UNIVERSITY OF SOUTHAMPTON

Supporting Development of Event-B Models

by

Renato Alexandre da Cruz Silva

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the

Faculty of Engineering and Applied Science
Department of Electronics and Computer Science

May 2012

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND APPLIED SCIENCE
DEPARTMENT OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

by Renato Alexandre da Cruz Silva

We believe that the task of developing large systems requires a formal approach. The complexity of these systems demands techniques and tool support to simplify the task of formal development. Often large systems are a combination of sub-components that can be seen as modules. Event-B is a formal methodology that allows the development of distributed systems. Despite several benefits of using Event-B, modularisation and reuse of existing models are not fully supported. We propose three techniques supporting the reuse of models and their respective proof obligations in order to develop specifications of large systems: *composition*, *generic instantiation* and *decomposition*. Such techniques are studied and tool support is defined as plug-ins by taking advantage of the extensibility features of the Event-B toolset (Rodin platform).

Composition allows the combination of different sub-components and refinement is possible. A shared event approach is followed where sub-components events are composed, communicating via common parameters and without variable sharing. By reusing sub-components, proof obligations required for a valid composition are expressed and we show that composition is monotonic. A tool is developed reinforcing the conditions that allow the monotonicity and generating the respective proof obligations.

Generic Instantiation allows a generic model (a machine or a refinement chain) to be instantiated into a suitable development. Generic model proof obligations are reused, avoiding re-proof and its refinement comes for free. An instantiation constructor is developed where the generic free identifiers (variables and constants) are renamed and carrier sets are replaced to fit the instance.

Decomposition allows the splitting of a model into several sub-components in a shared event or shared variable style. Both styles are monotonic and sub-components can be further refined independently, allowing team development. Proof obligations of the original model are split into the different sub-components which usually results in simpler and easier to discharge proof obligations. Decomposition is supported by a practical tool permitting the use of both styles.

We expect to close the gap between the use of formal methods in academia and industry. In this thesis we address the important aspect of having tools supporting well-studied formal techniques that are easy to use by model developers.

Contents

List of Figures

List of Tables

Acknowledgements

First of all, I would like to thank my parents António and Idalina Silva for their support when I decided to jump to this PhD adventure. The adventure required me to move to a different country and you always supported me whenever I needed: MUITO OBRIGADO PAI e MÃE.

In terms of work, I cannot express my gratitude enough to my supervisor Michael Butler for the inspiration, guidance, cordiality, support and patience throughout the PhD: THANK YOU VERY VERY MUCH Prof. Michael! I also would like to thank Stefan Hallerstede, Laurent Voisin, Thai Son Hoang, Nicolas Beauger, Carine Pascal, Jean-Raymond Abrial, Issam Maamria and Matthias Schmalz. Moreover John Colley, Lucas Cordeiro, Abdolbaghi Rezazadeh and Colin Snook for the valuable discussions. Professor Benedita Malheiro at ISEP was my inspiration to follow this path in my career which probably would not have happened without the support of Jose Manuel Baptista, also my professor at ISEP: thank you to both. A special thanks to Aryldo Russo (Dinho) for the interesting discussions about railways and for providing the requirements used in the case study chapter. I would also like to acknowledge my examiners, Jim Woodcock and Julian Rathke for their valuable review and suggestions for the improvement of this document. This work would not have been possible without the financial support of Fundação Ciência e Tecnologia (FCT-Portugal) through a Doctoral Degree Grant.

My family in general was very supportive and for that I thank them. In particular my younger sister Eluisa Vanessa Silva, my brother Edson Cláudio Ferreira and Manuela Silva. Also my cousins Bruno Correia, Paulo Silva, Mário Silva, my sisters in law Ana Cotovio, Mónica Cotovio and Raquel Silva not forgetting my beautiful nieces Alexia Silva, Paula Silva, Sarah Ferreira and more recently Debora Silva.

I want to thank my close friends which I consider as my brothers. Despite the distance they always believed that I could do it: Renato Gonçalves, Pedro Sousa, Bruno Rodrigues, António Quiemba, Miguel Kay, Edson Miranda and Tassia Carvalho.

My friends that helped and made my days so much better, for their friendship and for being the way they are: Vasthi Alonso, Erofili Grapsa, Alinne Veiga and more recently Mac Adamarczuk you all are stars! Also I would like to thank my dear friend Rosália Peixoto for the support despite the distance. My lunch buddies Matthew Jones and David Davies for the amazing entertaining time spent. In general the Southampton University Volleyball Club and Solent SGTV Volleyball Club squads: thanks for everything.

Last but not least I would also like to thank my girlfriend (future Dr.) Hannah Warren for the love and support during the PhD. You are definitely one of a kind and I am sure my path without you would not have been the same.

To life and death: my niece Sarah Silva born in the first year of my PhD and my beloved uncle Manuel Silva that past away in the last year of my PhD

Chapter 1

Introduction

This thesis investigates techniques that allow support of formal developments in Event-B [3, 9]. In particular we focus in reuse of developments, in the avoidance of re-proofs and respective tool support. We begin by studying other formal languages and the respective formal support for three techniques: *composition*, *generic instantiation* and *decomposition*. Afterwards and based on the previous study we apply the use of such techniques to Event-B. Case studies and respective tool support for each technique are developed in the *Rodin* platform, an application targeting developments in Event-B.

1.1 Thesis contribution

1.1.1 Overview

We believe that the development of large, complex and/or critical systems should be done using formal methodology. The development of such systems usually is complex and they must be ensured to work as desired avoiding failures that could lead to serious consequences or even life-threatening situations. Formal methods are used to help the development and modelling of these systems, which itself can be a hard task to accomplish. Several formal notations can be used for modelling systems. We use **Event-B**, a recent formal method with growing popularity used for modelling discrete systems. Event-B results from an evolution of other formal methods notations like the B-Method [158] and Action Systems [26]. Event-B is suitable for modelling parallel, reactive and distributed system and not restricted to software development unlike the "parent" B-method, including a richer notion of refinement. As we are mostly interested in distributed systems, this seems a suitable notation to be used.

However as a recent notation, Event-B lacks some features and mechanisms. We address in particular the lack of reusability mechanisms like avoiding proof obligations (POs) re-proof. We believe that reusability is very useful specially in large developments and we address these missing mechanisms.

1.1.2 Contributions

This thesis contributes to the development of systems, in particular large, distributed systems. It is necessary to envisage mechanisms that simplify the correct development of large systems according to their specifications and having tool support eases such a complex task. We propose three techniques for Event-B that help the development of these kind of systems: *composition*, *generic instantiation* and *decomposition*. Individual models can be composed in a shared event style through the *composition* technique. Proof obligations in the individual models can be reused to minimise the proof effort on the resulting composed model. Through *generic instantiation* an existing model can be used as generic and instantiated to be used in other developments. The new instances inherit the generic properties and respective POs. *Decomposition* allows the partitioning of a model into several sub-models as an architecture feature and/or to simplify and more easily discharge POs. The three techniques support *reuse* of existing sub-components taking advantage of their properties (reuse of models and avoiding re-proof). Necessary POs are defined and simplified using the existing POs associated to the individual sub-components. The Rodin platform serves as a host for the plug-ins developed to give tool support to each of the techniques. We present the work developed for these techniques starting from the theory behind each one of them, the application to case studies and the extension to tool support.

This chapter introduces the contribution of this thesis and the necessary background to understand the rest of the document. The technical details start with the introduction of formal methods in Sect. 1.2. Several formal methods relevant to our thesis are introduced in Sect. 1.3. Refinement is briefly covered including a comparison to different formal methods in Sect. 1.4. Section 1.5 introduces the formal method that we use for our work as well as a brief view of the Rodin platform, the toolset for Event-B. We finish this chapter by covering the background related to our contribution: *composition* in Sect. 1.6 and *decomposition* in Sect. 1.7.

Next we describe in more detail what formal methods are and show some examples.

1.2 Formal Methods

Formal Methods use rigorous mathematical techniques to reason about systems' behaviour. It can be applied to software and hardware systems and formal specification expresses, in precise mathematical terms, whether a future computer based system or program is working correctly. Formal methods ensures that a program fulfills its formal specification. This is specially important in the development of critical safety systems [38]. On a top-down development, the application of formal methods can be divided into 3 steps [5]:

- Creation of requirement documents
- Development of the *Abstraction Model* (first model representing a system through the use of formal notation) and the steps toward the *Concrete Model* (model which is closer to what the system will be, but still represented by formal notation)
- Converting the *Concrete Model* into an *Implementation*. On a programming software project, there already exist tools that automatically do this task.

[5, 7] use some formal methods case studies in industry and discuss how requirement documents, system models and executable code fit on a project's life cycle.

Formal methods can differ in several aspects, like syntax (specification language), semantics or applications. Classifications can be drawn from the different notations. A possible classification is to distinguish formal methods in terms of behaviour, i.e. state-based or event-based approaches [1, 35, 74]:

- State-Based behaviour: the system is described by a sequence of state changes. A state is a set of assignments to a set of components (frequently variables). This kind of systems usually are rooted in logic and close to how imperative programming languages that deal with state. This approach forces a close examination of how the real system is represented in the model [1, 31]. Examples of formal methods with a state-based behaviour are Z [173], VDM [105] or B [4].
- Event-Based behaviour: the system is described by a sequence of operations. The specification is manipulated algebraically while defining the actions [1]. Event-based systems are used to develop and integrate systems that are loosely coupled (suitable for large-scale distributed applications). The integrated systems can communicate by generating and receiving event notifications [74]. Examples of formal methods with a event-based behaviour are CSP (Communicating System Processes) [92] or CCS (Calculus of Communicating Systems) [126].

A state-based system usually changes state through the execution of events. An event-based approach expresses the evolution of the system by defining the enabled operations. Event-based view is suitable for message-passing distributed systems while state-based view is suitable for design of parallel algorithms [42]. Not always it is possible to make a very clear distinction of these two situations: depending on the viewpoint a formal notation is seen, it can show both characteristics.

[112, 113] suggest another classification for formal notations based on common characteristic of modelling languages from a system re-engineering point of view:

- **Model based**: a system is described by explicitly defining state and operations. It progresses through the execution of operations that change the system from one state to another. There is no explicit representation of concurrency and some functional requirements cannot be expressed (temporal requirements). Several stated-based formal methods are also model based as are the examples of Z, VDM, B or Event-B [9].
- **Logic based**: Logics are used to describe desirable properties of the system such as specification, temporal or probabilistic behaviour. The validity of these properties relies on the associated axiom system. The final executable specification can be used for simulation and prototype construction. Logic can be augmented with some concrete programming constructs to obtain a wide-spectrum formalism. In that case, correctness refinement steps are applied during the construction of such systems. Examples of logic based modelling languages are Hoare Logic [65], Weakest Precondition Calculus [64], Modal Logic [133] or Temporal Logic [119].
- **Algebraic Approach**: Explicit definition of operations is given by describing the behaviour of different operations without any definition of state. Like model-based notations, concurrency it is not explicitly expressed. Examples are OBJ [84] or LARCH [86].
- **Process Algebra Approach**: Concurrent systems are explicitly represented. The system behaviour is constrained by all observable communication between processes. Examples are: CSP, CCS, ACP (Algebra of Communicating Processes) [28] or LOTOS (Language of Temporal Ordering Specification) [98].
- **Net based**: Graphical languages are combined with formal semantics, bringing some advantages to system creation/development. Graphical notation are popular resulting from the simplicity of defining specifications for systems without requiring a deep understanding of the underlaying framework. Examples are: Petri Net [143], StateCharts [95] or UML-B [170].

The classification of the formal notation helps when deciding which formal methods is suitable for a particular system development. The next section gives an overview of other formal methods (related to Event-B or) relevant to our developed work.

1.3 Overview of some formal methods

Event-B is a formal method that allows the specification and modelling of reactive systems (see Sect. 1.5). Nevertheless other formal methods are available for implementing different kind of systems. We overview some formal methods related to Event-B and in particular to our work:

- CSP
- VDM
- Action Systems
- Classical B
- Z

These formal notations are briefly introduced in the following sections.

1.3.1 Communicating Sequential Processes - CSP

CSP is a process algebra formal method that allows modelling of parallel processing and interaction between systems [91]. The basic concept in CSP considers a process as a mathematical abstraction of interactions between the system and its environment. The behaviour of the system is described through independent *Processes* in an event-based view. A set of events in which a process P can engage is called its alphabet, written αP and represents the visible interface between the process and its environment [53]. The processes are constrained in the way they can engage in the events of its alphabet, using CSP process term language [43]. A process interacts with its environment by synchronously engaging in atomic events. A sequence of events is described using a prefix operator '$\rightarrow$'. For instance, $a \rightarrow P$ describes the process that engages in the event a and then behaves as process P. The environment can decide between two processes using the choice operator '$[]$'. For instance, $P \; [] \; Q$ represents the process that offers the choice to the environment between behaving as process P or as process Q. There is also a non-deterministic choice operator '$\sqcap$': $P \sqcap Q$ represents the process that internally chooses between behaving as P or Q, without any environment control. There are several operations that can be applied to traces [92] like concatenation, interleaving, subscripting, reversal among others. We describe here in more detail the concatenation and interleaving operations as they will be used later on.

Concatenation Let s be a sequence, each of whose elements is itself a sequence. Then $\frown/s$ is obtained by concatenating all the elements together in the original order. A definition can be given by means of the following laws (distributive operator) [92]:

- $\frown/\langle\rangle = \langle\rangle$
- $\frown/\langle s\rangle = s$

- $\frown/(s \frown t) = (\frown/s) \frown (\frown/t)$

Interleaving A sequence s is an interleaving of two sequences t and u if it can be split into a series of subsequences, with alternate subsequences extracted from t and u. For example $s = \langle 1, 6, 3, 1, 5, 4, 2, 7 \rangle$ is an interleaving of t and u, where $t = \langle 1, 6, 5, 2, 7 \rangle$ and $u = \langle 3, 1, 4 \rangle$. A recursive definition can be given by means of the following laws [92]:

- $\langle\rangle$ *interleaves* $(t, u) \equiv (t = \langle\rangle \wedge u = \langle\rangle)$
- s *interleaves* $(t, u) \equiv s$ *interleaves* (u, t)
- $(\langle x \rangle \frown s)$ *interleaves* $(t, u) \equiv (t \neq \langle\rangle \wedge t0 = x \wedge s$ *interleaves* $(t^{'}, u)) \vee$
  $(u \neq \langle\rangle \wedge u0 = x \wedge s$ *interleaves* $(t, u^{'}))$,
  where $t^{'}$ (same for $u^{'}$) is the tail of sequence t (u).

CSP allows the refinement of models. The refinement depends on the semantic model of the language which is used [153] and respective granularity:

- Traces refinement: The coarsest used relationship is based on the sequences of events which a process can perform (the traces of the process). A process Q is a traces refinement of another, P, if all the possible sequences of communications which Q can do are also possible for P. The previous trace refinement can be expressed as $P \sqsubseteq_T Q \mathrel{\widehat{=}} traces(Q) \subseteq traces(P)$.

- Failures refinement: A finer distinction between processes can be made by constraining events. An implementation can constrain events permitted to block as well as events that can be performed. A failure is a pair (s, X), where s is a trace of the process and X is a set of events the process can refuse to perform at that point (refusal). A state of a process is deadlocked if it can refuse to do every event and STOP is the simplest deadlocked process. Deadlock is also commonly introduced when parallel processes do not succeed in synchronising on the same event. Failures refinement between processes P and Q can be expressed as $P \sqsubseteq_F Q \mathrel{\widehat{=}} failures(Q) \subseteq failures(P)$.

- Failures-Divergences refinement: The failures model does not model processes that might livelock (i.e., perform an infinite sequence of internal actions) and so may never subsequently engage in a visible event. The failures-divergences model meets this requirement by adding the concept of divergences. The divergences of a process are the set of traces after which the process may livelock. This gives two major enhancements: the ability to analyse systems which have the potential to never perform another visible event and assert this does not occur in the situations being considered; and use divergence in the specification to describe "do not care" situations. Formally, after a divergence, a process is considered as acting chaotically and is able to do or refuse anything. This means that processes are considered to be identical after they have diverged. A failures-divergences refinement between processes P and Q can be expressed as $P \sqsubseteq_{FD} Q \mathrel{\widehat{=}} failures(Q) \subseteq failures(P) \wedge divergences(Q) \subseteq divergences(P)$.

- Infinite traces refinement: The infinite-traces model of CSP was introduced by Roscoe [152]. It extends the failures-divergences model by including all possible infinite behaviours of a process. A process model now has components (A, F, D, I) where A, F and D are as in the failures-divergences model and I is some subset of A^w, the set of infinite sequences of elements of A, the alphabet of the process. An infinite traces refinement between processes P and Q can be expressed as $P \sqsubseteq_{FDI} Q \mathrel{\widehat{=}} failures(Q) \subseteq failures(P) \wedge divergences(Q) \subseteq divergences(P) \wedge infinites(Q) \subseteq infinites(P)$.

The semantics of an expression P is written $(\alpha(P), \mathcal{F}[\![P]\!], \mathcal{D}[\![P]\!], \mathcal{I}[\![P]\!])$, or $[\![P]\!]$ for short. The semantics function is used to justify the algebraic laws: for expressions P, Q, $P = Q$ iff $[\![P]\!] = [\![Q]\!]$ [53].

There are some tools available for CSP. FDR2 (Failures/Divergence Refinement 2) is a refinement checker for establishing properties of models expressed in CSP. Also ProBE, an animator for CSP processes, allows the user to explore the behaviour of models interactively. These two tools are developed by Formal Systems Europe [80]. Adelaide Refinement Checker (ARC) [136] is a CSP refinement checker developed by the Formal Modelling and Verification Group at The University of Adelaide. Occam Transformation System is an automated tool to assist in carrying out algebraic transformations.

