are some constraints for extending the train’s head and removing the train’s rear.

Finally, we define an additional operation new_train to create a new train when the train enters the network from a particular section.

new_train_Type : new_train \in SECTION \rightarrow TRAIN

**System Model Using Train Abstract Data Type**

Using the train abstract data type, the system can be straightforwardly modelled. Let TRAIN_ID be the set of possible IDs for trains in the network. The variable trains represents the trains currently monitored by the systems, which is a mapping from train IDs to actual trains. Initially, trains is assigned the empty set \( \emptyset \).

variables : trains invariants :
trains \in TRAIN_ID \rightarrow TRAIN

Three events enter, extend_head, remove_rear are used to model the different cases where a train enter the network, a train extends its head to a new section, and a train removes its rear section.

\[
\begin{align*}
\text{enter} & \quad \text{any } t, s \text{ where } t \notin \text{dom}(\text{trains}) \land s \in \text{SECTION} \\
& \quad \text{then} \\
& \quad \quad \text{trains}(t) := \text{new_train}(s) \\
& \quad \text{end} \\
\text{extend_head} & \quad \text{any } t, s \text{ where } t \in \text{dom}(\text{trains}) \land s \notin \text{area}(\text{trains}(t)) \\
& \quad \text{then} \\
& \quad \quad \text{trains}(t) := \text{add_head}(\text{trains}(t) \mapsto s) \\
& \quad \text{end} \\
\text{remove_rear} & \quad \text{any } t \text{ where } t \in \text{dom}(\text{trains}) \land \text{head}(\text{trains}(t)) \neq \text{rear}(\text{trains}(t)) \\
& \quad \text{then} \\
& \quad \quad \text{trains}(t) := \text{front}(\text{trains}(t)) \\
& \quad \text{end}
\end{align*}
\]

In particular the guard of extend_head states that the new section \( s \) is not already occupied by the train \( t \), and the guard of remove_rear states that the head and the rear of the train \( t \) are in different sections. Moreover, these events lead us to the following constraints about the domain of operations add_head and front.

\[
\begin{align*}
\text{add_head_dom} : & \quad \text{dom}(\text{add_head}) = \{ t \mapsto s \mid t \in \text{TRAIN} \land s \notin \text{area}(t) \} \\
\text{front_dom} : & \quad \text{dom}(\text{front}) = \{ t \mid t \in \text{TRAIN} \land \text{head}(t) \neq \text{rear}(t) \}
\end{align*}
\]
An important invariant captures requirement SAF 1, stating that for any two distinct trains \( t_1, t_2 \), they do not occupy the same section.

\[
\forall t_1, t_2 \cdot t_1 \in \text{dom}(\text{trains}) \land t_2 \in \text{dom}(\text{trains}) \land t_1 \neq t_2 \Rightarrow \text{area}(\text{trains}(t_1)) \cap \text{area}(\text{trains}(t_2)) = \emptyset
\]

The invariant leads to the following additional guard for \text{enter} and \text{extend\_head}

\[
\forall t_1 \cdot t_1 \in \text{dom}(\text{trains}) \Rightarrow s \notin \text{area}(\text{trains}(t_1))
\]

While proving the correctness of our model, we discovered the following required constraints on the train abstract data type. These constraints are formalised by additional axioms over the abstract data type’s operations.

- **\text{area\_add\_head}**: \( \forall t, s \cdot t \mapsto s \in \text{dom}(\text{add\_head}) \Rightarrow \text{area}(\text{add\_head}(t \mapsto s)) = \text{area}(t) \cup \{s\} \)
- **\text{area\_front}**: \( \forall t \cdot t \in \text{dom}(\text{front}) \Rightarrow \text{area}(\text{front}(t)) = \text{area}(t) \setminus \{\text{rear}(t)\} \)
- **\text{area\_new\_train}**: \( \forall s \cdot s \in \text{SECTION} \Rightarrow \text{area}(\text{new\_train}(s)) = \{s\} \)

In order to specify the fact that the trains do not derail, SAF 2, we introduce another operation, \text{connection}, on the train abstract data type to specify the connections of the sections belonging to a train. The typing information for \text{connection} is as follows.

- **\text{connection\_Type}**: \( \text{connection} \in \text{TRAIN} \to (\text{SECTION} \leftrightarrow \text{SECTION}) \)

The invariant corresponding to SAF 2 is

\[
\forall t \cdot t \in \text{dom}(\text{trains}) \Rightarrow \text{connection}(\text{trains}(t)) \subseteq \text{NETWORK}
\]

where \text{NETWORK} is a constant describing the topology of the actual network. An additional guard is added to event \text{extend\_head} as follows.

\[s \mapsto \text{head}(\text{trains}(t)) \in \text{NETWORK}\]

Again, we discovered additional constraints on the operation \text{connection} while proving the model.

- **\text{connection\_add\_head}**: \( \forall t, s \cdot t \mapsto s \in \text{dom}(\text{add\_head}) \Rightarrow \text{connection}(\text{add\_head}(t \mapsto s)) = \text{connection}(t) \cup \{s \mapsto \text{head}(t)\} \)
- **\text{connection\_front}**: \( \forall t \cdot t \in \text{dom}(\text{front}) \Rightarrow \text{connection}(\text{front}(t)) \subseteq \text{connection}(t) \)
- **\text{connection\_new\_train}**: \( \forall s \cdot s \in \text{SECTION} \Rightarrow \text{connection}(\text{new\_train}(s)) = \emptyset \)

Note that axiom \text{connection\_front} does not specify exactly how a train’s connection is changed when the rear is removed. It only specifies that the connection will not be enlarged. This suffices for proving the no-derailment property of the system.
Generic Instantiation  We now need to find a representation for the train data type. This is the point where the role of the formal method expert becomes prominent. As mentioned before, different data structures can be used to represent the train abstract data type. We present here a solution where a train is represented by a function from an integer interval to the set of sections. Each train is associated with a tuple \((a,b,f)\), where the interval \(a .. b\) represents the domain of a total injective function \(f\).

\[
\text{train}_\text{Def} : \text{TRAIN} = \{a \mapsto b \mapsto f \mid a \in \mathbb{Z} \land a \leq b \land f \in a .. b \mapsto \text{SECTION}\}
\]

The train’s head is located at the lower end of the interval \((a)\) and its rear at the upper end \((b)\). Injectivity guarantees that the sequence cannot include a section twice at different positions. The operations on the train data type are defined accordingly.

- **head**\(_\text{Def} \) : \(\text{head} = \{a, b, f : a \mapsto b \mapsto f \in \text{TRAIN} \mid (a \mapsto b \mapsto f) \mapsto f(a)\}\)
- **rear**\(_\text{Def} \) : \(\text{rear} = \{a, b, f : a \mapsto b \mapsto f \in \text{TRAIN} \mid (a \mapsto b \mapsto f) \mapsto f(b)\}\)
- **area**\(_\text{Def} \) : \(\text{area} = \{a, b, f : a \mapsto b \mapsto f \in \text{TRAIN} \mid (a \mapsto b \mapsto f) \mapsto f[a .. b]\}\)
- **add_head**\(_\text{Def} \) : \(\text{add\_head} = \{a, b, f, s : a \mapsto b \mapsto f \in \text{TRAIN} \land s \notin [a .. b] \mid (a \mapsto b \mapsto f) \mapsto s \mapsto ((a - 1) \mapsto b \mapsto f \cup \{(a - 1) \mapsto s\})\}\)
- **front**\(_\text{Def} \) : \(\text{front} = \{a, b, f : a \mapsto b \mapsto f \in \text{TRAIN} \land a \neq b \mid (a \mapsto b \mapsto f) \mapsto (a \mapsto (b - 1) \mapsto \{(b) \mapsto f\})\}\)
- **new_train**\(_\text{Def} \) : \(\text{new\_train} = \{s : s \in \text{SECTION} \mid s \mapsto (1 \mapsto 1 \mapsto \{1 \mapsto s\})\}\)
- **connection**\(_\text{Def} \) : \(\text{connection} = \{a, b, f : a \mapsto b \mapsto f \in \text{TRAIN} \mid (a \mapsto b \mapsto f) \mapsto \{i : i \in a .. b - 1 \mid t(i) \mapsto t(i + 1)\}\}\)

By instantiating the abstract type \(\text{TRAIN}\_\text{TYPE}\) to \(\mathbb{Z} \times \mathbb{Z} \times (\mathbb{Z} \times \text{SECTION})\) and other abstract constants with the concrete constants of the same name, we can prove that the constraints of the train abstract data type (abstract axioms) are derivable from the definition of the train data type.

For instantiating the train abstract data type, we used the prototype plug-in for generic instantiation.

5 Related Work

Generic instantiation in Event-B has been introduced in [3] and is further elaborated in [8]. Both papers illustrate the use of generic instantiation for reusing formal models by combining it with existing techniques like refinement and composition. In this paper, we illustrate another application of generic instantiation for algebraically modelling abstract data types. In particular, the abstract development and the concrete instantiated development enable the separation of concerns between domain experts and formal methods experts. The domain experts can work with the abstract models, stating the assumptions under which the systems work correctly. The formal method experts use generic instantiation to prove that the actual implementations satisfy the assumptions as required by the domain experts.

A similar form of generic instantiation is also available in classical B [2]. A development in classical B also contains abstract data which must be finalised when...
the final software products are deployed. This finalisation process is an instantiation step, involving validating that the actual data satisfies the assumptions stated in the formal model [5]. We illustrate here (together with other work [3,8]) that generic instantiation is also useful during the stepwise development of the formal models, not just as the last realisation step in deploying the formal models.

Recent development of the Theory Plug-in [6] allows users to extend the mathematical languages of Event-B, e.g., by including new data types. Theorems about new data types can be stated and used later by a dedicated tactic associated with the Theory Plug-in. There is also a clear distinction between the theory modules (capturing data structures and their properties) and the Event-B models making use of the newly defined data structures. This distinction also enables a collaboration between domain experts and formal methods experts: the domain experts work with the Event-B models while the formal methods experts work with the theory modules. The difference with our approach is the order in which the work is carried out. With the Theory Plug-in, the domain experts rely on the theory developed by the formal methods experts. In our approach, the input for the formal methods experts are the abstract models that are developed by the domain experts, including the assumptions stated as axioms on the abstract carrier sets and constants. Another difference is that we can have different implementations for the abstract data types.

Our approach is similar to work on algebraic specification [7]. In this domain, a specification contains a collection of sorts, operations, and axioms constraining the operations. Specifications can be enriched by additional sorts, operations, or axioms. Furthermore, to develop programs from specifications, the specifications are transformed via a sequence of small refinement steps. During these steps, the operations are “coded” until the specification becomes a concrete description of a program. For each such refinement step, it is required to prove that the code of the operations satisfy the axioms constraining them. An algebraic specification therefore corresponds to an Event-B context, while the refinement of the algebraic specifications is similar to generic instantiation in Event-B. The main difference between algebraic specification and Event-B is that there is no corresponding elements to Event-B machines. In particular, we make use of the dynamic information of Event-B machines to derive the necessary axioms on the abstract data types.

6 Conclusion and Future Work

In this paper we presented our approach to modeling abstract data types and their implementation in Event-B. Using abstract data types allows us to hide irrelevant details that are not important for the domain expert. The domain expert can focus on modelling the functionality of the system which is his core competence. Abstract data types thereby have a similar purpose to programming interfaces in programming languages. The instantiation of the abstract data type is left to an Event-B expert. The way we introduced the concept of abstract data
types in our approach allows us to utilise generic instantiation which handles both the substitution of the abstract data type by the chosen data structure as well as the generation of the needed proof obligations to guarantee that the chosen structure is a valid instance of the abstract data type.

We successfully applied our approach to the example in this paper as well as a substantially more complex version of it. Further investigation is needed on the scalability of the approach, which is essential for its applicability in industrial development processes. Furthermore, we are interested in applying our approach outside the domain of railway systems to obtain evidence for its generality.

References

Development of Fault Tolerant MAS with Cooperative Error Recovery by Refinement in Event-B

Inna Pereverzeva¹,², Elena Troubitsyna², and Linas Laibinis²

¹ Turku Centre for Computer Science
² Åbo Akademi University, Finland
{inna.pereverzeva, elena.troubitsyna, linas.laibinis}@abo.fi

Abstract. Designing fault tolerance mechanisms for multi-agent systems is a notoriously difficult task. In this paper we present an approach to formal development of a fault tolerant multi-agent system by refinement in Event-B. We demonstrate how to formally specify cooperative error recovery and dynamic reconfiguration in Event-B. Moreover, we discuss how to express and verify essential properties of a fault tolerant multi-agent system while refining it. The approach is illustrated by a case study – a multi-robotic system.

Keywords: Event-B, formal modelling, refinement, fault tolerance, multi-agent system

1 Introduction

Multi-agent systems (MAS) and in particular the agent cooperation have been a subject of an active research over the last decade. In this paper we focus on studying the fault tolerance aspects of agent cooperation. Namely, we discuss how to express and verify essential properties of a fault tolerant MAS. Moreover, we show by example how to formally derive a specification of a MAS that relies on dynamic reconfiguration and cooperative error recovery to achieve fault tolerance.

In this paper, we present a formal development of a cleaning multi-robotic system. The system has a heterogenous architecture consisting of several stationary devices, base stations, that coordinate the work of respective groups of robots. Since both base stations and robots can fail, the main objective of our formal development is to formally specify cooperative error recovery and verify that the proposed design ensures goal reachability. The proposed development approach ensures goal reachability “by construction”. It is based on refinement in Event-B [1] – a formal top-down approach to correct-by-construction system development. In this paper we demonstrate how to formally define a system goal and, in a stepwise manner, derive a detailed specification of the system architecture.

The paper is structured as follows. In Section 2 we briefly overview the Event-B formalism. In Section 3 we define the main principles of formal reasoning
about goal-oriented MAS, describe the requirements for our case study – a multi-
robotic system – and outline the development strategy. Section 4 presents a
formal development of the system and demonstrates how to express and verify
its properties during the refinement process. Finally, in Section 5 we overview
the related work and discuss the achieved results.

2 Modelling and Refinement in Event-B

Event-B is a state-based formal approach that promotes the
correct-by-construction development paradigm and formal verification by the-
orem proving [1]. In Event-B, a system model is specified using the notion of
an abstract state machine. An abstract state machine encapsulates the model
state represented as a collection of variables, and defines operations on this
state, i.e., it describes the behaviour of the modelled system. A machine may
have the accompanying component, called context. A context may include user-
defined carrier sets, constants and their properties (model axioms). In Event-B,
the model variables are strongly typed by the constraining predicates called in-
vvariants. Moreover, the invariants specify important properties that should be
preserved during the system execution.

The dynamic behaviour of the system is defined by the set of atomic events.
Generally, an event can be defined as

\[
\text{evt} \equiv \text{any } v_l \text{ where } g \text{ then } S \text{ end}
\]

where \( v_l \) is a list of new local variables, \( g \) is the guard, and \( S \) is the action. The
guard is a state predicate that defines the conditions under which the action
can be executed. In general, the action of an event is a parallel composition of
deterministic or non-deterministic assignments.

The Event-B refinement process allows us to gradually introduce implementa-
tion details, while preserving functional correctness. The consistency of Event-B
models, i.e., invariant preservation, correctness of refinement steps, should be
formally demonstrated by discharging relevant proof obligations. The verification
efforts, in particular, automatic generation and proving of the required
proof obligations, are significantly facilitated by the Rodin platform [10]. Proof-
based verification as well as reliance on abstraction and decomposition adopted
in Event-B offers the designers a scalable support for the development of such
complex distributed systems as MAS.

3 Multi-Agent Systems

Our paper focuses on formal modelling and development of MAS that should
function autonomously, i.e., without human intervention. Typically, the main
task or goal that such a MAS should accomplish is split between the deployed
agents. Since agents may fail, to ensure success of the overall goal, we should in-
corporate some fault tolerance mechanisms into the system design. These mech-
anisms rely on cooperative error recovery that allows the system dynamically
reallocate functions from the failed agents to the healthy ones. A large number
of failure modes and scenarios makes verification of goal reachability in the pres-
ence of cooperative error recovery quite difficult and time-consuming. Therefore,
there is a clear need for rigorous approaches that support scalable design and verification in a systematic manner.

3.1 Towards a Formalisation of a Goal-Oriented MAS

Let us now describe more formally the properties that a MAS is expected to satisfy.

1. Let us to denote the system state space as $\Sigma$. Then the main goal $G$ that the system aims at accomplishing can be associated with a specific predicate over $\Sigma$:

   $G : \Sigma \to BOOL$.

   In other words, the system goal is reached in a particular state $\sigma$ if and only if $G(\sigma) = TRUE$.

2. The system goal $G$ can usually be decomposed into a set of subgoals $SG_i$, where $i \in 1..n$. We suppose that there exists a precise relationships, $Expr$, between reachability of the main goal and that of the subgoals such that:

   $G(\sigma) = TRUE \Leftrightarrow Expr(SG_1(\sigma),...,SG_n(\sigma)) = TRUE$.

3. We assume that the system is stable with respect to its goals (subgoals), i.e., once a particular goal (subgoal) is reached, it stays reached. In B models, this property can be formulated as an invariant (using auxiliary variables to refer to the relevant part of the previous system state $\sigma_{prev}$) of the form:

   $G(\sigma_{prev}) = TRUE \Rightarrow G(\sigma) = TRUE$.

4. In multi-agent systems, (sub)goals are usually achieved by system agents. Often a specific (sub)goal should be accomplished only by a particular subset of agents. We call such agents eligible. Formally, for each subgoal $SG_i$, we define a eligibility function, $SG_i.Elig$:

   $SG_i.Elig : AGENTS \times \Sigma \to BOOL$,

   where $AGENTS$ denotes a set of all the system agents. In practice, such a function often checks whether a particular agent belongs to a specific class of agents responsible for achieving the subgoal. Moreover, it also determines whether the agent is able to perform the required task, i.e., it has not failed.

5. Since MAS are distributed, we assume that the knowledge about the (sub)goal reachability is shared among the agents. In other words, each agent has its own local copy of it. We model this by a family of functions $Agent_{SG_i}$, where $i \in 1..n$:

   $Agent_{SG_i} : AGENTS \times \Sigma \to BOOL$.

   The local and global knowledge must be consistent, i.e.,

   $SG_i(\sigma) = FALSE \Rightarrow \forall a \in AGENTS. Agent_{SG_i}(a,\sigma) = FALSE$.  \hspace{1cm} (1)

   In practice, it means that the information about reaching a particular subgoal by an agent should be broadcasted to the other agents.
6. The essential property of the considered MAS is eventual reachability of its main goal. In B models, such reachability is typically abstractly modelled by a single event reaching the desired system state. The event is then refined by the group of events terminating in the desired state. To prove termination, the natural number expression, variant, should be defined and shown to be decreased by the refined events. We assume that there exists a variant expression $V_i, V_i \in \Sigma \rightarrow \text{NAT}$, for each subgoal $SG_i$ of the system.

Since system agents may fail before reaching the assigned (sub)goal, to prove eventual goal reachability, we need to introduce various agent cooperative recovery scenarios that allow the active agents to take over the failed ones. We will consider several such scenarios later in this paper.

To exemplify a goal-oriented development of MAS, next we present our case study – a multi-robotic system. We start by informally defining the system requirements. Then we demonstrate how to formally develop such a system in Event-B and prove its essential properties.

3.2 A Case Study: A Multi-Robotic System

The goal of the multi-robotic system is to get a certain territory cleaned by the robots. The territory is divided into several zones, which in turn are further divided into a number of sectors. Each zone has a base station that coordinates the cleaning activities within the zone. In general, one base station might coordinate several zones. In its turn, each base station supervises a number of robots attached to it by assigning cleaning tasks to them.

A robot is an autonomous electro-mechanical device that can move and clean. A base station may assign a robot a specific sector to clean. Upon receiving the task, the robot autonomously moves to this sector and performs cleaning. After successfully completing its mission, the robot returns back to the base station to receive a new task. The base station keeps track of the cleaned and non-cleaned sectors. Moreover, the base stations periodically exchange the information about their cleaned sectors.

While performing the given assignment, a robot may fail. Subsequently it leads to a failure to clean the assigned sector. We assume that a base station is able to detect all the failed robots attached to it. In case of a robot failure, the base station may assign another active robot to perform the failed task.

A base station might fail as well. We assume that a failure of a base station can be detected by the others stations. In that case, the healthy base stations redistribute control over the zones and robots coordinated by the failed station.

Let us now formulate the main requirements and properties associated with the multi-robotic system that is informally described above.

(PR1) The main system goal: the whole territory has to be cleaned.

(PR2) To clean the territory, every its zone has to be cleaned.

(PR3) To clean a zone, every its sector has to be cleaned.

(PR4) Every cleaned sector (zone) remains cleaned during the system execution.

(PR5) No two robots should clean the same sector. In other words, a robot gets only non-assigned and non-cleaned sectors to clean.
(PR6) The information about the cleaned sectors stored in any base station has to be consistent with the current state of the territory. More specifically, if a base station considers a particular sector in some zone to be cleaned, then this sector is marked as cleaned in the memory of the base station responsible for it. Also, if a sector is marked as non-cleaned in the memory of the base station responsible for it, then any base station sees it as non-cleaned.

(PR7) Base station cooperation: if a base station has been detected as failed then some base station will take the responsibility for all the zones and robots of the failed base station.

(PR8) Base station cooperation: if a base station has no more active robots, a group of robot is sent to this base station from another base station.

(PR9) Base station cooperation: if a base station has cleaned all its zones, its active robots may be reallocated under control of another base station.

The last three requirements essentially describe the cooperative recovery mechanisms that we assume to be present in the described multi-robot system.

3.3 Formal Development Strategy

In the next section we will present a formal Event-B development of the described multi-system robotic system. We demonstrate how to specify and verify the given properties (PR1)–(PR9). Let us now give a short overview of this development and highlight formal techniques used to ensure the proposed properties.

We start with a very abstract model, essentially representing the system behaviour as a process iteratively trying to achieve the main goal (PR1). The next couple of data refinement steps decompose the main goal into a set of subgoals, i.e., reformulate it in terms of zones and sectors. We will define the gluing invariants establishing a formal relationship between goals and the corresponding subgoals. Thus, we will define a relation $Expr$, described in Section 3.1.

While the specification remains highly abstract, we postpone the proof of goal reachability property by defining the corresponding events as anticipated. Once, as a result of the refinement process, the model becomes sufficiently detailed, we change the event statuses into convergent and prove their termination. To achieve this, we need to define a variant – a natural number expression – and show that the execution of any of these events decreases it.

Next we introduce the agent types – base stations and robots. The base stations coordinate execution of the tasks required to achieve the corresponding subgoal, while the robots execute the tasks allocated to them. We formally define the relationships between different types of agents, as well as agents and respective subgoals. These relationships are specified and proved as invariant properties of the model.

The consequent refinement steps explicitly introduce agent failures, the information exchange as well as cooperation activities between the agents. The integrity between the local and the global information stored within base stations is again formulated and proved as model invariant properties.

We assume that communication between the base stations as well as the robots and the base stations is reliable. In other words, messages are always
transmitted correctly without any loss or errors. The main focus of our development is on specifying and verifying the cooperative recovery mechanisms.

4 Development of a Multi-Robotic System in Event-B

4.1 Modelling system goals and subgoals

Abstract model. Our initial model abstractly represents the behaviour of the described multi-robotic system. We aim at ensuring the property (PR1). We define a variable \( \text{goal} \in \text{STATE} \) that models the current state of the system goal, where \( \text{STATE} = \{\text{incompl}, \text{compl}\} \). In the process of achieving the goal, modeled by the event \( \text{Body} \), the variable \( \text{goal} \) may eventually change its value from \( \text{incompl} \) to \( \text{compl} \). The value \( \text{compl} \) corresponds to the situation when the goal is achieved, i.e., the whole territory is cleaned. The system continues its execution until the whole territory is not cleaned, i.e., while \( \text{goal} \) stays \( \text{incompl} \).

\[
\text{Body} \equiv \begin{cases} \text{status anticipated when goal} \neq \text{compl then goal } \in \text{STATE} \end{cases}
\]

First refinement. In our first refinement step we elaborate on the process of cleaning the territory. Specifically, we assume that the whole territory is divided into \( n \) zones, where \( n \in \mathbb{N} \) and \( n \geq 1 \), and aim at ensuring the property (PR2). We augment our model with a representation of subgoals. We associate the notion of a subgoal with the process of cleaning a particular zone. A subgoal is achieved only when the corresponding zone is cleaned. A new variable \( \text{zones} \) represents the current subgoal status for every zone: \( \text{zones} \in 1..n \rightarrow \text{STATE} \).

To establish the relationship between goal and subgoals and formalise the property (PR2) per se, we formulate the gluing invariant:

\[
\text{goal} = \text{compl} \Leftrightarrow \text{zones}[1..n] = \{\text{compl}\}.
\]

The invariant can be understood as follows: the territory is considered to be cleaned if and only if its every zone is cleaned. In this case, the \( \text{Expr} \), defined in the Section 3, becomes a conjunction of the subgoals. To model cleaning of a zone(s), we refine the abstract event \( \text{Body} \). We model it in such a way that, while a certain subgoal is reached, it stays reached. Hence we ensure the property (PR4).