1.3.2 Vienna Development Method - VDM

Vienna Development Method (VDM) is a model-oriented notation developed while a research group of IBM laboratory in Vienna was working on compiler developments and language designs. It consists of a formal modelling language,VDM-SL, which is a combination of data definitions, state variables and a set of operations describing the specification of systems and state variables invariants verified before and after the execution of an operation [111]. Unlike other notations like Z or B, VDM has a three values logic which allows explicit treatment of undefinedness. The VDM syntax can be described using ASCII or mathematic notation. More recently an extension of VDM was developed, VDM^{++}, supporting object-oriented design, concurrency and capable of modelling real-time distributed systems [77].

A VDM development is made up of state descriptions at successive levels of abstraction and implementation steps which link to the state description. The implementation of an abstract state description S_a by means of a more concrete one S_c describes [111]:

- either a *data reification*, i.e. how the state variables of S_c implement the ones of S_a;
- or an *operation decomposition*, i.e. how the operations of S_c implement the ones of S_a into a computer language algorithm (*implementation*).

While modelling a specification using VDM, in particular for the operations, predicates precondition and postcondition are written explicitly. The state of variables before and after an operation usually is defined. To refer to a before value it is used the "~" decoration on the relevant variable [77]. VDM objects must be validated by proof obligations [111] and for an operation to be valid, the satisfiability rule (a sentence is satisfiable if there is some interpretation under which it is true) must be met [101]. VDM formal development uses data reification from abstract to concrete model [equivalent to data refinement] but also uses operation decomposition to develop (abstract) implicit specifications of operations and functions into algorithms that can be directly implemented in a computer language of choice. In general operation decomposition it is applied after the data reification [101].

In terms of tools, VDMTools [59] is the leading commercial tool for VDM-SL and VDM^{++} developed by CSK Systems. Overture [134] is a community-based open source initiative aimed at providing free tool support for VDM^{++} on top of the Eclipse platform. Its aim is to develop a framework for interoperable tools that may be useful for industrial application, research and education.

1.3.3 Action Systems

Action Systems was introduced by Back and Kurki-Suonio [26]. It provides a general description of reactive systems, capable of modelling terminating, aborting and infinitely repeating systems. Arbitrary sequential programs can be used to describe an atomic action. A basic action system $P = (A, v, P_i, P_a)$ consists of a list of labels A, a list of variables v, a set of labelled statements (actions) $P_a = \{P_\alpha \mid \alpha \in A\}$ and an initialisation statement P_i. Each action $\alpha \in A$ is of the form [26]:

$$\mathit{action}\ \alpha : g_\alpha(x) \to y := S(x, y). \tag{1.1}$$

The action guard g_α is a condition that the enabling variables x must satisfy for action α to be enabled. The effect of the action is to assign new values $S(x, y)$ to the update variables y. Actions are atomic which means that when an action is executing no other action of the system occurs until the first action is complete. Taking the view that an action system engages in an action jointly with the surrounding environment allows the environment to observe the executed actions and not the state of the action system itself [41].

Back and von Wright [27] describe how Action Systems can be used on parallel and distributed systems in a stepwise manner by giving a behavioural semantics in terms of execution traces. Back [24] suggests that sequential programs could also be implemented in a parallel fashion: two or more actions can be executed in parallel, as long as the (atomic) actions do not have variables in common. Butler [41] exposes a composition using Action Systems from an event-based point of view based on CSP synchronisation. Woodcock and Morgan [189] give two proof methods which are sound and jointly complete in terms of CSP failure-divergences semantics for state-based concurrent systems using the weakest precondition *wp* approach proposed by Morgan [129]. The weakest precondition is briefly explained below.

Weakest Precondition For guarded command G, command com, and postcondition Q:

$$wp(G \rightarrow com, Q) \mathrel{\widehat{=}} G \Rightarrow wp(com, Q).$$

Whereas $wp(com, Q)$ characterises the states from which com is certain to establish Q, we need the states from which com could possibly establish Q. Morgan [129] defined the *conjugate weakest precondition* as follows:

$$\neg wp(com, \neg Q)$$

because in those states it is not certain that com will establish $\neg Q$. Note that we are taking the view that an aborting command could possibly establish anything. Therefore we can say that:

$$\overline{wp}(com, Q) \mathrel{\widehat{=}} \neg wp(com, \neg Q) \tag{1.2}$$

Although $wp(com, true)$ implies termination of com, (1.2) shows that $\overline{wp}(com, true)$ does not. For any action α let G be its guard. Then

$$G \equiv \overline{wp}(\alpha, true).$$

Butler [53] augments Back and von Wright [27] and Woodcock and Morgan [189] works by defining the semantics of Action Systems in terms of the CSP infinite-traces semantic model:

Definition 1.1. For action system $P = (A, v, P_i, P_a)$,

$$\{[P]\} \mathrel{\widehat{=}} (A, \mathcal{F}\{[P]\}, \mathcal{D}\{[P]\}, \mathcal{I}\{[P]\})$$

A failure is a pair of the form (s, X), where $s \in A^*$ (set of finite sequences of elements of A) is an event-trace and $X \subseteq A$ is a refusal set. If (s, X) is in P after initialisation,then P could engage in the action trace s and then refuse all actions X. Trace s is a divergence if $P_{\langle i \rangle s}$ aborts. For an infinite trace $u \in A^w$ (A^w is the set of infinite traces for alphabet A) and $P_u \mathrel{\widehat{=}} (i \mid i \in \mathbb{N} \cdot P_{u_i})$, $\mathcal{I}\{[P]\})$ are those $u \in A^w$ in which the execution of all the

$P_{\langle i \rangle u}$ in sequence is possible. Operationally, for this to be possible in some state it must be the case that state S_0 is enabled and that execution of S_0 could result in a state in which S_1 is enabled and so on for each S_i [53].

Event-B is inspired by Action Systems and as a consequence several similarities can be drawn (atomic actions, state based, modelling reactive systems) for these two formal methodologies. More information regarding composition follows in Sect. 1.6.

1.3.4 Classical B

Classical B (or B-Method) [4] created by Abrial is a formal approach for the specification and development of computer software systems [158] and can be seen as a parent of Event-B. A system specification is defined by *machines* that have *variables* defining the state space. The state progresses with the execution of *operations*. Operations can have preconditions, guards (or both) and postconditions. Properties of the system can be expressed by means of predicates called *invariants*. The B-Method can be seen as both state-based (explicit notion of “state” expressed by variables) or event-based (operations occurring nondeterministically). The development of models usually follows a top-bottom style (Event-B inherits this style as seen in Sect. 1.5) where the most abstract model is simple. More details and complexity are added throughout stepwise refinements. Classical B defines three basic components: *abstract machine*, *refinement* and *implementation*. The last component, implementation, is a special kind of refinement machine from which code can be produced, respecting the original abstract specification. The refinement in Classical B is one to one: one abstract operation is refined by one concrete operation and it is not possible to introduce new operations unlike Event-B. Classical B has been used widely in both academic [169, 17] and industrial developments [5].

Different ways and different tools exist for generating the output code like the B-Toolkit [20] or Atelier B [19]. The B-Method focus on software systems and consequently the final result - implementation model - although similar to other refinement steps, includes programming constructs for common languages (e.g. C and Ada) with some restrictions on the used syntax.

1.3.5 Z

The Z notation [173] is a state-based formal method, which uses mathematical techniques to represent and describe computing systems: hardware and software. A system contains a set of state variables and some operations that change the variables values. A model that is characterised by the operations is called an Abstract Data Type (ADT) and Z follows this style. Z can be used to describe object-oriented programs since the state variables and operations can be compared to instance variables and methods, respectively [100]. Z serves as basis for other notations (for instance, classical B) and several variants adapted to object-oriented programming (an example is Object-Z [168] which is an object-oriented extension of Z). Z includes two notations [100]: notation for ordinary discrete mathematics and notation that provides structure to the mathematical text - *paragraphs*. The most important and more used paragraph is a macro-like abbreviation and naming construct called *schema*. Z defines the requirements through the use of mathematic entities such as sets, relations/functions or sequences. A schema consists of three parts: *name* which identifies the schema and it is used when composed with other schemas; *signature* which is a collection of variables introducing data types and created by declarations and providing a vocabulary for making mathematical statements; *predicate* (or constraint) that defines relations between signatures elements using predicates (describing abstractly the effect of each operation in the proposed system). One of the ways to represent a schema *StateSpace* is represented here (the shortest one) [37]: $StateSpace \mathrel{\widehat=} [x_1 : S_1; ...; x_n : S_n \mid Inv(x_1, ..., x_n)]$. $x_1...x_n$ are state variables, $S_1...S_n$ are expressions that represent variable types. $Inv(x_1, ..., x_n)$ are the state invariants. Schemas are used to define the static and dynamic feature of a system. The static part includes the possible states and rules that should be preserved during the system execution (invariant clauses). The dynamic part consists of available operations and changes on the state after the execution of an operation, as well as on relationships between input and output.

Research has been undertaken to adjust Z to model concurrent systems [37, 76]. Some of these results are: Fischer's CSP-OZ [75] developed to combine CSP properties with Object-Z; *Circus* [186, 185], developed by Woodcock and Cavalcanti providing formal support for the specification of data, behaviour aspects of concurrent systems and allowing refinement through the use of a syntactical approach in opposition to a semantic one; TLZ [109] developed by Lamport that combines Temporal Logic of Actions (TLA) [108] and Z; Taguchi and Araki [179] combine Z and CCS to specify concurrent systems, among others.

The Community Z Tools (CZT) project [57] is an open source project providing an integrated toolset to support Z, with some support for Z extensions such as Object-Z, Circus, and TCOZ. Another Z tool is Fastest [79] which is a model-based testing tool. The tool receives a Z specification and generates (almost automatically) test cases derived from the specification. $f$uzz [175, 174] is Spivey's typechecker for the original Z language. It includes style files for LaTeX and it is available as part of the Z Word Tools [180].

All the previous formal methods have something in common: the use of *refinement* to describe a specification. Refinement plays an important role in formal developments in particular on a top-down style. Initially we have an initial abstract and simplistic view of the modelled system. Refinement allows the introduction of more details in the state system respecting the initial abstract view. We discuss more about refinement in the next section.

1.4 Refinement

Refinement allows the construction of a model in a gradual way, making it closer to an implementation [15]. At same time, the overall correctness of the system should be preserved. A property P' is said to refine a property P if $P' \sqsubseteq P$. The initial model is defined as the *abstract model*. A model that maintains the properties of the abstract model and adds more details is defined as a *concrete model*. The states in the abstract model are linked to the concrete ones. The refinement process can be repeated so it can be applied over a concrete model generating an even more concrete model. All formal notations presented in the previous sections have the notion of refinement although sometimes named differently (in VDM it is known as *reification*). Operations in B, VDM and Z are "refined" on a one-to-one basis: one abstract operation is refined by only one concrete operation. Event-B, as it will be seen in the Sect. 1.5, is more flexible as it inherits a refinement property from Action Systems and CSP where it is possible to introduce new events during the stepwise refinement steps. *Gluing invariants* are predicates used to link the abstract and concrete states. In Event-B, refinement can also be applied to a machine and respective context(s) separately. It is possible to *extend* contexts by adding new sets, constants or axioms to an existing context as long as the abstract context properties are kept [150] (see Sect. 1.5).

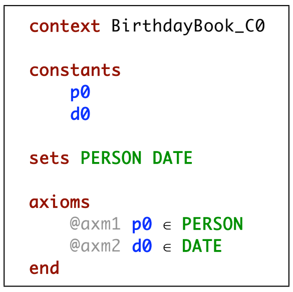
Proof obligations are generated and discharged during the refinement process to preserve the abstract properties in the concrete model: concrete events must keep the behaviour of the respective abstract ones; the new model should not introduce divergence and the invariants of the concrete model should be preserved for every event enabled (the semantic of these proof obligations are described in Sect. 1.5.3). New events refining an implicit event which does nothing (*skip*) [15] can be added in a refinement step. All the abstract events must be refined in the concrete model. A constraint for the refinement is that the concrete machine should not deadlock before the abstract machine, otherwise the concrete machine might not achieve what the abstract machine had previously required. The formalization of the described constraints can be found in [15].

Next section focus on the Event-B language and properties which will help understand the rest of the document. The refinement POs for Event-B are also described in the following section.

1.5 Event-B

Event-B is a formal methodology that uses mathematical techniques based on set theory and first order logic. It is a notation and method used for modelling discrete systems resulting from an evolution of other formal methods notations like classical B and Action Systems. The justifications and explanations for such notation can be found in [87]. Event-B is suitable for modelling parallel, reactive and distributed systems and can be seen as a state-based formal method due to the close relation to classical B. Event-B models can be developed in the Rodin modelling tool [151, 71] and we discuss it briefly in Sect. 1.5.7. The semantics of a model developed in Event-B is given by means of its proof obligations (cf. Sect. 1.5.3). These obligations have to be discharged to show consistency of the model with respect to some behavioural semantics. Abrial [9] expresses these behavioural semantics as state trace semantics.

An abstract Event-B specification is divided into two parts: a static part called *context* and a dynamic part called *machine*. A context *Ctx* consists of *carrier sets* s (similar to types [15]), *constants* c, *axioms* (assertions constraining constants and carriers sets) and *theorems* $A(s, c)$. (Identifiers occurring free in a formula are indicated in parentheses). An example of a context can be seen in Fig. 1.1.



```
context BirthdayBook_C0

constants
    p0
    d0

sets PERSON DATE

axioms
    @axm1 p0 ∈ PERSON
    @axm2 d0 ∈ DATE
end
```

FIGURE 1.1: Context BirthdayBook_C0

A *model* is defined by a machine M that *sees* a context *Ctx*. A machine usually contains global state *variables* v as well as *invariants* and (machine) *theorems* $I(s, c, v)$ that define the dynamic properties of the specification by constraining v. Possible state changes are described by means of *events*: when their conditions are satisfied, optional local variables (parameters) can be used and state variables may be updated. An example of a birthday book machine can be seen in Fig. 1.2.

An abstract Event-B specification can be refined by adding more details and becoming more concrete (see Fig. 1.3 where machine N refines machine M).

Refinement allows the introduction of more details in small steps. Otherwise the specification development would have to be done in one single step with the possible consequence of becoming complicated, hard to reason about (dealing with all the details of

```
machine BirthdayBook_M0 sees BirthdayBook_C0

variables birthday

invariants
  @inv1 birthday ∈ PERSON ⇸ DATE

events
  event INITIALISATION
    then
      @act1 birthday ≔ {p0↦d0}
  end

  event AddBirthday
    any p d
    where
      @grd1 p ∈ PERSON
      @grd2 d ∈ DATE
      @grd3 p ∉ dom (birthday)
    then
      @act1 birthday ≔ birthday ∪ {p ↦ d}
  end
end
```

FIGURE 1.2: Machine BirthdayBook_M0

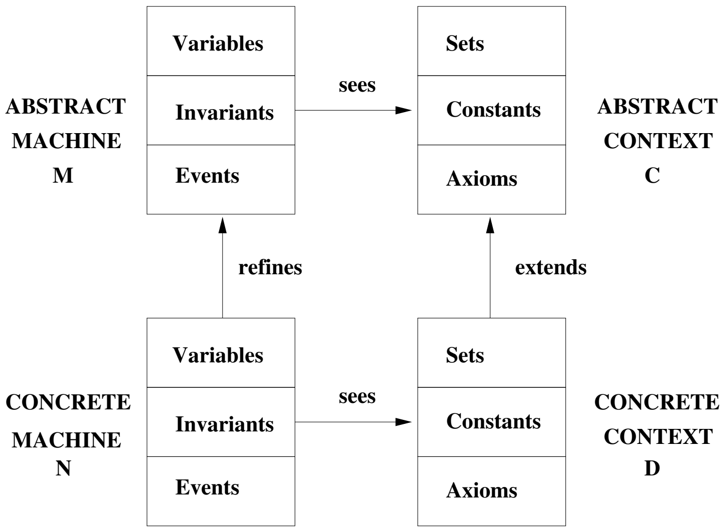


FIGURE 1.3: Machine and context refinement

implementation at once) and most important, hard to *understand* [88]. Concrete models are expressed through the refinement of events, introduction of new variables w and consequently the introduction of *gluing invariants*: invariants that relate abstract and concrete states (variables). Therefore abstract variables can exist in a concrete model or disappear and be replaced by a concrete variables. In that case, a gluing invariant is required to relate the abstract and concrete variable. Abstract contexts can be *extended* by concrete contexts allowing the introduction of new carrier sets, constants and axioms. As an example, see Fig. 1.4 where machine $BirthdayBook\_M0$ is refined with the introduction of a new variable *reminder* (relation between variable *birthday* and a reminding date; the same birthday can have multiple reminding dates). Note that we do not change the original abstract event: we only "extend" it; the abstract event

AddBirthday is extended by appending a concrete guard *grd*4 and a concrete action *act*4.

```
machine BirthdayBook_M1 refines BirthdayBook_M0
sees BirthdayBook_C0

variables birthday reminder

invariants
    @inv1 reminder ∈ birthday ↔ DATE

events
  event INITIALISATION extends INITIALISATION
  then
    @act2 reminder ≔∅
  end

  event AddBirthday extends AddBirthday
  any r
  when
    @grd4 r ∈ DATE
  then
    @act4 reminder(p↦d)≔r
  end
end
```

Figure 1.4: Machine BirthdayBook_M1

Proof obligations arise to verify the consistency of a model. For instance, there are proof obligations to establish the refinement relationship between two machines, and to establish invariant preservation by the events. We reason about a system specification through its proof obligations. The reasoning verifies that the specification is sound wrt some behavioural semantic and that system properties are always satisfied [88]. The logic used in Event-B is typed set theory built on first-order predicate logic, and allows the definition of partial functions. As such, it is necessary that the used proof system handles well-definedness. In [122], it is shown that it is possible to reason about partiality without abandoning the well-understood domain of two-valued predicate logic. In that approach, the reasoning is achieved by extending the standard calculus with derived proof rules that preserve well-definedness across proofs [116]. The proof calculus outlined in [122] is the one used to reason in Event-B.

1.5.1 Preliminaries

A full definition of the mathematical language of Event-B may be found in [16]. Here we give a very brief overview of the structure of the mathematical language to help motivate the remaining sections and chapters.

Event-B distinguishes predicates and expressions as separate syntactic categories. Predicates are defined in term of the usual basic predicates ($\top, \bot, A = B, x \in S, y \leq z$, etc), predicate combinators ($\neg, \wedge, \vee$, etc) and quantiers ($\forall, \exists$). Expressions are defined in terms of constants (0, $\varnothing$, etc), (logical) variables (x, y, etc) and operators (+, $\cup$, etc). Basic predicates have expressions as arguments. For example in the predicate

$E \in S$, both E and S are expressions. Expression operators may have expressions as arguments. For example, the set union operator has two expressions as arguments, i.e., $S \cup T$. Expression operators may also have predicates as arguments. For example, set comprehension is defined in terms of a predicate P , i.e., { x | P } [11].

1.5.1.1 Notation

The naming conventions that we use throughout this thesis are shown in the following tables:

Context	Ctx
Constant	c
Carrier Set	s
Axiom/Theorem	$A(c, s)$

(a) Context Elements

Machine	M
Abstract Variable	v
Concrete Variable	w
(Abstract) Invariant/Theorem	$I(c, s, v)$
(Concrete) Invariant/Theorem	$J(c, s, v, w)$

(b) Machine Elements

Event	evt
(Abstract) Parameter	p
(Concrete) Parameter	q
(Abstract) Guard	$G(c, s, p, v)$
(Concrete) Guard	$H(c, s, q, w)$
Parameter Witness	$W(c, s, p, q, w, w')$
Variable Witness	$W(c, s, q, v', w, w')$
(Abstract) Action	$S(c, s, p, v, v')$
(Concrete) Action	$T(c, s, q, w, w')$

(c) Event Elements

1.5.1.2 Types

All expressions have a type which is one of three forms:

- a basic set, that is a predefined set ($\mathbb{Z}$ or BOOL) or a carrier set provided by the user (i.e., an identifier);
- a power set of another type, P(α);
- a cartesian product of two types, $\alpha \times \beta$

These are the types currently built-in to the Rodin tool [11]. User-defined types can be defines as *carrier sets* and the only implicit assumption is that they are not empty [156]. An expression E has a type **type**(E) provided E satisfies typing rules. Each expression operator has a typing rule which we write in the form of an inference rule. For example, the following typing rule for the set union operator specifies that $S \cup T$ has type ($\mathbb{P}\,\alpha$) provided both S and T have type $\mathbb{P}(\alpha)$:

$$\frac{\mathbf{type}(S) = \mathbb{P}(\alpha) \qquad \mathbf{type}(T) = \mathbb{P}(\alpha)}{\mathbf{type}(S \cup T) = \mathbb{P}(\alpha)}$$

This rule is polymorphic on the type variable α which means that union is a polymorphic operator. It should be noted that an expression of type BOOL is not a predicate. The type BOOL consists of the values *TRUE* and *FALSE*, both of which are expressions. These are different to the basic predicates $\top$ and $\bot$. The *bool* operator is used to convert a predicate into a boolean expression, i.e., $bool(x > y)$. A boolean expression E is converted to a predicate by writing $E = TRUE$. We have that $bool(\top) = TRUE$.

1.5.1.3 Functions

There exists a relation between operators and function application in Event-B. The type of an Event-B function f is $\mathbb{P}(\mathbf{type}(A) \times \mathbf{type}(B))$. The functionality of a partial function $f \in A \rightarrowtail\!\!\!\!\!+ B$ is specified with an additional property and a uniqueness condition:

$$\forall x, y, y' \cdot x \mapsto y \in f \wedge x \mapsto y' \in f \Rightarrow y = y'$$

The domain of f, written $dom(f)$, is the set $\{x \mid \exists y \cdot x \mapsto y \in f\}$. Application of f to x is written $f(x)$ which is well-defined provided $x \in dom(f)$. Note that f is not an operator itself: it is simply an expression. The operator involved here is implicit: it is the function application operator that takes two arguments, f and x. An explicit operator for a function application could have been written as $apply(f, x)$, where *apply* is the operator and f and x are the arguments. But in the Rodin tool, the shorthand $f(x)$ must be used.

Variables in the mathematical language are typed by set expressions. This means, for example, that a variable may represent a function since a function is a special case of a set (of pairs). Variables may not represent expression operators or predicates in the mathematical language. This means that, while we can quantify over sets (including functions), we cannot quantify over operators or predicates.

1.5.1.4 Well-Definedness

Ill-defined terms arise in the presence of partial functions. They result from the application of functions to terms outside their domain. When ill-definedness is a concern, the adopted reasoning framework has to cope with it. Different approaches exist to reason in the presence of partial functions. Each of these approaches has its own specialised proof calculus. In [122], it is shown that it is possible to reason about partiality without abandoning the well-understood domain of two-valued predicate logic. In that approach, the reasoning is achieved by extending the standard calculus with derived proof rules that preserve well-definedness across proofs [116].

Along with typing rules as defined above, all expression operators come with well-definedness predicates. We write $\mathbf{WD}(E)$ for the well-definedness predicate of expression E. Table 1.1 gives examples of well-definedness conditions for several operators, including the function application operator.

Expression	Well-Definedness Conditions
$F(E)$	$\mathbf{WD}(F), \mathbf{WD}(E), F \in x \nrightarrow y, E \in dom(F)$
F/E	$\mathbf{WD}(F), \mathbf{WD}(E), E \neq 0$
$card(E)$	$\mathbf{WD}(E), finite(E)$
$S \cup T$	$\mathbf{WD}(S), \mathbf{WD}(T)$
$F \mod E$	$\mathbf{WD}(F), \mathbf{WD}(E), E \neq 0$
$min(S)$	$\mathbf{WD}(S), S \neq \varnothing, \exists x \cdot (\forall n \cdot n \in S \Rightarrow x \leq n))$
$max(S)$	$\mathbf{WD}(S), S \neq \varnothing, \exists x \cdot (\forall n \cdot n \in S \Rightarrow x \geq n))$

Table 1.1: Some expressions and respective well-definedness conditions

From the Table 1.1, it can be seen that an expression $F(E)$ is well-defined provided both F and E are well-defined, that F is a partial function and that E is in the domain of F. In the Rodin tool, well-definedness conditions give rise to proof obligations for expressions that appear in models. The well-definedness conditions are themselves written as predicates in the Event-B mathematical language. [125] gives the full list of expression, predicates and respective well-definedness conditions available in the Rodin tool. A well-defined sequent of the form $H \vdash_D G$ is defined as follows:

$$H \vdash_D G \mathrel{\widehat{=}} D(H), D(G), H \vdash G.$$

That is, the well-definedness of H and G is assumed when proving $H \vdash G$. Generally speaking, when proving a sequent $H \vdash G$, the approach suggests proving its validity as well as its well-definedness:

$$\begin{aligned} WD_D : &\quad \vdash_D D(H \vdash G) \\ Validity_D : &\quad H \vdash_D G \end{aligned}$$

where $D(H \vdash G)$ is defined as $D(\forall x \cdot H \Rightarrow G)$ such that x are the free variables of H and G [116]. A proof rule is said to preserve well-definedness (WD) iff its consequent and antecedents only contain well-defined sequents (i.e., $\vdash_D$ sequents). For a generic proof rule where $H_1 \ldots H_n$ (standing for a conjunction), $n > 0$, are a sequence of (possible empty) sequents and G is also a sequent, with an optional name r:

$$\frac{\vdash H_1 \ldots H_n}{\vdash G}\ r$$

then, the same proof rule can be rewritten including the well-definedness conditions (and the well-definedness operator D [122]) as:

$$\cfrac{\cfrac{\vdash_D D(H_1)\ldots D(H_n)}{\vdash_D D(H_1\ldots H_n)}\wedge goal_D \qquad \vdash_D H_1\ldots H_n}{\vdash_D G}\; r_D$$

Additional details about the use of well-definedness in a first order predicate calculus can be found in [122].

Other works also study the well-definedness of partial functions: Fitzgerald and Jones [78] discuss a connection between the classical First-order Predicate Calculus(FoPC) and the Logic of Partial Functions (LPF). It is claimed that theorems in LPF using weak equality can be straightforwardly translated into ones that are true in FoPC; translation in the other direction results, in general, in more complicated expressions but in many cases these can be readily simplified. [18, 120] discuss the semantics of Z in relation to first order logic, particularly regarding undefinedness and proofs.

With regard to recursive functions, they are not supported in the Rodin platform. The theory plug-in [115] that allows the extension of the mathematical language [11, 51], allows the definition of new (possibly recursive) operators as well as the necessary well-definedness conditions. Nevertheless this is currently work in progress.

Next we extend this initial description by focusing on the kind of events, the types of variables assignments and parameters. In the end we outline the existing proof obligations for Event-B models. These details are necessary further on to explain our contributions.

1.5.2 Events

In Event-B, events specify changes to variables and the conditions under which they may occur. Events occur as soon, and as long, as its firing condition (guards) are set [3]. An event *evt* is expressed by *parameters* (local variables to the event) p, by *guards* $G(s,c,p,v)$ and *actions* $S(s,c,p,v,v')$:

$$evt \mathrel{\widehat{=}} \textbf{ANY}\ p\ \textbf{WHERE}\ G(s,c,p,v)\ \textbf{THEN}\ S(s,c,p,v,v')\ \textbf{END}.$$

When the guard $G(s,c,p,v)$ is true then the event *evt* is enabled and therefore the action $S(s,c,p,v,v')$ can update the set of variables v to v' (after-state of v). A more general definition is as follows: events may *occur* atomically when its *guards* are true and as a result the state is updated through the execution of actions. The guard of an event states the necessary conditions under which an event may occur and an action describes how the state variables evolve when an event occurs [88]. A mandatory event called $INITIALISATION$, with $TRUE$ as guard, defines the initial state of the machine. The system state progresses when events are enabled and occur. More details about events, guards and actions can be found in [15].

We consider three kind of events depending on when they are introduced during the development of a model:

- convergent: new events introduced after a refinement. These kind of events refine *skip* and require a variant (see Section 1.5.3.2) to ensure non-divergence.
- anticipated: event declared in anticipation that does not need to decrease a variant (but must not increase it either); it only decreases the variant when it becomes convergent in a further refinement [9].
- ordinary: neither convergent nor anticipated.

The majority of developments use ordinary or convergent events but anticipated events can be useful when modelling. Anticipated events are used to avoid a technical difficulty of using abstract variables in a new event during a refinement step. By declaring the new event in anticipation in an abstract refinement, this technicality is circumvented. As mentioned, new events can be introduced in a refinement of an abstract model (similar to CSP hiding operator where some events are hidden from the environment) representing internal events. These new events may (optionally) be defined as convergent or anticipated.

For variable assignments in an action, there are three simple forms [13] described in Table 1.2. v and v_1 are some variables, $E(\dots)$ denotes an expression, p are parameters

Assignment	Before-After Predicate (BAP)
$v := E(p, v_1)$	$v' := E(p, v_1)$
$v :\in E(p, v_1)$	$v' :\in E(p, v_1)$
$v :\mid S(p, v_1, v')$	$S(p, v_1, v')$

Table 1.2: Event-B assignments

and $S(\dots)$ is a predicate. The before-after predicate (BAP) denotes the relationship holding between the state variables of the model just before (denoted by v) and just after (denoted by v') applying a substitution. The first row in Tab. 1.2 corresponds to a deterministic substitution while the other two are non-deterministic substitutions. In the second row, the assignment is non-deterministic and based on the expression $E(\dots)$ (for instance, assigning a value to v from a non-empty set). The third row assigns a value to v according to the predicate defined and it is also considered non-deterministic. Variables that do not appear on the left-hand side of an assignment of an action are not changed (variables v_1). The last row is the most general form of assignment and all the other assignments can be expressed in this manner.

Concrete models are expressed through the refinement of events. An abstract event $evt1$ is refined by $evt2$ if the guard $H(s, c, q, w)$ of $evt2$ is stronger (*guard strengthening*) than the guard $G(s, c, p, v)$ of $evt1$ and the gluing invariant $J(s, c, v, w)$ establishes a *simulation* of $evt2$ ($T(s, c, p, w, w'))$ by $evt1$ ($S(s, c, p, v, v')$):

$$evt1 \mathrel{\widehat{=}} \textbf{ANY}\ p\ \textbf{WHERE}\ G(s,c,p,v)\ \textbf{THEN}\ S(s,c,p,v,v')\ \textbf{END}$$

$$\begin{aligned}evt2 \mathrel{\widehat{=}} \textbf{ANY}\ q\ \textbf{WHERE}\ & H(s,c,q,w)\\ \textbf{WITH}\ & p : W_1(p,s,c,w,q)\\ & v' : W_2(v',s,c,w,q,w')\\ \textbf{THEN}\ & T(s,c,p,w,w')\ \textbf{END}.\end{aligned}$$

The gluing invariant must be preserved by all events: invariants are supposed to hold whenever a variable value changes in an event (invariant preservation PO). Moreover the guard strengthening PO is preserved as follows: when a concrete event is enabled, then so it is the corresponding abstract one. Finally the simulation PO is proved if the occurrence of the concrete event does not contradict what the corresponding abstract event does. In addition, for event refinements it must be shown that it is possible to choose a value for the abstract parameter p such that $G(s, c, p, v)$ holds and the gluing invariant $J(s, c, v, w)$ is re-established. Possible values of the abstract parameter p are given as *witness* predicates $W_1(p, s, c, q, w)$ in concrete events [90, 87]. A witness is necessary for each disappearing abstract parameter of an abstract event in the abstract event. Moreover a witness $W_2(v', s, c, w, q, w')$ is needed for each disappearing abstract variable v' that has a non-deterministic assignment (see the third row of Table 1.2) [9]. New events can be introduced in a refinement of an abstract machine. They must refine the implicit abstract dummy event *skip* and it may be proved that they do not collectively diverge by being always enabled and preventing abstract events to occur. The divergence is avoided if each new event decreases a *variant* [14]. The variant must be well-founded, may be an integer or a finite set and it is bounded. One variant per event that must be decremented by that same event. To preserve refinement, consistency proof obligations are defined as described in Sect. 1.5.3.

1.5.3 Proof obligations

Proof obligations have a two-fold purpose. On the one hand, they show that a model is sound with respect to some behavioural semantics. On the other hand, they serve to verify properties of the model [88]. In Event-B, there are different kind of proof obligations generated during a model development. A list of standard POs for contexts and machines is defined in [9, 88, 87]. Here we only cover the relevant POs for our work. In Event-B, refinement is defined in terms of POs and these correspond to standard forward simulation [10]. We shall use a generic model illustrated in Fig. 1.5 to describe the POs. Backward simulation is currently not supported.

Context *Ctx* is characterised by constants c, carrier sets s and axioms *A(s,c)*. This context is seen by all the involved machines. The abstract machine M contains a set of

```
CONTEXT Ctx
CONSTANTS $c$
SETS $s$
AXIOMS $A(s,c)$
```

(d) Context Ctx

```
MACHINE M SEES Ctx
VARIABLES $v$
INVARIANT $I(s,c,v)$
EVENT $evt \mathrel{\widehat{=}}$
    ANY p WHERE
        $G(p,s,c,v)$
    THEN
        $v :| S(p,s,c,v,v')$
    END
```

(e) Machine M

```
MACHINE N REFINES M
SEES Ctx
VARIABLES $w$
INVARIANT $J(s,c,v,w)$
VARIANT $n(s,c,w)$
EVENT $evt1$ REFINES evt $\mathrel{\widehat{=}}$
    ANY q WHERE
        $H(q,s,c,w)$
    WITH
        $p : W_1(p,s,c,w,q)$
        $v' : W_2(v',s,c,w,q,w')$
    THEN
        $w :| T(q,s,c,w,w')$
    END
convergent EVENT $evt2 \mathrel{\widehat{=}}$
    ANY q WHERE
        $H_2(q,s,c,w)$
    THEN
        $w :| T_2(q,s,c,w,w')$
    END
```

(f) Machine N

FIGURE 1.5: Context *Ctx* seen by machine *M* and respective refinement *N*

variables v, a list of invariants and local theorems *I(s,c,v)* and an event *evt* defined by the parameter p, guards *G(p,s,c,v)* and before-after predicates $S(p,s,c,v,v')$ [considered in the non-deterministic form as defined in the third row of Table 1.2] over the set of variables v. Machine N refining M, contains a set of variables w and a set of additional (concrete) invariants and theorems $J(s,c,v,w)$. Event *evt1* refines abstract event *evt* and a new convergent event *evt2* is introduced in this refinement. We assume that variables v and w are pairwise disjoint and the same happens to parameters p and q. A proof obligation is a sequent of the shape:

$$\boxed{\begin{array}{l} Hypotheses \\ \vdash Goal \end{array}}$$

Hypotheses and goal are defined by predicates such as invariants, theorems, axioms or guards. Based on the previous, we define the standard proof obligations in Event-B [9]. These proof obligations are divided into consistency POs and refinement POs as described below.

1.5.3.1 Consistency POs

Consistency POs are required to be always verified for each machine. Consistency is expressed by the feasibility and invariant preservation POs for each event [88]. Moreover well-definedness POs are generated for each potential ill-defined term (such as axioms, theorems, invariants, variant, guards, actions).

Invariant Preservation (INV): This kind of proof obligation ensures that each invariant is preserved by each event. The hypotheses include axioms, invariants, local theorems, guards and before-after predicates of that event. The goal is each individual invariant from the set of existing invariants. In Fig. 1.5(e), for event *evt* and each of the invariants $i(s, c, v)$ in *I(s,c,v)*, the respective proof obligation rule is given by (1.3).

$$\boxed{\begin{array}{rl} & A(s,c) \\ & I(s,c,v) \\ evt/inv/INV: & G(p,s,c,v) \\ & S(p,s,c,v,v') \\ & \vdash i(s,c,v') \end{array}} \tag{1.3}$$

$i(s, c, v')$ is one of the invariants where variables v are modified to v'.