Second refinement. Next we further decompose system subgoals into a set of subsubgoals. We assume that each zone in our system is divided into \( k \) sectors, where \( k \in \mathbb{N} \) and \( k \geq 1 \), and aim at formalising the property (PR3). We establish the relationship between the notion of a subsubgoal (or simply a task) and the process of cleaning a particular sector. A task is completed when the corresponding sector is cleaned. A new variable \( \text{territory} \) represents the current status of each sector:

\[
\text{territory} \in 1..n \rightarrow (1..k \rightarrow \text{STATE}).
\]

The following gluing invariant expresses the relationship between subgoals and subsubgoals (tasks) and correspondingly ensures the property (PR3):

\[
\forall j \in 1..n \Rightarrow \text{zones}(j) = \text{compl} \Leftrightarrow \text{territory}(j)[1..k] = \{\text{compl}\}.
\]

The invariant says that a zone is cleaned if and only if each of its sectors is cleaned.

The refined event \( \text{Body} \) is now models cleaning of a previously non-cleaned sector:
Let us observe that the event \textit{Body} also preserves the property (PR4).

4.2 Introducing Agents

In the third refinement step we augment our model with a representation of agents. In the model context, we define the abstract finite set \textit{AGENTS} and its disjointed non-empty subsets \textit{RB} and \textit{BS} that represent the robots and the base stations respectively. To define a relationship between a zone and its supervising base station, we introduce the variable \textit{responsible}:

\[
\text{responsible} \in 1 \ldots n \rightarrow \text{BS}.
\]

Each active robot is supervised by a certain base station. We model this relationship between robots and their supervised station by a variable \textit{attached}, defined as a partial function:

\[
\text{attached} \in \text{RB} \rightarrow \text{BS}.
\]

To coordinate the cleaning process, a base station stores the information about its own cleaned sectors and updates the information about the status of the other cleaned sectors. We assume that each base station has a “map” – the knowledge about all sectors of the whole territory. To model this, we introduce a new variable \textit{local\_map}:

\[
\text{local\_map} \in \text{BS} \rightarrow (1 \ldots n \rightarrow (1 \ldots k \rightarrow \text{STATE})).
\]

The abstract variable \textit{territory} represents the global knowledge on the whole territory. For any sector and zone, this global knowledge has to be consistent with the information stored by the base stations. In particular, if in the local knowledge of any base station a sector is marked as cleaned, then it should be cleaned according to the global knowledge as well. To establish those relationships, we formulate and prove the following invariant:

\[
\forall bs, z, s \cdot bs \in \text{ran}(\text{responsible}) \land z \in 1 \ldots n \land s \in 1 \ldots k \Rightarrow
\text{territory}(z)(s) = \text{incompl} \Rightarrow \text{local\_map}(bs)(z)(s) = \text{incompl}.
\]

For each base station, the local information about its zones and sectors always coincides with the global knowledge about the corresponding zones and sectors:

\[
\forall bs, z, s \cdot bs \in \text{ran}(\text{responsible}) \land z \in 1 \ldots n \land \text{responsible}(z) = bs \land s \in 1 \ldots k \Rightarrow
\text{territory}(z)(s) = \text{incompl} \Leftrightarrow \text{local\_map}(bs)(z)(s) = \text{incompl}.
\]

All together, these three invariants formalise the property (PR6). It easy to see that these invariants are special cases of the property (1), formulated in the Section 3.

A base station assigns a cleaning task to its attached robots. Here, we have to ensure the property (PR5) – no two robots can clean the certain sector at the same time. We introduce a number of new variables and an event \textit{NewTask} to model this behaviour.

The robot failures have some impact on execution of the cleaning process. The task cannot be performed if the robot assigned for it has failed. To reflect this behaviour, we refine the event \textit{Body} by two events \textit{TaskSuccess} and \textit{TaskFailure}, which respectively model successful and unsuccessful execution of the task.

\[
\text{Body} \equiv \text{refines Body status anticipated}
\]

\[
\text{any } z, s, \text{result}
\]

\[
\text{when } z \in 1 \ldots n \land s \in 1 \ldots k \land \text{territory}(z)(s) \neq \text{compl} \land \text{result} \in \text{STATE}
\]

\[
\text{then territory}(z) := \text{territory}(z) \uplus \{s \mapsto \text{result}\}
\]
At this refinement step, we are ready to demonstrate that the events TaskSuccess and TaskFailure converge. To prove it, we define the variant expression over system variables, counter + card(dom(assigned)), and prove that it is decreased by new events. An auxiliary variable counter stores the number of all non-cleaned sectors of the whole territory, see [8] for details.

A base station keeps track of the cleaned and non-cleaned sectors and repeatedly receives the information from the other base stations about their cleaned sectors. The knowledge is inaccurate for the period when the information is sent but not yet received. In this refinement step, we abstractly model receiving the information by a base station. We introduce a new event UpdateMap to model updating of the local map of a base station.

In this refinement step we also introduce an abstract representation of the base station cooperation defined by the property (PR7). Namely, we allow to reassign a group of robots from one base station to another. We define such a behaviour in the event ReassignRB. In the next refinement steps we will elaborate on this event and define the conditions under which this behaviour takes place.

Additionally, we model a possible redistribution between the base stations their pre-assigned responsibility for zones and robots. This behaviour is defined in the new event GetAdditionalResponsibility presented below. The guard of the event defines the conditions when such a change is allowed. A base station can take the responsibility for a set of new zones if it has the accurate knowledge about these zones, i.e., the information about their cleaned and non-cleaned sectors.

**4.3 Modelling of Broadcasting**

In next, fourth refinement step we aim at defining an abstract model of broadcasting. After receiving a notification from a robot about successful cleaning the assigned sector, a base station updates its local map and broadcasts the message about the cleaned sector to the other base stations. In its turn, upon receiving the message, each base station correspondingly updates its own local map. A new relational variable msg models the message broadcasting buffer:

\[ msg \in BS \leftrightarrow (1 \ldots n \times 1 \ldots k). \]
If a message \((bs \mapsto (z \mapsto s))\) belongs to this buffer then the sector \(s\) from the zone \(z\) has been cleaned. The first element of the message, \(bs\), determines to which base station the message is sent. If there are no messages in the \(msg\) buffer for any particular base station then the local map of this base station is accurate, i.e., it coincides with the global knowledge about the territory:

\[
\forall bs, z, s \in 1..n \land s \in 1..k \land bs \in ran(responsible) \land (bs \mapsto (z \mapsto s)) \notin msg \Rightarrow
\]

\[
\text{territory}(z)(s) = local\_map(bs)(z)(s),
\]

\[
\forall bs \in ran(responsible) \land bs \notin dom(msg) \Rightarrow
\]

\[
(\forall z, s \in 1..n \land s \in 1..k \Rightarrow \text{territory}(z)(s) = local\_map(bs)(z)(s)).
\]

After receiving a notification about successful cleaning of a sector, a base station marks this sector as cleaned in its local map and then broadcasts the message about it to other base stations. To model this, we refine the abstract events \(\text{TaskSuccess}\) and \(\text{UpdateMap}\).

### 4.4 Introducing Robot and Base Station Failures

**Fifth refinement.** Now we aim at modelling possible robot failures. We elaborate on the abstract events concerning robot and zone reassigning. We start by partitioning the robots into active and failed ones. The current set of all active robots is defined by a new variable \(active\). Initially all robots are active, i.e., \(active = RB\). A new event \(\text{RobotFailure}\) models possible robot failures that can happen at any time during system execution. We make an assumption that the last active robot can not fail and add the corresponding guard \(\text{card}(active) > 1\) to the event \(\text{RobotFailure}\) to restrict possible robot failures. In practice, the constraint to have at least one operational agent associated with our model can be validated by probabilistic modelling of goal reachability, which is planned as a future work. Let us also note that for multi-robotic systems with many homogeneous agents this constraint is usually satisfied.

A base station monitors all its robots and detects the failed ones. The abstract event \(\text{TaskFailure}\) abstractly models such robot detection.

To formalise the property \((\text{PR8})\), we should model a situation when some base station does not have active robots anymore. In that case, some group of active robots has to be sent to this base station from another base station. This behaviour is modelled by the event \(\text{ReassignNewBStoRBs}\) that refines the abstract event \(\text{ReassignRB}\):

\[
\text{ReassignNewBStoRBs} \equiv \text{refines ReassignRB}
\]

\[
\text{any} \ bs_1, bs_2, rbs
\]

\[
\text{when} \ bs_1 \in BS \land bs_2 \in BS \land rbs \subseteq active \land \text{ran}(rbs \subseteq attached) = \{bs\} \land rbs \neq \emptyset \land \text{ran}(rbs \subseteq assign_z) = \{0\} \land bs_2 \in ran(responsible) \land bs_1 \in ran(responsible) \land
\]

\[
bs_2 \neq bs_1 \land bs_1 \in \text{ran}(rbs \subseteq attached) \land \text{dom}(attached \uparrow (bs_1)) = 1 \land active
\]

\[
\text{then} \ attached := attached \uparrow (rbs \times \{bs_2\}) \end
\]

This event can be further refined by a concrete procedure to choose a particular base station that will share its robots (e.g., based on load balancing).

Moreover, to ensure the property \((\text{PR9})\), we consider the situation when all the sectors for which a base station is responsible are cleaned. In that case, all the active robots of the base station may be sent to some other base station that still has some unfinished cleaning to coordinate. This functionality is specified by the event \(\text{SendRobotsToBS}\) (a refinement of the event \(\text{ReassignRB}\)).
Sixth refinement. In the final refinement step presented in the paper, we aim at specifying the base station failures. Each base station might be either operating or failed. We introduce a new variable $operating \subseteq BS$ to define the set of all operating base stations. We also introduce a new event $BaseStationFailure$ to model a possible base station failure. We again make an assumption that the last active base station can not fail.

In the fourth refinement step we modelled by the event $GetAdditionalResponsibility$ that a base station can take over the responsibility for the robots and zones of another base station. Now we can refine this event by introducing an additional condition – only if a base station is detected as failed, another base station can take over its responsibility for the respective zones and robots:

\[
\text{GetAdditionalResponsibility} \equiv \text{refines GetAdditionalResponsibility}
\]
\[
\text{any } bs_i, bs_j, za, rbs
\]
\[
\text{when } bs_i \in BS \land bs_j \in \text{operating} \land zs = \text{dom}(\text{responsible}(bs_i)) \land bs_i \neq bs_j \land rbs \subseteq \text{active} \land rbs = \text{dom}(\text{attached}(bs_j)) \land bs_j \notin \text{dom}(\text{msg}) \land bs_j \notin \text{operating}
\]
\[
\text{then responsible} := \text{responsible} - (zs \times \{bs_j\}) \parallel \text{attached} := \text{attached} - (rbs \times \{bs_j\}) \parallel\]
\[
\text{asgn}_s := \text{asgn}_s - (rbs \times \{0\}) \parallel \text{asgn}_z := \text{asgn}_z - (rbs \times \{0\}) \parallel \text{local_map}(bs_i) := \emptyset
\]

As a result of the presented refinement chain, we arrived at a centralised model of the multi-robotic system. We can further refine the system to derive its distributed implementation, relying on the modularisation extension of Event-B to achieve this.

To verify correctness of the models we discharged more than 230 proof obligations. Around 80% of them have been proved automatically by the Rodin platform and the rest have been proved manually in the Rodin interactive proving environment.

5 Conclusions and Related Work

Formal modelling of MAS has been undertaken in [12, 11]. The authors have proposed an extension of the Unity framework to explicitly define such concepts as mobility and context-awareness. Our modelling have pursued a different goal – we have aimed at formally guaranteeing that the specified agent behaviour achieves the pre-defined goals.

Formal modelling of fault tolerant MAS in Event-B has been also undertaken by Ball and Butler [2]. They have proposed a number of informally described patterns that allow the designers to incorporate well-known (static) fault tolerance mechanisms into formal models. In our approach, we have implemented a more advanced fault tolerance scheme that relies on goal reallocation and dynamic reconfiguration to guarantee goal reachability.

The foundational work on goal-oriented development has been done by van Lamsweerde [5]. The original motivation behind the goal-oriented development was to structure the system requirements and derive properties in the form of temporal logic formulas. Over the last decade, the goal-oriented approach has received several extensions that allow the designers to link it with formal modelling [6, 7, 9]. These works aimed at expressing temporal logic properties in Event-B. In our work, we have relied on goals to facilitate structuring of the system behaviour and derived a detailed system model that satisfies the desired properties by refinement.
The theoretical aspects of modelling reachability has been studied in [3]. A work similar to our but in the context of discovering a distributed topology is presented in [4]. In our work, reasoning about liveness property has been put in the context of goal-oriented development.

In this paper we have presented an approach to formal development of a fault tolerant MAS with cooperative error recovery by refinement in Event-B. The formal development has allowed us to uncover missing requirements and rigorously define the relationships between agents. It has also facilitated a systematic derivation of a complex mechanism for cooperative error recovery.

Our approach has demonstrated a number of advantages comparing to various process-algebraic approaches used for modelling MAS. We relied on a proof-based verification that allowed us to derive a quite complex model of the behaviour of a multi-agent robotic system. We did not need to impose restrictions on the size of the model, number of agents etc. We could comfortably express intricate relationships between the system goals and the employed agents. Therefore, we believe that Event-B and the associated tool set will provide a suitable framework for formal modelling of complex MAS.

References

Building on the DEPLOY Legacy:  
Code Generation and Simulation

A. Edmunds and J. Colley and M. Butler

1University of Southampton, UK
ae2@ecs.soton.ac.uk

Abstract. The RODIN, and DEPLOY projects laid solid foundations for further theoretical, and practical (methodological and tooling) advances with Event-B. Our current interest is the co-simulation of cyber-physical systems using Event-B. Using this approach we aim to simulate various features of the environment separately, in order to exercise deployable code. This paper has two contributions, the first is the extension of the code generation work of DEPLOY, where we add the ability to generate code from Event-B state-machine diagrams. The second describes how we may use code, generated from state-machines, to simulate the environment, and simulate concurrently executing state-machines, in a single task. We show how we can instrument the code to guide the simulation, by controlling the relative rate that non-deterministic transitions are traversed in the simulation.

1 Introduction

This paper describes activities undertaken during the early part of the ADVANCE [8] project. Building on the RODIN, and DEPLOY projects [9], we are working to better understand the issues arising in a development when modelling with Event-B, and animating with ProB, in tandem with a multi-simulation strategy. Some of DEPLOY’s industrial partners were interested in the formal development of multi-tasking, embedded control systems. We developed an approach for automatically generating code from Event-B models, for these types of systems [3]. In this paper we also present an extension to this work.

Event-B uses set-theory, predicate logic and refinement to model discrete systems. The basic structural elements of Event-B models are contexts and machines. Contexts describe the static aspects of a system, using sets, constants, and axioms. The contents of a Context can be made visible to a machine. Machines describe the dynamic aspects of a system, in the form of state variables, and guarded events, which update state. Required properties are specified using the invariants clause. The invariants give rise to proof obligations.

In the remainder of this section, we describe the Event-B representation of state-machines, and Tasking Event-B, our existing code-generation approach. We introduce a case study in Section 2. In Section 3 we describe the new code generation feature for Event-B state-machine diagrams. In Section 4 we describe
how we use a single task, generated from state-machines, to simulate the environment and concurrently executing state-machines. We show how we guide the simulation, using additional guards on the transition implementations, to control the relative rate that non-deterministic transitions are traversed. We conclude with Section 5.

State-machine diagrams can be added to a machine. Each contains an initial state, typically contains one or more transitions, one or more other states, and possibly a final state. A transition 'elaborates' one or more events; that is, a transition describes the atomic state updates that occur during the change from one state to the next. We use an example of an automotive engine stop-start controller, loosely based on [5], to illustrate our approach. The system aims to save fuel by switching the engine off when the car is stationary. Fig. 1 is an example of a state-machine diagram, EngMode. Initially the state-machine is in the ENG_OFF state, and may go the ENG_CRANKING state via transitions s1 or userStart, and so on. In the properties we define 'translation type' as Enumeration. The underlying Event-B model, uses a set-partition of the states, as shown below. The current state of the state-machine is recorded in a variable EngMode ∈ EngMode STATES, where EngMode STATES is a partition of the states of the EngMode state-machine,

\[
\text{partition}(\text{EngMode STATES}, \{\text{ENG\_STOPPING}\}, \{\text{ENG\_CRANKING}\}, \{\text{ENG\_RUNNING}\}, \{\text{ENG\_OFF}\})
\] (1)

1.1 Tasking Event-B

Tasking Event-B is an extension to Event-B; where Event-B elements are restricted to implementable types. If required we use decomposition to separate the system into sub-components. At an appropriate stage we introduce implementation specific constructs to guide code generation. These constructs are underpinned by Event-B operational semantics; Tasking Event-B introduces three main constructs: AutoTask, Environ, and Shared Machines. AutoTask Machines model controller tasks (in the implementation). Environ Machines model
the environment, and Shared Machines provide a protected resource for sharing data between tasks.

Tasks bodies are specified using the syntax shown in Fig. 2. We can use (:) sequence, (if-elsif-else) branching, (do) looping, and text output to the console.

```
TaskBody ::= 
  TaskBody ; TaskBody 
  | if EventName 
    (elseif EventName)* 
     else EventName 
  | while EventName 
  | output String VariableName 

EventName ::= String 
VariableName ::= String
```

Fig. 2. Task Body Syntax

1.2 Translation of a Task Body

To simplify the discussion, our example uses a single tasking approach. We will not consider here the issue of multi-tasking. We therefore need only to give a brief overview of AutoTask Machine translation, since it will not be synchronized with a Shared Machine. Given an event \( E \equiv g \rightarrow a \), we map action \( a \) to a program statement \( a' \), and guard \( g \) to a condition \( g' \), if \( g \) exists. The guard should be \( \top \) for events used in sequences, but may be any implementable predicate for use in branching and looping statements. An example translation of branching follows, where events \( e_1 \equiv g_1 \rightarrow a_1 \) and \( e_2 \equiv g_2 \rightarrow a_2 \), are used in the task body,

\[
\text{if } e_1 \text{ else } e_2 \text{ endif} \\
\sim \text{ if } g'_1 \text{ then } a'_1 \text{ else } a'_2 \text{ end if;}
\]

The branching construct of the task body contains events \( e_1 \) and \( e_2 \), and translates to a branching construct in the program code. The guard \( g'_2 \) does not appear in the code, but to ensure that the modeller's intentions are correctly implemented a proof obligation can be generated to ensure that \( g_2 \leftrightarrow \neg g_1 \). The tool could be augmented to generate proof obligations automatically, to show that branch guards are disjoint and complete.

2 The Automotive Stop-Start Model

A typical approach to multi-tasking in hybrid systems, relies on a write-read-process protocol. The shared variable store, shown in Fig. 3 is used by the various modules; to write to, and then read from. In such a system, each task keeps a local copy of the parts of the state that it needs to deal with. In the
write-read-process protocol, all tasks write to the store, all tasks then read from the store. Only when all tasks have updated their local copies of shared state, can processing take place. The task iterates these steps in a loop. In our tool we simulate the concurrent implementation using sequential code generated from a single AutoTask Machine. The deployable modules of Fig. [3] can be implemented in a multi-tasking environment if the execution order of the protocol is preserved.

In our sequential simulation, we use a single AutoTask Machine, which contains both controller and environment state-machines; and define write and read behaviour in the machine’s task-body construct. We have already seen the Stop-Start (SSE) system’s EngMode state-machine, in Fig. [1]. In addition to this we have Clutch, Gear and Steering environment state-machines. There are three controller modules, the SSE Module that decides whether to issue stop or start commands based on the engine state, and values determined by the HMI Controls module. HMI Controls monitors the clutch, gear, and steering controls to see if automatic stop or start should be enabled.

![Fig. 3. Overview of the Stop-Start Architecture](image)

### 2.1 The Task-body

We have defined the state-machines of the system and we can now specify the IO between the modules via the shared variable store. The store contains a copy of all of the variables involved in IO between modules. Each of the modules may send data and receive data from the variable store. If we take, as an example, the engine’s IO, we output the engine state and speed to the shared variable store. All variables in the store are prefixed ‘STO’, and variables in the engine module (other than state names) are prefixed ‘ENG’, so the following event updates the shared variable store’s copies of the engine state. Each state-machine has a send (write) and receive (read) event which has the state-machine name and send or recv as a suffix.

\[
\begin{align*}
\text{Eng}._\text{send} & \triangleq \text{STO}_\text{EngMode} := \text{EngMode} \\
& \parallel \text{STO}_\text{EngineSpeed} := \text{ENG}_\text{EngineSpeed}
\end{align*}
\]
2.2 Modelling Starting and Stopping the Engine

The EngMode state-machine keeps track of the engine mode, i.e. off, running, cranking, or stopping. The engine is initially in the ENG_OFF state. We model the ultimate task of the SSE system, the automatic engine start, with the $s1$ event. This is enabled after receiving an engine start order from the Stop-Start Controller module (the SSE Module’s SSEMode state-machine, introduced later). The $s1$ event follows,

$$s1 \triangleq \text{when } \text{EngMode} = \text{ENG_OFF} \land \text{ENG_Start_Order} = \text{TRUE} \quad \text{then } \text{EngMode} := \text{ENG_CRANKING}$$

The predicate and action involving EngMode are generated automatically in the translation from the state-machine diagram. The guard with ENG_Start_Order is added by the developer to indicate that the engine should enter the cranking state when a Start Order has been received. The engine may also be started manually, as modelled by the userStart event. When the engine is running at a sufficient rate $s3$ sets the engine state to ENG_RUNNING,

$$s3 \triangleq \text{when } \text{EngMode} = \text{ENG_CRANKING} \land \text{Eng_EngineSpeed} >= \text{Eng_Idle_Speed} \quad \text{then } \text{EngMode} := \text{ENG_RUNNING}$$

When the engine is running, it can be stopped automatically by the SSE module. The HMI_CoIsControls module checks to see if it is in neutral gear, steering not-used, and clutch released. If it is, HMI_Stop_EnaT sets HMI_Stop_Ena to true. This is eventually passed to the SSEMode module via the shared store.

$$\text{HMI_Stop_EnaT} \triangleq \text{when } \text{HMI_Gear} = \text{NEUTRAL} \land \text{HMI_Steer} = \text{NOT_USED} \land \text{HMI_Clutch} = \text{RELEASED} \land \text{HMI_CoIsControlsSM} = \text{HMI_OPERATION} \quad \text{then } \text{HMI_Stop_Ena} := \text{TRUE} \parallel \text{HMI_Strt_Req} := \text{FALSE}$$

Event $t7$ elaborates a transition of the SSE state-machine diagram, setting SSE_Stop_Order and SSE_Start_Order. This is copied to the variable store, and then read by the engine module.

$$t7 \triangleq \text{when } \text{SSEMode} = \text{SSE_OPERATION} \land \text{SSE_Stop_Req} = \text{TRUE} \land \text{SSE_EngMode} = \text{ENG_RUNNING} \land \text{SSE_Stop_Ena} = \text{TRUE} \quad \text{then } \text{SSEMode} := \text{SSE_STOPPING} \parallel \text{SSE_Stop_Order} := \text{TRUE} \parallel \text{SSE_Start_Order} := \text{FALSE}$$
We specify the sequence of events in the Task Body in the ‘usual’ Tasking Event-B style, seen in Fig. 4. We have specified that send events occur before the read events. This is necessary to ensure the latest state is made available for the state-machine evaluation. The Task Body is periodic, and generates a loop in the implementation. The order of processing is as follows: 1) Initialisation of state. 2) Evaluate state-machines. 3) Send updated values to the variable store. 4) Read updated values from the variable store; then go to 2, and repeat. The sequence \{4, 2, 3\}, in the task body, corresponds to the read-process-write protocol, which follows initialisation (and initial sends to the variable store). Fig. 4 also shows the output clause, for text output to the console. The next section provides details of the translation to Ada code.