Feasibility (FIS): It ensures that each non-deterministic action is feasible for a particular event. The hypotheses include axioms, invariants, local theorems and guards of that event. The goal ensures that values exist for variables v' such that the before-after predicate $S(p, s, c, v, v')$ is feasible. In Fig. 1.5(e), for event *evt* and each of the actions *act*, this proof obligation is given by (1.4).

$$\boxed{\begin{array}{rl} & A(s,c) \\ & I(s,c,v) \\ evt/act/FIS: & G(p,s,c,v) \\ & \vdash \exists v' \cdot S(p,s,c,v,v') \end{array}} \tag{1.4}$$

Well-Definedness (WD) It ensures that any axiom (WD/AXM), theorem (WD/THM), invariant(WD/INV), guard(WD/GRD), action(WD/ACT), variant(WD/VWD) or witness p in an event $evt(evt/p/WWD)$ is indeed well-defined. It varies with the potentially ill-defined expression as seen in Table 1.1. An important property for WD proof obligations is that they are ordered. For example, the WD conditions for an invariant depends only on the previous defined invariants: for the WD of i_k, which is the invariant k from the set of invariants I, the hypotheses that can be assumed are $i_1 \dots i_{k-1}$.

1.5.3.2 Refinement POs

The refinement POs are required when the an abstract machine is refined by a more concrete one. Besides the consistency POs, refinement POs are additional obligations required to be discharged to ensure valid refinements. As mentioned in Section 1.4, in Event-B refinement requires concrete events to keep the behaviour of the respective abstract ones. Proof rule (1.5) expresses the refinement PO for each concrete event.

Refinement (REF): For each concrete event, the refinement PO reinforces that abstract actions are simulated by the concrete ones, that each abstract guard is at least as weak as the concrete one and that when an abstract variable is data refined by a concrete one and disappears, gluing invariants exist linking the abstract and concrete variables.

$$\boxed{\mathit{evt1/REF}: \begin{array}{l} A(s,c) \\ I(s,c,v) \\ J(s,c,v,w) \\ H(q,s,c,w) \\ T(q,s,c,w,w') \\ \vdash \exists v' \cdot G(p,s,c,v) \land S(p,s,c,v,v') \land J(s,c,v',w') \end{array}} \tag{1.5}$$

The use of witnesses (cf. Sect. 1.5.2) allows the separation of the previous proof rule in three parts: proof rules Gluing Invariant Preservation (1.7), Guard Strengthening (1.8) and Simulation (1.9). In practice, when discharging POs, it is simpler to deal with one part of the refinement PO at a time instead of dealing with all at once. We do not address the technical parts about the *partition* of the refinement POs but more details can be found in [10]. When non-deterministic witnesses are used, a proof obligation is generated to ensure that the witness is feasible.

Non-Deterministic Witness (WFIS): It ensures that each witness proposed in the concrete event indeed exists, in particular when the witness is a non-deterministic predicate. Witness are used when an abstract parameter is refined and disappears (being replaced by another parameter, a variable or an expression) or when an abstract variable that is assigned non-determistically is refined and disappears. In both cases, witnesses should related the refined element with a concrete representation (parameter, variable, expression) and this proof obligation ensures that the substitution is indeed feasible. The hypotheses include axioms, invariants and theorems (abstract and concrete), concrete guards and before-after predicate for witness. The goal is to confirm that the witness indeed exists. In Fig. 1.5(f), for convergent event *evt2* and witness p, this proof obligation is given by (1.6).

$$\boxed{\mathit{evt2/p/WFIS}: \begin{array}{l} A(s,c) \\ I(s,c,v) \\ J(s,c,v,w) \\ H_2(q,s,c,w) \\ T(q,s,c,w,w') \\ \vdash \exists p \cdot W_1(p,s,c,w,q) \end{array}} \tag{1.6}$$

With the use of witnesses, the refinement PO (1.5) can be split in three parts (which in practice makes the POs easier to manage and discharge). These three proof rules are presented below.

Gluing Invariant Preservation (INV): In a refinement, concrete invariants must be preserved for each concrete event. The hypotheses include axioms, abstract invariants and theorems plus concrete invariants and theorems, concrete guards, witnesses predicates for variables and concrete before-after predicates. The goal is each concrete invariant from the set of invariants in the refinement. In Fig. 1.5(f), for event *evt1* and each of the invariants $j(s,c,v,w)$ in *J(s,c,v,w)*, the respective proof obligation rule is given by (1.7).

$$\boxed{\begin{array}{ll} & A(s,c) \\ & I(s,c,v) \\ & J(s,c,v,w) \\ \textit{evt/inv/INV}: & H(q,s,c,w) \\ & W_2(v',s,c,w,q,w') \\ & T(q,s,c,w,w') \\ & \vdash j(s,c,v',w') \end{array}} \tag{1.7}$$

Guard Strengthening (GRD): It ensures that each abstract guard is at least as weak as the concrete one in the refining event. As a consequence, when a concrete event is enabled, the corresponding abstract one is also enabled. The hypotheses include axioms, abstract invariants and theorems, concrete invariants and theorems, concrete guards and witness predicates for parameters. The goal is each individual abstract guard from the set of abstract guards. In Fig. 1.5(f), for event *evt1* and each of the abstract guards *g(p,s,c,v)*, this proof obligation is given by (1.8).

$$\boxed{\begin{array}{ll} & A(s,c) \\ & I(s,c,v) \\ \textit{evt1/grd/GRD}: & J(s,c,v,w) \\ & H(q,s,c,w) \\ & W_1(p,s,c,w,q) \\ & \vdash g(p,s,c,v) \end{array}} \tag{1.8}$$

Simulation (SIM): It ensures that each action in a concrete event simulates the corresponding abstract action. When a concrete action is executed, the corresponding abstract one should not be contradicted. The hypotheses include axioms, abstract invariants and theorems, concrete invariants and theorems, concrete guards, witness predicates for refined parameters, witness predicate for refined abstract variables and the concrete before-after predicate for each concrete event. The goal is each individual abstract before-after predicate from the set of abstract assignments. In Fig. 1.5(f), for event *evt1* and one of the respective actions *act*, this proof obligation is given by (1.9).

$$\boxed{\mathit{evt1/act/SIM}:\ \begin{array}{l} A(s,c)\\ I(s,c,v)\\ J(s,c,v,w)\\ H(q,s,c,w)\\ W_1(p,s,c,w,q)\\ W_2(v',s,c,w,q,w')\\ T(q,s,c,w,w')\\ \vdash S(p,s,c,v,v') \end{array}} \tag{1.9}$$

When dealing with convergency and divergency, a variant is required to ensure that new events are not enabled forever. Otherwise, that possibly would not allow abstract events to occur resulting in the introduction of divergency to the model. The solution for this situation is the addition of a variant as described below.

Numeric Variant (NAT): It ensures that under the guards of each convergent or anticipated event, a proposed numeric variant is indeed a natural number. Also applicable to finiteness of set variants (FIN). The hypotheses include axioms, invariants and theorems (abstract and concrete) and guards for each convergent (or anticipated) event. The goal is to prove that the numeric variant is a natural number. In Fig. 1.5(f), for convergent event *evt2*, this proof obligation is given by (1.10).

$$\boxed{\mathit{evt2/NAT}:\ \begin{array}{l} A(s,c)\\ I(s,c,v)\\ J(s,c,v,w)\\ H_2(q,s,c,w)\\ \vdash n(s,c,w) \in \mathbb{N} \end{array}} \tag{1.10}$$

Numeric Variant Decreasing (VAR): It ensures that convergent events decrease the proposed numeric variant. Also applicable to finiteness of set variants (FIN). The hypotheses include axioms, invariants and theorems (abstract and concrete) and guards for each convergent (or anticipated) event. The goal is to prove that after the assignments the numeric variant decreases. In Fig. 1.5(f), for convergent event *evt2*, this proof obligation is given by (1.11).

$$\boxed{\mathit{evt2/VAR}:\ \begin{array}{l} A(s,c)\\ I(s,c,v)\\ J(s,c,v,w)\\ H_2(q,s,c,w)\\ T(q,s,c,w,w')\\ \vdash n(s,c,w') < n(s,c,w) \end{array}} \tag{1.11}$$

1.5.3.3 Enabledness PO

All the previous proof obligations are supported by the Event-B tool (Rodin platform described in Sect. 1.5.7). Nevertheless there is another proof obligation that is not supported by Rodin but it can be important when modelling a system: *enabledness*. Following the CSP notation for channels [153], we distinguish between parameters with an input (represented in CSP as "!") or output (represented in CSP as "?") behaviour. This distinction is important in particular for the generation of enabledness proof obligations during refinements. The enabledness proof obligation is given by [41] (described in that work as the *progress condition*):

$$\boxed{G \wedge J \Rightarrow H \vee H_N} \tag{1.12}$$

where G are the abstract guards, J are the gluing invariants, H are the concrete guards of refined events and H_N are the guards of the new events. The guards of the abstract event imply the guards of the concrete event or any of the new events guards. If an event is disabled in the concrete model, it should be disabled in the abstract model. Reducing the nondeterminism of individual events may result in reducing internal nondeterminism. The choice between a range of output values may be reduced during a refinement because the external choice is preserved. But the range of input values in a refinement must be preserved [41]. Using an example, let us consider event *Add1* in machine M illustrated by Fig. 1.6(a).

```
MACHINE M
VARIABLES s
INVARIANT s ⊆ ℕ

EVENT Add1 ≙
    ANY p WHERE
        p ∈ 0..9
    THEN
        s := s ∪ {p}
    END
```

(a) Machine S and event *Add1*

```
EVENT Add2 ≙
    ANY pWHERE
        p ∈ 0..5
    THEN
        s := s ∪ {p}
    END
```

(b) Event *Add2*

```
EVENT Add3 ≙
    ANY pWHERE
        p ∈ ∅
    THEN
        s := s ∪ {p}
    END
```

(c) Event *Add3*

FIGURE 1.6: Machine M and events *Add1*, *Add2* and *Add3*

If we consider that parameter p is an input parameter and that event *Add2* refines *Add1*, the enabledness PO resulting from (1.12) is given by:

$$p \in 0..9 \wedge s \subseteq \mathbb{N} \Rightarrow p \in 0..5$$

The previous PO cannot be proved and therefore the enabledness is violated. The concrete guard is strengthened and some abstract conditions ($x \in 6..9$) are not accepted in the concrete event. If we consider p as an output parameter and again *Add2* refining *Add1*, the enabledness proof obligations is:

$$p \in 0..9 \wedge s \subseteq \mathbb{N} \Rightarrow \exists p \cdot p \in 0..5$$

which can be easily proved as there exists a value for p between 0..5 from the hypotheses ($p \in 0..9$). But if we consider *Add3* as a refinement of *Add2*, the enabledness proof obligation is:

$$p \in 0..5 \land s \subseteq \mathbb{N} \Rightarrow (\exists p \cdot p \in \varnothing).$$

The enabledness is violated because we cannot prove this PO: there is no value of p that satisfies the concrete guard.

1.5.4 Feasibility and Initialisation

Contexts contain the static part of an Event-B model. It may contain carrier sets, constants, axioms and theorems. Carrier sets, that are user-defined types, only have a built-in assumption that they are not empty. Other assumptions about it can be added as axioms (e.g. carrier set s is finite: $finite(s)$).

An Event-B model is initialised by an event *initialisation* with no guards. This event does not have guards because the initialisation must always be possible. Moreover the expressions on the right-hand side of the initialisation actions cannot refer to any variable of the model, since the model is being *initialised* [9]. Returning to the birthday book example in Sect. 1.5, this action is a valid initialisation:

- $birthday :\mid birthday' = \{p0 \mapsto d0\}$

and this is an invalid initialisation:

- $birthday :\mid birthday' = birthday \cup \{p0 \mapsto d0\}$, because the right-hand side of the assignment refers to state *birthday* that have not been initialised yet.

The initialisation event cannot preserve the invariants because before that event, the system state *does not exist*; the initialisation event must establish the invariant for the first time. Thus, the other events, that are only possible after initialisation has taken place, can be enabled when the invariants hold. The invariant proof obligation for this invariant establishment is almost identical to the proof obligation rule INV (see Sect. 1.5.3) except that the invariants are not mentioned in the hypotheses of the sequent as described by PO rule (1.13) [9]. The initialisation provides a witness for the satisfiability of the invariants.

$$\boxed{\begin{array}{ll} & A(s,c) \\ \mathit{INITIALISATION/inv/INV}: & S(s,c,v') \\ & \vdash i(s,c,v') \end{array}} \tag{1.13}$$

Note that axioms in contexts do not generate proof obligations. Consequently they can introduce false assumptions and in that case, anything can be proved. To tackle this issue, *sanity tests* such as checking if a predicate that is clearly false can be discharged (e.g., $(1 = 0)$) can be used. If yes, a false predicate exists in the model and the properties and assertions may not hold. Note that this situation is different from introducing an invariant or a theorem that are clearly false: the corresponding PO for that invariant/theorem should not be found provable.

Events have feasibility proof obligations for non-deterministic actions as seen in Sect. 1.5.3. Moreover, the introduction of a guard that is always false results in that event being always disabled. Currently proof obligations are only generated for safety properties. Because the enabledness property is a liveness property, no proof obligation is generated to verify that situation. Nevertheless ProB [141], that is a model checker for the Rodin platform (see Sect. 1.5.7) allows the verification of enabledness considering small finite sets.

1.5.5 Event-B and Action Systems

In Event-B a system is specified as an abstract machine consisting of some state variables and some events (guarded actions) acting on that state. This is essentially the same structure as an action system which describes the behaviour of a parallel reactive system in terms of the guarded actions that can take place during its execution. As described in Sect. 1.3.3, an action in Action Systems is a predicate transformer that maps postconditions to preconditions. Event-B events are similar but from a more specific view where guards correspond to preconditions and the occurrence of the event lead to postconditions. We can compare both by defining the weakest preconditions (as described in Sect. 1.3.3) for events and actions respectively. We write $wp_M(\alpha, Q)$ for the weakest precondition guaranteeing that the event with label $\alpha \in A$ (A being the finite set of labels of machine M) will establish postcondition Q. An event labelled α from machine M has a canonical form in terms of a guard and a before-after predicate as follows [9]:

$$\textbf{event } \alpha \mathrel{\widehat{=}} \textbf{WHEN } G(v) \textbf{ THEN } v :\mid BA(v, v') \textbf{ END}.$$

The weakest precondition of this canonical form is [48]:

$$wp_M(\alpha, Q) \mathrel{\widehat{=}} G(v) \Rightarrow (\forall v' \cdot BA(v, v') \Rightarrow Q[v'/v]). \tag{1.14}$$

An action α from a basic action system $P = (A, v, P_i, P_a)$, where $\alpha \in A$ has a canonical form in terms of a guard and a before-after predicate as follows [26]:

$$action\ \alpha : G(v) \rightarrow v := BA(v, v').$$

The weakest precondition of this canonical form is [128, 129]:

$$\begin{aligned} wp(G(v) \rightarrow BA(v,v'), Q) &\mathrel{\widehat{=}} G(v) \Rightarrow wp(BA(v,v'), Q[v'/v]) \\ &\equiv G(v) \Rightarrow \forall v \cdot BA(v,v') \Rightarrow Q[v'/v]). \end{aligned} \tag{1.15}$$

The weakest precondition semantics (1.14) and (1.15) are equivalent. This occurs because Event-B can be seen as a realisation of the generic Action Systems formalism: both are predicate transformers mapping preconditions to postconditions.

1.5.6 CSP Semantics for Event-B Machines

Morgan's CSP semantics for Action Systems [129] allows traces, failures and divergences to be defined for action systems in terms of sequences of actions that can and cannot engage in. Butler [53] extends that work to include unbounded nondeterminism and defines the infinite traces for Action Systems. Schneider *et al* [159] developed a CSP viewpoint of Event-B refinement for traces, divergences and infinite traces (TDI). The notion of *traces* here refers to a finite sequence of events from a machine's alphabet (e.g. $tr \in \alpha M^*$), where alphabet are the observations of possible occurrences of events of M. The CSP semantics is also based on the weakest precondition semantics of events. The syntax used is slightly different from Sect. 1.3.3. For example, a sequence of actions $\langle act1, act2 \rangle$ occurs in exactly those states satisfying $\overline{wp}(act1; act2, true)$. That could be also expressed as [129, 159]:

$$\neg[act1, act2]false \equiv \overline{wp}(act1; act2, true).$$

Let S be a statement (of an event). Then $[S]Q$ denotes the weakest precondition for statement S to establish postcondition Q. Observe that for the case $Q = true$ we have $[when\ G(v)\ then\ v : |BA(v,v')\ end]true = true$. The semantics of machine M can be defined in terms of:

Traces The traces of a machine M are those sequences of events $tr = \langle a1, ..., an \rangle$ which are possible for M (after initialisation init): those that do not establish false:

$$traces(M) = \{tr \mid \neg[init; tr]false\} \tag{1.16}$$

Divergences A sequence of events tr is a divergence if the sequence of events is not guaranteed to terminate, i.e. $\neg[init; tr]true$. Thus

$$divergences(M) = \{tr \mid \neg[init; tr]true\} \tag{1.17}$$

Any Event-B machine M with events of the form given above in Sect. 1.5.2 is divergence-free (use of anticipated, convergence clause). This is because $[evt]true =$

$true$ for such events (and for init), and so $[init; tr]true = true$. Thus no potential divergence tr meets the condition $\neg[init; tr]true$.

Infinite Traces An infinite sequence of events $u = \langle u0, u1, ...\rangle$ is an infinite trace of M if there is an infinite sequence of predicates Pi such that $\neg[init](\neg P_0)$ (i.e. some execution of init reaches a state where P_0 holds), and $P_i \Rightarrow \neg[ui](\neg P_{i+1})$ for each i (i.e. if P_i holds then some execution of ui can reach a state where P_{i+1} holds).

$$infinites(M) = \{u \mid \exists\langle P_i\rangle_{i\in\mathbb{N}}\cdot\neg[init](\neg P_0) \wedge \forall i\cdot P_i \Rightarrow \neg[ui](\neg P_{i+1}) \qquad (1.18)$$

Moreover, the failures semantics of machine M can also be defined.