3 Translating State-Machines to Ada Code

To illustrate the translation process we show the Ada implementation, we have seen how state-machine states are modelled by an enumeration partition, and we use this in the implementation. The partition of Equation 1 is translated to the following Ada code.

```
package StopStart01b_Globals is
type EngMode_STATES is(
    ENG_STOPPING, ENG_CRANKING,
    ENG_RUNNING, ENG_OFF);

We create the package StopStart01b_Globals to store the global constants and types. The type EngMode_STATES is an enumeration of the state-machine states. Recall also, that we generate a state variable EngMode which is typed as EngMode \in EngMode_STATES, to keep track of the state; it has the initial value Eng.OFF. We use the diagram and the initialisation event to generate the following code:

EngMode : EngMode_STATES := Eng.OFF;
```
The main program invokes the state-machine implementations in a loop, once per cycle. Each state-machine diagram maps to a procedure. State-machine procedures are called exactly once before the sends to, and reads, from the variable store. The evaluation of each state-machine procedure is independent of the other state-machines, since each keeps a local copy of the state, copied from the variable store. Each state-machine procedure has a state variable $v$, states $w_i$, and implemented actions $a_i$. To each state-machine procedure, we add to a case statement,

\[
\begin{align*}
\text{case } v \text{ is when } w_1 &= a_1; \\
\text{when } w_2 &= a_2; \ldots \\
\text{when } w_n &= \text{null};
\end{align*}
\]

Translation of our example gives rise to the following code,

\[
\begin{align*}
\text{procedure EngModestateMachine is } & \\
\text{begin } & \\
\text{case EngMode is } & \\
\text{when } \text{ENG\_STOPPING} &= > & \\
\text{if } ((\text{ENG\_EngineSpeed} = 0)) & \text{ then } & \\
\text{EngMode} & := \text{ENG\_OFF}; & \quad \text{–– s5} \\
\text{elsif } ((\text{ENG\_Start\_Order} = \text{true})) & \text{ then } & \\
\text{EngMode} & := \text{ENG\_CRANKING}; & \quad \text{–– s6} \\
\text{else null; } & \quad & \\
\text{end if; } & \\
\text{when } \text{ENG\_CRANKING} &= > & \\
\text{if } ((\text{ENG\_EngineSpeed} = 0)) & \text{ then } & \\
\text{EngMode} & := \text{ENG\_OFF}; & \quad \text{–– s2} \\
\text{elsif } ((\text{ENG\_EngineSpeed} \geq \text{Eng\_Idle\_Speed})) & \text{ then } & \\
\text{EngMode} & := \text{ENG\_RUNNING}; & \quad \text{–– s3} \\
\text{else null; } & \quad & \\
\text{end if; } & \\
\text{when } \text{ENG\_RUNNING} &= > \ldots & \\
\text{end case; } & \\
\text{end EngModestateMachine; }
\end{align*}
\]

We can see that each of the case’s when statements contains a branching statement. This is because each state of the state-machine has at least two branches; a do-nothing transition, plus one or more outgoing transitions. The do-nothing transition is not explicitly shown on the diagram. A do-nothing transition can be added to each state, since adding a skip event is a valid refinement. It is implemented by the else null; branch. Other branches are translated from states with
more than one outgoing transition. This may be seen in the *ENG_STOPPING* case in the example. The branch conditions are mapped from the guards of the events (*s5* and *s6*) that elaborate the outgoing transitions.

4 Manipulating State Machine Transitions

The generated code from our example is compiled to an executable file and run. We have implementable code for the controller state-machines, and a simulation of the environment from the environment state-machines. When executing, we find that most of the state remains unexplored, and this is due to the non-determinism in the state-machines. This section identifies how we can guide a simulation, by reducing the non-determinism in the generated state-machine by modifying the branch conditions.

For the controller state-machines, each state’s outgoing transitions are disjoint and complete; in other words, a transition is always taken in the simulation. However, in the environment, it is unlikely that the clutch changes state so frequently. We do have the implicit *do-nothing* transition on environment state-machine states, but we need this to happen more often than the other transitions. We must have some control over the relative rate that non-deterministic transitions are traversed in the simulation. As it stands, any outgoing transition is equally likely to occur. To solve this in the simulation, we introduce an enabling variable \( q \in 0 \ldots n \) and a random variable \( r \in 0 \ldots n \), and use the random variable in a case-statement’s branch conditions. Variable \( q \) is calculated once at the beginning of the simulation, but a new random variable \( r \) is calculated at each state-machine evaluation. The event \( g \rightarrow a \) in Event-B terms is implemented as a branch \( g \wedge r = q \rightarrow a \) in a case-statement.

We now suggest how we may generate, and use the variables \( q \) and \( r \) in simulation. This aspect is work in progress, but we believe the approach will be useful for generating test scenarios, and therefore will help to improve test coverage. By adding a guard to the branch condition we can influence the path taken through the code during simulation. In effect, we reduce the non-determinism in the state-machines, which allows us to guide the simulation, and therefore the exploration of the state-space.