Failures A failure is a pair comprising a trace and a refusal; a refusal is a set of actions. Let R be a refusal. The behaviour (tr, R) is observed whenever the process first engages in all the actions in tr and then may refuse any action in R. The failures $tr : A^*; R : A$ of the action system $(A, init)$ are those for which

$$failures(M) = \{tr \mid \neg[init; tr]gd(R)\} \qquad (1.19)$$

is true initially, where A^* is a set of sequences with elements in A and $gd(R)$ is the disjunction of the guards of the actions in R. Thus R can be refused if $init$ then tr can reach a state in which no guard of any action in R is true [129].

Like some other formal notations, Event-B has tool support. The tool is called Rodin and it is briefly described below.

1.5.7 Rodin Platform

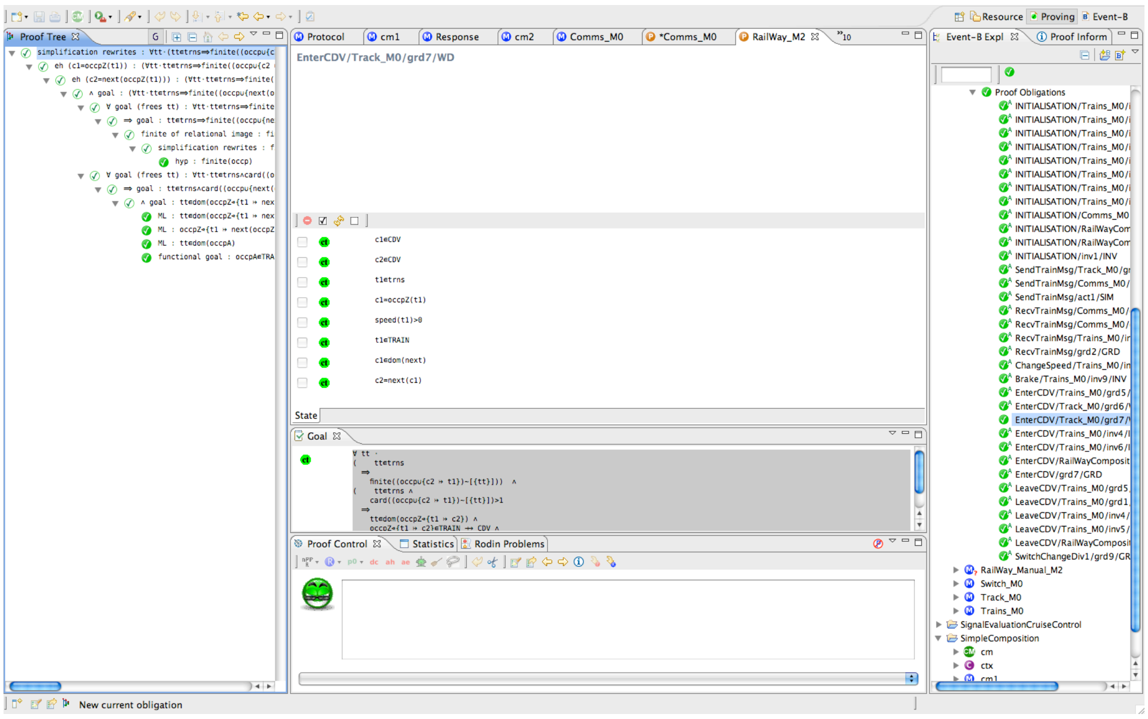
The Rodin platform [151] is the result of an EU research project[1]. It is a software tool, based on modern software programming tools developed to use Event-B notation [49, 13]. DEPLOY[2] is a continuation of this project and addresses scaling methodologies in requirements validation, requirements evolution, reuse, resilience, and scaling tooling in simulation, analysis and verification of formal models. Rodin was created to help the development of specifications based on the idea that a large complex or critical project should be started by modelling and reasoning about the specification. Moreover, formal reasoning is achieved through the generation of proof obligations. The (ambitious) purpose is to give more options to the industry when using formal methods and decrease the criticism that affects the formal methods [6]. Rodin strives to be a high usability tool showing that modelling does not have to be cumbersome nor hard to achieve.

[1]RODIN - Rigorous Open Development Environment for Open Systems: EU IST Project IST-511599

[2]DEPLOY - Industrial deployment of system engineering methods providing high dependability and productivity (2008 - 2011): FP VII Project 214158 under Strategic Objective IST-2007.1.2. Further information and downloadable tools are available at http://www.deploy-project.eu/

Besides formally validating the specifications according to some user defined properties (invariants), the main idea is to increase the understanding of the system that is being modelled. Therefore discharging the proof obligations correspond to the formal validation that the created system matches the requirements [49].

Rodin is an open source tool, based on the Eclipse Platform [66] and a complement for a rigorous modelling development [49]. The intention is to allow the tool to be customised according to the industry requirements by permitting the integration of functionalities considered necessary. Rodin supports a Static Checker that validates model properties. A Proof Generator is used to generate proofs obligations and these proofs can be discharged by an *Automatic Prover* (which is a theorem prover that discharges automatically as many proofs as possible as seen in Fig. 1.7). Proofs that are not automatically discharged have to be proved interactively. Another Rodin feature is the high level of extensibility reflected by, for instance, the ability to extend the default theorem prover (B4free provers provided by ClearSy [21]), model checking (ProB provided by University of Düsserdorf [141]) or even animate models (Brama provided by ClearSy [39] and ProB). Applying the UML framework using Event-B, it is also another approach developed using plug-in technology, where the concept of object oriented and classes are introduced and "merged" with Event-B notation [182, 170, 169, 171]. Figure 1.8 shows a screenshot of the user interface for Rodin Platform.



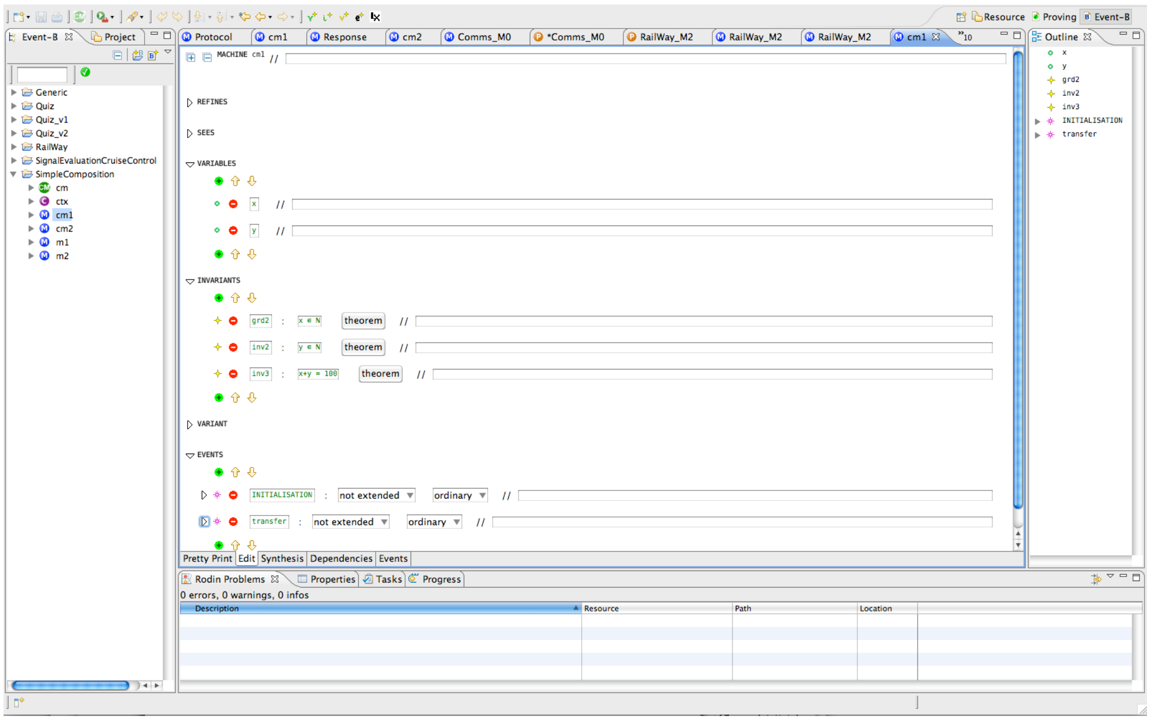
```
simplification rewrites : ∀tt·(tt∈trns⇒finite((occpu{c
  eh (c1=occpZ(t1)) : (∀tt·tt∈trns⇒finite((occpu{c2
    eh (c2=next(occpZ(t1))) : (∀tt·tt∈trns⇒finite(
      ∧ goal : (∀tt·tt∈trns⇒finite((occpu{next(o
        ∀ goal (frees tt) : ∀tt·tt∈trns⇒finite
          ⇒ goal : tt∈trns⇒finite((occpu{ne
            finite of relational image : fi
              simplification rewrites : f
                hyp : finite(occp)
        ∀ goal (frees tt) : ∀tt·tt∈trns∧card((o
          ⇒ goal : tt∈trns∧card((occpu{next(
            ∧ goal : tt∈dom(occpZ⊲{t1 ↦ nex
              ML : tt∈dom(occpZ⊲{t1 ↦ nex
              ML : occpZ⊲{t1 ↦ next(occpZ
              ML : tt∈dom(occpA)
              functional goal : occpA∈TRA
```

```
EnterCDV/Track_M0/grd7/WD

c1∈CDV
c2∈CDV
t1∈trns
c1=occpZ(t1)
speed(t1)>0
t1∈TRAIN
c1∈dom(next)
c2=next(c1)
```

```
∀ tt ·
(    tt∈trns
  ⇒
   finite((occpu{c2 ↦ t1})∼[{tt}]))  ∧
(    tt∈trns ∧
   card((occpu{c2 ↦ t1})∼[{tt}])>1
  ⇒
   tt∈dom(occpZ⊲{t1 ↦ c2}) ∧
   occpZ⊲{t1 ↦ c2}∈TRAIN ⇸ CDV ∧
```

```
Proof Obligations
  INITIALISATION/Trains_M0/i
  INITIALISATION/Trains_M0/i
  INITIALISATION/Trains_M0/i
  INITIALISATION/Trains_M0/i
  INITIALISATION/Trains_M0/i
  INITIALISATION/Trains_M0/i
  INITIALISATION/Trains_M0/i
  INITIALISATION/Trains_M0/i
  INITIALISATION/Comms_M0
  INITIALISATION/RailWayCom
  INITIALISATION/RailWayCom
  INITIALISATION/inv1/INV
  SendTrainMsg/Track_M0/gr
  SendTrainMsg/Comms_M0/
  SendTrainMsg/act1/SIM
  RecvTrainMsg/Comms_M0/
  RecvTrainMsg/Comms_M0/
  RecvTrainMsg/Trains_M0/ir
  RecvTrainMsg/grd2/GRD
  ChangeSpeed/Trains_M0/in
  Brake/Trains_M0/inv9/INV
  EnterCDV/Trains_M0/grd5/
  EnterCDV/Track_M0/grd6/
  EnterCDV/Track_M0/grd7/
  EnterCDV/Trains_M0/inv4/I
  EnterCDV/Trains_M0/inv6/I
  EnterCDV/RailWayComposit
  EnterCDV/grd7/GRD
  LeaveCDV/Trains_M0/grd5
  LeaveCDV/Trains_M0/grd1
  LeaveCDV/Trains_M0/inv4/
  LeaveCDV/Trains_M0/inv5/
  LeaveCDV/RailWayComposit
  SwitchChangeDiv1/grd9/GR
RailWay_Manual_M2
Switch_M0
Track_M0
Trains_M0
SignalEvaluationCruiseControl
SimpleComposition
  cm
  ctx
  cm1
```

FIGURE 1.7: The Proof Obligation Perspective: on the left, it is shown the proof tree of the selected PO; on the middle, on the top window are the hypotheses of the selected PO and just below the respective goal. Below the goal window are the buttons used to interactively discharge a PO; on the right, are the list of generated POs. Having all the POs green, it means that all the POs are discharged.



```
MACHINE cm1 //

▷ REFINES

▷ SEES

▽ VARIABLES
    x //
    y //

▽ INVARIANTS
    grd2 : x ∈ N   theorem //
    inv2 : y ∈ N   theorem //
    inv3 : x+y = 100   theorem //

▷ VARIANT

▽ EVENTS
    ▷ INITIALISATION : not extended ▼  ordinary ▼ //
    ▷ transfer : not extended ▼  ordinary ▼ //
```

FIGURE 1.8: The Event-B Perspective: on the left, the list of projects where the last one is expanded, showing several machines and a context; in the middle window, a view of a machine *cm*1 where the sections of variables, invariants and events are expanded and can be edited.

Next we cover the background of some of our contributions: composition and decomposition.

1.6 Composition

Composition has several definitions depending on the context. In a computer science context, (functional) composition can be defined as the act or mechanism of combining simple functions to build more complicated ones. It derives from a usual mathematical step of composing functions where the result of each function is passed as the argument of the next, and the result of the last one is the result of the whole. Engineering suggests another perspective of composition: ability to interact with sub-components. It is possible to represent concurrently-executing systems. In the formal methods context, in particular specifications, composition is the capacity to model the interaction of sub-components generating larger and more concrete specifications. Several formal methods define the interaction of specifications based on shared state or shared events (operations) [53]. Another possibility is a combination of the previous two approaches (sometimes called fusion composition [25]). The next sub-sections describe these different kind of interactions in different formal methodologies.

1.6.1 Shared State Composition

Shared state composition allows the interaction of sub-components by state sharing. Because variables usually define the state of a system, this composition is also known as shared variable. Back [23] using Action Systems applies the interaction of sub-components through external variables sharing. In that work, local variables are kept distinct and the global variables are shared among the processes in the parallel composition. Composing action systems $P = (v, P_{Ai}, P_A,)$ and $Q = (w, Q_{Bi}, Q_B)$ can be represented as follows:

$$P \parallel Q \mathrel{\widehat{=}} ((v, w), P_{Ai} \cup Q_{Bi}, P_A \cup Q_B)$$

The set of variables v and w are merged and the actions of both action systems (P_{Ai} and Q_{Bi} for the initialisation plus P_A and Q_B) are executed in parallel. The actions of $P \parallel Q$ are the union of both sets of actions and the interaction occurs when global variables are shared. Furthermore, under certain conditions parallel composition is considered monotonic w.r.t. data refinement [24]. If P' is a refinement of P, then $P' \parallel Q$ is a refinement of $P \parallel Q$ under a condition R (abstraction relation) as long as the interleaved execution of actions from Q preserves R.

Abadi and Lamport in [1] propose a shared variable composition as a conjunction of properties. Composition of systems means interaction within their environments and a system behaves properly only if its environment does. A system guarantees the properties M and L only under the environment assumption E. This can be described as $E \Rightarrow M \cap L$, where M and L are the safety and liveness properties of the system respectively.

There are some approaches for the development of composition using VDM [111, 77, 101]. One of the approaches is based on rely/guarantee conditions [104] where two state predicates are added as pre and postconditions of a specification, allowing *interference* between systems. This extension of VDM developed by Jones [103] permits the specification and development of concurrent shared-variable systems [187]. In this approach, a specification can then be described as:

$$(P, R, G, Q)$$

where P corresponds to the precondition and is a condition describing a set of states, while R, G, Q are rely-condition, guarantee-condition and postcondition respectively. The last three are conditions of state-transitions (predicates of two states: before and after state). A rely-condition states the postcondition that the rest of the system may achieve for any atomic step. Similarly, the guarantee-condition is the postcondition for any atomic step made by the operation itself [187]. The guarantee condition of parallel processes should imply the guarantee condition of the overall operation. [183] describes further work for composition using VDM combining ideas in concurrent separation logic and the rely/guarantee formalism. Assume-guarantee [81] is a similar style to rely-guarantee.

The B-Method includes a syntax for composition. There are some keywords that can be used to compose models like **Includes**, **Imports**, **Sees** and **Uses**. [158] describes the use of such keywords and restrictions. When a machine has a number of included machines (using the **Includes** keyword), several operations from different machines can be called in parallel. Combining operations results in the conjunction of the preconditions and the body of the parallel combination will be the parallel combination of all the bodies. This can be expressed as follows:

$$\begin{aligned}&\textbf{PRE}\ P1\ \textbf{THEN}\ S1\ \textbf{END} \parallel \textbf{PRE}\ P2\ \textbf{THEN}\ S2\ \textbf{END}\\ &= \textbf{PRE}\ P1 \wedge P2\ \textbf{THEN}\ S1 \parallel S2\ \textbf{END}\end{aligned}$$

where $P1, P2$ are preconditions and $S1, S2$ are operation statements. The preconditions are conjoined and the postconditions are called in parallel. Potet and Rouzaud [140] use some of these keywords to prove the correctness of composed specifications under certain restrictions. Blazy et al [33] use classical B to define specification patterns to be used as reuse mechanisms. One of the reuse mechanisms is composition where two patterns can be associated using the keyword **Extends** and proof obligations are generated when necessary for each kind of composition: juxtaposition (patterns are composed without defining any link between them), composition with inter-pattern links (relations between variables of the composed patterns can be added) and unification (some variables of the composed patterns can be merged and shared).

More recently, Abrial et al [124, 15] proposed a state-based decomposition for Event-B where the splitting of a system in sub-components (machines) is achieved using variables. In this case, decomposition is considered the inverse operation of composition and one can go from one to another and vice-versa. Figure 1.9 shows the decomposition of machine M(Fig. 1.9(a)) into machines $M1$(Fig. 1.9(b)) and $M2$(Fig. 1.9(c)). In a shared variable decomposition, just like the name suggests, variables can be shared as a consequence of the events' decomposition. Therefore, the events $evt1$ and $evt2$ from machine M are allocated to machine $M1$ and the rest of the events ($evt3$ and $evt4$) are allocated to machine $M2$. Variable $v2$ (Fig. 1.9(a)) is shared by events $evt2$ and $evt3$ that belong to different sub-components after decomposition (Figs. 1.9(b) and 1.9(c) respectively). Therefore $v2$ is considered a *shared variable*. In addition to the events allocated to each sub-component, it is necessary to introduce additional *external events* to each sub-component, to simulate how the shared variable is handled in the other sub-component. An external event is created based on the original event but only referring to shared variables. They have to be refined by the original events [15]. Other variables become parameters in that same event. $evt3\_ext$ is added to machine $M1$ and $evt2\_ext$

is added to machine $M2$, respectively. Sub-components $M1$ and $M2$ can be refined independently but shared variables must always be present and cannot be data-refined. The re-composition of the (refined) sub-components should always be possible (although not necessary) resulting in a refinement of the original system [124].

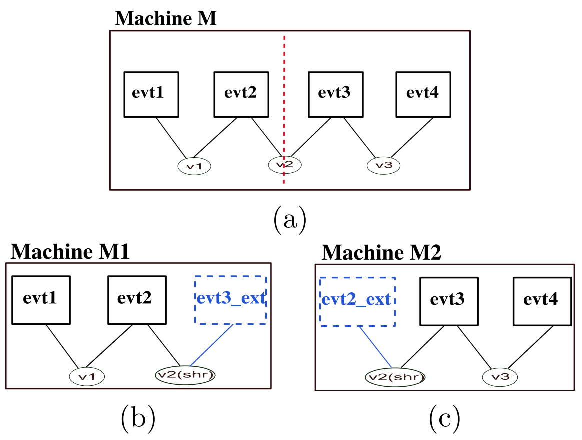


FIGURE 1.9: Shared Variable Decomposition of Machine S in Machines T and W with shared variable *v2*

While studying the several approaches for the composition of systems, we realised that there is a strong similarity between the rely/guarantee approach proposed by Jones and the shared variable decomposition for Event-B proposed by Abrial. The constraint originated by the shared variables and external events corresponds to the rely condition while the internal events correspond to the guarantee conditions as depicted in Fig. 1.10.

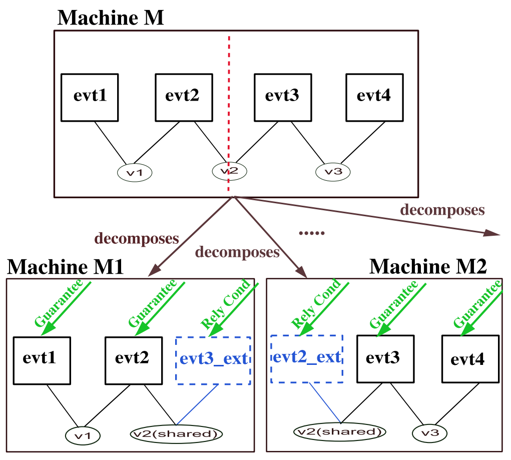


FIGURE 1.10: Shared Variable Decomposition Result

From $M1$ viewpoint (similar for $M2$), $evt3\_ext$ is the rely condition that modifications in the state in event $evt3$ are preserved in $M1$ and consequently $evt2$ is guaranteed to behave as the original one. Thus it is possible to make a correlation between these two approaches. Further study is required to use the developed worked on rely/guarantee for VDM in the shared variable decomposition for Event-B and we intend to do it in the future.

1.6.2 Shared Event Composition

The shared event approach is suitable for the development of distributed systems [42]: sub-components interact through synchronised events in parallel. Even for formal notation where the models have an explicit state space, the communication occurs at the event/operation level. CSP is an event-based methodology used for modelling distributed systems. Because of CSP's stateless approach, several works try to combine state-based and event-based approaches, as are the examples of combining CSP and B [43, 50, 181] or combining CSP with object oriented classes [75, 131]. The parallel composition of two processes P and Q is expressed as $P \parallel Q$. Events for process P are represented by their alphabet αP (similar to Q). The interaction happens by synchronising common events in $\alpha P \cap \alpha Q$, while events not in $\alpha P \cap \alpha Q$ can occur independently. An example of a synchronisation between events is represented as follows [53]:

$$(a \rightarrow P) \parallel (a \rightarrow Q) = a \rightarrow (P \parallel Q)$$

Events common to P and Q become single events in $P \parallel Q$. In CSP there exists a special class of events known as *communication* which is an event described by a pair $c.v$: c is the name of the *channel* on which the communication occurs and v is the *value* of the message to be communicated. A process ready to input (receives) any value x on the channel c, and then behave like Q can be described as: $c?x \rightarrow Q_x$. Similarly a process that outputs (sends) a value v on the channel c and then behaves like P can be defined as: $c!v \rightarrow P$ [153]. Channels can be considered members of the alphabet of the process and used for communication in only one direction and between two processes [92]. If two processes P and Q are composed in parallel and both have a common channel c, interaction happens whenever both processes are ready to engage in the common channel. If P is ready for $c!v$ and process Q is ready for $c?x$, v can be passed from P to Q [41]:

$$(c!v \rightarrow P) \parallel (c?x \rightarrow Q_x) = c!v \rightarrow (P \parallel Q_v)$$

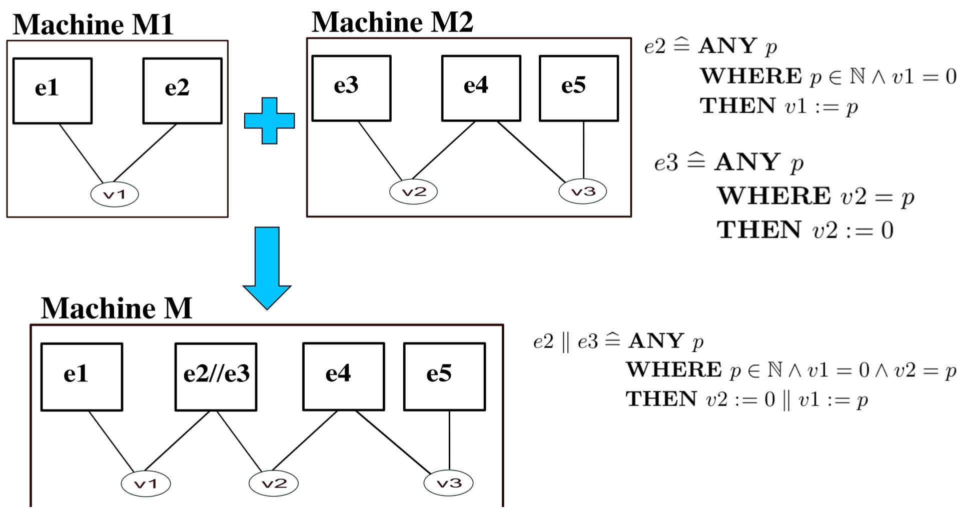
As expected the result is an output channel and the process Q receives the value v. This can also be applied for channels with input-input behaviour. The laws that govern the behaviour of $P \parallel Q$ are simple and regular. Some of these laws are described below although there are more properties defined in [92]:

- Commutativity: $P \parallel Q = Q \parallel P$, there is a logical symmetry between a process and its environment.
- Associativity: $(P \parallel Q) \parallel R = P \parallel (Q \parallel R)$, so when three processes are assembled, it does not matter in which order they are put together.

- Monotonicity: If $P \sqsubseteq P'$ then $P \parallel Q \sqsubseteq P' \parallel Q'$, for any Q. Components that are part of the parallel operation can be refined independently while preserving the parallel relationship.

In Z, it is possible to create big schemas based on small ones. That can be seen as composition, where specifications are reused, creating more complex systems. Since Z permits the refinement of specifications, composition and refinement can be applied at the same time to a model. [173] describes how the combination of schemas can be achieved, assuming that overloading - possibility that two distinct variables in the same scope might have identical names - is forbidden. The piping operator ($\gg$) is used to describe operations that have almost independent effects on two disjoint sets of state variables. If we compose the schemas $Op1$ and $Op2$ using the piping operator: $Op1 \gg Op2$, the output parameters of $Op1$ are matched with the inputs of $Op2$ and hidden, while the other components are merged as they would be in $Op1 \wedge Op2$. Another approach for the composition is through the use of views [34, 99]. A view is a partial specification of the entire system and can be evaluated directly from the requirements. Partial means that unnecessary details of the system's behaviour tackled by other views should be omitted. An advantage of views is that they can be constructed and analysed independently of other views. The interaction between views uses the schema calculus and standard logic operators. Views can be connected by an invariant relating their state (state-based approach), or connected by synchronising their operations (event-based approach) or a mix of both. [34] discusses a similar approach using views, but the composition is through coupling schemas (relation between different state spaces). By relating several state schemas and respecting some properties, it is described how the composition can be achieved based on three techniques: data refinement, view composition and viewpoint unification. Circus (that combines Z and CSP) programs are sequences of paragraphs: channel declarations, channel set definitions, Z paragraphs, or process definitions. A system is defined as a process that encapsulates some state and communicates through channels. The generic channel declaration $\mathbf{channel}[T]c : T$ declares a family of channels c and $[T]$ determines the type of the values that are communicated through channel c. An action can be a schema, a guarded command, an invocation of another action, or a combination of these constructs using CSP operators. The CSP operators of sequence, external and internal choice, parallelism, interleaving, their corresponding iterated operators, and hiding can be used to compose actions [54, 155]. The prefixing operator is standard, but a guard construction may be associated with it. For instance, given a Z predicate p , if the condition p is true , the action $p \;\&\; c?x \rightarrow A$ inputs a value through channel c and assigns it to the variable x , and then behaves like A, which has the variable x in scope. If, however, the condition p is false , the same action blocks. Such enabling conditions like p may be associated with any action. The CSP operators of sequence, external and internal choice, parallelism, interleaving, their corresponding iterated operators, and hiding may also be used to compose actions.

Butler [45] proposes a shared event composition for Event-B inspired by CSP and Action Systems with event sharing as seen in Fig. 1.11. In this kind of composition, machines with independent state spaces (variable sharing is not permitted) can be composed by sharing events. Since it is based on CSP synchronisation, this composition also inherits the CSP properties for the channel communication described above. As aforementioned in Sect. 1.5.5, Action Systems and Event-B are related. Based on that relation, Butler [42, 45] defined the relation between the parallel composition of actions (including composition with Value-Passing) in Action Systems and the B operations/Event-B events. This definition is described by Definition 1.2 as described below. In Fig. 1.11, machine *M1* has events *e1* and *e2* and variable *v1*. Machine *M2* has events *e3*, *e4* and *e5* and variables *v2* and *v3*. These two machines can be composed originating machine *M*. In particular, events *e2* and *e3* can be composed. Moreover, in case both events have a common parameter p, this can be used for message passing between machines $M1$ and $M2$. The composition of synchronised Action Systems actions (using Event-B syntax for actions) generates a new action whose guard is the conjunction of the original guards and the actions are executed in parallel [42]:



Machine M1
e1
e2
v1
Machine M2
e3
e4
e5
v2
v3
Machine M
e1
e2//e3
e4
e5
v1
v2
v3
$e2 \mathrel{\widehat{=}}$ **ANY** p **WHERE** $p \in \mathbb{N} \wedge v1 = 0$ **THEN** $v1 := p$
$e3 \mathrel{\widehat{=}}$ **ANY** p **WHERE** $v2 = p$ **THEN** $v2 := 0$
$e2 \parallel e3 \mathrel{\widehat{=}}$ **ANY** p **WHERE** $p \in \mathbb{N} \wedge v1 = 0 \wedge v2 = p$ **THEN** $v2 := 0 \parallel v1 := p$

FIGURE 1.11: Shared Event Composition of machines *M1* and *M2* into *M* with composition of events *e2* and *e3*

Definition 1.2. If both events $evt1$ and $evt2$ have a parameter p:

$evt1 \mathrel{\widehat{=}}$ **ANY** $p?, x$ **WHERE** $p? \in C \wedge G(p?, x, m)$ **THEN** $S(p?, x, m)$ **END**
$evt2 \mathrel{\widehat{=}}$ **ANY** $p!, y$ **WHERE** $H(p!, y, n)$ **THEN** $T(p!, y, n)$ **END**

then:

$evt1 \parallel evt2 \mathrel{\widehat{=}}$
ANY $p!, x, y$ **WHERE** $p! \in C \wedge G(p!, x, m) \;\wedge\; H(p!, y, n)$
THEN $S(p!, x, m) \parallel T(p!, y, n)$ **END**

where x, y, p are sets of parameters from each of the actions $evt1$ and $evt2$. Action $evt1$ has $p?$ as an input parameter and $evt2$ has $p!$ as an output parameter and the resulting composition is $p!$ itself an output parameter (like in CSP). This property can be used to model value-passing systems: the parameter $p!$ is written in $evt2$ and its value is used as input for parameter $p?$ to be used in $G(p?, x, m)$ and $S(p?, x, m)$. An interpretation of this composition is that the value p is sent from $evt2$ and received in $evt1$. The fusion of parameters in only possible if the types of the shared parameter match or are a subset of each other:

$$p! \in C \wedge p? \in D \Rightarrow C \cap D \neq \varnothing \tag{1.20}$$

where C and D are types (carrier sets). Actions with shared parameters of type input can also be composed and the shared parameter has input behaviour. Actions with shared parameters of type output cannot be composed since this could lead to a deadlock state [42].

A relation between the infinite-traces semantics of CSP and Action Systems is defined in Definition 1.1. Based on that definition, the event-based parallel composition of action systems can be shown to correspond to the CSP parallel-composition of processes as described by Theorem 1.1 and proved in [53]: the infinite-traces semantic of syntactic parallel composition of actions is equal to the infinite-traces semantic composition of individual actions.

Theorem 1.1 (see [53, Theorem 5.17]). Let $\{\![M]\!\}$ represent the infinite-traces semantics of action system M (similar for $\{\![N]\!\}$ and action system N). Then the infinite-traces semantics of CSP can be applied to Action Systems according to Definition 1.1: the infinite-traces semantics of action system M in parallel with N, $M \parallel N$ is given as [3]:

$$\{\![M \parallel N]\!\} = \{\![M]\!\} \parallel \{\![N]\!\} \tag{1.21}$$

In order to give a CSP semantics to Event-B we simply treat an Event-B machine as an action system. Doing this just requires treating an Event-B event as a predicate transformer as shown previously in Section 1.5.5. Therefore with respect to infinite-traces semantics, the parallel composition of action systems corresponds to the parallel composition of Event-B events. Moreover the properties of the parallel composition of action systems are also inherited in the parallel composition of machines. The most relevants are *commutativity* along with *monotonicity*: if M and N are Event-B machines and $M \sqsubseteq M'$ then $M \parallel N \sqsubseteq M' \parallel N$, for any N. Therefore machines M and N can be refined independently while the properties of the parallel composition $M \parallel N$ are still preserved. This is one of the most important and powerful properties that shared event composition in Event-B inherits from Action Systems and CSP. The monotonicity property for the shared event composition in Event-B is proved by means of proof

[3]This theorem is shown and proved in [53], theorem 5.17 on page 67.

obligations in Sect. 2.3.4.

1.6.3 Fusion Composition

Fusion composition is another kind of composition which can be seen as a combination of the previous two approaches. Back and Butler [25] extend the notion of a product operator for refinement calculus of Back [22]. The *fusion* operator is introduced as a generalisation of the product operator preserving the monotonicity and conjunctivity properties. The fusion operator can be used to conjoin two specifications into a larger specification that refines both specifications within their combined termination condition. As a result the non-determinism is reduced on the termination behaviour of both specifications. Poppleton [138] follows the previous work by proposing a composition using the fusion operator as a way to reuse existing models for Event-B. A proposal for development of feature oriented specifications [139, 70] uses the fusion operator. Consider machines *M1* and *M2* in Fig. 1.12 which are fused by combining variables and events, generating machine *M*. Machine *M1* has a set of variables v (variables x are assigned in the event e and variables y are kept unchanged), a context defined by carrier sets s, constant c, axioms A_1 and invariant I_1. Similarly, machine *M2* has variables z (divided in a and b) and same context properties except the axioms A_2 and invariant I_2. The union of the variables of each model corresponds to the set of variables of machine M. The common events (we consider that events e and f are common) are composed similarly to shared event composition described in Sect. 1.6.2.

Machine M1
$v = x \cup y$ $/* \textit{variables} */$
$s, c, A_1(s, c)$ $/* \textit{context} */$
$I_1(s, c, v)$ $/* \textit{invariant} */$
$\textit{event} : e \mathrel{\widehat{=}}$
  ANY $p1$ **WHERE** $G_1(p1, v)$
  THEN $x := S_1(p1, v)$ **END**

(a)

Machine M2
$z = a \cup b$ $/* \textit{variables} */$
$s, c, A_2(s, c)$ $/* \textit{context} */$
$I_2(s, c, z)$ $/* \textit{invariant} */$
$\textit{event} : f \mathrel{\widehat{=}}$
  ANY $p2$ **WHERE** $G_2(p2, z)$
  THEN $a := S_2(p1, z)$ **END**

(b)

Machine M
$v, z = x \cup y \cup a \cup b$ $/* \textit{variables} */$
$s, c, A_1(s, c) \wedge A_2(s, c)$ $/* \textit{context} */$
$I_1(s, c, v) \wedge I_2(s, c, z)$ $/* \textit{invariant} */$
$\textit{event} : e \odot f \mathrel{\widehat{=}}$
  ANY $p1, p2$ **WHERE** $G_1(p1, v) \wedge G_2(p2, z)$
  THEN $x := S_1(p1, v) \parallel a := S_2(p1, z)$ **END**

(c)

FIGURE 1.12: Fusion Composition of machines *M1* and *M2* into machine *M*

Event fusion preserves the refinement properties of the model [138] and as a requirement, shared variables should be refined in the same functional manner in both machines.

Decomposition, that can be seen as the inverse operation of composition, is briefly discussed in the next section.