One question is, how to choose a value of \( n \)? We could base it on the total number of outgoing transitions of the state involved, but this would not give a large enough value. A typical state may have four transitions, plus a *do-nothing* transition, so a random number \( r \in 0 \ldots 4 \) could be used. However, we wish to manipulate the probability of a branch being taken, so that a branch is very unlikely to be taken; therefore, a much larger value for \( n \) is required. So, we calculate \( n \) based on the number of tests that would be required, for test coverage of all transitions, in all states. Likewise, the value of \( q \) must be unique within the case-statement; we just allocate an arbitrary, but unique value, close to \( n \). In future work we will investigate how we could modify \( n \) during simulation runs, and use this value to reduce the probability of a simulation traversing previously explored state. In the code fragment below, we add the probabilistic
condition to the branch of the case-statement, where \( r = \text{StartStop01b\_random} \) (\( \text{StartStop01b\_random} \) is a random variable in the implementation code) and \( q = 3990 \).

```python
    case EngMode is
      when ENG_STOPPING =>
        if ((ENG_EngineSpeed = 0)) and (StopStart01b\_random = 3990) then
          EngMode := ENG_OFF;
        
Adding the branch condition gives us control over the likelihood that a particular transition from a state will be taken when the state-machine is evaluated. We manually modify the conditions, to affect the behaviour of the simulation. We may wish to focus on exploring the state in a particular region. For instance, to test an engine-stop scenario, we require that the engine is in the ENG_RUNNING state, the gear is in NEUTRAL, the clutch is in the RELEASED state, and the steering NOT USED. Fig. 5 shows that we want large probabilities of transitions leading to the states that we want, and small ones departing.

For a given simulation run we can define attracting and repelling states. Here, ENG_RUNNING is an attracting state; that is, we want the state-machine to be in that state or moving towards it most of the time. To achieve this we can adjust the branch conditions, to increase the probability of the transitions that lead to that state, being taken. For instance to increase the probability of the engine going from ENG_OFF to ENG_CRANKING we can modify the statement to read (\( \text{StopStart01b\_random} < 3990 \)). In addition to this, we propose to record the navigated transitions, for transition coverage analysis. So, we will be able to use the data also, to guide the simulation. We show two simulation runs here, with the text output defined in the Task Body, Run 1 uses the ‘unmodified’, generated code; it simply loops and never reaches the ENG_RUNNING state. With the branch conditions modified, as described, Run 2 shows the simulation cycling from ENG_RUNNING to ENG_OFF; and with the indicator lamp changing to inform the driver of the situation.
5 Conclusions

We have shown how we generate Ada code from State-machines, and illustrated the approach with a case study based on an automotive engine controller, automatic stop-start system. We describe how we simulate the environment, and a multi-tasking implementation. We gain an insight into how we adjust the conditions to provide meaningful simulation runs, which should be useful in the ADVANCE project. In future work we intend to record the transition coverage, and feed this back to the simulator, to ensure all transitions are covered. We will also investigate the interaction between the generated code, environment simulations, and ProB.

References

9. The DEPLOY Project Team. Project Website. at http://www.deploy-project.eu/
Lessons Learned/Sharing the Experience of Developing a Metro System Case Study

Renato Silva
University of Southampton,
ras07r@ecs.soton.ac.uk

Abstract. In this document we share the experiences gained throughout the development of a metro system case study. The model is constructed in Event-B using its respective tool set, the Rodin platform. Starting from requirements, adding more details to the model in a stepwise manner through refinement, we identify some keys points and available plugins necessary for modelling large systems (requirement engineering, decomposition, generic instantiation, among others), which ones are lacking plus strengths and weaknesses of the tool.

Keywords: Event-B, Rodin, requirements, refinement, decomposition, generic instantiation

1 Introduction

Event-B [1] is a formal method that allows modelling and refinement of systems. From the experiences during DEPLOY1, there exists a natural instinct to model a system such that it mimics its implementation. That is not always the best approach: models should be used to understand the system and its behaviour; the implementation should be seen as an independent task. This document aims to guide modellers by describing the experiences gained throughout the development of a metro system case study, suggesting “rules of thumb”, modelling techniques and assessing the current tool support (Rodin platform [2]).

We build a metro system model in a “top-down” style, in Event-B based on safety properties, starting from an abstraction view of the system and gradually augmented it with more details. Generic instantiation [3,4,5] and decomposition [6] are techniques used in the case study, simplifying the formal development by reusing existing models and avoiding re-proofs. Some requirements are based on real ones for metro system carriage doors.

A brief overview of the Event-B language is given in Section 2. The construction of the metro system model is described in Section 3, including a discussion of the keys points for building of a formal model such as requirements, abstraction, refinement, proofs, decomposition, generic instantiation in the Rodin platform. We finish with conclusions and related work in Section 4.

1 DEPLOY - Industrial deployment of system engineering methods providing high dependability and productivity - supported by the EU Commission (Grant 214158)
2 Background

Event-B is a formal modelling method for developing correct-by-construction hardware and software systems. An Event-B specification is divided into two parts: a static part called context and a dynamic part called machine. A machine SEES as many contexts as desired. A context consists of sets, constants and assumptions (axioms) of the system. An Event-B model is a state transition system where the state corresponds to variables $v$ and transitions are represented by a collection of events $\text{evt}$ in machines. The most general form of an event is: $\text{evt} \triangleq \text{any } t \text{ where } G(t,v) \text{ then } S(t,v,v') \text{ end}$, where $t$ is a set of parameters, $G(t,v)$ is the enabling condition (called guard) and $S(t,v,v')$ is a before-after predicate computing after state $v'$. Essential to Event-B is the formulation of invariants $I(v)$: safety conditions/properties to be preserved at all times. Proof obligations (PO) are generated for all system transitions to validate and ensure that these conditions are preserved. Because Event-B advocates the use of refinement, additional PO (forward refinement) [1] are generated to ensure that concrete refinements preserve the abstract models’ properties. The Event-B toolset is Rodin [2], result of an EU research project\(^2\): software tool, based on modern software programming tools created to help the development of specifications based on the idea that large complex or critical projects should start with modelling and reasoning about its specification.

3 Case study construction

In this section the steps followed throughout the construction of our model are described. The safety-critical metro system case study describes a formal approach for the development of embedded controllers for a metro\(^3\). Butler [7] makes a description of embedded controllers for a railway using classical B. Our starting point is based on that work but applied to a metro system. That work goes as far as our first decomposition. We augment it by refining sub-components, adding requirements and instantiating emergency and service doors in carriages.

3.1 Requirements

Requirements analysis [8] in systems engineering, encompasses tasks that go into determining the needs or conditions to meet for a new or altered product, taking account possible conflicting requirements of the various stakeholders, such as beneficiaries or users. There are several techniques to deal with requirements and they vary according to projects’ domains. Moreover guidelines [8] have been developed to achieve this goal. Nevertheless requirements are often described in an informal manner. Consequently it is hard to reason about each requirement: experienced people are able to detect contradictions and uncertainties but it is not

---

\(^2\) RODIN: Rigorous Open Development Environment for Open Systems (EU IST Proj)

\(^3\) The Event-B model built is available at http://eprints.ecs.soton.ac.uk/23135/
guaranteed that all will be uncovered. Moreover, within the formal methods do-
main, it is hard to trace informal requirements with the model/implementation.

Although not available when we developed this case study, a requirement plug-in (ProR [9,10]) now exists for the Rodin platform, supporting ReqIF 1.0.1 Standard. Benefits of ProR are incremental creation of hierarchical require-
ments structures from informal requirements or providing traceability between
requirements and formal models. Furthermore, the system description, mixing
general and informal artefacts may contain assumptions about the environment
or requirements properties and ProR can reason about them (possibly uncover-
ing contradictions and uncertainties).

Our metro system is characterised by trains, tracks circuits (also called sec-
tions or CDV and a communication entity (comms) that allows the interaction
between trains and tracks. The trains circulate in sections and before a train
enters or leaves a section, a permission notification must be received. In case of
hazard situations, trains receive braking notifications. Track is responsible for
controlling the sections, changing switch directions (switch is a special section
that connects different routes and can be either divergent or convergent) and
sending signalling messages to the communication entity. These are the main
requirements for this case study (some described in Fig. 1):

1. Route sections are all connected and cannot have empty gaps (inv1).
2. There are no loops in the route sections: sections cannot introduce loops
   (thm3). Moreover no circularity is allowed (via transitive closure: thm4).
3. Switches cannot be connected and can be either divergence or convergent.
4. Non-switches have at most one successor and at most one predecessor section.
5. Trains circulate in tracks (inv4, inv5, inv7), preserving transitive closure.
6. Trains occupy at least one section plus a safety distance (inv4).
7. Trains cannot be in the same section at the same time (trains crashed: inv13).
8. Comms handles messages exchanged between trains and tracks. Trains head-
    ing to an occupied section receive a negative access and braking message.
9. As part of the safety requirements, all trains have an emergency button.
10. While the emergency button is enabled, the train cannot speed up (braking).
11. If a train door is opened, then the train is stopped (in a platform or due to
    an emergency). In contrast, if the train is moving, then its doors are closed.

3.2 Abstraction

Following a “top-down” design, the development starts with an abstraction
model: description that encompasses the main aspects and goals the system
intends to answer, obstructing itself from the implementation and other details.
Getting a good abstraction is a very hard task requiring an accurate under-
standing of the system. Moreover the abstraction is the basis of the development
playing a crucial role in the entire model. A good abstraction is often not achieved
at first attempt even for experienced developers. It may change throughout the

---

development to fit additional requirements that came into play on a later stage or when, after a few refinements, it does not fit exactly as initially desired. No tools are available that help finding the right abstraction mainly because each system has its specific properties. It often relies on experience and empirical research. Nevertheless we believe that systems can be categorised according to some common properties, architecture and behaviours and therefore having a abstraction template repository could be helpful when starting a model development. Abstraction templates could then be customised according to specific needs. Unfortunately such repository does not yet exist, requiring further investigation beyond the scope of this paper.

For our abstraction model (Fig. 1), we focus on the main properties: tracks are divided into sections that are connected (Reqs. 1, 2, 3, 4); trains circulate in tracks (Req. 5); the most important (safety) global property introduced initially states that trains cannot be in the same section at the same time (Reqs. 6, 7).

---

3.3 Refinement

Refinement allows the construction of a model in a gradual way, making it closer to an implementation [3]. At same time, the overall correctness of the system is preserved. Our case study heavily uses refinement as seen in Fig. 2. At each refinement step, new requirements are introduced to the model and consequently new invariants, variables, events are introduced or refined. For instance, for refinement Train M1, the invariants and properties imposed are:

1. There is a limit to the number of carriages per train.
2. If a carriage alarm is activated, the train’s emergency button is also active.
3. The sum of carriage doors corresponds to the doors of a train.
4. Trains have states: maintenance, manual, automatic.
5. If a train is not in a maintenance state, then it must have the correct number of carriages and the leader carriage must be defined already.
6. If a train is in maintenance, then it must be stopped.
7. The emergency brake is activated if a train exceeds the maximum speed.

**Do it right at first/Recursion** As for abstraction, refinement steps are not reached at first attempt. They evolve, accommodate different requirements and also change, impacting previous refinements. And that comes with a cost: a change in the abstraction, affects all the following refinements and the adjustment to each refinement level has to be done manually, which is cumbersome. In our case, the emergency brake requirement (Req. 9) was only added after we had reached the first decomposition. The consequences propagated to the abstraction, impacting most events and manual re-proving (which delayed for a few days the progress achieved before). This is a limitation of the refinement process in the tool that does not propagate the changes, requiring improvements.

### 3.4 Proofs and model construction

Proofs play an important role in formal modelling, checking that properties and behaviours are preserved. There is always a compromise between representing a system, avoiding complex proofs and tool limitations. Despite the plug-ins available for automated proof solving (AtelierB provers [11], Relevance Filter[12]), complex proofs tend to be avoided. From our experience, a complex proof hard (but not impossible) to discharge, often means that the model is overcomplicated and may be rewritten/simplified. When building Train_M2, train doors were represented as \((DOOR,CARRIAGE; train\_carriage)^{-1}\), where \(DOOR,CARRIAGE \in DOOR\rightarrow CARRIAGE\) represents carriage doors and \(train\_carriage \in CARRIAGE\rightarrow trns\) represents the train carriages. Although that relation is enough to describe which doors are part of a train, from a proof viewpoint was very unsuccessful. By rewriting train doors as variable \(door\_train\_carriage = (DOOR,CARRIAGE; train\_carriage)^{-1}\) and invariants \(door\_train\_carriage \in trns \leftrightarrow DOOR,\) \(door\_train\_carriage^{-1} \in DOOR \rightarrow trns\), we solved the issue.

From a tool viewpoint, there is a direct relation between the number of PO per refinement and performance. Our criteria to choose which properties to add per refinement were directly related with the PO generated per refinement: if over 150 PO, additional properties were stated in new refinements. Train requirements are spread over 4 refinement steps for that reason. Improvements have been made in terms of tool performance in the latest releases but large developments (over 15 refinements and large number of events) are still affected.

### 3.5 Decomposition

The “top-down” style of development used in Event-B allows the introduction of new events and data-refinement of variables during refinement steps. A consequence of this development style is an increasing complexity of the refinement
process when dealing with many events and state variables. Model decomposition [6] addresses such complexity by cutting a large model into smaller components. Two methods have been identified for the Event-B decomposition and are supported by a Rodin plug-in [6]: shared variable [3] and shared event [13]. Because decomposition is monotonic [13], the generated sub-components can be further refined independently: sub-components can be used to further refined the original model or be used in other models. Moreover team development can be introduced: different developers can share parts of the same original model by working independently in parallel with the resulting decomposition sub-components. Decomposition also partition PO which are expected to be easier to discharge in sub-components. In our model, decomposition is used for following reasons: separation of aspects; model architectural decision; tool performance: building/proving is faster for separated models than for monolithics.

Decomposition is recursively used as seen in Fig. 2: splitting the initial monolithic model into three parts (Train, Middleware and Track) from an architectural point of view (separation of aspects); splitting Train_M4 into Leader-Carriage (due to the number of POs and separation of aspects) and Carriage and later on to decompose Carriage into CarriageInterface and CarriageDoor (Fig. 3(b)). Although we could have used either decomposition styles, we used the shared event style mainly because in that manner, we did not constrain the refinement of variables (like it happens for shared variables).

Unfortunately the decomposition process does not propagate modifications on the original machine and consequently, decomposed components need to be regenerated if the original component is modified. If the decomposed components have been refined, than the modifications need to be reflected in those refinements (notified via errors or PO being generated or requiring reproving). We believe that the decomposition tool requires improvements in terms of propagation changes to minimise the overall impact that is inevitable.

![Fig. 2. Overall view of the metro system development](image-url)
3.6 Generic Instantiation

Generic Instantiation can be seen as a way of reusing components and solving difficulties raised by the construction of large and complex models [3]. Generic developments (single machine or a chain of refinements) are reused, originating components with similar properties instead of starting from scratch. Reusability occurs via the instantiation and parameterisation of patterns. [4] proposes a generic instantiation approach for Event-B by instantiating machines. The goal is to reuse a pattern as an instance in an existing development (problem) consisting of a chain of refinement of machines \( S_0 \) to \( S_k \) (\( S \) stands for Specific problem) as seen in Fig. 3(a). The instance sees the parameterisation context \( C_{IG} \) (that

![Generic Instantiation Diagram](image)

(a) Instantiation of \( G_0 .. G_j \) via parameterisation context \( C_{IG} \) creating instance \( IG \) to fit problem \( S_0 .. S_k \).

(b) Carriage Refinement Diagram and Door Instantiation

Fig. 3.Generic Instantiation

extends the specific problem context \( C_S \) containing the replacement properties for the elements in context \( C_{Gi} \). Variables, events and parameters can be renamed to fit new or existing elements in the specific problem. The correctness of the instantiation relies on reusing the pattern PO and ensuring that assumptions in the context parameterisation are satisfied in the instance. In our case study, an existing development of carriage doors (\( GCDoor_M0 .. GCDoor_M2 \)) is used as a pattern with all the related PO previously discharged. The pattern is instantiated and parameterised accordingly into emergency doors and service doors (Fig. 3(b)). The main pattern requirements are:

1. Doors have a state associated: open (train must be stopped) or closed.
2. When adding/removing a carriage to a train, doors must be closed for safety.
3. Actions involving the doors may result from commands (open, close, isolate, remove_isolated) sent from the central door control.
4. Doors must be closed and locked before a train starts moving.
5. Doors are opened by the following devices: manual\_platform, manual\_internal or automatic\_central\_door.

6. Doors can get obstructed when closed automatically (people/object obstruction). If an obstruction is detected, a second attempt is made to close them.

7. Doors can be isolated in case of malfunction or for safety reasons.

8. If a door is obstructed, then it must be in a state corresponding to open.

These requirements are shared between both emergency and service doors highlighting the use of instantiation. Additional requirements for each kind of door can be added in further refinements (emergency doors are only available for emergencies, do not respond to standard open command, etc). For our case, the instantiation was manual. Nevertheless currently a generic instantiation prototype is available [5]. The tool needs to mature and requires improvements in terms of matching the pattern and the last refinement of problem. In this case study, the matching was manually achieved through decomposition.

**Animation/Model Checker and Code Generation** Although we are mainly interested in safety properties, ProB model checker [14] proved to be a very useful tool. At some stages, all PO were discharged but ProB showed that the system was deadlocked. In larger developments, these situations may occur frequently. Therefore we suggest safety properties preservation (via PO) and running ProB to confirm deadlock freeness. Another option, to be addressed by ADVANCE5 is to introduce liveness properties (e.g. enabledness). Regarding implementation, a code generation plug-in6 [15] (Event-B to Ada or C) is available.

**Statistics** Table 1 describes the statistics of the model in terms of variables, events and PO (including automatically discharged) for each refinement. Almost 3/4 of the PO were discharged automatically. The case study conditions

<table>
<thead>
<tr>
<th>Var</th>
<th>Events</th>
<th>PO/Auto</th>
</tr>
</thead>
<tbody>
<tr>
<td>TransitiveClosureCtx</td>
<td>10/10</td>
<td></td>
</tr>
<tr>
<td>MetroSystem_C0</td>
<td>5/3</td>
<td></td>
</tr>
<tr>
<td>MetroSystem_C1</td>
<td>7/10</td>
<td>7/64</td>
</tr>
<tr>
<td>MetroSystem_M0</td>
<td>13/17</td>
<td>17/17</td>
</tr>
<tr>
<td>MetroSystem_M1</td>
<td>12/17</td>
<td>78/57</td>
</tr>
<tr>
<td>MetroSystem_M2</td>
<td>12/17</td>
<td>24/22</td>
</tr>
<tr>
<td>Track</td>
<td>4/10</td>
<td>0/0</td>
</tr>
<tr>
<td>Train</td>
<td>7/14</td>
<td>0/0</td>
</tr>
<tr>
<td>Middleware</td>
<td>1/4</td>
<td>0/0</td>
</tr>
<tr>
<td>Train_M1</td>
<td>9/10</td>
<td>74/22</td>
</tr>
<tr>
<td>GCDoor_M0</td>
<td>10/17</td>
<td>117/132</td>
</tr>
<tr>
<td>GCDoor_M1</td>
<td>10/17</td>
<td>117/132</td>
</tr>
<tr>
<td>GCDoor_M2</td>
<td>13/21</td>
<td>155/79</td>
</tr>
<tr>
<td>Total</td>
<td>909/678</td>
<td>74.6%</td>
</tr>
</tbody>
</table>

Table 1. Statistics of the metro system case study


were the following: Rodin v2.1 (Auto Builder: OFF; Auto Prover: OFF), Model Decomposition v1.2.1 and Shared Event Composition plug-in v1.3.1. Generic instantiation was done manually (tool support was not available), ProB v2.1.2.

4 Related Work and Conclusions

From the experience of developing formal models involving a large number of refinements, development tools reach a saturation point where it is not possible to edit the model due to the high amount of resources required (or very slowly). Decomposition is a possible solution that alleviates the issue by splitting the model into tool manageable dimensions, separating concerns, decreasing the number of events and variables per sub-component which results in more manageable models. Generic instantiation reuses pattern and respective PO per instance.

The experience of modelling a metro system in Event-B using the Rodin platform and its plugins, is shared in terms of model design and assessment of available tools. Requirements are defined and modelled through refinement. As an architectural decision and to alleviate the problem of modelling a monolithic component, the model is decomposed several times. Benefiting from an existing development for carriage doors GCDoor, this pattern is used to instantiate two kind of carriage doors: service and emergency doors. The refinement of Carriage is decomposed, originating CarriageDoor that matches with pattern GCDoor_M0. Although the instantiation is similar for both cases, the resulting instances can be further refined independently. Generic instantiation minimises the proving effort reusing the pattern GCDoor PO (in the overall 257). Therefore we achieve our goal of reusing existing developments and discharging as little PO as possible. Even the interactive proofs were relatively easy to discharge once the correct tactic was discovered. This task would be more difficult without decomposition due to the elevated number of hypotheses to be considered. Nevertheless the effort of discharging PO could be further minimised by having an easy way to reuse PO tactics. A limitation of this model is not addressing liveness properties through proofs which would enrich the model.

Although we use Event-B, these techniques are generic enough to suit other formal notations and other case studies. Formal methods has been widely used to validate requirements of real systems. The systems are formally described and properties are checked to be preserved whenever a system transition occurs. Usually this result in complex models with several properties to be preserved, therefore structuring and reusability are pursued to facilitate the development. Lutz [16] describes the reuse of formal methods when analysing the requirements and designing the software between two spacecrafts’ formal models. Stepney et al. [17] propose patterns to be applied to formal methods in system engineering. Using the Z notation, several patterns (and anti-patterns) are identified and catalogued to fit particular kind of models. These patterns introduce structure to the models and aim to aid formal model developers to choose the best approach to model a system, using some examples. Although the patterns are expressed for Z, they are generic enough to be applied to other notations. Comparing with
the development of our case study, the instantiation of service and emergency
doors corresponds to the Z promotion, where a global system is specified in
terms of multiple instances of local states and operations. Although there is
not an explicit separation of local and global states in our case study, service
and emergency doors states are connected to the state of *CarriageDoor* and
we even use decomposition, instantiation and refactoring to fit into a specific
pattern. Stepney [17] suggests template support and architecture patterns to be
supported by tools. We agree and aim to address this issue in the future by
having categorised templates customised according to the modeller’s needs.

References

University Press (2010)
2. Rodin: RODIN project Homepage. [http://rodin.cs.ncl.ac.uk](http://rodin.cs.ncl.ac.uk) (September
3. Abrial, J.R., Hallerstede, S.: Refinement, Decomposition, and Instantiation of
Instantiation. In Breitman, K., Cavalcanti, A., eds.: Formal Methods and Software
Engineering. Volume 5885 of LNCS. Springer Berlin / Heidelberg, Rio de Janeiro,
Brazil (December 2009) 466–484
2009 Seventh IEEE International Conference on Software Engineering and Formal
Methods 0 (2009) 210–219
Software: Practice and Experience 41(2) (February 2011) 199–208
355–366
Requirements into Specifications. Submitted to Science of Computer Programming
(2012)
accessed 27-July-2010.
14. ProB: ProB. [http://www.stups.uni-duesseldorf.de/ProB/overview.php](http://www.stups.uni-duesseldorf.de/ProB/overview.php)
15. Edmunds, A., Butler, M.: Tasking Event-B: An Extension to Event-B for Gener-
17. Stepney, S., Polack, F., Toyn, L.: An Outline Pattern Language for Z. In: Proceed-
ings of the 3rd International Conference on Formal Specification and Development
Abstract. We show how the event-based notation offered by Event-B may be augmented by algorithmic modelling constructs without disrupting the refinement-based development process.

1 Introduction

One of the lessons of the DEPLOY project [5] is that the industrial application of formal modelling cannot fully succeed by employing just one notation, paradigm and methodology. In the case of Event-B [2], one of the language strong sides at the level of abstract design - a simple and versatile notation suitable for a wide range of abstractions - makes the language difficult to apply to concrete designs. Unstructured event-based models often become unwieldy long and verbose when design and implementation decisions are added.

In this paper we discuss a proposal to extend the event-based notation of Event-B with algorithmic constructs that permit an efficient specification of a large class of concrete designs.

Our extension, language SLP (sequential composition, loop, parallel composition), is a compact formal modelling notation with strictly defined syntax and semantics. To stay on the same technological platform as Event-B, we define the language semantics as a list of FOL verification conditions. We adopt without changes the mathematical language of Event-B - the part of the notation used to define predicates and expressions. The languages also borrows the notation and the atomicity assumption of Event-B substitutions.

Rather than a replacement or a simple superposition of algorithmic and event styles we propose to have a seamless connection between Event-B and SLP where a high-level event specification is gradually transformed into an algorithmic specification with explicit concurrency and control flow (see Fig. 1).

The defining difference between SLP and Event-B is that the latter is data-driven while the former features explicit control flow for sequential computation and units of concurrency for concurrent computations. This requires a departure from a flat machine structure, apt for inductive reasoning but often onerous in practice for large models, to a hierarchical model with nested naming scopes delineating verification concerns.

2 Syntax

An SLP model is made of the following three main parts. The first, taken verbatim from Event-B, provides definitions of types, axiom, variables, invariants
Fig. 1. The SLP approach promotes a gradual transition from an event-based to an algorithmic specification.

and theorems. This part may also contain Event-B events in the case of a mixed Event-B/SLP model.

The second part is the definition of environment activities. In SLP, we take a view that actions performed by an environment must be explicitly defined as such. This is not just a syntactic notion - SLP offers differing refinement rules (not discussed in this paper) for environment and system activities.

The final part is the definition of the behaviour of a modelled system. It takes the form of a list of so-called process definitions - concurrent units of system behaviour. The body of a process is defined by a succession of atomic state updates (substitutions, in the Event-B terminology) connected by the typical algorithmic control structures - sequential composition, if and loop. A process body runs in an infinite loop until it explicitly executes a termination command.

The processes of a system and environment activities execute concurrently. They interact by reading and writing shared (global) variables. A system process may also define its private (local) variables to deal with computations that do not need to be exposed to environment or other system processes. For a given process, the universe of the process is the set of all other processes and all the environments.

The following is the top-level structure of an SLP specification:

\[
slp := \langle invdef \rangle^* \\
\langle environment \rangle^* \\
\langle process \rangle^+ \\
\langle invdef \rangle := (invariant | theorem) \langle label \rangle : \langle predicate \rangle
\]

To simplify the presentation, we omit the declaration of constants and sets while variable declarations are deduced from invariants.\(^1\) All the variables defined at the global level are seen by system and environment processes. These should be the variables used to model input/output between the environment and system components. Like in Event-B, we split invariant conditions to label and partition invariant preservation conditions.

An environment is a labelled pair of a rely and guarantee predicates. Like invariants, rely and guarantees are labelled.

\(^1\) Note that this is our preferred concrete syntax. The abstract syntax for these elements is exactly that of Event-B
\[\text{environment} \ ::= \text{environment} \ \text{label} \ \text{reldef}^* \ \text{gardef}^* \ \text{end};\]

\[\text{reldef} \ ::= \text{rely} \ \text{label} : \langle \text{predicate} \rangle;\]

\[\text{gardef} \ ::= \text{guar} \ \text{label} : \langle \text{predicate} \rangle;\]

The following is an example of an environment describing the behaviour of a temperature sensor. The environment may update value of \(t\) (current temperature) by changing it in some small increments defined by constant \(\Delta\). A rely predicate is omitted and assumed to be \(\top\).

\[\text{environment temp_sensor} \]
\[\text{guar1 is } t' \in t - \Delta \ldots t + \Delta\]
\[\text{end}\]

A system activity, called a process, follows the template of an environment but may also define local variables and concrete behaviour specification.

\[\text{process} \ ::= \text{process} \ \text{label} \ \text{reldef}^* \ \text{gardef}^* \ \text{invdef}^* \ \langle \text{block} \rangle? \ \text{end};\]

Informally, the body of a process is the implementation that is shown to tolerate the interference defined by the process rely and satisfy the obligation of the process guarantee. In an extreme case of a solipsistic process there may be no rely and guarantee predicates so that the process has no specific obligations to its universe. Such a process specifies a sequential algorithm that runs till completion without any interaction.

Continuing the theme of the sensor example, with the syntax discussed, we can already define a small but meaningful specification. The temperature sensor \(t\) belongs to the environment while the system controls the heater modelled by variable \textit{heater}:

\[
\begin{align*}
\text{invariant} &\; \text{temp} : \; t \in \mathbb{Z} \\
\text{invariant} &\; \text{heater} : \; h \in \text{BOOL} \\
\text{environment} &\; \text{temp_sensor} : \\
\text{guar1} &\; \text{is } t' \in t - \Delta \ldots t + \Delta \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{process} &\; \text{heater_control} \\
\text{rely} &\; \text{rel1} : \; t \in \text{SAFE_TEMP} \\
\text{guar1} &\; : \; t > \text{TEMP_HIGH} \land h = \text{TRUE} \Rightarrow h' = \text{FALSE} \\
\text{guar2} &\; : \; t < \text{TEMP_LOW} \land h = \text{FALSE} \Rightarrow h' = \text{TRUE} \\
\text{end}
\end{align*}
\]

There may be any number of \textit{environment} and \textit{process} parts. One may, for instance, add an alarm process to detect an abnormal temperature range.

\[
\begin{align*}
\text{invariant} &\; \text{alarm} : \; \text{alarm} \in \text{BOOL} \\
\text{process} &\; \text{alarm_control} \\
\text{guar1} &\; : \; \text{alarm}' = \text{bool}(t \notin \text{SAFE_TEMP}) \\
\text{end}
\end{align*}
\]
Note that the rely of heater_control is not always satisfied by the sensor behaviour. A system process is temporarily disabled if its rely is broken by an environment. A process, however, may not violate the rely of another process or an environment.

The body of a process describes how the activity defined by its guarantee predicate is realised. The following operators are used to build the body of a process:

\begin{align*}
\langle \text{block} \rangle & := \langle \text{action} \rangle ; \langle \text{block} \rangle ; \\
\langle \text{action} \rangle & := \langle \text{statement} \rangle \text{ atomic? refin} \text{es? with?} \\
\langle \text{statement} \rangle & := \langle \text{substitution} \rangle | \langle \text{if} \rangle | \langle \text{loop} \rangle | \langle \text{begin} \rangle | \langle \text{assert} \rangle | \text{stop} ; \\
\langle \text{if} \rangle & := \text{if } \langle \text{predicate} \rangle \text{ then } \langle \text{block} \rangle \\
& \quad \text{ elseif } \langle \text{predicate} \rangle \text{ then } \langle \text{block} \rangle \ast \\
& \quad \text{ else } \langle \text{block} \rangle ? \text{ end} ; \\
\langle \text{loop} \rangle & := \text{while } \langle \text{predicate} \rangle \\
& \quad \text{ invdef } \ast \\
& \quad \text{ var } \langle \text{expression} \rangle \\
& \quad \text{ then } \langle \text{block} \rangle \text{ end} ; \\
\langle \text{begin} \rangle & := \text{begin } \langle \text{invdef} \rangle \ast \langle \text{block} \rangle \text{ end} ; \\
\langle \text{assert} \rangle & := \langle \text{assert} \rangle (\langle \text{label} \rangle :)? \langle \text{predicate} \rangle \ast
\end{align*}

Most of the syntax is self explanatory. The \text{stop} statement terminates a process; \text{assert} \ p \ asserts the truth of \ p; \langle \text{substitution} \rangle \text{ and } \langle \text{expression} \rangle \text{ are Event-B substitution and expression elements (see Rodin Deliverable D7 [7] for concrete definitions). Block } \langle \text{begin} \rangle \text{ defines the scope of visibility for local variables. Elements } \text{atomic}, \langle \text{refines} \rangle \text{ and } \langle \text{with} \rangle \text{ are used to define the refinement relationship between SLP models but are not discussed in this paper. A trivial implementation of heater\_control retells the implications in the process guarantee as an if statement:}

\begin{verbatim}
process heater\_control
\begin{align*}
\text{rely } \text{rel1: } & t \in \text{SAFE\_TEMP} \\
\text{guar} \text{ guar1: } & t > \text{TEMP\_HIGH} + \delta \land h = \text{TRUE } \Rightarrow h' = \text{FALSE} \\
\text{guar} \text{ guar2: } & t < \text{TEMP\_LOW} - \delta \land h = \text{FALSE } \Rightarrow h' = \text{TRUE} \\
\text{if } & t > \text{TEMP\_HIGH} + \delta \land h = \text{TRUE } \text{then} \\
& \text{ act1: } h' := \text{FALSE} \\
\text{elseif } & t < \text{TEMP\_LOW} - \delta \land h = \text{FALSE } \text{then} \\
& \text{ act2: } h' := \text{TRUE}
\end{align*}
\end{verbatim}

2.1 Semantics

Similar to Event-B, the semantics of SLP is given as a list of verification conditions called proof obligations. We discuss only the consistency conditions showing
that the SLP part of an Even-B/SLP model does not violate invariants and introduce deadlocks and divergences. Informally, the purpose of consistency proof obligations is to establish the following three facts:

– when control is passed to a statement, the state update defined by the statement may take place;
– any statement does not take the system outside of the safety invariant bounds;
– a statement eventually terminates.

We begin by cataloging the major syntactic elements of a specification. The following are coming from Event-B and are shared between Event-B and SLP models: constants \( c \), carrier sets \( s \), axioms \( P(c,s) \), global variables \( v \) and invariant \( I(c,s,v) \).

There are elements specific to SLP. Taking the viewpoint of a substitution \( S \) located somewhere in the body of a process, they are: the rely \( R(c,s,v,v') \) and guarantee \( G(c,s,v,v') \) of a current process; process variables \( u \) (must be distinct from \( v \)); process invariant \( T(c,s,v,u) \); variables defined in enclosing \begin{itemize} \item \texttt{begin...} and \texttt{while...} blocks, \( w = \{w_1, \ldots, w_i\} \) (all distinct); \texttt{begin...} and \texttt{while...} block invariants \( B_j(c,s,v,u,w_1, \ldots, w_i) \); assertion predicate \( A(c,s,v,u,w) \) expressed directly in a preceding \texttt{assert} or derived from other kind of a preceding statement; and, finally, the substitution itself - \( S(c,s,v,u,w,v',u',w') \).

The following shorthand is used to identify syntactic element in the context of substitution \( S \). Assume that \( S \) is contained inside \( i \) nested blocks \begin{itemize} \item \texttt{begin...} and \texttt{while...} \end{itemize} that define some local variables \( u \) and \( w \). In the scope of \( S(...) \) the actual invariant is \( I_i \), as defined below. The invariant defines the state space \( \Omega_i \) on which the update defined by \( S \) takes the effect: \( \{z \mid S(z)\} \subseteq \Omega_i \times \Omega_i \).

\[
I_i = \begin{pmatrix}
P(c,s) \\
I(c,s,v) \\
T(c,s,v,u) \\
\bigwedge_{j \leq 1} B_j(c,s,v,u,w_1, \ldots, w_i)
\end{pmatrix}
\]
\[
\begin{align*}
A &= A(c,s,v,u,w) \\
S &= S(c,s,v,u,w,v',u',w') \\
\Omega_i &= \{z \mid I_i(z)\} \\
\Omega_i^\sqrt &= \Omega_i \cup \{\sqrt\}
\end{align*}
\]

Extended state \( \Omega_i \cup \{\sqrt\} \) adds a termination symbol \( \sqrt \) from which no continuation is possible. Globally, the set of all names spaces forms a tree such that the state of an inner wholly contains the state of outer space: \( \Omega_0 \subseteq \Omega_1 \subseteq \cdots \subseteq \Omega_n \) where \( \Omega_0 \) is the state of a name space of containing just global variables and \( \Omega_n \) is the state of some current block within the body of a process.

To define verification conditions, we convert SLP statements into relations describing the connection between previous and next states. All the partial state update relations are treated as guarded relations (i.e., never applied outside of their domain) and loops are required to terminate to ensure total correctness. We write \( \parallel I_i \) meaning \( [I_i] \), \( I \) meaning \( [I] \), \( A \) for \( [A] \) and so on.
a \; assert \; p \; ; \; b \; := \; [(p) \triangleright (b)]^o

\[ \begin{align*}
[\ldots]_i \in \Omega_i \leftrightarrow \Omega_i^\prime \\
\text{stop}_i := \Omega_i \times \{\sqrt{\cdot}\}
\end{align*} \]

\[ \begin{align*}
[\text{assert}_i p]_i := & \; \text{id}([p] \cap \Omega_i) \\
a \; ; \; \text{assert}_i p \; ; \; b \; := & \; [(p) \triangleright (b)]^o \\
a \; ; \; b \; := & \; [b]_i \circ [a]_i \\
[s_1 \ldots s_n]_i := & \; [s_1]_i \cup \cdots \cup [s_n]_i \\
u := E(v) \; := & \; \{u \mapsto E(v)\}^o \\
u \in E(v) \; := & \; \{u \mapsto u' \mid u' \in E(v)\}^o \\
u : E(v, v') \; := & \; \{u \mapsto u' \mid E(v, v')\}^o
\end{align*} \]

\[ \begin{align*}
\text{if } c_0 \text{ then } b_0 \\
\text{elseif } c_1 \text{ then } b_1 \\
\cdots \\
\text{elseif } c_k \text{ then } b_k \\
\text{else } b_e \end{align*} \]

\[ \begin{align*}
\text{while } c \\
\text{invariant } LI \\
\text{var } V \\
\text{then } b \end{align*} \]

\[ \begin{align*}
\text{begin} \\
\text{invariant } BI \\
b \end{align*} \]

\[ \begin{align*}
\text{assert } \neg c \land LI \land \text{trm}\left(C, LI, V, [b]_{i+1}\right)_i \]

\[ \begin{align*}
\left([BI] \triangleright [b]\right)_{i+1} \cap (\Omega_i \times \Omega_i)
\end{align*} \]

Operator \( r^o \) extends a relation \( r \subset \Omega_i \times \Omega_i^\prime \) to a total relation \( r' \subset \Omega_i \leftrightarrow \Omega_i^\prime \) so that mappings not covered by \( r \) are taken from \( \text{id}(\Omega_i) \): \( r^o = \{x \mapsto y \mid x \mapsto y \in r \lor x \mapsto y \in \text{id}(\Omega_i) \setminus r\} = \text{id}(\Omega_i) \Leftrightarrow r \). Also, as a shorthand, for some predicate \( x \in \Omega_i \rightarrow \text{BOOL} \) we write \( [x] \) to mean a set of elements satisfying \( x \): \( [x] = \{e \mid x(e)\} \).

In the definition of a loop, \( V \in \Omega_i \rightarrow \mathbb{N} \) is a loop variant and \( \text{trm} \) is a termination condition expressing that the variant value is decreased by each loop iteration: \( \text{trm}(C, LI, V, b) \equiv \forall x, y \cdot x \in V[b[s]] \land y \in V[s] \Rightarrow x < y \), where \( st \equiv l \cap [LI \land c] \).

Note the two rules for sequential composition. The \( a \; ; \\ b \) case defines the conventional sequence-to-relational-join rule. The preceding rule (of a higher precedence) makes the sequential composition 'forgetful' when an assertion is placed between two statements: the information about previous statements is dropped and the focus is placed on the last statement and the preceding assertion. One reason for this is that a chain of substitutions may lead to a large and intractable set of hypothesis preventing efficient automated proof and introduce an undesirable interdependency between substitutions where a change in one substitution could invalidate proofs done for successive substitutions. An assertion breaks such a chain making the proof context smaller. Another reason, specific to our technique of refining Event-B into SLP, is the use of assertions...
to prove that the set of enabling states of a refined substitution does not grow larger in a refined model.

The following is a list of the more important proof obligation, given, for brevity, in a relational form.

**Well-definedness** SLP mirrors the Event-B approach of proving that each partial relation is well-guarded. In other words, we prove that a relation defined by statement $a$ may be applied to a current state: $(\Pi \cap A) \downarrow \bigl[ a \bigr] \neq \emptyset$.

**Feasibility of rely** The rely must not contradict an invariant: $\Pi \triangleleft R \subseteq I \times I$.

**Closure of rely** Conditions involving rely invariably require tolerating any number of rely iterations. To simplify corresponding proof obligations we insists that a rely relation $R$ is reflexively and transitively closed: $\text{id}(\Omega_i) \subseteq R \land R \circ R \subseteq R$.

**Invariant preservation** Invariant properties of model variables are assumed to hold before every substitution. It must be proven that all invariants known in the scope of a substitution are re-established by the substitution: $\bigl[ a \bigr](\Pi \cap A) \subseteq \Pi$.

Not that when statement $a$ is located in the body of a loop $\Pi$ also includes the loop invariant.

**Variant** A loop variant is based on the same principles as Event-B variant and is embedded into the rule converting a loop into a relational form.

**Establishing guarantee** A substitution executed by a process must agree with a process guarantee. Formally, any state update would be covered by a 'promise' expressed in the guarantee: $(\Pi \cap A) \triangleleft \bigl[ a \bigr] \subseteq G$.

**Establishing assertion** An asserted condition $A_n$ must be implied by a previous assertion or a statement. We must take into the account the fact that between previous and current statements the universe might have changed its state. For this, the latest locally known state is 'blurred' by the rely condition of a process.

- if two assertions follow each other then the second must be contained in the first: $(A \triangleleft R)[\Pi] \subseteq A_n$;
- otherwise, if an assertion is preceded by a substitution, the preceding substitution after-state must imply the assertion: $(R \circ \bigl[ a \bigr])[\Pi] \subseteq A_n$;
- otherwise, an assertion must be established by an invariant: $R[\Pi] \subseteq A_n$.

**Process compatibility** All non-environment processes must be compatible w.r.t. their rely/guarantee conditions: $I \triangleleft G_A \subseteq R_B$. 

Kyoto 54

November 13, 2012
3 From Event-B to SLP

SLP is not a standalone formalism and is meant to complement the Event-B notation when one needs to obtain a detailed design expressed in terms of parallel processes and algorithmic constructs. Thus, there is always a stage when a pure Event-B specification undergoes a transformation into an Event-B/SLP specification.

One simple case of Event-B to Event-B/SLP refinement is introducing environments and processes operating on new variables. In a general case, the Event-B part is refined to make use of new variables so that there is an information flow between the two parts. Naturally, there are no specific proof obligations for this case: one only needs to discharge the consistency conditions.

A more interesting situation is the replacement of Event-B events with SLP constructs. Of all possibilities, we shall only consider the simplest one: refinement of a set of events by new (rather than existing) environments and processes.

**New environment** (process) A new SLP environment (process) may be defined to refine one or more abstract Event-B events; refined events disappear from a model. The relevant proof obligation is that a process guarantee is contained in the behaviour of refined events: \((I \cap R) \triangleleft G \subseteq [e_1]_R \cap \cdots \cap [e_n]_R\).

**New concrete process** A sub-set of machine events may be refined into a process with a body. We focus on a simpler case when this is done in a single refinement step. Without loss of generality, we consider the case of refinement where substitutions of a process body coincide exactly with substitutions of refined events, in other words, a refinement that forms a process from events without any further behavioural or data refinement that may take place in following refinement steps.

Let \(E\) be the set of events of machine \(M\) describing the behaviour of a prospective process \(P\) and \(\text{tr}(M) \models E\) be the machine traces limited to events \(E\). Let \(\text{tr}(P)\) be a set of traces of a new process in terms where each trace element is the list of labels of parallel substitution parts. It is easy to define a mapping \(f\) from the alphabet of \(\text{tr}(P)\) to set \(E\) (it is not necessarily a one-to-one mapping but this does not pose problems). If one can prove that \(f(\text{tr}(P)) \subseteq \text{tr}(M) \models E\) and, separately, that process \(P\) does not introduce new divergences then process \(P\) is declared to refine events \(E\). We have previously shown how to convert a statement of the form \(f(\text{tr}(P)) \subseteq \text{tr}(M) \models E\) into a list of FOL theorems [3, 4].

4 Small Example

We illustrate the Event-B/SLP hybrid modelling by showing a simple case of Event-B to SLP refinement. The model computes the greatest common divisor (GCD) of two numbers. Function gcd \(\in \mathbb{N} \times \mathbb{N} \to \mathbb{N}\) axiomatically satisfies the following properties:
axm1 : ∀a, b : a, b ∈ N ∧ a > b ⇒ gcd(a, b) = gcd(a - b, b)
axm2 : ∀a, b : a, b ∈ N ∧ b > a ⇒ gcd(a, b) = gcd(a, b - a)
axm3 : ∀a : a ∈ N ⇒ gcd(a, a) = a

At an abstract level, one may use the constant function gcd to compute the result in one step:

```plaintext
machine gcd0
variables r, x₁, x₂
invariant r ∈ N ∧ x₁ ∈ N ∧ x₂ ∈ N
initialisation r := ∈ N \parallel x₁ := ∈ N \parallel x₂ := ∈ N
events
gcd = begin r := gcd(x₁ ↦ x₂) end
end
```

Variables x₁ and x₂ serve as input values and r holds the result. The following is a typical Event-B refinement based on the unfolding of an atomic abstract step into a sequence of concrete computations.