1.7 Decomposition

The development of specifications in a “top-down” style starts with an abstract model of the envisaged system. Throughout refinements the initial model becomes less abstract and more concrete, closer to an implementation. As a consequence, there is a better view of the system as a whole and design decisions can be taken. Nonetheless refinements of a system bring complexity and tractability problems when the model augments in a way that becomes cumbersome to manage [124]. Decomposition is precisely the process by which a single model can be split into various sub-components in a systematic fashion. The complexity of the whole model is decreased by studying, and thus refining, each sub-component independently of the others [124]. The independent sub-components can be developed in parallel which is attractive in an industrial environment. As a result of the attractive benefits of decomposition, it is a topic of interest that has been explored in several areas like mathematics, in different areas of engineering and also in different formal methodologies. There is a strong relation between composition and decomposition: they can be seen as the inverse operation of each other. Therefore the related work is very often interleaved as we present below.

Abadi and Lamport [2] suggest a decomposition of concurrent systems (interleaving and non-interleaving representation) in the style of “composition is conjunction” using TLA [108]. The goal is to facilitate the decomposition of complete systems and respective proofs by reasoning about the sub-components when the environment conditions are safety properties.

Moore [127] suggests a decomposition of system requirements and respective proofs using the CSP Trace Model. The method emphasizes the decomposition of high-level requirements and reasons about the safety of non-divergent processes. The only way a process can communicate with another process executing concurrently is through CSP-like communication channels; no shared variables are permitted. The method proceeds iteratively, until the appropriate requirements for the component processes and the minimal set of synchronization requirements are found.

Jian [101] uses a combination of data reification and operation decomposition in VDM (DD-VDM) to reason about data decomposition. Data decomposition is based on the ideas of model splitting, modularisation and operation decomposition. The operations in the sub-models are viewed as the operations working on the whole model and rules are added in DD-VDM concerning the interaction of several sub-models. [102] is the continuation of that work by developing parallel object-oriented programs in the VDM framework.

Butler [40] suggests a decomposition approach for Action Systems with value-passing, internal actions and parallel composition as described in Sect. 1.6.2. As a continuation of that approach, Butler and Waldén [52] combine Action Systems and classical B to derive parallel and distributed systems.

Two methods have been identified for the Event-B decomposition: shared variable (Fig. 1.9) and shared event (Fig. 1.13). The shared event decomposition can be seen as the inverse operation of the shared event composition described in Sect. 1.6.2. In this case, the decomposition requires the definition of which variables are allocated to which sub-component (in Fig. 1.13, $v1$ is allocated to machine $M1$ and variables $v2, v3$ are allocated to machine $M2$). Event $evt2$ is *shared* since uses variables $v1$ and $v2$ allocated to different sub-components. During the decomposition, $evt2$ is decomposed into $evt2'$ (containing only guards and actions related to $v1$) and $evt2''$ (containing only guards and actions related to $v2$). We follow the shared event decomposition approach and in a pragmatic way, we aim to study and specify a decomposition tool. Because shared-event decomposition is monotonic [45], the generated sub-components can be further refined independently. So $M1$ and $M2$ can be refined independently into $M11$, $M12$... and $M21$, $M22$... respectively . Therefore we can introduce team development: several developers share parts of the same model and can work independently in parallel (we show this option is our case study in Chapter 6). Besides alleviating problems when dealing with complex specifications, decomposition also partition the proof obligations which are expected to be easier to be discharged in the sub-components. Next we discuss in more detail the shared event decomposition before introducing our contribution in Chapter 4.

1.7.1 Shared Event Decomposition

In Event-B, decomposition of a component (specification) corresponds to distributing events and variables among the sub-components. Shared event decomposition does not permit variable sharing and an event can be split into different sub-components as seen in Fig. 1.13. The sub-components can be further refined independently according to the monotonicity property of decomposition [45].

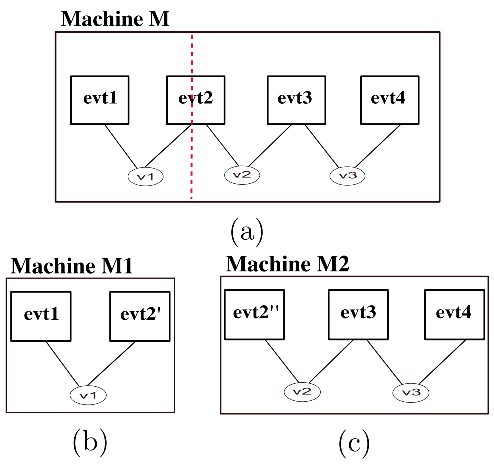


FIGURE 1.13: Shared event decomposition of machine M into machines $M1$ and $M2$ with shared event $evt2$

The decomposition can be seen as syntactic and semantic: syntactic because the sub-components are a consequence of the syntactic partition of the component ; semantic in the sense that the sub-components can lose some information (invariants that relate sub-components) but the behaviour of the recomposed sub-components is the same as the non-decomposed component (i.e. the recomposition is a valid refinement of the abstract component).

Figure 1.14 shows the decomposition of machines *M1* into *M3_0* and *M4_0*. *M3_0* and *M4_0* are refined independently until *M3_m* and *M4_n* are reached. It should be possible to recompose *M3_m* and *M4_n* into *cM2* and proved that *cM2* is a refinement of *M1*. This is equivalent to express the monotonicity property of decomposition as:

$$M1 \sqsubseteq (M3\_0 \parallel M4\_0) \sqsubseteq (M3\_m \parallel M4\_n)$$

The shared event parallel composition of *M3_0* and *M4_0* refines *M1*. Also the parallel composition of the individual refinements of *M3_0* and *M4_0* (*M3_m* and *M4_n* respectively) are a refinement of the *M1*.

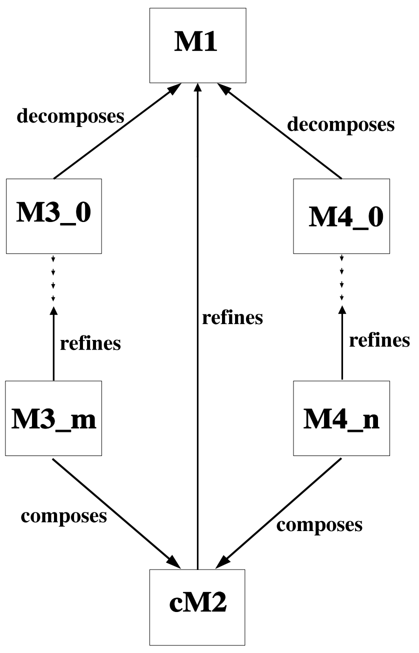


FIGURE 1.14: Decomposition, Recomposition and Refinement

Consider machine M in Fig. 1.15(a) containing variables v, z and events evt_1, evt_2 and evt_3. Each event has a parameter p_i, guards G_{ij} and assignments to variables using predicates S_{ij}, where i and j are indexes corresponding to events elements. Machine M is decomposed into machines *M1* and *M2* as seen in Fig. 1.15. Variable v is allocated to machine *M1* and variable z is allocated to machine *M2* meaning that event evt_1 (that only depends on that variable) is part of *M1* and event evt_2 (only dependent on z) is part of *M2*. Event evt_3 uses both variables so the event is split in two parts: guards

```
MACHINE M
VARIABLES $v, z$
EVENT $evt_1 \triangleq$
    ANY $p1$ WHERE
        $G_1(v, p1)$
    THEN
        $v := S_1(v, p1)$
    END
EVENT $evt_2 \triangleq$
    ANY $p2$ WHERE
        $G_2(z, p2)$
    THEN
        $z := S_2(z, p2)$
    END
EVENT $evt_3 \triangleq$
    ANY $p3$ WHERE
        $G_{31}(v, p3)$
        $G_{32}(z, p3)$
    THEN
        $v := S_{31}(v, p3)$
        $z := S_{32}(z, p3)$
    END
```

(a) Machine M

```
MACHINE M1
VARIABLES $v$
EVENT $evt_1 \triangleq$
    ANY $p1$ WHERE
        $G_1(v, p1)$
    THEN
        $v := S_1(v, p1)$
    END
EVENT $evt_3 \triangleq$
    ANY $p3$ WHERE
        $G_{31}(v, p3)$
    THEN
        $v := S_{31}(v, p3)$
    END
```

(b) Machine M1

```
MACHINE M2
VARIABLES $z$
EVENT $evt_2 \triangleq$
    ANY $p2$ WHERE
        $G_2(z, p2)$
    THEN
        z := $S_z(z, p2)$
    END
EVENT $evt_3 \triangleq$
    ANY $p3$ WHERE
        $G_{32}(z, p3)$
    THEN
        $z := S_{32}(z, p3)$
    END
```

(c) Machine M2

FIGURE 1.15: Machines *M1* and *M2* resulting from the shared event decomposition of machine *M*

and actions related with variable v are decomposed into machine $M1$ and guards and actions related to variable z are stored in machine $M2$.

Event evt_3 from machine M has a parameter $p3$. During the decomposition $p3$ is shared between the sub-events and allows the interaction between the sub-components $M1$ and $M2$. This correspond to modelling value-passing systems as described in [40, 41] for Action Systems or in [42] for B and in [45] for Event-B.

1.7.2 Shared Variable Decomposition

In Event-B, the shared variable decomposition allows variable sharing and external events are introduced in the sub-components to ensure that the behaviour of the shared variables is maintained in all sub-components. Such approach is suitable for designing parallel algorithms [42] (an example can be found in [90]). The re-composition of the (refined) sub-components should always be possible resulting in a refinement of the original system. Therefore what was described in Fig. 1.14 can also be applied to the shared variable decomposition and it is proved in [8].

Consider again machine M in Fig. 1.16(a) containing variables v, z and events evt_1, evt_2 and evt_3. Machine M is shared variable decomposed into machines *M1* and *M2*. Event evt_1 is allocated to machine *M1* and events evt_2, evt_3 are allocated to machine *M2*. Consequently variable v is shared. Event evt_3_ext must be added to machine *M1* to ensure that the behaviour of (shared) variable v in the machine M is preserved in that sub-component. Similarly, in machine $M2$, event evt_1_ext is added to simulate the behaviour of v from the machine $M2$. Machines $M1$ and $M2$ can be further refined

```
MACHINE M
VARIABLES $v, z$
EVENT $evt_1 \widehat{=}$
    ANY $p1$ WHERE
        $G_1(v, p1)$
    THEN
        $v := S_1(v, p1)$
    END
EVENT $evt_2 \widehat{=}$
    ANY $p2$ WHERE
        $G_2(z, p2)$
    THEN
        $z := S_2(z, p2)$
    END
EVENT $evt_3 \widehat{=}$
    ANY $p3$ WHERE
        $G_{31}(v, p3)$
        $G_{32}(z, p3)$
    THEN
        $v := S_{31}(v, p3)$
        $z := S_{32}(z, p3)$
    END
```

(a) Machine M

```
MACHINE M1
VARIABLES $v$ /*shared var*/
EVENT $evt_1 \widehat{=}$
    ANY p1 WHERE
        $G_1(v, p)$
    THEN
        $v := S_1(v, p)$
    END
EVENT $evt_3\_ext \widehat{=}$
    ANY $p3$ WHERE
        $G_{31}(v, p3)$
    THEN
        $v := S_3(v, p3)$
    END
```

(b) Machine M1

```
MACHINE M2
VARIABLES $z$
                $v$ /*shared var*/
EVENT $evt_1\_ext \widehat{=}$
    ANY $p1$ WHERE
        $G_1(v, p1)$
    THEN
        $v := S_1(v, p1)$
EVENT $evt_2 \widehat{=}$
    ANY $p2$ WHERE
        $G_2(z, p2)$
    THEN
        $z := S_2(z, p2)$
    END
EVENT $evt_3 \widehat{=}$
    ANY $p3$ WHERE
        $G_{31}(v, p3)$
        $G_{32}(z, p3)$
    THEN
        $v := S_{31}(v, p3)$
        $z := S_{32}(z, p3)$
    END
```

(c) Machine M2

FIGURE 1.16: Machines *M1* and *M2* resulting from the shared variable decomposition of machine *M*

independently as long as the external events and shared variables are present. Moreover, the shared variables and the external events cannot be refined.

The following chapters describe our work applied to three reuse mechanisms: *composition*, *generic instantiation* and *decomposition*. Each chapter contains a small case study applying the respective mechanism. A more complex case study is presented in the end to illustrate the use of the reuse mechanisms when developing models.

Chapter 2

Shared Event Composition for Event-B

The development of a system can start with the creation of a specification. Following this viewpoint, we claim that often a specification can be constructed from the combination of specifications. The combination of specifications can be seen as *composition*. Event-B is a formal method that allows modelling and refinement of systems. The combination, reuse and validation of component specifications is not currently supported in Event-B. We extend the Event-B formalism using shared event composition as an option for developing distributed systems. Refinement is used in the development of specifications using *composed machines* and we prove that properties and proof obligations of specifications can be reused to ensure valid composed specifications. The main contribution of this work is the Event-B extension to support shared event composition including the definition of static checks and proof obligations (POs) for a composed machine. Composition and refinement are coupled to gradually develop a model in a stepwise manner. Moreover, composition is the preliminary work towards decomposition (described in Chapter 4) as it defines a methodology for (de)composing specifications. We explore the composition of specifications by defining properties and (reuse of) proof obligations. These contributions are supported by a tool developed in the Rodin platform (*parallel composition plug-in* [162]). This chapter is based on papers accepted for the B workshop running in parallel with FM 2011 (International Symposium on Formal Methods) [161] and in FMCO 2010 (International Symposia on Formal Methods for Components and Objects) [164].

2.1 Introduction

In a "top-down" style, the initial model abstracts the most important behaviour and state of the system. Systems can often be seen as a combination and interaction of several sub-specifications (hereafter called sub-components) where each sub-component has its own functionality aspect. This view introduces *modularity* in the system: different sub-components represent a particular functionality and changes in the sub-components are accommodated more gracefully [99] in the system specification. We use *composition* to structure specifications through the interaction of sub-components seen as independent modules. This use of composition is not new in other formal notations: examples are [191, 106, 138] as described in Sect. 1.6. Here we express how we can (re)use composition for building specifications in Event-B through sub-components (modules) interaction, benefiting from their properties and proof obligations inspired by views in Z [99]. The interesting part of composition involves the interaction of sub-components which occurs by shared state, shared operations or a combination of both (for example, fusion composition) as discussed in Chapter 1. Although sub-components usually have states, in our approach we mainly focus on their (visible) operations similar to the CSP view [129, 53]. Therefore we follow a *shared event composition approach* where events/operations from different sub-components are synchronised in parallel. We constrain sub-components to have independent state spaces and consequently avoid dealing with sub-components that have intersecting states like it happens in a shared state approach [144, 145].

This chapter is structured as follows: Sect. 2.2 introduces the notion and properties for shared event composition. The notion of composed machine, respective static checks, proof obligations and the monotonicity property are introduced in Sect. 2.3. Section 2.4 illustrates the application of the shared event composition to a distributed system case study: file transfer system. Related work is described in Sect. 2.5. Conclusions and future work are drawn in Sect. 2.6.

2.2 Shared Event Approach

Sub-component specifications, that are part of a full system specification, deal with a particular aspect of the system being modelled. Sub-component interaction must be verified to comply with the desired behavioural semantic of the system. The interaction usually occurs as a shared state, shared event or a combination of both as described in Sect. 1.6. The kind of interaction usually depends on the characteristics of the specified system. For instance, when specifying an automated teller machine (ATM) system, *user* and *cashMachine* can have separate specifications. Both specifications can define variables to describe the used debit/credit cards for the transactions and the composition of these two specifications can interact through *shared variables*: the variables representing the cards. On the other hand, a shared event composition allows sub-components to interact through synchronised events. The specification *user* can have an event that defines the personal identification number (PIN) of the card: *user_defines_PIN*. *cashMachine* can contain an event that changes the card PIN: *change_PIN_card*. Furthermore an additional sub-component *serverBankValidation* can have events defining when a bank operation is enabled. One of these events can be *validate_user_operation_card*. A shared event composition of these specifications results in a new event *user_change_PIN* that allows the introduction of a new PIN for a particular card when the conditions defined in *validate_user_operation_card* are enabled. Such event could be specified by composing events *user_define_PIN*, *change_PIN_card* and *validate_user_operation_card*.

Here we focus on the developments using shared event composition only, where composition is treated as the *conjunction* of individual elements' properties: conjunction of individual invariants, union of variables and synchronisation of events. Events when synchronised are composed as described in Def. 1.2. Machine properties are merged by the conjunction of invariants as seen in Def. 2.1.

Definition 2.1. Let machines $M1 \dots Mm$ have variables $v1 \dots vm$ respectively. Then if machines $M1 \dots Mm$ are composed in parallel, the invariant of the composed machine $M1 \parallel \dots \parallel Mm$ is given as:

$$I(M1 \parallel \cdots \parallel Mm) \mathrel{\widehat{=}} I_1(s, c, v1) \wedge \cdots \wedge I_m(s, c, vm). \tag{2.1}$$

When sub-components are composed it is desirable to define properties that relate the individual sub-components allowing interactions. These properties are expressed by adding *composition invariants* $I_{CM}(s, c, v1, \dots, vm)$ to the composed machine constraining the variables of all machines being composed. Therefore a more complete version of the conjunction of invariants is described in Def. 2.2.

Definition 2.2. The invariant of the parallel composition of machines $M1$ to Mm with variables $v1$ to vm respectively is the conjunction of the individual invariants (Def. 2.1) and the composition invariant $I_{CM}(s, c, v1, \dots, vm)$:

$$I(M1 \parallel \cdots \parallel Mm) \mathrel{\widehat{=}} I_1(s, c, v1) \wedge \cdots \wedge I_m(s, c, vm) \wedge I_{CM}(s, c, v1, \dots, vm). \tag{2.2}$$

In Fig. 1.11, *composed machine M* can have as invariant the conjunction of the individual invariants as defined by Def. 2.2: $I(M1 \parallel M2) \mathrel{\widehat{=}} I_{M1}(s, c, v1) \wedge I_{M2}(s, c, v2, v3)$ plus possible composition invariant $I_{CM}(s, c, v1, v2, v3)$.