```plaintext
refinement gcd1a
refines gcd0
variables r, x₁, x₂, y₁, y₂, pc
invariant
y₁ ∈ N ∧ y₂ ∈ N
pc ∈ 1..5
pc = 2 ⇒ gcd(x₁ ↦ x₂) = gcd(y₁ ↦ x₂) ∧ y₁ > 0 ∧ x₂ > 0
pc = 3 ⇒ gcd(x₁ ↦ x₂) = gcd(y₁ ↦ y₂) ∧ y₁ > 0 ∧ y₂ > 0
pc = 4 ⇒ gcd(x₁ ↦ x₂) = gcd(y₁ ↦ y₂) ∧ y₁ > 0 ∧ y₂ > 0
initialisation \ldots \parallel y₁ := ∈ N \parallel y₂ := ∈ N \parallel pc := 1
events
copy₁ = when pc = 1 then y₁ := x₁ \parallel pc := 2 end
copy₂ = when pc = 2 then y₂ := x₂ \parallel pc := 3 end
sub₁ = when y₁ > y₂ ∧ pc ∈ {3, 4} then y₁ := y₁ - y₂ \parallel pc := 4 end
sub₂ = when y₂ > y₁ ∧ pc ∈ {3, 4} then y₂ := y₂ - y₁ \parallel pc := 4 end
gcd = when y₁ = y₂ ∧ pc = 4 then r := y₁ end
end
```

Events sub₁ and sub₂ form the body of a loop. An auxiliary variable pc is used to simulate control flow; variables x₁, x₂, y₁, y₂ are introduced to describe the concrete computation steps. Note how the after state of each event is encoded in model invariant. The repeating template v = C⇒\ldots in invariants is an indicator that an event-based specification is used to simulate concrete control flow.

The SLP version of the same refinement step is given below. Here we have an explicit loop construct containing a two-branch if that makes for a more concise specification without the need to propagate state properties via an invariant.
refinement gcd1b
refines gcd0
variables r, x1, x2, y1, y2
invariant y1 ∈ N ∧ y2 ∈ N
initialisation ... || y1 ∈ N || y2 ∈ N
process main
  y1 := x1 || y2 := x2 ;
  while y1 ≠ y2 then
    invariant gcd(x1 ↦ x2) = gcd(y1 ↦ y2) ∧ y1 > 0 ∧ y2 > 0
    if y1 > y2 then y1 := y1 − y2
    elseif y2 > y1 then y2 := y2 − y1 end
  end ;
  r := y1
end
end

Essential to the proof of refinement is the last case of sequential composition
where control is passed from a loop to an assignment saving the final result. The
relational interpretation of the loop asserts the loop invariant and the negation
of the loop condition which immediately give that r = y1 = gcd(x1 ↦ x2).

5 Conclusion

The implementation language of B-Method, B0 [1] is one of the inspirations for
this works. There are, however, important differences in both aims and tech-
niques employed: B0 allows a modeller to write more detailed bodies of abstract
operations using the concepts from programming languages. In contrast, in SLP,
the main development technique is an aggregation of several abstract events into
a body of a process. This means that a data-driven design of Event-B may be
refined into an algorithmic design whereas in B0 it would have to remain data-
driven at the top level. Equally important is an explicit treatment of concurrency
that becomes more and more relevant topic in embedded systems design. We
use rely/guarantee [6] approach to model cooperation of concurrent processes
via shared variables.

Event-B is rather obviously lacking in means of control flow specification.
One solution is the integration of two narrowly specialised two notation, i.e.,
CSP∥B that combines B and CSP [9]. Another is extension of the basic notation
with means to explicitly define control flow, i.e., the Flow plug-in for Rodin [3].
In this paper we followed a different direction with a premise that a deficiency of
a notation in a certain area is best rectified by coming up with a new notation.

This leads us to the following crucial point: to make Event-B applicable
in any given problem domain it may be necessary to (1) design a specialised
concrete syntax exposing Event-B method in a way tailored to the problem
domain (for example, a graphical notation like the one offered by UML-B [8])
and (2) devise a specialised notation and refinement rules for concrete designs,
like the one shown in this paper. The use of Event-B for an abstract design
puts a development on a solid and well-studied platform. But concrete designs
incorporating implementation decision must offer the concepts, terminology and
structuring principles already employed and recognised in the target problem domain. In this sense, the language defined in this paper is merely a technological demonstration that such a direction is viable.

References

5. Industrial deployment of system engineering methods providing high dependability and productivity (DEPLOY). IST FP7 project, online at http://www.deploy-project.eu/.
Proof Hints for Event-B

Thai Son Hoang
Institute of Information Security, ETH Zurich

Abstract. Interactive proofs are often considered as costs of formal modelling activity. In an incremental development environment such as the Rodin platform for Event-B, information from proof attempts is important input for adapting the model. This paper considers the idea of using interactive proofs to “improve” the model, in particular, to convert them into automatic ones. We propose to lift some essential proof information from the interactive proofs into the model as what we called proof hints. In particular, proof hints are not only for the purpose of proofs: it helps to understand the formal models better.

Keywords: Event-B, formal modelling, proof hints, the Rodin platform

1 Introduction

Event-B is a refinement-based modelling method, which can be used to develop various types of systems. Starting with an abstract specification, several refinement steps gradually introduce more details into the formal models in a consistent manner. An important part of an Event-B formal model is the verification conditions generated as proof obligations. The task of discharging these obligations is first given to some automated provers. Afterwards, remaining undischarged proof obligations are required to be proved interactively. Typically, manual proofs are considered as “costs” of development, given the required human interaction for produced them, and the difficulty in maintaining them when the formal models evolve.

As the size of models grows, the complexity of the associated proof obligations also increases, hence interactive proofs are inevitable. Improving the performance of the automatic provers has been considered with some success [3,4,2]. Despite the improvement in the percentage of automatic proofs, interactive proofs are still an obstacle in developing and maintaining formal models.

In this paper, we attempt to answer the following question. Can we improve our formal models in such a way that helps the proofs? After all, modelling using refinement is also a way of structuring the proof of correctness of the models. We propose some notions to encapsulate essential proof details extracted from interactive proofs within the formal model. We call the additional information to the formal models “proof hints”.

Some form of proof hints already exists in Event-B, e.g., “witnesses” or “theorems”. These useful features are designed not only to help with proving the correctness of the model but also to give more information about the particular
model, i.e., why it is correct. In fact, “proof hints” should help to understand the formal model better.

We consider the current state of Rodin Platform (Rodin) and show two kinds of useful proof information that can be included in the formal models, namely, to select hypotheses and to perform a proof by cases.

**Select hypotheses** Indicates that some facts (e.g., invariants or axioms) are required for discharging the obligations.

**Perform a proof by cases** Indicates that the proof can be discharged by consider different cases.

We show that the effect of the proof hints can be “simulated” at the moment by some modelling “tricks” in Event-B.

In the long term, we propose to have an extension to Event-B and to Rodin, to have proof hints as a part of the model and to implement a plug-in for interpreting the proof hints and applying these hints appropriately during proofs.

**Organisation.** The rest of the paper is structured as follows. Section 2 gives some background on Event-B and Rodin. Section 3 presents our ideas of proof hints. Section 4 illustrates proof hints by means of two examples. Section 5 discusses some proposals for the tool support. We give some conclusions in Section 6.

### 2 Background

#### 2.1 The Event-B Modelling Method

An Event-B model corresponds to a discrete transition system and is divided into two parts: a static part called context and a dynamic part called machine. A context may contain carrier sets (types), constants, axioms (assumptions about sets and constants). For clarity, we omit references to context in the sequel.

Machines may contain variables, invariants, and events. Variables \(v\) define the state of a machine and are constrained by invariants \(I(v)\). An event \(e\) can be represented as \(e \equiv \text{any } x \text{ where } G(x, v) \text{ then } Q(x, v) \text{ end}\), where \(x\) stands for the event’s parameters, thus allowing for state changes. The guard \(G(x, v)\) states the necessary condition under which an event may occur. The action \(Q(x, v)\) describes how state variables \(v\) evolve when the event occurs. The short form \(e \equiv \text{when } G(v) \text{ then } Q(v) \text{ end}\) is used when the event does not have any parameters, and we write \(e \equiv \text{begin } Q(v) \text{ end}\) when, in addition, the event’s guard equals true. A dedicated event in the last form is used for the initialisation event (init). The action of an event is composed of one or more assignments of the form: \(v := E(x, v), \ v :\in E(x, v), \ v :\mid P(x, v, v')\), specifying \(v\) becomes \(E(x, v)\), \(v\) becomes an element of \(E(x, v)\), and \(v\) becomes such that \(P(x, v, v')\) holds, respectively. All assignments of an action \(Q(x, v)\) occur simultaneously. As a result, each event \(e\) is associated with a before-after predicate, denoted as \(Q(x, v, v')\).
The invariant preservation proof obligation (INV) states that invariants are maintained whenever variables are updated. For each event \( e \), the corresponding proof obligation is as follows.

\[
I(v), G(x, v), Q(x, v, v') \vdash I(v') \quad \text{(INV)}
\]

All predicate modelling elements, e.g., axioms, invariants, guards, can be also declared as theorems. Theorems need to be proved when they are declared. As an example, a theorem in guard must be proved to be a consequence of axioms, invariants, and previously declared guards.

Machine refinement is a mechanism for introducing details about the dynamic properties of a model [1]. The states of the abstract machine \( M \) are related to the states of the concrete machine \( N \) by gluing invariants \( J(v, w) \), where \( v \) and \( w \) are the variables of the abstract and concrete machine, respectively. Each event \( e \) of the abstract machine is refined by a concrete event \( f \) (later we will relax this constraint). Assume that the concrete event is of the following form

\[
f \equiv \text{any } y \text{ where } H(y, w) \text{ then } R(y, w) \text{ end}.
\]

Somewhat simplifying, we can say that \( e \) refines \( f \) if the gluing invariant establish a simulation of \( f \) by the \( e \). This is presented as the following obligation.

\[
I(v), J(v, w), H(y, w), R(y, w, w') \vdash \exists v'. G(x, v) \land Q(x, v, v') \land J(v', w') \quad \text{(REF)}
\]

In order to split the above proof obligation, Event-B introduces the notion of “witnesses” for \( x \) and \( v' \). The witnesses are predicates \( W_1(x, y, v, w, w') \) (for \( x \)) and \( W_2(v', y, v, w, w') \) (for \( w' \)), which are required to be feasible, i.e., satisfying the following proof obligations.

\[
I(v), J(v, w), H(y, w) \vdash \exists x. W_1(x, y, v, w) \quad \text{(WFIS1)}
\]

\[
I(v), J(v, w), H(y, w), R(y, w, w') \vdash \exists v'. W_2(v', y, v, w, w') \quad \text{(WFIS2)}
\]

The witnesses supply instances of \( x \) and \( v' \) (provided that they exist) for instantiating the proof obligation \( \text{REF} \). Given the witnesses, the refinement proof obligation \( \text{REF} \) can be split into the following proof obligations.

\[
I(v), J(v, w), H(y, w), W_1(x, y, v, w, w') \vdash G(x, v) \quad \text{(GRD)}
\]

\[
I(v), J(v, w), H(y, w), R(y, w, w'), W_1(x, y, v, w, w'), W_2(v', y, v, w, w') \vdash Q(x, v, v') \quad \text{(SIM)}
\]

\[
I(v), J(v, w), H(y, w), R(y, w, w'), W_1(x, y, v, w, w'), W_2(v', y, v, w, w') \vdash J(v', w') \quad \text{(INV)}
\]

In the course of refinement, new events are often introduced into a model. New events must not modify abstract variable \( v \). When an abstract event \( e \) is refined by more than one concrete events \( f \), we say that the abstract event \( e \) is split and prove that each concrete \( f \) is a valid refinement of the abstract event. A concrete event \( f \) can refine two (or more) abstract events \( e \), provided that the abstract events are only different in guards. We say that the abstract events are merged into the concrete event \( f \). It is required to prove that the guard of \( f \) is stronger than the disjunction of the guards of the abstract events.
2.2 Proving with the Rodin Platform

Rodin is an Eclipse-based tool chain for analysing and reasoning about Event-B models. Models are developed incrementally within the platform. Two main activities of developers are modelling and proving (as illustrated in Fig. 1). Proof obligations are generated from modelling and are input to the proving activity. Information about proof attempts from proving are input to the modelling activity to “improve” the models. In particular, failed proof attempts usually indicate some problems with the models and give hints on how the models can be fixed, e.g., to strengthen the guard of some events or to add some missing invariants into the models.

![Fig. 1. Developing Event-B models using Rodin](image)

Obligations are proved either automatically or manually. In automatic mode, Rodin uses some predefined proof tactics made up of internal and external provers to discharge the obligations. In interactive mode, the user “guides” the proof attempts by applying some simple proof steps to simplify the obligations before invoking some trusted external provers to finish the proofs. As interactive proofs require manually interventions, it is usually considered as some costs of developing formal models. Moreover, maintenance of interactive proofs is difficult: a change in the formal model more often invalidates the interactive proofs. As a result, improving the rate of automatic proofs will also help to maintain the models better.

We consider some common interactive proof steps, e.g., to add hypothesis, to select hypotheses, and to perform a proof by cases.

**Add hypothesis** This proof step corresponds to the following proof rule.

\[
\begin{array}{c}
\text{H} \vdash P \\
\text{H, P} \vdash G
\end{array}
\]

\[H \vdash G\text{ CUT}\]

The rule allows add \(P\) as a hypothesis, provided that it can be proved from the current hypotheses \(H\).

**Select hypotheses** Rodin has a notion of selected hypotheses which is used by some external provers. Often, too many irrelevant hypotheses will have
negative effect on the performance of external provers. By restricting the set of hypotheses available to these external provers, the user helps the external provers to concentrate on using only some relevant hypotheses. An example for selected hypotheses are guards of an event: they are by default selected for proof obligations related to the event.

**Perform a proof by cases** This proof step allows user to perform a proof by cases, with respect to some condition $P$.

\[
\begin{array}{c}
H, P \vdash G \\
H, \neg P \vdash G \\
\hline
H \vdash G
\end{array}
\]

The proof is split into two branches accordingly.

3 Proof Hints

There are existing work for improving the rate of automatic proofs. Recently, some links to external provers such as Isabelle [4], SMT [2] have been added to Rodin. Selected hypotheses can be calculated according to some heuristic [3]. However, interactive proofs are still unavoidable. We look at the problem from a different angle: to convert interactive proofs into automatic proofs by improving the formal models, essentially exposing some proof information in the formal models. We call these additional proof information **proof hints**.

There is already several such proof information existing in the Event-B models, normally being seen as part of the model rather than some exposed proof information.

- Theorems in the model is a special case of *adding a hypothesis* in an interactive proof.
- Automatic *selection of guards* for the event’s proof obligations.
- The witnesses can be seen as some hints for manually *instantiating* the existential goal of the general proof obligation $\text{REF}$, which results in three sub-obligations $\text{GRD}$, $\text{SIM}$, and $\text{INV}$.

In principle, any proof information can be lifted to be proof hints, part of the model. However, revealing the actual proof is certainly undesirable: this could have negative effects on the understanding of the model. In fact we want only to exposing *essential* information about the proofs. We believe that the criteria for proof hints should be as follows.

1. They should help to *automate* more proofs.
2. They should help to better *understand* the model.

While the first criteria is straightforward, our emphasis is on the second criteria. Once again, low level proof information is irrelevant for understanding of the model. We only want to have essential key important proof steps as hints, in order to justify about the correctness of the formal models.
4 Some Useful Proof Hints

This section presents two kinds of proof hints, namely to select hypotheses and to perform a proof by cases.

4.1 Select Hypotheses

During an interactive proof section, the developer can complete the proof by selecting some hypotheses and invoke one of the provers that uses only selected hypothesis, e.g., AterlierB P0. The solution to make the proof become automatic is to (somehow) give hints to Rodin to select these additional hypotheses automatically.

Example Consider the following specification containing two variables \( x \) and \( y \). The machine has a single event\(^1\) called set, which assign \( y + 1 \) to \( x \) when \( x \) is either 1 or 2.

\[
\begin{align*}
\text{variables: } & x, y \\
\text{invariants:} & \\
& \text{hypSel0}_1: \ x \in \mathbb{N} \\
& \text{hypSel0}_2: \ x \neq 0 \Rightarrow y \in \mathbb{N}
\end{align*}
\]

\[
\begin{align*}
\text{set} & \\
\text{when} & \text{grd1: } \ x \in \{1, 2\} \\
\text{then} & \text{act1: } \ x := y + 1 \\
\text{end}
\end{align*}
\]

We are interested in proof obligation \text{set/hypSel0} \_1/INV stating that event set maintains the invariant \text{hypSel0} \_1.

\[
x \in \mathbb{N}, x \neq 0 \Rightarrow y \in \mathbb{N}, x \in \{1, 2\} \vdash y + 1 \in \mathbb{N}
\]

The proof obligation cannot be discharged automatically. In particular, by default, the selected hypotheses are \text{hypSel0} \_1 and \text{grd1}. The obligation can be discharged by selecting \text{hypSel0} \_2, and invoke external provers using selected hypotheses, such as AterlierB P0.

Workaround A simple workaround to have \text{hypSel0} \_2 being selected automatically is to add the invariant as a \textit{theorem} in guard of set.

\[
\begin{align*}
\text{set} & \\
\text{when} & \text{grd1: } \ x \in \{1, 2\} \\
\text{thm1: } & x \neq 0 \Rightarrow y \in \mathbb{N} \\
\text{then} & \text{act1: } \ x := y + 1 \\
\text{end}
\end{align*}
\]

\(^1\) For clarity we omit the initialisation event \text{init}. 

---

1 Kyoto 64 November 13, 2012
The additional theorem can be removed in further subsequent refinement if necessary.

**Proposal** The disadvantages of the above approach are as follows.

- This introduces extra proof obligations to prove that the copies of the invariants are theorems in guard (even though those proof obligations are discharged automatically).
- Recopying the text of the invariants is error-prone.
- Reformulating invariants leads to the need for changing the text of the extra theorems.

Our proposal is to have a specific “proof hint” for events. This form of proof hints will specify the invariant/theorem need to be used automatically. For example, we could extend event set with the following hint of using hypSel0_2 for the maintenance of hypSel0_1.

```plaintext
set
  when x ∈ {1, 2}
  then x := y + 1
  hints use hypSel0_2 for hypSel0_1
end
```

### 4.2 Perform a Proof by Cases

Sometimes, during an interactive proving session, the user suggests a predicate \( P \) in order to do a “proof by cases”. The subsequent branches of the proof can be discharged easily. It is hence desirable to have this hint about performing a proof by cases in the model. Automatic provers often do not apply the case splits automatically, since this potentially leads to blow up in terms of the number of sub-goals.

**Example** Consider the following specification with three variables \( a, b, c \).

```plaintext
variables: a, b, c

invariants:
  case0_1 : a ≤ c
  case0_2 : a ≠ 1 ⇒ b = a + 1
  case0_3 : a = 1 ⇒ b ≤ c

set
  begin
    a := b - 1
  end
```
The interesting proof obligation to look at is \( \text{set/case0\_1/INV} \).

\[
a \leq c, a \neq 1 \Rightarrow b = a + 1, a = 1 \Rightarrow b \leq c \vdash b - 1 \leq c
\]

Informally, the reasoning follows the cases of either \( a = 1 \) or \( a \neq 1 \) and applying \text{case0\_2, case0\_3} accordingly. Hence we would like to give the hints about the case split. The obligation is not discharged by the default automatic prover within Rodin.

**Workaround** In order to “simulate” the effect of introducing this proof hints, we first split the event into two sub-events, guarded by corresponding conditions.

\[
\begin{align*}
\text{set\_case1} & \equiv \text{when } a = 1 \text{ then } a := b - 1 \text{ end} \\
\text{set\_case2} & \equiv \text{when } a \neq 1 \text{ then } a := b - 1 \text{ end}
\end{align*}
\]

The original event \text{set} can be obtained by merging the above two events using refinement.

\[
\begin{align*}
\text{set} & \text{ refines set\_case1, set\_case2} \\
\text{begin} & \quad a := b - 1 \\
\text{end}
\end{align*}
\]

which leads to a trivial proof obligation to prove about merging events.

**Proposal** There are several disadvantages of the workaround:

- Splitting of event and merging using refinement is artificial.
- Splitting of events leads to double number of proof obligations (those that does not need the case split).

Our proposal is to provide a hint directly in the model about the case split.

\[
\begin{align*}
\text{set} & \begin{align*}
& \text{begin} \\
& \quad a := b - 1 \\
& \text{hints} \\
& \quad \text{split case using } a = 1 \text{ for case0\_1}
\end{align*} \text{end}
\end{align*}
\]

### 5 Ideas on Tool Support

Given the extensibility of Rodin, having proof hints as additional elements of Event-B models would be straightforward. How the hints are interpreted and work with the automatic provers of Rodin is a more challenging topic. There are some options for the implementation of the “hint-interpreter”.

---

Kyoto 66 November 13, 2012
The first option is to have the interpreter to effect the generated proof obligations. For example, two different proof obligations are generated according to the “proof-by-cases” hint to replace the original proof obligation. This requires to alter the *Proof Obligation Generator (POG)* of *Rodin* to take into account the hints. The second option is to leave the original proof obligations untouched and to incorporate the hints at the start of a proof, *i.e.*, they can be applied before the automatic provers are invoked.

At the moment, we are investigating these options for tool support. The first option is similar to witnesses in event refinement, to split proof obligation *REF* into obligations *GRD*, *SIM*, and *INV*. As a result, it changes the generated proof obligations of Event-B models, which might be undesirable. In particular, the number of proof obligations generated can be different depending on whether proof hints are present. The second option applying proof hints at the beginning of each proof, works for the two illustrated proof hints presented within this paper. In general, we might want to have more general proof hints that should be apply in the middle of a proof, or even combining different hints in one proof.

### 6 Conclusion

We presented some ideas about the notion of “proof hints” for Event-B and discusses the possibilities of extending the supporting *Rodin platform*. In a broad term, proof hints essentially are proof information that are added to the model. We proposed two kinds of proof hints in this paper, for suggesting selected hypotheses and performing proof-by-cases. We presented some workaround at the moment to “simulate” the proof hints and to automate some proofs in the current version of *Rodin*. The illustrated examples are simple and seem to be unrealistic. However, they are extracted from some large development (adapted accordingly). Their simplicity is not too illustrate the weakness of *Rodin’s* automatic provers, but rather to support the argument that formal proofs of systems are challenging tasks.

Often when describing an Event-B formal model, we also need to explain why the model is correct, *e.g.*, why guard strengthening or invariant preservation is satisfied. Proof hints should give some information (but not too much) to answer the questions about the correctness of models. It would be the ultimate goal of having self-explained formal models, not only in terms of how they works (*e.g.*, events) and what their properties are (*e.g.*, invariants), but also why they are correct.

We do not propose proof hints as a way to avoid interactive proofs. More often, we need to perform some interactive proof steps, in order to figure out or understand why the obligation can be discharged. The role of proof hints therefore is to convert some interactive proofs into automatic ones, helping the model to be more resilient to changes.

In the long term, we might want to extract the essence of proofs as some high-level structured proofs (similar to Isabelle/Isar [5]). This requires more
investigation in terms of the usefulness of such an approach, and subsequent tool support.

References

Towards Refinement Strategy Planning for Event-B

Tsutomu Kobayashi¹ and Shinichi Honiden²

¹ The University of Tokyo, Japan
² National Institute of Informatics, Japan
{t-kobayashi,honiden}@nii.ac.jp

Abstract. Event-B is a formal approach oriented to system modeling and analysis. It supports refinement mechanism that enables stepwise modeling and verification of a system. By using refinement, the complexity of verification can be spread and mitigated. In common development using Event-B, a specification written in a natural language is examined before modeling in order to plan the modeling and refinement strategy. After that, starting from a simple abstract model, concrete models in several different abstraction levels are constructed by gradually introducing complex structures and concepts. Although users of Event-B have to plan how to abstract the specification for the construction of each model, guidelines for such a planning have not been suggested. Specifically, some elements in a model often require that other elements are included in the model because of semantics constraints of Event-B. As such requirements introduces many elements at once, non-experts of Event-B often make refinement rough though rough refinement does not mitigate the complexity of verification well. In response to the problem, a method is proposed to plan what models are constructed in each abstraction level. The method calculates plans that mitigate the complexity well considering the semantics constraints of Event-B and the relationships between elements in a system.

Keywords: Formal Methods, Refinement, Event-B, Specifications, Abstraction

1 Introduction

Event-B [2] is a formal approach oriented to system-level analysis and modeling. Event-B users specify models in a notation based on set theory and first order logic and check the model by proving.

The most notable feature of Event-B is the support for refinement mechanism. In a refinement process, users first construct a simple and highly abstract model. After checking the consistency in the abstract model, more complex model is constructed and the consistency between the abstract model and the complex model is checked. Usually complex models are constructed by introducing new aspects and properties of a system. Users gradually construct
concrete models by repeating this step. Refinement enables users to construct a complex model more simply rather than to construct the complex model at once. Therefore, the burden of a model construction is mitigated by Event-B.

As Event-B is an effective method, it has been attracting more and more attentions from the industry. For example, many companies in Japan are interested in formal methods including Event-B, while technical and administrative guidelines are constructed through cooperation of a national institute and software vendors [1].

Refinement enables users to spread the complexity of modeling over some steps. However, it is necessary to properly define how elements are gradually introduced, mitigating the complexity while complying with semantics constraints of Event-B. In this paper, we propose a method that considers the constraints and relationships between elements of a system and plans what models should be constructed for an effective refinement. Thus the method enables ordinary users to leverage the refinement mechanism in a simple way.

The remainder of this paper is organized as follows. Section 2 describes the problem and the cause of the problem. Section 3 describes the proposed method together with exemplification on an example. Section 4 shows related work as well as future work, before concluding remarks in Section 5.

2 Problem and Approach

In usual modeling in Event-B, users first read the specification of the system written in a natural language. Then models in several abstraction levels are constructed gradually. Event-B supports checking constructed models but does not guide modeling explicitly. Thus users have to plan what models are constructed in each abstraction level. Specification of a system is composed of a set of statements about property of the system. We call such statements artifacts. For example, a specification of a library management system may include an artifact “There are no loaned books in the open stack.”. Usually, Event-B models include invariants that correspond to a subset of artifacts of the system. Thus, users have to plan which subset of artifacts of the system should be reflected to models of each abstraction level. When constructing a concrete model, new artifacts are added to artifacts of the abstract model. Therefore users need to plan which artifacts are introduced to each abstraction level.

In Event-B models, artifacts are expressed as invariants using the formal language of Event-B. Thus, in order to express an artifact, it is necessary to introduce elements (e.g. career sets, constants, variables, and events) that correspond to terms appeared in the artifact. We call such terms phenomena. For instance, an artifact “There are no loaned books in the open stack.” can be expressed like “openstack ∩ dom(loan_state) = ∅” (Where openstack ⊆ books and loan_state ∈ books ⊃ members) only when openstack and loan_state are already introduced to the model.

Such constraints on the introduction of phenomena exist not only between an artifact and phenomena but also between a phenomenon and phenomena. That
is, in some cases, an introduction of a phenomenon requires an introduction of some other phenomena. The causes of such constraints on introduction include the following two facts.

Firstly, all variables and constants in an Event-B model have to be typed as a primitive type (a built-in data type or an element of a career set as a user-defined type) or a pair that is recursively built from primitive types. A primitive type is atomic and not expressible by using other primitive types. For example, if a variable \( \text{var} \) is typed as \( S \) by using a career set \( S \) in an abstract model and typed as \( T \to \mathbb{N} \) by using a career set \( T \) and a built-in data type \( \mathbb{N} \) in a concrete model, a type error will occur. Thus, any typing statement of a phenomenon should not be changed through refinement. Therefore, a phenomenon corresponds to a variable or a constant can be introduced only when phenomena that are necessary to type the variable or constant are already introduced. For instance, in order to introduce a phenomenon “loan state” and express its type as \( \text{loan} \cdot \text{state} \in \text{books} \to \text{members} \), the introduction of career sets \( \text{books} \) and \( \text{members} \) is needed. Thus, it is also required to introduce phenomena that corresponds to career sets \( \text{books} \) and \( \text{members} \).

Secondly, Event-B has several criteria for consistency between an abstract model and a concrete model. In modeling in Event-B, users can confirm consistency of models by proving them (proof obligations). There is a proof obligation named EQL, which requires that if a variable is included in both an abstract model and a concrete model there must not be a state transition such that it is included in the concrete model but not included in the abstract model. In order to make a refinement consistent EQL proof obligation must hold. Therefore a phenomenon corresponds to a variable can be introduced in a model only when state transitions that change the value of the variable are already introduced. For example, the introduction of a variable that corresponds to a phenomenon “loan state” requires an introduction of all state transitions included in the behavior of the system (e.g. “loaning a book from the open stack”, “returning a book”, “loaning a reserved book”).

For these reasons, an introduction of an artifact to a model requires introductions of 1) phenomena that appear in the artifact and 2) phenomena that are required by 1). Let \( A = (A_i)_{i=0,1,...} \) be a sequence of sets of artifacts reflected to the \( n \)th model, \( \text{req}_a(a) \) be the phenomena required by an introduction of an artifact \( a \), and \( \text{req}_a(A_i) = \bigcup_{a \in A_i} \text{req}_a(a) \) for a set of artifacts \( A_i \). When an artifact is introduced, phenomena that are not introduced yet but required for an introduction of the artifact are also introduced. Then, let \( \text{int}_p(A) = \text{req}_a(A_i) \setminus \text{req}_a(A_{i-1}) \) for a set of artifacts \( A_i \). It denotes newly installed phenomena to the \( i \)th model in a sequence of artifacts sets \( (A_i)_{i=0,1,...} \).

Consider the introduction order of artifacts \( a \) and \( b \) such that \( \text{req}_a(a) = \{p_1, \ldots, p_{10}\} \), \( \text{req}_a(b) = \{p_6, \ldots, p_{10}, q\} \) to an empty model (Figure 1). For \( A_1 = \{a\} \), \( A_2 = \{a, b\} \), the newly introduced phenomena will be as follows: \( \text{int}_p(A) = \text{req}_a(a) = \{p_1, \ldots, p_{10}\} \), \( \text{int}_p(A) = \text{req}_a(b) \setminus \text{req}_a(a) = \{q\} \). In contrast, For \( B_1 = \{b\} \), \( B_2 = \{a, b\} \), the newly introduced phenomena will

\(^3\) In this paper, the relative complement of a set \( T \) in a set \( S \) is denoted by \( S \setminus T \).
be as follows: \(\text{int}_p(A) = \text{req}_a(b) = \{p_6, \cdots, p_{10}, q\}\), \(\text{int}_p(B) = \text{req}_a(a) \setminus \text{req}_a(b) = \{p_1, \cdots, p_5\}\).

**Fig. 1.** Difference between Two Introduction Orders

Therefore, \(\langle \text{int}_p(A) \rangle, \langle \text{int}_p(B) \rangle \rangle = (10, 1)\), whereas \(\langle \text{int}_p(A) \rangle, \langle \text{int}_p(B) \rangle \rangle = (6, 5)\). Thus, the number of newly introduced phenomena in each model will vary depending on the introduction order of artifacts. As refinement is a mechanism to mitigate the complexity of modeling by spreading the introduction of phenomena over some steps, how much the numbers of newly introduced phenomena are dispersed is useful information for users to plan refinement. Thus, we define the *effectiveness* of an introduction order of artifacts as in Definition 1. For instance, the order \((a, b)\) is more effective than \((b, a)\) in the above example. Moreover, an order \((A_i)_{i=0, \cdots, 3}\) such that \((\text{num}_{A,i})_{i=1,2,3} = (3, 1, 2)\) is more effective than an order \((B_i)_{i=0, \cdots, 3}\) such that \((\text{num}_{B,i})_{i=1,2,3} = (0, 3, 3)\) since \((\text{num}_{A,i})_{i=1,2,3} = (3, 2, 1)\) is smaller than \((\text{num}_{B,i})_{i=1,2,3} = (3, 3, 0)\) in lexicographical order.

**Definition 1.** Let a sequence \((\text{num}_{A,i})_{i=1,\ldots,|A|}\) be a history of the number of phenomena newly introduced in each refinement (i.e. \(\text{num}_{A,i} = |\text{int}_p(A)|\)) and a sequence \((\text{num}_{A,i})_{i=1,\ldots,|A|}\) be a sorted permutation of \((\text{num}_{A,i})_{i=1,\ldots,|A|}\) in descending order. Then, for a sequence \((A_i)_{i=0,\ldots,|A|}\) and \((B_i)_{i=0,\ldots,|B|}\) such that \(\text{req}_a(A_i) = \text{req}_a(B_{i+1})\), \((A_i)_{i=0,\ldots,|A|}\) is called more effective than \((B_i)_{i=0,\ldots,|B|}\), if \((\text{num}_{A,i})_{i=1,\ldots,|A|}\) is smaller than \((\text{num}_{B,i})_{i=1,\ldots,|B|}\) in lexicographical order.

As in the above example, users should plan an introduction order of artifacts so that the refinement is effective. However, for this planning, users have to grasp and compare the constraints on introduction between phenomena over multi-
### Table 1. Artifacts of Library Management System

<table>
<thead>
<tr>
<th>Artifact</th>
<th>Phenomena Appeared in the Artifact</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>“Loan is done only for members” loan state, members</td>
</tr>
<tr>
<td>b</td>
<td>“Books on loan are not in the open stack” loan state, books, open stack state</td>
</tr>
<tr>
<td>c</td>
<td>“No reserved books are in the open stack” reservation state, books, open stack state</td>
</tr>
</tbody>
</table>

### Table 2. Events of Library Management System

<table>
<thead>
<tr>
<th>Event</th>
<th>Caused State Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>Loaning a reserved books Remove one from reservation state, Add one to loan state</td>
</tr>
<tr>
<td>p2</td>
<td>Returning a book Remove one from loan state</td>
</tr>
<tr>
<td>p3</td>
<td>Loaning a book from the open stack Remove one from open stack state, Add one to loan state</td>
</tr>
</tbody>
</table>

3 Method

#### 3.1 Derivation of Required Phenomena

As we viewed in Section 2, the effectiveness of refinement depends on the sets of phenomena required by artifacts. The phenomena required by an artifact depend on types and state transitions related to phenomena that appear in the artifact. In the proposed method, constraints on introduction between phenomena related to types and state transitions are assumed as the input. The output of the method is orders that maximize the effectiveness of refinement.

To illustrate the method, construction of a model of a library management system is described. The artifacts of the system are as shown in Table 1 and the events of the system are as shown in Table 2. Events represent behavior of the system. An event can cause multiple state transitions.

Phenomena can be classified into four kinds according to what kind of element in an Event-B model corresponds to the phenomenon. Phenomenon corresponds to a career set, a constant, a variable, or an event.

Let $P_S$, $P_C$, $P_V$, and $P_E$ be the set of phenomena that correspond to career sets, constants, variables, and events, respectively. Let $P = P_S \cup P_C \cup P_V \cup P_E$ and $T$ be the set of state transitions.

Let $\text{typed} : P \to 2^{P_S}$ be a set of career sets required for typing a constant or a variable, $\text{changed by} : P \to 2^T$ be a set of state transitions that change the value of a variable, and $\text{caused by} : T \to 2^{P_S}$ be a set of events that causes a state transition.
The proposed method takes these three functions as the input from the users. For example, constraints on introduction between phenomena in the library management system can be depicted as in Figure 2.

The set of career sets required for typing a phenomenon can be derived by tracing typed edges. The set of events that change a phenomenon can be derived by tracing changed_by edges and caused_by edges. Therefore, let reqₚ(p) be the phenomena required by an introduction of a phenomenon p, then reqₚ(p) can be derived by the input. That is,

\[
reqₚ(p) = typed(p) \cup \bigcup_{t \in changed_by(p)} caused_by(t) .
\] (1)

Thus, reqₐ(a) can be derived by the input. That is,

\[
reqₐ(a) = appear(a) \cup \bigcup_{p \in appear(a)} reqₚ(p) .
\] (2)

Where appear(a) denotes the phenomena that directly appear in an artifact a. By constraints on introduction depicted in Figure 2 and Equations (1) and (2), reqₐ of each artifact in the library management system is derived as follows:

\[
\begin{align*}
reqₐ(a) &= \{ p1, p2, p3, p5, p7, p8 \} \\
reqₐ(b) &= \{ p1, p2, p3, p5, p6, p7, p8 \} \\
reqₐ(c) &= \{ p1, p3, p4, p6, p7, p8 \}
\end{align*}
\]
3.2 Search for the Best Introduction Orders of Artifacts

Let $\text{int}_a(A) = A_i \setminus A_{i-1}$ ($1 < i$). It denotes newly introduced artifacts to the $i$th model. We assume that only one artifact is introduced through one refinement step. Then, $\text{int}_a(A) = 1$ ($1 < i$) and $\text{req}_{as}(A_i) = \text{req}_{a}(a) \cup \text{req}_{as}(A_{i-1})$ ($1 < i$) where $a = \text{int}_a(A)$.

In the proposed method, orders that correspond to sequences $(A_i)_{i=0,\ldots,|A|}$ that maximized the effectiveness of the refinement are obtained by Algorithm 1. All orders of artifacts introduction $(\text{int}_a(A))_{i=1,2,3}$ and the numbers of newly introduced phenomena $(\text{int}_{p_i}(A))_{i=1,2,3}$ in each refinement for the library management system is as shown in Table 3. In this case, the result of the algorithm represents the order $(\{a\}, \{b\}, \{c\})$.

The method uses breadth-first search with pruning as shown in the Algorithm 1. A node of the search tree represents an introduction order of artifacts. A structure that represents a node is composed of $\text{as}$ that represents the history of artifacts introduction, $\text{ps}$ that represents the set of phenomena introduced so far, $\text{nums}$ that represents the history of the number of introduced phenomena in each step, $\text{max}$ that represents the maximum of $\text{nums}$, and $\text{rest}$ that represents the number of phenomena not introduced yet.

The function $\text{CertainlyBetter}$ (Line 1–19) checks whether an introduction order of artifacts is certainly effective than the other order. This function is used for pruning (Line 32, 35). The number of phenomena introduced in a later refinement is at most the number of phenomena not introduced yet ($\text{maybe_better}\_\text{rest}$) since $\text{req}_{as}(A_i) \subseteq \text{req}_{as}(A_i)$ for all $i$. Therefore, an order is certainly better if the maximum number of introduced phenomena in each step of the order is at most less than the current maximum of the other order (Line 5–6). If both maximums are equal, the algorithm retries the checking on orders without the artifact corresponds to the maximum number (Line 7–15).
Algorithm 1 Search the Best Introduction Orders of Artifacts

1: function CertainlyBetter(maybe_better, maybe_worse)
2: if \((\text{maybe\_better.nums} = \{\}) \lor (\text{maybe\_worse.nums} = \{\}))\) then
3:   return false \(^\triangleright\) Not sure whether maybe\_better is better
4: else if \(\{\text{max(\{maybe\_better.max, maybe\_better.rest\})) < \text{maybe\_worse.max}\}\}\) then
5:   return true \(^\triangleright\) maybe\_better is certainly better
6: else if \(\text{maybe\_better.max} = \text{maybe\_worse.max}\) then
7:   new mb\_reduced, mw\_reduced
8:   mb\_reduced.nums \(\leftarrow\) maybe\_better.nums \(\setminus\) \{maybe\_better.max\}
9:   mw\_reduced.nums \(\leftarrow\) maybe\_worse.nums \(\setminus\) \{maybe\_worse.max\}
10:   mb\_reduced.max \(\leftarrow\) max(mb\_reduced.nums)
11:   mw\_reduced.max \(\leftarrow\) max(mw\_reduced.nums)
12:   mb\_reduced.rest \(\leftarrow\) maybe\_better.rest
13:   mw\_reduced.rest \(\leftarrow\) maybe\_worse.rest
14: return CertainlyBetter(mb\_reduced, mw\_reduced)
15: else
16: return false \(^\triangleright\) Not sure whether maybe\_better is better
17: end if
18: end function

19: function SearchBestOrder(artifacts, req\_as, n\_artif, n\_phen)
20: orders \(\leftarrow\) \{\{as : \{}, ps : \0, nums : \{}, max : 0, rest : n\_phen\}\}
21: repeat
22: for all order \(\in\) orders s.t. length(order.as) is minimum do
23:   orders \(\leftarrow\) orders \(\setminus\) \{order\}
24: for all a \(\in\) (artifacts \(\setminus\) order.as) do
25:   new neworder
26:   neworder.as \(\leftarrow\) append(order.as, \{a\})
27:   neworder.ps \(\leftarrow\) req\_as(neworder.as)
28:   neworder.nums \(\leftarrow\) append(order.nums, \{neworder.ps \(\setminus\) order.ps\})
29:   neworder.max \(\leftarrow\) max(neworder.nums)
30:   neworder.rest \(\leftarrow\) n\_phen - sum(neworder.nums)
31: if not (\(\exists o \in\) orders s.t. CertainlyBetter(o, neworder)) then
32:   orders \(\leftarrow\) orders \(\cup\) \{neworder\}
33: for all o \(\in\) orders do
34:   if CertainlyBetter(neworder, o) then
35:     orders \(\leftarrow\) orders \(\setminus\) \{o\}
36: end if
37: end for
38: end if
39: end for
40: end for
41: until \(\forall o \in\) orders, length(o.as) = n\_artif
42: return orders
43: end function
4 Discussion

4.1 Related Work

There are some studies on requirement engineering methods for modeling in Event-B. In [9], Problem Frames and Event-B are applied successfully on an industrial project. The authors constructed a problem diagram before modeling in Event-B. They associated elaborations of phenomena in problem diagram with a data refinement in Event-B. The work of [5] proposed an iterative process of requirement modeling and validation. The authors connected reasoning about artifacts with refinement in Event-B. In [13], Event-B specifications are derived from class and state-machine diagrams. However, refinement strategy planning is not covered in these studies.

There have been many studies which aim at deriving formal specification in other methods than Event-B from natural language specifications or diagrams like UML [3,4,6–8,10,12,14,15] but refinement is not considered in these studies either.

The authors of [11] proposed a method to derive an abstract specification of an event. In this study, patterns of correspondences between a KAOS goal model and an event in Event-B are provided. The patterns also consider a part of proof obligations that will be generated. From the point of view of refinement strategy planning, this method transforms a refinement strategy planning for an event into a refinement strategy planning in a KAOS goal model. On the other hand, our method plans refinement strategy of the whole model by considering the constraints on introduction between elements in the system. Thus our approach can be considered as complementary to this work.

4.2 Future Work

Further refinement of the proposed method is the primary part of the future work.

First, we assumed every artifact is not changed through refinement. However, that is not the case in realistic situations. There are many cases that some artifacts are strengthened in concrete models by using newly introduced phenomena.

Refinement of events is also neglected. In realistic situations, many events are refined through refinements. For example, both event “Loaning a reserved books” and “Loaning a book from the open stack” can refine an event that only includes “Add one to loan state” state transition.

Moreover, as we viewed in Section 2, we assumed that the complexity of modeling can be measured only by the number of phenomena. However this criterion is too rough. For instance, the importance of the number of events and that of variables are different. Thus, finer analysis of complexity of modeling is needed.
5 Conclusion

This paper has aimed at resolving complexities in planning of refinement strategy by considering semantics constraints of Event-B. Refinement strategy planning is an important and difficult phase in modeling in Event-B. Therefore, the proposed method facilitates ordinary users to leverage Event-B. Although much work remains as discussed in Section 4.2, we believe this work promotes systematic use of formal specifications, more independently from specific knowledge and skills.

References

Formally Checking Large Data Sets in the Railways

Thierry Lecomte, Lilian Burdy, Michael Leuschel

1 ClearSy, Aix en Provence, France
Thierry.lecomte@clearsy.com

2 Formal Mind, Dusseldorf, Germany
leuschel@cs.uni-duesseldorf.de

Abstract. This article presents industrial experience of validating large data sets against specification written using the B / Event-B mathematical language and the ProB model checker.

Keywords: B mathematical language, ProB model checker, data validation.

1 Introduction

Historically, the B Method [1] was introduced in the late 80’s to design correctly safe software. Promoted and supported by RATP1, B and Atelier B, the tool implementing it, have been successfully applied to the industry of transportation. Figure 1 depicts the worldwide implementations of the B technology for safety critical software, mainly as automatic train controllers for metros. Today, Alstom Transport Information Solutions, Siemens Transportation Systems and Technicatome-Areva are the main actors in the development of B safety-critical software. They share a product-based strategy and reuse as much as possible existing B models to develop future metros.

In the mid ‘90s Event-B [2] enlarged the scope of B to analyse, study and specify not only software, but also whole systems. Event-B has been influenced by the work done earlier on Action Systems [13] by the Finnish School (Action System however remained an academic project). Event-B is the synthesis between B and Action System. It extends the usage of B to systems that might contain software but also hardware and pieces of equipment. In that respect, one of the outcome of Event-B is the proved definition of systems architectures and, more generally, the proved development of, so called, “system studies” [7][8][9][10][11], which are performed before the specification and design of the software. This enlargement allows one to perform failure studies right from the beginning in a large system development. Event-B has been used to perform system level safety studies in the Railways [12],

1 Régie Autonome des Transports Parisiens : operates bus and metro public transport in Paris
allowing to formally verify part of the whole system specification, hence contributing to improve the overall level of confidence of the railways system being built.

Fig. 1. Worldwide implementations (2012) of systems embedding software generated from B models.

However, if the verification of Event-B system specification or B software specification is quite easily reachable by semi-automated proof\(^2\), verifying embedded data against properties\(^3\) may turn out to be a nightmare in case of large data sets. For the Meteor metro (line 14, Paris), software and data were kept together in a B project [5]. Demonstrating data correctness regarding expected properties was really difficult as it requires to iterate over large sets of variables and constants (and their domains) and the Atelier B main theorem prover\(^4\) is not designed for this activity [6], that requires more a model checker or constraint solver rather than a theorem prover. Later on, software and data started to be developed and validated within two different processes, in order to avoid a new compilation if the data are modified but not the software. Data validation started to be entirely human, leading to painful, error-prone, long-term activities (usually more than six months to manually check 100 000 data against 200 rules)

In this article, we present a formal approach, based on the B/Event-B mathematical language and the ProB model checker and constraint solver, designed and experimented by Alstom Transport Information Solutions for the validation of railways data.

\(^2\) Automatic theorem provers usually demonstrate 90-95 % of « well written” B models, the remaining has to be demonstrated during interactive sessions with the tool.

\(^3\) In the case of a metro, these data may represent the topology of the tracks, the position of the signals, switches, etc.

\(^4\) That is used by both Atelier B and Rodin platform
2 The Genesis

Verifying railways systems covers many aspects and requires a large number of cross-verification, performed by a wide range of actors including the designer of the system, the company in charge of its exploitation, the certification body, etc. Even if complete automation is not possible, any automatic verification is welcome as it helps to improve the overall level of confidence. Indeed a railways system is a collection of highly dependent sub-system specification and these dependencies need to be checked. They may be based on railways signalling rules (that are specific to every country or even every company in a single country), on rolling stock features (constant or variable train size or configuration) and exploitation conditions.

In France, AQL RATP laboratory initiated the development of a generic tool, OVADO\(^5\), to verify trackside data for the metro line 1 in Paris that is being automated\(^6\). This tool, based on the PredicateB predicate evaluator\(^7\), is able to parse data (XML, csv or text-based formats), load rules and verify that data complies with rules. Initially tested on line 13 configuration data, the tool has been able to check 400 definitions and 125 rules in 5 minutes. In Fig 2, we see on the left a data (called E_a_trainDynamicDeparture_minimum_speed) that associates to a train (referred to by an integer index) its minimum speed (a floating point value). It is declared as a total function, indexes and minimum speed being reachable in an excel file (A7 containing the first index and AM7 the first minimum speed). On the right, a named property is described in natural and in mathematical languages. This property may refer to data previously defined.

\[
\text{Fig. 2. Example of data definition and property}
\]

However the PredicateB tool is just a calculator able to manipulate B/Event-B mathematical language predicates: it is not able to find all possible values for any non-deterministic substitution or to find all counter-examples. Moreover the way the errors are displayed may lead to difficult analysis when the faulty predicate is complex.

\(^5\) Tool for checking the B properties on railway invariants, initially developed by ClearSy
\(^6\) A specific tool, initially developed for validating line 14 data, representing more than 300 000 lines of C++ code, was too difficult to maintain and to adapt to other lines. It was not reused for other lines.
\(^7\) Hosted by the Rodin SVN Sourceforge service (http://rodin-b-sharp.svn.sourceforge.net)
During the DEPLOY project\(^8\), the University of Düsseldorf and Siemens Transportation Systems have elaborated a new approach, based on the ProB model checker to dramatically reduce validation duration from about six months to some minutes [3][4]. Data is extracted from ADA source code and properties come from B models. In the case of the San Juan project, 79 files with a total of 23,000 lines of B are parsed to extract 226 properties and 147 assertions. The verification took 1017 seconds and led to the discovery of 4 false formulas. ProB was then experimented with great success on several projects: Roissy Charles de Gaule airport shuttle, Barcelona line 9, San Paolo line 4, Paris line 1 and Algiers line 1. At that occasion, ProB was slightly improved in order to deal with large scale problems and well validated in order to ease its acceptance by a certification body. However analysing false properties remains difficult. In Fig 3, a failed invariant is listed on the left (the one that is rewritten as false) while the counterexample is shown on the right (the values used for the data that lead to the breaking of the invariant).

![Fig. 3. A false property and its graphical representation](image)

### 3 DTVT

Alstom Transport Information Solutions decided to experiment a new approach by reusing successful features of previous experiments. A new tool, DTVT, is defined and implemented. Its structure is presented in figure 4.

![Fig. 4. DTVT tool structure](image)

\(^8\) [http://www.deploy-project.eu/](http://www.deploy-project.eu/)
Input data is in csv format. Data items are identified through their container file and their name. For example, Curvatures_Cap!BeginValueCm refers to the variable BeginValueCm in the file Curvatures_Cap.xls (see figure 5).

Supported basic types are INT, BOOL and STRING. Data items are sequences of these basic types. Values are extracted from xls files (see figure 6, the positions are expressed in centimeters).

The verification rules are expressed using the B mathematical language and structured as B operations. Instead of having to deal with too large, quantified predicates, a verification rule is decomposed in small steps that allow displaying accurate error message helping to determine the source of the error.

A rule is composed of one or several COUNTEREXAMPLEs. COUNTEREXAMPLEs are evaluated in the order they are defined. Keyword COUNTEREXAMPLE is followed by a formatted message (%1, %2, %3, etc.)
represent the value of the first, second, third parameter of the following ANY substitution).
The ANY substitution allows to filter data or to calculate values. In figure 7, the first rule computes the number of couples of the sequence ATC_Equipment_Type whose second element is the string “Trackside OMAP”.
The ANY substitution is followed by an EXPECTED field. If some values of the parameters of the ANY substitution satisfy the predicate of that substitution but don’t satisfy the predicate of the EXPECTED field, the error message is displayed with its parameters instantiated. In figure 7, the error message of the second rule displays the value of urbalisSectorID (%1) and nb (%3).

ProB is the central tool for the verification. It has been modified in order to produce a file containing all counter examples detected (see figure 5) and slightly improved to better support some B keywords.

DTVT has been experimented with success on several ongoing developments (Mexico, Toronto, Sao Paulo, and Panama) to verify up to 50,000 Excel cells against up to 200 rules. A first round allowed defining required concepts, intermediate constructs (predicates used by several rules) and formalizing a set of generic rules that are shared by all projects. During the next rounds, specific project rules and data files were added. A complete verification is performed in about 10 minutes, including the verification report. The process is completely automatic and can be replayed without any human intervention when data values are modified.

5 Conclusion

Data validation appears to be of paramount importance in safety critical systems. The results obtained in this domain during the DEPLOY project have allowed to create and experiment with success on real scale projects a method for validating data against properties, based on the ProB model-checker and constraint solver.
References

5. Patrick Behm, Paul Benoit, Alain Faivre, Jean-marc Meynadier (1999), *Météor: A Successful Application of B in a Large Project*
A recipe for timed Event-B specifications

Alexei Iliasov, Jeremy Bryans

Newcastle University

Abstract. We present a novel approach to the description of real-time requirements in Event-B, based on the relativistic time model of Gottfried Leibniz. The approach is surprisingly useful, and has led to some significant results. We illustrate the approach with several modelling recipes for the specification of real-time systems in Event-B.

1 Introduction

In the design and modelling of systems from user specifications, it is common to find some proportion of the user requirements expressed in terms of real-time. In work on business information systems, for example, real-time requirements are a natural way to express high-level constraints on business processes [8]. In scheduling or performance analysis, real-time is the natural language for stating requirements.

We use the Event-B language [5] to explore an alternative model of time, the Leibnizian model [11]. According to Leibniz, time is not a fundamental dimension, but is used to distinguish the changes in an observed entity. In the Newtonian model time is an observable attribute of an entity, and may be used to distinguish an entity in the past from an entity in the future, even if the entities are otherwise identical. In the Leibnizian model, in which time is not a directly observable attribute, these may only be distinguished if some other observable attribute has changed. In other words, in the Leibnizian model, time-related changes are transformations of the entity itself. If nothing changes, time is not observed to pass, and therefore (to the observer) time does not pass. The Newtonian model permits time to change without a change in the observed entity.

The dichotomy of the Leibnizian model, in which two separate entities are necessary in order to define the notion of time, suggests that all the time-related properties may be isolated in the observer part leaving the part being observed to deal with functional properties. This has important practical implications: the formulation of timing constraints does not have to be notationally tied with the description of behaviour so that existing methods, semantics and tools may be employed in specifying functional properties.

This difference in time interpretation has significant consequences for the definition of a timed semantics, and for the specification of timing constraints in Event B. We present the semantic model briefly, using examples to illustrate the important points. We also give a series of “recipes” to show how the Leibnizian time model could be used by a model developer to introduce time into Event-B developments.
2 Background

An Event-B development starts with a compact, often trivial abstraction. The cornerstone of the Event-B method is a stepwise development that facilitates a gradual design of a complex system via a number of correctness-preserving refinement steps. The general form of an Event-B model (or machine) is shown in Fig. 1. A machine encapsulates a state space, defined by machine variables, and provides transitions on the state, as described by machine events. Events are characterised by a list of parameters $p$, a state predicate $G$ called an event guard, and a next-state relation $S$.

The invariant clause defines the properties of a system, expressed as state predicates, that must be preserved during the system lifetime. The states defined by an invariant are called the safe states of a system. A correct model is proven to never leave its safe states. Data types $s$, constants $c$ and relevant axioms are defined in a separate component called a context, and included into a machine with the sees clause.

The consistency of a machine as well as the correctness of refinement steps is demonstrated by discharging relevant proof obligations which, collectively, define the Event-B proof semantics [5]. The Rodin Platform [18], a tool supporting Event-B, is an integrated environment that automatically generates necessary proof obligations and provides a number of automated provers and solvers along with an interactive proof environment.

An Event-B machine defines a state transition system. Let $\Omega = \{v \mid I(c, s, v)\}$ be the (safe) states of a machine where $v$ and $I(c, s, v)$ are the variables and the invariant of a machine. The relational form of an event $e$ is $[e]_R \equiv \{v \mapsto v' \mid \exists p \cdot (G_e(c, s, p, v) \land S_e(c, s, p, v, v'))\}$.

**Definition 1 (Event-B transition system).** A machine defines a transition system $(\Omega, f, \omega_0)$ where $f : \Omega \to \mathcal{P}(\Omega)$ is defined as $f = (\bigcup e [e]_R)$; the set of initial states $\omega_0 \subseteq \Omega$ is defined by the initialisation predicate $S_I : \omega_0 = \{v' \mid S_I(c, s, v')\}$. 

\[
\begin{array}{c}
\text{machine } M \\
\text{sees } Context \\
\text{variables } v \\
\text{invariant } I(c, s, v) \\
\text{initialisation } S_I(c, s, v') \\
\text{events} \\
\quad e = \text{any } p \text{ where } G_e(c, s, p, v) \text{ then } S_e(c, s, p, v, v') \text{ end} \\
\end{array}
\]

**Fig. 1.** Event-B model structure.
3 Leibnizian Time

In this section we formally define some essential concepts of the Leibnizian time model. We illustrate them with a timed specification of a lossless buffer, which we return to throughout the paper. For brevity, we omit the theorem proofs. Proofs and machine-checked models of the example are available at [3].

A fundamental concept is that of a process, which we define as a transition system.

**Definition 2 (Process).** A process $P$ is a tuple $(\alpha_P, p, \iota_P)$ where $\alpha_P$ is a process alphabet, $p \subseteq \alpha_P \times \alpha_P$ is a transition relation and $\iota_P$ is the set of initial states.

Time only appears when we put together two processes and let them interact in a certain way. The nature of the interaction is what intuitively may be regarded as an observation of one process by another.

**Definition 3 (Observation connection).** An observation connection between processes $C$ and $S$ is a relation $\varphi \subseteq \alpha_S \times \alpha_C$.

A timed system is formed of pair of processes where one process, an observer, is said to observe another process, a subject. In the definition above, $C$ is an observer and $S$ is a subject.

**Definition 4 (Timed system).** An observer process $C$, a subject process $S$ and an observation connection $\varphi$ define a timed system $C \leq \varphi \leq S$.

The first technique we give extends an untimed Event-B model to a timed system, by defining a timed observer in an associated context. We illustrate this technique in Example 1.

**Recipe 1 (Event-B timed system)** An timed Event-B system $C \cdot \varphi \cdot S$ is a pair of a machine $S$ and context $C$ of the following form.

```
machine S
  sees C
  variables v
  invariant I(v)
  initialisation R(v')
  events
    $E_i =$ any $p_i$ where $G_i(p_i, v)$
    then $S_i(p_i, v, v')$
  end
end

context C
  sets $\alpha_C$
  constants $c, \varphi, \iota_C$
  axioms
    $\iota_C \subseteq \alpha_C$
    $c \subseteq \alpha_C \times \alpha_C$
    $\varphi \subseteq \{v \mid I(v)\} \times \alpha_C$
  ...
end
```
Subject $S$ is an arbitrary Event-B machine defining a vector of variables $v$. Set \( \{v \mid I(v)\} \) defines the possible states of the machine. Observer $C$ is axiomatically defined in a context. The context defines a sort $\alpha_C$, a transition relation $c$ and an observation connection $\varphi$ which relates states from set $\{v \mid I(v)\}$ to observer states. Further axioms and theorems may added, to more precisely characterise the observer model.

Example 1 (Buffer). A *lossy* buffer with the capacity to store one element of type $V$ is defined by machine $BUF$, as shown below.

```
machine BUF
  sees def, $C_0$
  variables b
  invariant $b \in V$
  initialisation $b :\in V$
  events
    wr = any $v$ where $v \in V1$
      then $b := v$
    end
  end
  rd = begin $b := \text{nil}$ end
end
```

The constant $\text{nil} \in V$ and sets $V1 = V \setminus \{\text{nil}\}$, $V1 \neq \emptyset$ are defined in context $\text{def}$. Event $wr$ updates the value of the stored element; event $rd$ consumes a buffered element and sets the buffer contents to $\text{nil}$ to indicate that the buffer is now empty. The events are always enabled and thus $BUF$ permits arbitrary interleavings of the operations. Such operations may be implemented by unsynchronised concurrent activities. The write operation may happen arbitrary often thus potentially overwriting a previous value before it is read.

A *lossless* buffer is defined with the following timed Event-B system.

```
$C_0 \cdot \varphi \cdot Buf$
```

The observation model rules out the possibility of event $wr$ writing into a non-empty buffer. We shall substantiate this claim in Example 2.

An interpretation of a timed system gives a precise meaning to the phenomenon of observation. Essentially, an observation prohibits behaviours that an observer does not expect to see.

Definition 5 (Interpretation of a timed system). *Given a timed system* $C \cdot \varphi \cdot S$ *where* $S = (\alpha_S, s, \iota_S)$ *and* $C = (\alpha_C, c, \iota_C)$, *its interpretation is a process* $\mathbb{I}(C \cdot \varphi \cdot S) \equiv (\varphi, \tau(C \cdot \varphi \cdot S), (\iota_S \times \iota_C) \cap \varphi)$.
where transition relation \(\tau(C \cdot \varphi \cdot S) \subseteq (\alpha S \times \alpha C) \times (\alpha S \times \alpha C)\) is such that a mapping \((u \mapsto t) \mapsto (u' \mapsto t') \in (\alpha S \times \alpha C) \times (\alpha S \times \alpha C)\) belongs to \(\tau(C \cdot \varphi \cdot S)\) if and only if the following properties hold

(a) \(u \mapsto u' \in s\) (a transition of a subject process)
(b) \(t \mapsto t' \in c\) (a transition of an observer process)
(c) \(u \mapsto t, u' \mapsto t' \in \varphi\) (subject and observer transitions are linked via the observation connection)

One could say that an observer is a historian with a preconceived idea about subject process behaviour. An observer would not tolerate a subject that does not follow a certain plan or timetable. Note the use of \(\varphi \subseteq \alpha S \times \alpha C\) to define the alphabet of a timed system interpretation. Whenever we speak about a timed system we always imply, unless specifically indicated otherwise, that the timed system permits an interpretation.

It is essential to note that (despite the nomenclature) the observer is an integral part of the timed system, and does not have a merely passive role. The observer characterises the timing constraints that the developer wishes to impose on an otherwise untimed system, and permits only interpretations that conform to these constraints.

Recipe 2 (Consistency) It may happen that a proof of liveness and timing properties is merely a consequence of an incompatibility between the observer and the subject process. This incompatibility results in a vacuous interpretation of a timed system that defines no common state transitions. To avoid this problem, it is sufficient to exhibit an initialisation of the timed system. For a timed system \(C \cdot \varphi \cdot S\) one needs to prove that

\[
\exists x, y : x \mapsto y \in \alpha S \times \alpha C \land x \mapsto y \in \varphi \tag{1}
\]

Condition 1 is called the consistency proof obligation of a timed system. \(\square\)

The consistency condition holds for the system in Example 1; one possible witness is mapping \(\text{nil} \mapsto \text{nil}\).

We give now the condition under which an event may be safely removed from a timed system without affecting the overall behaviour.

Recipe 3 (Relation empty) Consider a timed system \(C \cdot \varphi \cdot S\) with Event-B machine \(S\) defining some event \(E_i\):

\[
E_i = \text{any } p_i \text{ where } G_i(p_i, v) \text{ then } S_i(p_i, v, v') \text{ end}
\]

Let \(S'\) be a machine identical to \(S\) except that \(E_i\) is suppressed:

\[
E_i = \text{any } p_i \text{ where } \bot \text{ then } S_i(p_i, v, v') \text{ end}
\]

Timed systems \(C \cdot \varphi \cdot S\) and \(C \cdot \varphi \cdot S'\) are equivalent provided the following condition is satisfied.
\( (\varphi[\text{before}(E_i)] \times \varphi[\text{after}(E_i)]) \cap c = \emptyset \) \hfill (2)

where \( \text{before}(e) \) corresponds to the enabling states defined by an event guard and \( \text{after}(e) \) is a set of possible new states computed by an event:

\[
\begin{align*}
\text{before}(e) &= \{ v \mid I(v) \land \exists p_i \cdot G_i(p_i, v) \} \\
\text{after}(e) &= \{ v' \mid I(v) \land \exists p_i \cdot (G_i(p_i, v) \land S_i(p_i, v, v')) \}
\end{align*}
\]

The technique allows one to prove that after removing event \( E_i \) the overall timed system does not become less live since the \( E_i \) is already prevented from occurring by an observer.

**Example 2 (Buffer, contd.).** We can apply the event removal technique to prove that timed system \( C_0 \cdot \varphi \cdot \text{BUF} \) from Example 1 does indeed define a lossless buffer.

To make the buffer lossless, we need to rule out the possibility of event \( wr \) writing into a non-empty buffer. That is, event \( wr \) should not happen when \( b \neq \text{nil} \). Event \( wr \) may be represented (via a trivial case of refinement) by the following two events.

\[
\begin{align*}
\text{wr} &= \text{refines wr any } v \text{ where } b = \text{nil} \land v \in V1 \text{ then } b := v \text{ end} \\
\text{owr} &= \text{refines wr any } v \text{ where } b \neq \text{nil} \land v \in V1 \text{ then } b := v \text{ end}
\end{align*}
\]

It is possible to prove that \( \text{owr} \) is not a part of the timed system \( C_0 \cdot \varphi \cdot \text{BUF} \) by showing that Condition 2 holds for \( \text{owr} \):

\[
(\varphi[\text{before}(\text{owr})] \times \varphi[\text{after}(\text{owr})]) \cap c = \emptyset
\]

which expands to \( \varphi[\{ b \mid b \in V1 \land (\exists v \cdot v \in V1) \}] \times \varphi[\{ b' \mid b \in V1 \land (\exists v \cdot v \in V1 \land b' = v) \}] \cap c = \emptyset \). Since \( V1 \) is not empty we have that \( \exists v \cdot v \in V1 \Leftrightarrow \top \) and also \( V1 = \{ b \mid b \in V1 \} \). The condition simplifies to \( \varphi[V1] \times \varphi[V1] \cap c = \emptyset \Leftrightarrow \varphi[V1] \times \varphi[V1] \cap (V \times V \setminus (V1 \times V1)) = \emptyset \Leftrightarrow \top \). Hence, we can replace machine \( \text{BUF} \) in \( C_0 \cdot \varphi \cdot \text{BUF} \) with the following machine \( \text{BUF}' \):

\[
\text{machine } \text{BUF}'
\]

\[
\begin{align*}
\ldots
\text{events}
\text{wr} &= \text{any } v \text{ where } b = \text{nil} \land v \in V1 \text{ then } b := v \text{ end} \\
\text{rd} &= \text{begin } b := \text{nil} \text{ end}
\end{align*}
\]

It is trivial to see that \( \text{BUF}' \) defines a lossless buffer. Hence, \( C_0 \cdot \varphi \cdot \text{BUF} \) is also a lossless buffer. \( \square \)

It is often advantageous to deal with an observer that is cooperative enough to completely accept any execution of a subject process. Then one knows a priori that something happens in a subject process for every possible point of time defined by an observer.
Definition 6 (Strictness). A timed system \( \mathcal{A} = (\alpha C, c, \iota C, s, \iota S) \) is strict if for every \( u \mapsto t \in \alpha S \times \alpha C \) and \( t \mapsto t' \in c \) there exists some \( u' \) such that \( (u \mapsto t) \mapsto (u' \mapsto t') \in \tau \mathcal{A} \) and \( \iota C \subseteq \varphi[i S] \).

In a system with a strict observer, an observation connection is also a simulation relation [3].

Example 3 (Buffer, contd.). Observer \( C_0 \) permits a concise abstraction however there is an even simpler observer that achieves the same effect. Notice that \( C_0 \leq \varphi \leq \text{BUF} \) defines three transitions classes: reading a value and setting buffer to 0 (\( V^+ \times \{\text{nil}\} \)); reading an empty buffer (\( \{\text{nil} \mapsto \text{nil}\} \)); writing into an empty buffer (\( \{\text{nil}\} \times V^+ \)). We shall exploit this property and define a new observer \( C_1 \) such that these three classes are the kernels of new observation connection \( \varphi_1 \):

\[
\text{context } C_1 \\
\text{extends def } \\
\text{sets } \alpha C_1 \\
\text{constants } c_1, \varphi_1, \iota C_1, E, F \\
\text{axioms} \\
\quad \text{partition}(\alpha C_1, \{E, F\}) \\
\quad \iota C_1 = \{E, F\} \\
\quad c_1 = \{E \mapsto E, E \mapsto F, F \mapsto E\} \\
\quad \varphi_1 = V^1 \times \{F\} \cup \{\text{nil}\} \times \{E\}
\]

It is not hard to see that event removal condition also holds for \( C_1 \cdot \varphi_1 \cdot \text{BUF} \); \( \varphi_1[\text{before(our)}] \times \varphi_1[\text{after(our)}] \cap c_1 = \emptyset \Leftrightarrow (\{F\} \times \{F\}) \cap c_1 = \emptyset \Leftrightarrow \top \). It is easy to see that, unlike \( C_0 \cdot \varphi \cdot \text{BUF} \), system \( C_1 \cdot \varphi_1 \cdot \text{BUF} \) is strict. □

The fourth recipe allows a developer to show that a state which is possible in the untimed process is ruled out by the timing constraints. We give the theory of the technique and demonstrate it with a simple example.

Recipe 4 (Point empty) Consider a timed system \( C \cdot \varphi \cdot S \) and a subject state \( w \in \alpha S \). If one can show that \( \varphi \) does not project \( w \) into anything at all in \( \alpha C \) then, by the Definition 5 of timed system interpretation, any state \( \chi \in \alpha C \times \alpha S \) where \( \text{pr}_{\{2\}}(\chi) = \{w\} \) is not a state of \( C \cdot \varphi \cdot S \).

Thus, a subject state not projected by \( \varphi \) is not reachable in a timed system. A proof that assumes the existence of such a state may be discharged by deriving a contradiction with the following rule.

\[
\forall W : W \subseteq \Omega \land \varphi[W] = \emptyset \Rightarrow \bot \tag{3}
\]

where \( \Omega = \{v \mid I(v)\} \) is the set of subject states. □

Example 4 (Mutex). In this example we describe a very simple mutual exclusion algorithm that works due to a rigid scheduling of the involved threads. The state of a thread \( p \) is defined by \( s(p) \) and is one of the following values: ’out’, denoting
that \( p \) is outside of a critical section and not trying to enter it; 'prep', telling that the thread is about to enter the critical section; and 'in' for the states when the thread is in the critical section.

\[
\text{machine MTX}
\]

\[
\text{variables } s
\]

\[
\text{invariant}
\]

\[
\begin{align*}
\text{inv}1 : s \in P &\rightarrow \{\text{out}, \text{prep}, \text{in}\} \\
\text{inv}2 : \text{card}(s^{-1}[\{\text{in}\}]) &\leq 1
\end{align*}
\]

\[
\text{initialisation } s := P \times \{\text{out}\}
\]

\[
\text{events}
\]

\[
\begin{align*}
\text{prepare} &= \text{any } p \text{ where } p \in P \land s(p) = \text{out} \text{ then } s(p) := \text{prep} \text{ end} \\
\text{enter} &= \text{any } p \text{ where } p \in P \land s(p) = \text{prep} \text{ then } s(p) := \text{in} \text{ end} \\
\text{leave} &= \text{any } p \text{ where } p \in P \land s(p) = \text{in} \text{ then } s(p) := \text{out} \text{ end}
\end{align*}
\]

where set \( P \) of processes is finite. Invariant inv2 expresses the property of mutual exclusion. We employ the following observer process to define that no two processes may be, at the same time, at stages 'prep' and 'in':

\[
\text{context } C
\]

\[
\begin{align*}
\text{...}
\end{align*}
\]

\[
\begin{align*}
\text{c} \subseteq aC \times aC \\
S = \mathcal{P}(P \times \{\text{out}, \text{prep}, \text{in}\}) \\
\varphi \subseteq S \times aC \\
\text{axm5 : } &\forall t, q \cdot t, q \in P \land t \neq q \Rightarrow [a(t) = \text{prep} \land a(q) = \text{in}] = \emptyset
\end{align*}
\]

end

where \( [P(\omega)] = \varphi[\{\omega \mid P(\omega)\}] \). The only non-trivial proof obligation in this model is the preservation of inv2 by event enter. It asks to prove, for some process \( p \), that entering the critical does not violate safety invariant inv2.

\[
\text{card}(s^{-1}[\{\text{in}\}]) \leq 1 \land s(p) = \text{prep} \models \text{card}((s \leftrightarrow \{p \mapsto \text{in}\})^{-1}[\{\text{in}\}]) \leq 1
\]

The condition cannot be discharged within the scope of the subject model alone. We need to bring in the constraints of the observer model to demonstrate the condition. We proceed by replacing \( \text{card}((s \leftrightarrow \{p \mapsto \text{in}\})^{-1}[\{\text{in}\}]) \leq 1 \) with a stronger goal \( s^{-1}[\{\text{in}\}] = \emptyset \) and continue with a proof by contradiction. The negation of \( s^{-1}[\{\text{in}\}] = \emptyset \) in hypothesis gives

\[
s(p) = \text{prep} \land s(x) = \text{in} \land x \neq p \models \bot
\]

A state where one process is in the critical section and the other is about to enter the critical section is disallowed by the observer (axm5) so that the point empty technique may be used to discharge the condition. Instantiating axm5 with \( t = p, q = x \) we have \( \varphi[\{a \cdot a \in S \land a(p) = \text{prep} \land a(x) = \text{in} \mid a\}] = \emptyset \) which gives us set \( W \) to instantiate Condition 3 and derive a contradiction in hypothesis.
One way to realise observer $C$ is by defining it to be cyclic scheduler that allows processes to access the critical section at fixed time intervals.

**Recipe 5 (Point merge)** This technique is a generalisation of the empty point technique. It is used to derive a contradiction when a subject state, defined by the intersection of states of two or more concurrent threads disagrees with the observation model. The following lemma states how to make a transition from a set of statements about individual thread states to a statement about a time point when such a state configuration may be observed.

**Lemma 1 (Point merge).** Let $W$ and $P_i$ be non-empty subject process states such that $W = \{v \mid W(v)\}$, $P_i = \{v \mid P_i(v)\}$ where $W(v)$ and $P_i(v)$ are predicates over subject process state space and it holds that $W \Rightarrow \bigwedge_i P_i$. Then there exist time points $t_i \in [P_i] \cap [W]$ such that $\forall i, j: t_i = t_j$.


The proof technique is to show that no two states from $P_i$ and $P_j$, $i \neq j$ may be observed at the same time (due to some timing conditions). Then the existence of a time point common for the two states $P_i$ and $P_j$ gives a contradiction.

We have applied the point merge technique in the proof of Fischer’s timing-based algorithm of mutual exclusion [19, 4]. The complete Event-B development of the algorithm is available at [2].

4 Discussion

We have presented a summary of our ideas on how the Leibnizian model of time may be used to construct timed Event-B specifications. Our approach offers a homogenous technique to time modelling where properties of timed models are expressed and proven in a gradual, refinement-based manner. The approach is a conservative extension of Event-B. No notational or semantical changes are necessary and the existing modelling tools have proven adequate.

Our technique does not dictate any specific time domain: we let a modeller choose the most appropriate abstraction of time – a simple scheduler, a fictious integer clock or a dense time clock. Both dense and discrete time domains are supported so that the approach may be used as a part of a toolchain with a wide range of potential roles including expressing scheduling properties and hard real-time constraints. The approach has proven to be quite efficient and intuitive: we were able to tackle several large case studies and, as far as we are aware, our models are simpler and require a lower verification effort while all proofs are completely machine-checked.

Due to space constraints, we did not present a larger case study although one such case study is available at [2]. Many recipes were not discussed. These include rules for demonstrating the realisability of a timed specification and several refinement-related recipes. We plan to provide a plug-in to the Rodin Toolkit [18] for automated generation of the timed systems proof obligations and a template-based assistant for constructing various kinds of observer processes.
References

Dependability-Explicit Engineering with Event-B: Overview of Recent Achievements

Elena Troubitsyna
Åbo Akademi University, Turku, Finland
{Elena.Troubitsyna}@abo.fi

Abstract. Event-B has been actively used within the EU Deploy project to model dependable systems from various application domains. As a result, we have created a number of formal approaches to explicitly reason about dependability in the refinement process. In this paper we overview the work on formal engineering of dependable systems carried out in the Deploy project. We outline our approaches to integrating safety analysis into the development process, modelling fault tolerant systems and probabilistic dependability evaluation. We discuss achievements and challenges in development of dependable systems within the Event-B framework.

Keywords: Formal modelling, dependability, safety, fault tolerance, Event-B, refinement, probabilistic verification.

1 Introduction

Nowadays we tend to place increasing reliance on computer-based systems and software which they are running. The degree of reliance that we can justifiably place on a system is expressed by the notion of dependability [1]. However, the analysis of recent software-caused accidents has shown that the current development process is inadequate for achieving high degree of dependability. While a number of existing methods and tools address certain aspects of dependable systems development, there is still a lack of a general viable dependability-explicit techniques for developing software for complex systems.

To address this issue, in the FP7 EU Deploy project [2] we have proposed a number of approaches that allow the designers to explicitly address dependability throughout the entire system development by refinement in Event-B [3]. In this paper we briefly overview the approaches that have been mainly proposed by the researchers from Åbo Akademi University. The goal of this paper is to present some evidences that Event-B constitutes a suitable framework for formal dependability-explicit development.

2 Engineering Dependable Systems with Event-B

2.1 Event-B
Currently, complexity is perceived as the main threat to dependability. To cope with the system complexity, we need scalable formal techniques to explicitly
address various dependability aspects throughout the entire development cycle. It is widely recognised that system complexity can be managed via abstract modelling, decomposition and iterative development. Event-B [3] is a formal top-down development approach to correct-by-construction system development. Development in Event-B starts from defining a high-level specification that represents the system behavior and properties in a highly abstract way. The main development technique – refinement – allows us to ensure that a concrete specification preserves the globally observable behaviour and properties of the abstract specification, i.e., verify correctness with respect to the abstract specification. Verification of each refinement step is done by proofs. The Rodin platform [4] automates modelling and verification in Event-B.

2.2 Dependability in System Development

The notion of dependability encompasses a wide range of system properties. Traditionally, dependability can by characterised by such attributes [1] as reliability, safety, availability, maintainability, confidentiality and integrity. In the Deploy project, the main focus has been on developing techniques addressing safety, reliability and availability.

The system dependability is impaired by failures, errors and faults [1]. To break the chain of propagation of a fault – a physical defect or malfunction of a system component – towards the system boundary, the system designers employ a variety of techniques to avoid and remove faults, as well as tolerate and forecast them. Let us now discuss the ways in which Event-B facilitates development of dependable systems.

The main purpose of fault prevention (or fault avoidance) techniques is to avoid occurrence or introduction of faults during the development process. Development in Event-B allows the designers to better understand the system requirements and properties and express them in precise mathematical way. The verification that proceeds hand-in-hand with the modelling enables early identification of design errors and avoid dependability-impairing failures.

Fault tolerance methods are used to design a system in such a way that it is capable of functioning despite the presence of faults. While specifying fault tolerant systems in Event-B, we model not only nominal system behaviour but also failure occurrence and fault tolerance as an intrinsic part of the system specification. It allows us to formally underpin fault assumptions and rigorously define fault tolerance mechanisms.

Fault removal is a set of techniques for identifying and removing the causes of errors. The fault removal process at the development stage starts with system verification, which is followed by diagnosis and correction steps. While modelling systems in Event-B, we rely on proofs, probabilistic extension of Event-B and associated probabilistic model checking to verify correctness of functional behaviour and satisfaction of the desired dependability attributes.

Fault forecasting aims at evaluation of the impact of fault occurrence and activation on the system behaviour. Such an evaluation has qualitative and quantitative aspects. The qualitative analysis helps to designate and classify failure
modes as well as identify combinations of faults of components that may potentially lead to a system failure. We have demonstrated that how a seamless integration between Event-B and various techniques for safety analysis facilitate qualitative assessment of the impact of faults on the system dependability. The probabilistic extension of Event-B allows for the quantitative assessment of to what extent certain attributes of dependability are satisfied.

Therefore, we believe that Event-B constitutes a suitable and versatile framework for creating a rigorous dependability-explicit development process. Next we overview in a more details our contributions to attaining establishing dependability-explicit development process with Event-B.

3 Formal development of fault tolerant mode-rich systems

A widely used mechanism for achieving fault tolerance is based on the notion of modes. In our work [10–12], we have proposed an approach to formal development of fault tolerant mode-rich systems. Such systems achieve fault tolerance by rollbacking to specific degraded modes. The proposed formal development process allows the designers to develop a system in a layered fashion. Essentially, it consists of a number of steps gradually unfolding system architectural layers by refinement. Moreover, we prove the consistency between mode transitions on adjacent architectural layers, which significantly improves scalability of verification. It has been noted that testing and model checking of the systems with complex mode transition schemes suffers from poor scalability. We have overcome this problem by relying on incremental verification of global mode consistency properties by proof and hence achieved a good scalability.

In our approach to modelling mode-rich systems [10–12], we have focused on verification of consistency of a predefined mode logic. In [13], we have proposed to conduct Failure Modes and Effects Analysis (FMEA) of each operational mode to identify mode transitions required to implement fault tolerance. Fault tolerance is achieved by two different means – transitions to a more degraded mode and dynamic reconfiguration using redundant components. Furthermore, we have investigated a complex interplay between the states of components during reconfiguration and the system modes.

4 Goal-oriented refinement of reconfigurable systems

In [5, 6, 17], we have investigated the problem of ensuring safety and fault tolerance of mobile agent systems. The work has resulted in defining the modelling patterns to represent agent roles in dynamic scopes and deriving the logical conditions to ensure system dependability.

In [7], we have continued our study of multi-agent systems and have proposed a goal-oriented approach to development of multi-agent systems. It is currently
recognized that the goal-oriented development facilitates design of complex dynamically adaptable systems. In goal-oriented development the system requirements are defined in terms of goals – the functional and non-functional objectives that a system should achieve. Often changes in system operational environment, e.g., caused by failures of agents – independent system components of various types – might hinder achieving the desired goals. In [ADA] we have proposed a formal development approach that ensures goal reachability “by construction”. Essentially, our approach allows the developers to define system goals at different levels of abstraction and guarantee goal reachability despite agent failures. We have derived refinement patterns modelling the mechanisms for dynamic system reconfiguration by reallocating goals from failed agents to healthy ones and, per se, guarantee dependability. We believe that our approach offers a scalable technique for formal development of dynamically reconfigurable dependable systems.

While refining a reconfigurable system, we had to assume that sufficient amount of agents would remain operational to achieve the desired goals. In [8], we have demonstrated how to integrate probabilistic analysis to quantitatively assess the likelihood of goal reachability despite failures. The rigorous refinement process has allowed us to establish the precise relationships between component failures and goal reachability. We have assessed the derived reconfigurable system architecture to quantitatively verify that it achieves the desired reliability and performance objectives. This was accomplished by relying on the probabilistic extension of Event-B to verify reliability and performance properties using PRISM model checker [9].

5 Integrating Safety Analysis into Formal Development

In [14], we have demonstrated how to combine formal modelling and refinement with Failure Modes and Effects Analysis (FMEA). We have defined a set of patterns formalising the requirements derived from FMEA as well as automated their integration into the formal specification. The proposed approach facilitates formal development and improves traceability of safety requirements. The approach proposed in this paper allows us to automate the formal development process via two main steps: choice of suitable patterns that generically define FMEA result, and instantiation of chosen patterns with model-specific information. Our approach allows the developers to verify (by proofs) that safety invariants are preserved in spite of identified component failure modes. Hence we believe that it provides a useful support for formal development and improves traceability of safety requirements.

The use of an evidence generated from formal analysis is still an open issue in the system certification process. Sometimes the formal proofs deemed to be too complex and cause doubts regarding their trustworthiness as the evidence in safety cases of safety-critical systems. Another open issue related to the formal modelling process is whether the obtained formal model adequately represents safety requirements described in a system specification. In our work[18] we pro-
posed an approach to linking formal modelling in Event-B with safety cases. We give the classification of safety requirements and define how each class can be represented in a formal specification. The approach allows the developers to obtain a consistent system specification that facilitate deriving a sufficient safety case.

The systems, whose components are susceptible to various kinds of faults, never are "absolutely" safe, i.e., certain combinations of failures may lead to an occurrence of a hazard – a potentially dangerous situation breaching safety requirements. To demonstrate that the probability of a hazard occurrence is acceptably low, in [15] we have presented a formal approach to integrating quantitative safety analysis into formal system development by refinement in Event-B. Essentially, our approach can be seen as a process of extracting a fault tree – a logical representation of a hazardous situation in terms of the primitives used at different abstraction layers. Eventually, we arrive at the representation of a hazard in terms of the failures of basic system components, which allows us to calculate probability of a hazard occurrence. The proposed approach is based on a probabilistic extension of Event-B [16]. It enables development of systems that are not only correct but also safe by construction.

6 Quantitative Assessment of Dependability

To facilitate dependability-explicit development in the probabilistic Event-B [16], we strengthened the notion of Event-B refinement by requiring that a refined model, besides being a proper functional refinement of its more abstract counterpart, also satisfies a number of quantitative constraints. These constraints ensure that the refined model improves (or at least preserves) the current probabilistic measures of system dependability attributes. In our work, these additional constraints are usually derived from the fundamental properties of Markov processes. To validate the proposed approaches, in Deploy we have conducted a number of case studies including formal development and quantitative assessment of a fault tolerant satellite system, formal modelling integrated with safety analysis of a radio-based railway crossing controller, service-oriented system etc. This work allows the designers to evaluate the impact of the chosen design decisions on system dependability.

7 Discussion

Our work on formal engineering of dependable systems in the EU Deploy project has resulted in two types of approaches:

- the approaches that focus on creating modelling patterns and guidelines for representing and verifying certain resilience-related behavior
- the approaches that integrate (external) techniques for safety and reliability analysis into the formal development process of Event-B.
A tight cooperation with the Deploy industrial partners has allowed us to gain rich experience in modelling dependable systems from the transportation, aerospace and business information system domains. The development of industrial-scale systems has emphasized the need for scalability in formal modelling and automatic tool support. It has fostered the research on modularisation and decomposition techniques for Event-B as well as development of various plug-ins. Moreover, it has led to understanding importance of heterogenous modelling techniques to address a variety of dependability aspects.

In general, we believe that Event-B offers a powerful formal technique for engineering dependable systems. To leverage scalability and industrial relevance of the method, we will continue to enlarge the set of modelling patterns for representing various dependability aspects, strengthening automatic tool support and enriching its capabilities via dedicated plug-ins to the Rodin platform.

References

9. PRISM, “Probabilistic symbolic model checker, online at http://www.prismmodelchecker.org/.”


Formal Data Validation with Event-B

Frédéric Badeau
Systerel
1090 rue René Descartes
Parc d’activités de la Duranne
13857 Aix-en-Provence cedex 3
frederic.badeau@systerel.fr

Marielle Doche-Petit
Systerel
1090 rue René Descartes
Parc d’activités de la Duranne
13857 Aix-en-Provence cedex 3
marielle.doche-petit@systerel.fr

October 10, 2012

Abstract

This article presents a verification and validation activity performed in an industrial context, to validate configuration data of a metro CBTC system by creating a formal B model of these configuration data and of their properties. A double tool chain is used to safely check whether a certain given input of configuration data fulfill its properties. One tool is based on some Rodin and open source plug-ins and the other tool is based on ProB.

1 Data configuration of CBTC

A Communication Based Train Control (CBTC) is a system used to safely control metro systems. It allows several train control modes including a fully automatic mode. It must achieve high safety-critical and high availability levels defined in the CENELEC standards (EN 50126 [5], EN 50128 [6], EN 50129 [7]).

A CBTC system comes with a lot of configuration data because these data should describe a large part of the metro railway network. Much equipment of the CBTC should be parametrized with these configuration data. Each piece of equipment typically uses from a few kilo-bytes to one or more megabytes of data.

Pieces of CBTC equipment are controlled by software parametrized by configuration data. Although the process presented here could be applied on both safety-critical and non safety-critical parts, it is only applied on safety-critical parts in order to reduce development costs. Safety-critical software is developed independently from its configuration data, thus a software version has to be validated only once, even if it is instantiated for several pieces of equipment. It also does not have to be validated again after some configuration data change. The data properties as required by a piece of software, are described as requirements in the software interface document. Every set of configuration data also has to be validated once.

For the CBTC system under development, this principle of separation between software and configuration data is taken to the extreme, since configuration data are not built-in into equipment, they are instead dynamically loaded at runtime through configuration messages.

Although configuration data are just built to parametrize some equipment, these data may be partial: each piece of equipment needs only access to its own specific data subset, otherwise some data can be shared between several pieces of equipment. So the development process of configuration data starts by developing a common database of configuration data representing the whole CBTC system. This database should have no or very low redundancy.

2 Validation of configuration data

To validate configuration data we build a B model of the configuration data and of their properties with basic B types and predicates. Then, given an input set of configuration data we use a double tool chain to evaluate all the predicates for these configuration data.

The process of validating configuration data is split into two major activities:

• the preparation phase which consists in modelling data and their properties and then verifying and testing the model,
• the actual validation of a given configuration data set against the obtained model.

The preparation activity takes by far the most time, whereas the actual validation is more about setting everything up and then running automated tools.

2.1 The real database

In one case, data are given in the XML format derived from a UML model. XML is well-suited to be interfaced with the predicates evaluation tools.\(^1\)

However, it is necessary to model and access data from binary messages derived from the XML database, because these messages contain the actual configuration data that are loaded into safety-critical equipment of the CBTC system.

Interfacing predicate evaluation tools directly with binary structures is not recommended, since it would depend too much on a particular binary structure and would be difficult to maintain. It has been decided to develop a tool to convert a binary structure into an XML file. To deal with safety issues, a reverse tool has also been independently developed to convert back the XML file into a binary file. To ensure that the binary/XML converter works fine, we apply the reverse XML/binary converter and check that the result is the same as the binary file we started from.

2.2 Modelling interface constants

The preparation activity starts by interfacing the data of the real database with B constants, which will provide the basic bricks of the formal model.

The following types\(^2\) of constants can be declared as basic interface constants to link elements of the real database to constants of the B properties:

- carrier sets,
- subsets of carrier sets,
- scalar data (mostly integers),
- functions from a scalar type to another scalar type,
- relations between a scalar type and another scalar type,
- functions from a scalar type to functions from a scalar type into another scalar type.

For all these cases, a constant is given by its name, a typing predicate in the B language (except for carrier sets as they define new scalar types) and an XPath request. When these requests are placed on some XML file, they select one value, or a set of values. In the case of functions and relations, several

\(^1\)Our approach can be adapted to other classical database interfaces like SQL, excel,...

\(^2\)Usually two-dimensional functions are enough to model everything we need from the real configuration database.
XPath requests are required, to select first a domain element, and then to select a range element. And in the case of a function of functions three XPath requests are required, two for the domain and one for the range.

Carrier sets are used to define every distinct type of objects handled in the database, such as lines, sections, blocks, block frontiers, track circuits, switches, signals (traffic lights), beacons, etc. In a CBTC system, a block is an elementary linear part of a railroad which defines a local one dimension coordinate system oriented by its two frontiers. By definition, the origin frontier is located at abscissa 0 and the destination frontier is located at abscissa block length.

\[
\text{SETS}
\]

\[
t_{\text{block}};
\]

\[
t_{\text{block}}_{\text{frontier}}
\]

Scalar data are used to define constant numbers of the system, such as special distances, speeds, accelerations, delays.

Functions and relations are used to define links between objects, such as the length of a block, the block a signal is located on, the signal abscissa, the blocks that follow a given block in a given direction. Functions and relations are the most common building units handled by the model.

The following interface constants represent the block frontier located at the origin or at the destination of a block. Destination and origin define the block orientation.

\[
f_{\text{block}}_{\text{orig}} \in t_{\text{block}} \rightarrow t_{\text{block}}_{\text{frontier}} \land
\]

\[
f_{\text{block}}_{\text{dest}} \in t_{\text{block}} \rightarrow t_{\text{block}}_{\text{frontier}}
\]

These interface constants do not have to be a straightforward copy of the real database that could be produced automatically. They are the first step of the model and should be the result of modelling choices in order to make the model as simple and as effective as possible to express properties. For instance, in the XML database a signal has an enumerated attribute to state its type between manoeuvre signal, spacing signal, or permanent signal. Keeping this attribute is something to do at a programming language level. In a set-theoretic model, we prefer to define constant subsets for each type of signal.

Concerning floating point values, we reach a limitation of the B Language, which only supports integers, so we use only fixed point integers isomorphic to integers in trivial way. The XML database contains floating point values. The conversion to integers can be done with XPath operators.

### 2.3 Modelling useful expressions

The second modelling step consists in building a library of useful formal expressions which will serve as building blocks for the whole model. These intermediate useful expressions are called definitions, they are similar to a LET in software-B. A definition is associated with a name, an informal description, and a B expression.

Although interface constants provide a certain flexibility, they may be too limited to define directly the data structures best suited to model properties. For instance, interface constant functions are limited to two-dimensional functions. Definitions based on interface constants are used to define the constants we really want for the B model.

Thanks to the definitions, the B model is easier to read, to understand and to verify. This breaking down mechanism into many intermediate steps does really work fine and does not come with any downside, so it may be used intensively. Especially, there is no limitation on the number of nested definitions used in a B model and the tools do not have any specific time efficiency issues due to nested definitions.

For the CBTC under development, a library of definitions was used to model graph functions. Indeed, a CBTC railway network is represented by an oriented graph of blocks. The following properties hold on blocks:

- A block has two frontiers noted up and down.
- The down frontier is located at the origin of the block at abscissa 0 and the up frontier is located at the destination of the block at abscissa block length.
- A block frontier may be connected to no other block, to one block, or to two blocks if there is a switch at this block frontier and if the block is at the narrow end of the switch. In the latter case, the two following blocks are called left block and right block depending on their position for an observer located on the narrow end and facing the switch point.

The four constants below represent the possible next block connected to a block depending on the direction (upward/downward) and the possible turning direction (left/right).
f_{next\ up\ block} ∈ t\ block → t\ block \land
f_{next\ down\ block} ∈ t\ block → t\ block \land
f_{next\ up\ block} ∈ t\ block → t\ block \land
f_{next\ down\ block} ∈ t\ block → t\ block

These functions are partial functions which allows to take account of the case where there is no following block (a convention then fixes the direction which gives the next block).

We define the relation of the possible next block in a direction (upward/downward) this way:

\[ r_{next\ block} = \text{f\ next\ block} ∪ \text{f\ next\ block} \land \]
\[ r_{next\ down\ block} = \text{f\ next\ down\ block} ∪ \text{f\ next\ down\ block} \land \]

The railway graph is then modelled by a relation associating an oriented block with every oriented block it is connected to. The fact that block orientation is arbitrary, meaning that a block oriented in a direction may be followed in this direction by a block oriented in the opposite direction makes the model more complex. However this is required to model correctly all possible railway network topologies.

Then we define the relation that associates to a block and a direction, each next block in the corresponding direction:

\[ r_{next\ block} = \]
\[ ((r_{next\ up\ block} ∩ (f\ block\ dest; f\ block\ origin^{-1})) \cup \{c\ up\ -> c\ up\}) ∪
((r_{next\ up\ block} ∩ (f\ block\ dest; f\ block\ dest^{-1})) \cup \{c\ up\ -> c\ down\}) ∪
((r_{next\ up\ block} ∩ (f\ block\ origin; f\ block\ dest^{-1})) \cup \{c\ down\ -> c\ down\}) ∪
((r_{next\ up\ block} ∩ (f\ block\ origin; f\ block\ origin^{-1})) \cup \{c\ down\ -> c\ up\})

Then, we build a library of functions dealing with this oriented graph. Some ordering functions define whether a position given by an abscissa on a block is located after another given position, with respect to a certain direction. They all use the iterate operator on the relation graph. Several ordering functions are defined depending on which block direction is used as the reference direction.

We define the relation that associates to a block and a direction, every downstream blocks in the corresponding direction:

\[ r_{block\ chain} = \text{closure}(r_{next\ block}) \]

The following function states whether a position 2 is located afterwards a position 1 in a given direction related to the orientation on block 1.

\[ f_{pos\ afterwards} = \]
\[ %dir1 -> block1 -> abs1 -> block2 -> abs2; \ (dir1 ∈ t\ dir\ ∧
block1 ∈ t\ block\ ∧
abs1 ∈ \text{INTEGER}\ ∧
block2 ∈ t\ block\ ∧
abs2 ∈ \text{INTEGER}\ ∧
bool\ (block1 = block2
dir1 \land
abs2 ≤ abs1, \ c\ up\ -> bool(abs1 ≤ abs2)) (dir1) = \text{TRUE}) ∧
(block1 \neq block2
dir1 \land
∃dir2\ (dir2 ∈ t\ dir\ ∧
(block1 -> dir1) (block2 -> dir2) ∈ r\ block\ chain)) \]

Many properties deal with distances on the railway graph. Although these properties seem straightforward in natural language, they are actually more complex than expected to handle, because there could be several paths between two positions. To handle distances, we define a zone, which is a graph sub-part, and we define library functions giving the zone obtained by starting from a direction and moving on to a certain maximum distance. Several functions are defined to suit all the properties dealing with distances.
2.4 Modelling properties

Every property requirement on configuration data coming from the relevant documents produced during system design should be modelled by a predicate, which is the last step of creating the B model. Every property is given a name, the design requirement it refers to, a predicate and a natural language description of the predicate. The requirement tag is used to build traceability tables in order to ensure that all requirements were properly processed.

Predicates should not be too complex to be easily verified. To do so, definitions should be used intensively. Actually, the production of definitions is partly done bottom-up, from interface constants to properties, and partly top-down, from properties to interface constants and partly by means of refactoring when we realise afterwards how to build a better model.

The following predicate corresponds to the requirement: position protected by a signal is located afterwards the signal regarding the signal direction.

\[
\forall \text{mansig} \cdot (\text{mansig} \in s_{\text{mansig}} \Rightarrow f_{\text{pos Afterwards}}(f_{\text{sig dir}}(\text{mansig}) \Rightarrow f_{\text{sig block}}(\text{mansig}) \Rightarrow f_{\text{sig abs}}(\text{mansig}) \Rightarrow f_{\text{sig prot seg}}(\text{mansig}) \Rightarrow f_{\text{sig prot abs}}(\text{sigman})) = \text{TRUE})
\]

All given requirements have been modelled into predicates using the process described here. However the calculus on integers was not completely satisfactory, even though we worked with fixed point reals, integer computation introduces errors when using integer division. So in some cases, we had to replace an equality predicate, by a definition stating that two integers are distant from less than a given epsilon. Those epsilon values were explicitly put back into XML input files, so that the safety department could agree, or not, with those values. Fortunately, all the computations were pretty basic, so the limitations of the B language were not too much of a constraint. However, if we had to model real floating point calculus with scientific operators like exponential or trigonometry operators, we would need to extend somehow the B language or the tools, but this is beyond the scope of this paper.

2.5 Verifying and testing properties

The B model is built property by property. After producing a property, the model developer exercises it against a nominal example through the tool chain based on Rodin plug-ins, since it is easier to use on unfinished models. If the property check fails, then the problem is tracked down just like software debugging. The Rodin plug-in tool displays the value (true/false) of all sub predicates, so we can explore the predicate, searching for something unexpected. This exploration is usually very effective and leads to the problem, which could be either a data, a model or a property error.

When the development of a property is complete, it should then be checked by a qualified person different from the person who formalised the property. This way, we make sure that the property is adequately formalised. However, experience shows that this process is not robust enough for complex properties. For these properties, we write one or more unit tests in order to test the different kinds of errors that could be expected from the predicate. Usually between one and three unit tests are written. Of course, this test strategy does not intend to be complete, but it is very efficient to point out small errors that could ruin the validation for that property. In one case, a definition providing a set of blocks was erroneously always equal to the empty set, implying that properties using the definition always succeeded.

2.6 Tool results

The model was written with no particular concern about the tools, except for one rule: when defining a variable, instead of first typing it with pure typing predicates and then adding constraint predicates, it is far more efficient to directly type it with a constraint set expression.

In this data validation process, a double tool chain is used in order to mitigate the risk of error in these tools.

- The first tool chain is based on the Rodin AST plug-in and on the predicate evaluator core plug-in. The interface plug-ins are proprietary one.

- The second tool chain is based on the ProB model checker. The interface part is also a proprietary one.

Both tools needed an adjustment of their evaluation strategy due to intensive use of closure and cartesian products with one argument being a large subset of integers. The tools had a tendency to
perform an early evaluation of sets to be more efficient, however such a strategy does not pay off for
closure or cartesian product evaluation for which a lazy strategy proves to be much more efficient.
After taking into account these two issues, both tools were very efficient. The validation of a set of
configuration data containing several millions of integers took only a few minutes. No contradictory
result was produced by the double chain which, given the high differentiation of the development tools
—both in process and implementation techniques— gives a great confidence in the results.

3 Pros and Cons of the Process

3.1 Pros

A major benefit of this work is that the formal modelling activity focuses on the capital issue of under-
standing as clearly as possible the configuration data and their properties in all possible situations
and not only for the simple situations one can think of or for the examples represented by diagrams of
input documents.

One other major benefit of this work is that all properties, however hard to define on the railway
graph, were entirely modelled in B. None was left to external error prone informal verification.

• Well suited for CBTC configuration data: properties are neither too simple nor too complex.
• Reasonable validation time for both tools (a few minutes).
• The tools can handle large data (several megabytes).
• It is far more interesting to focus on high level formal modelling and on debugging data sets than
to hard code verifications.
• It may also take less time to validate configuration data with predicate evaluation tools, than to
develop safety-critical specific tools.

3.2 Limitations

• Ill-suited for scientific calculus.
• From an industrial point of view, we are looking for other domains than railway safety-critical
systems where this process can be applied with the same success.

References

[4] Leuschel, Michael and Butler, Michael *ProB: An Automated Analysis Toolset for the B Method*, Inter-
[5] CENELEC, Railway applications. The specification and demonstration of reliability, availability,
maintainability and safety (RAMS), EN50126.
for railway control and protection systems, EN50128.
[7] CENELEC, Railway applications. Communication, signalling and processing systems. Safety re-
lated electronic systems for signalling, EN50129.