2.3 Composed Machines: Composition and Refinement

We define a new construct *composed machine*, representing the shared event composition of Event-B machines. We aim to have a construct that remains reactive to changes in the sub-components in a way that has a minimal effect on the entire specification. Consequently this representation of the composition is *structural*. The interaction of sub-components, following a "top-down" approach, can represent a *refinement* of an existing abstraction. In that case, to formalise the composition, it is necessary to define *composition POs* plus *refinement POs*. In the following sections, we introduce the structure of a composed machine, respective POs, prove the monotonicity property for shared event composition and describe the required static checks.

2.3.1 Structure of Composed Machines

A shared event composed machine is expressed as the parallel conjunction of sub-component properties. Composed machine CM defined by machines $M1, \dots, Mm$ can be seen in Fig. 2.1. Machines are composed in parallel including their properties and events: $CM \mathrel{\widehat{=}} M1 \parallel \cdots \parallel Mm$. Moreover:

- The composed machine variables are all the sub-component variables (v_1 from $M1$, v_2 from $M2$, ..., v_m from Mm) and are state-space disjoint.
- The invariants of the composed machine are defined as Def. 2.2.
- The composed events are defined according to Def. 1.2.

COMPOSED MACHINE CM **SEES** Ctx
INCLUDES $M_1, \dots, M_m$
VARIABLES $v_1, \dots, v_m$
INVARIANTS $I_{CM}(s, c, v_1, v_2, \dots, v_m)$
EVENTS
    $INITIALISATION \mathrel{\widehat{=}} M1.INITIALISATION \parallel \dots Mm.INITIALISATION$
    $evt_{11} \mathrel{\widehat{=}} M1.evt_{11} \parallel \dots Mm.evt_{m1}$
    $\dots$
    $evt_{1p} \mathrel{\widehat{=}} M1.evt_{1p} \parallel \dots Mm.evt_{m1}\ evt_{1p}$
END

FIGURE 2.1: Composed machine CM composing machines $M1$ to Mm seeing context Ctx

$I_{CM}(s, c, v_1, v_2, \dots, v_m)$ expresses the properties relating the states of sub-components. When a composed machine is used as a combination of composition and refinement, it refines an abstract model and just like in an ordinary machine, abstract events must be refined. For instance, a composed machine CM refining abstract machine $M0$ can be expressed as $(M0 \sqsubseteq CM) \equiv (M0 \sqsubseteq M1 \parallel \cdots \parallel Mm)$. The next section discusses static checks that are required in order to implement a tool for composition.

2.3.2 Static Checks

For the implementation of a tool for composition (Sect. 5.2), composed machines need to be validated against some well-formedness conditions. The shared event composition relies on these definitions:

- The state space of the composed machine is defined as the composition of the sub-components' state space.
- The invariant of the composed machine is defined as the conjunction of the individual invariants plus possible additional composition invariants.
- Sub-components can *communicate* via shared parameters during the parallel occurrence of events (composed events).

We distinguish between necessary technical conditions for the composition and methodological conditions (convenient and for simplicity). The technical conditions are as follows:

- Sub-component variables cannot be shared.
- A composed event is defined by events of the different sub-components.
- The same event can be composed more than once. It corresponds to different events' synchronisations.

The methodological conditions are:

- A composed machine is defined by at least one sub-component.
- Composed machines refining an abstraction do not introduce new events. For simplicity we restrict the introduction of new events during the composition since adding new events before or after the composition has a similar outcome to adding them during the composition.
- Variants are not required for composed machines. Only new events require variants and they are not allowed, as justified in the previous point.
- A composed event is defined by at least one event.
- When the composed machine refines an abstraction, the rules and refinement POs are applied similarly to standard machines.

These are the required conditions to build a valid composed machine. Next we present the required POs to verify composed machines.

2.3.3 Proof Obligations

POs play an important role in Event-B developments. For simplicity we define POs in terms of a composition of two machines $M_1(v1)$ and $M_2(v2)$ that refine machine $M_0(v0)$, but the rules generalise easily to the composition of n machines. Furthermore context elements in the formulas $(s, c, A(s, c))$ are not considered. The same proof obligations defined for standard machines (invariant preservation, well-definedness, refinement, etc) are defined for composed machines. We simplify the composed machines POs by assuming that the POs of the individual machines already hold. We just define the additional POs necessary to ensure that the composed machine satisfies all the standard POs. Therefore we consider that the POs of the machines to be composed (M_1 and M_2) hold. The same applies for the abstract machine M_0. Following the POs described in Sect. 1.5.3 for standard machines, the respective composition POs are described as follows.

2.3.3.1 Consistency

Consistency POs are required to be always verified. Consistency is expressed by the feasibility and invariant preservation POs for each composed event. In the composed machine, feasibility PO FIS_{CM} corresponds to the feasibility of all events from the individual machines that are composed. To show the feasibility proof obligation for a composed event, we compose event $evt1$ from machine $M1$ and event $evt2$ from machine $M2$: $evt1 \parallel evt2$. The feasibility proof obligation for the composed event $evt1 \parallel evt2$ is $FIS_{evt1\parallel evt2}$.

Theorem 2.1. Let FIS_{evt1} and FIS_{evt2} be the feasibility proof obligations for two different events $evt1$ and $evt2$ operating on disjoint variables v_1 and v_2 respectively. Then $FIS_{evt1\parallel evt2}$ holds if both FIS_{evt1} and FIS_{evt2} also hold.

From (1.4):

$$FIS_{evt1}: \quad FIS_{evt1H} \vdash FIS_{evt1G} \equiv I_1(v_1) \wedge G_1(p_1, v_1) \vdash \exists v_1' \cdot (S_1(p_1, v_1, v_1')) \tag{2.3}$$

$$FIS_{evt2}: \quad FIS_{evt2H} \vdash FIS_{evt2G} \equiv I_2(v_2) \wedge G_2(p_2, v_2) \vdash \exists v_2' \cdot (S_2(p_2, v_2, v_2')) \tag{2.4}$$

$$\begin{aligned} FIS_{evt1\parallel evt2}: \quad & FIS_{evt1\parallel evt2H} \vdash FIS_{evt1\parallel evt2G} \equiv \\ & I_{CM}(v_1, v_2) \wedge I_1(v_1) \wedge I_2(v_2) \\ & \wedge G_1(p_1, v_1) \wedge G_2(p_2, v_2) \\ & \vdash \exists v_1', v_2' \cdot (S_1(p_1, v_1, v_1') \wedge S_2(p_2, v_2, v_2')). \end{aligned} \tag{2.5}$$

Assume: FIS_{evt1} and FIS_{evt2}.
Prove: $FIS_{evt1\parallel evt2}$.

Proof. Assume the hypotheses of $FIS_{evt1\parallel evt2}$ ($FIS_{evt1\parallel evt2H}$):

$$I_{CM}(v_1, v_2)$$

$$I_1(v_1) \wedge G_1(p_1, v_1) \tag{2.6}$$

$$I_2(v_2) \wedge G_2(p_2, v_2). \tag{2.7}$$

Prove:

$$\exists v_1', v_2' \cdot (S_1(p_1, v_1, v_1') \wedge S_2(p_2, v_2, v_2')).$$

The proof proceeds as follows:

$$\begin{aligned}
&\exists v_1', v_2' \cdot (S_1(p_1, v_1, v_1') \wedge S_2(p_2, v_2, v_2')) && \\
\equiv\ &\exists v_1' \cdot (S_1(p_1, v_1, v_1')) && \\
&\wedge \exists v_2' \cdot (S_2(p_2, v_2, v_2')) && \{\text{disjoint v1 and v2}\} \\
\Leftarrow\ &(FIS_{evt1G} \wedge FIS_{evt2G}). && \{(2.3)+(2.6),(2.4)+(2.7)\}
\end{aligned}$$

□

Another consistency PO is invariant preservation. In the composed machine, invariant preservation PO INV_{CM} corresponds to the invariant preservation in all events from the individual machines that are composed. The invariant preservation proof obligation for the composed event $evt1 \parallel evt2$ is $INV_{evt1\parallel evt2}$. Note that $i(v')$ denotes the result of the substitution of variable v by the corresponding before-after predicate v' in invariant i.

Theorem 2.2. Let INV_{evt1} and INV_{evt2} be the invariant preservation proof obligations for two different events $evt1$ and $evt2$. Then for each individual predicate i_1, i_2 and i_{CM} from the set of invariants I in a composed machine, $INV_{evt1\parallel evt2}$ holds if both INV_{evt1} and INV_{evt2} also hold plus the composition invariant $I_{CM}(v_1, v_2)$ holds.

From (1.3):

$$INV_{evt1}: \quad INV_{evt1H} \vdash INV_{evt1G} \equiv I_1(v_1) \wedge G_1(p_1, v_1) \wedge S_1(p_1, v_1, v_1') \vdash i_1(v_1') \tag{2.8}$$

$$INV_{evt2}: \quad INV_{evt2H} \vdash INV_{evt2G} \equiv I_2(v_2) \wedge G_2(p_2, v_2) \wedge S_2(p_2, v_2, v_2') \vdash i_2(v_2') \tag{2.9}$$

$$\begin{aligned}
INV_{evt1\parallel evt2}: \quad & INV_{evt1\parallel evt2H} \vdash INV_{evt1\parallel evt2G} \equiv \\
& I_{CM}(v_1, v_2) \wedge I_1(v_1) \wedge I_2(v_2) \\
& \wedge G_1(p_1, v_1) \wedge G_2(p_2, v_2) \\
& \wedge S_1(p_1, v_1, v_1') \wedge S_2(p_2, v_2, v_2') \\
& \vdash i_1(v_1') \wedge i_2(v_2') \wedge i_{CM}(v_1', v_2').
\end{aligned} \tag{2.10}$$

Assume: INV_{evt1} and INV_{evt2}.
Prove: $INV_{evt1\parallel evt2}$.

Proof. Assume the hypotheses of $INV_{evt1\parallel evt2}$:

$$I_{CM}(v_1, v_2)$$

$$I_1(v_1) \wedge G_1(p_1, v_1) \wedge S_1(p_1, v_1, v_1') \tag{2.11}$$

$$I_2(v_2) \wedge G_2(p_2, v_2) \wedge S_2(p_2, v_2, v_2') \tag{2.12}$$

Prove:

$$i_1(v_1') \wedge i_2(v_2') \wedge i_{CM}(v_1', v_2').$$

The proof proceeds as follows:

$$\begin{aligned}
& i_1(v_1') \wedge i_2(v_2') \wedge i_{CM}(v_1', v_2') \\
\Leftarrow\ & INV_{evt1G} \\
& \wedge INV_{evt2G} \\
& \wedge i_{CM}(v_1', v_2'). && \{(2.8)+(2.11),(2.9)+(2.12)\}
\end{aligned}$$

In other words, composition invariants $I_{CM}(v_1, v_2)$ need to be verified but the invariant POs of the individual machines hold without having to be re-verified. □

Well-definedness POs are also applicable to the composed machines. Nevertheless in practice, well-definedness POs are only generated for $I_{CM}(v_1, v_2)$. Other expressions (guards, actions, etc) are verified in the individual machines [10].

2.3.3.2 Refinement

The refinement POs are only required when the composed machine refines an abstract machine. Machine M_0 with variables v_0, invariant $I_0(v_0)$ and abstract event evt_0 is refined by composed machine CM defined by abstract machines M_1 with variables w_1, invariant $I_1(w_1)$, event evt_1, M_2 (w_2 ; $I_2(w_2)$; evt_2) and composition invariant $J_{CM}(v_0, w_1, w_2)$. The composed event $evt1 \parallel evt2$ refines the abstract event evt_0. The refinement PO for a composed machine REF_{CM} results from the verification of the composition invariant preservation $J_{CM}(v_0', w_1', w_2')$, the verification of guard strengthening for $G_0(p_0, v_0)$ and simulation $S_0(p_0, v_0, v_0')$ for each refined event.

Theorem 2.3. Let composed event $evt1 \parallel evt2$ refine abstract event $evt0$. Then the refinement REF PO for $evt1 \parallel evt2$ consists in proving the guard strengthening of abstract guards, proving the simulation of the abstract variables (v_0') and preserving the gluing invariant ($J_{CM}(v_0', w_1', w_2')$) in the composed machine.

From (1.5):

$$INV_{evt1}:\quad I_1(w_1) \wedge H_1(q_1, w_1) \wedge T_1(q_1, w_1, w_1') \vdash i_1(w_1') \tag{2.13}$$

$$INV_{evt2}:\quad I_2(w_2) \wedge H_2(q_2, w_2) \wedge T_2(q_2, w_2, w_2') \vdash i_2(w_2') \tag{2.14}$$

$$\begin{aligned}
REF_{evt0 \sqsubseteq (evt1 \parallel evt2)}:\quad & I_0(v_0) \wedge I_1(w_1) \wedge I_2(w_2) \wedge J_{CM}(v_0, w_1, w_2) \\
& \wedge H_1(q_1, w_1) \wedge H_2(q_2, w_2) \\
& \wedge T_1(q_1, w_1, w_1') \wedge T_2(q_2, w_2, w_2') \\
& \vdash \exists v_0' \cdot G_0(p_0, v_0) \wedge S_0(p_0, v_0, v_0') \wedge i_1(w_1') \wedge i_2(w_2') \wedge J_{CM}(v_0', w_1', w_2').
\end{aligned} \tag{2.15}$$

Assume: INV_{evt1} (2.13) and INV_{evt2} (2.14).

Prove: $REF_{evt0 \sqsubseteq (evt1 \parallel evt2)}$.

Proof. Assume the hypotheses of $REF_{evt0 \sqsubseteq (evt1 \| evt2)}$:

$$\begin{aligned} & I_0(v_0) \wedge J_{CM}(v_0, w_1, w_2) \\ & I_1(w_1) \wedge H_1(q_1, w_1) \wedge T_1(q_1, w_1, w_1') \\ & I_2(w_2) \wedge H_2(q_2, w_2) \wedge T_2(q_2, w_2, w_2') \end{aligned}$$

Prove:

$$\vdash \exists v_0' \cdot G_0(p_0, v_0) \wedge S_0(p_0, v_0, v_0') \wedge I_1(w_1') \wedge I_2(w_2') \wedge J_{CM}(v_0', w_1', w_2').$$

The proof proceeds as follows:

$$\begin{aligned} & \exists v_0' \cdot G_0(p_0, v_0) \wedge S_0(p_0, v_0, v_0') \\ & \quad \wedge I_1(w_1') \wedge I_2(w_2') \wedge J_{CM}(v_0', w_1', w_2') \\ \equiv\ & G_0(p_0, v_0) \wedge I_1(w_1') \wedge I_2(w_2') \\ & \quad \wedge \exists v_0' \cdot (S_0(p_0, v_0, v_0') \wedge J_{CM}(v_0', w_1', w_2')) && \{\wedge \text{ goal; } v_0, w_1', w_2' \text{ are free variables}\} \\ \Leftarrow\ & G_0(p_0, v_0) \\ & \quad \wedge \exists v_0' \cdot (S_0(p_0, v_0, v_0') \wedge J_{CM}(v_0', w_1', w_2')) && \{\text{from (2.13) and (2.14)}\} \end{aligned}$$

□

As mentioned in Sect. 1.5.3, the refinement POs can be slit into separated POs using witnesses: guard strengthening, simulation and gluing invariant preservation. We separate the above refinement proof into these three kind of proof obligations.

Guard Strengthening For each composed event $evt1 \parallel evt2$, the guard strengthening PO GRD_{CM} refers to the relation between the conjunction of the guards of the composed event $H_1(q_1, w_1) \wedge H_2(q_2, w_2)$ and the guard of the abstract event $evt0$: $G_0(p_0, v_0)$.

For each abstract guard g_0 from the set of guards G_0 in an abstract machine, the GRD PO for each event requires verification that the concrete guards $H_1(q_1, w_1) \wedge H_2(q_2, w_2)$ are stronger than the abstract ones $G_0(p_0,v_0)$.

From (1.8), the proof rule to be verified is:

$$\begin{aligned} GRD_{evt0 \sqsubseteq (evt1 \| evt2)}: \quad & I_0(v_0) \wedge I_1(w_1) \wedge I_2(w_2) \wedge J_{CM}(v_0, w_1, w_2) \\ & \wedge H_1(q_1, w_1) \wedge H_2(q_2, w_2) \\ & \wedge W_1(p_0, w_1, w_2, q_1, q_2) \\ & \vdash g_0(p_0,v_0). \end{aligned}$$

Gluing Invariant Preservation For composed events, the gluing invariant preservation PO INV_{CM} requires that all the gluing invariants are preserved for each composed event (similar to the invariant preservation described in Sect. 2.3.3.1).

Theorem 2.4. Let the invariant in the composed machine be $I_1(w_1) \wedge I_2(w_2) \wedge J_{CM}(v_0, w_1, w_2)$. Then for each composed event $evt1 \parallel evt2$, only each predicate from the set of gluing invariants $J_{CM}(v_0, w_1, w_2)$ needs to be verified if INV_{evt1} and INV_{evt2} hold.

From (1.7):

$$\begin{aligned} INV_{evt1 \parallel evt2}: \quad & I_0(v_0) \wedge I_1(w_1) \wedge I_2(w_2) \wedge J_{CM}(v_0, w_1, w_2) \\ & \wedge H_1(q_1, w_1) \wedge H_2(q_2, w_2) \\ & \wedge W_2(v_0', w_1, w_2, q_1, q_2, w_1', w_2') \\ & \wedge T_1(q_1, w_1, w_1') \wedge T_2(q_2, w_2, w_2') \\ & \vdash i_1(w_1') \wedge i_2(w_2') \wedge j_{CM}(v_0', w_1', w_2'). \end{aligned} \tag{2.16}$$

Assume: INV_{evt1} (2.13) and INV_{evt2} (2.14).
Prove: $INV_{evt1 \parallel evt2}$.

The proof proceeds as follows:

$$\begin{aligned} & i_1(w_1') \wedge i_2(w_2') \wedge j_{CM}(v_0', w_1', w_2') \\ \equiv\ & j_{CM}(v_0', w_1', w_2')) && \{\text{from (2.13) and (2.14)}\} \end{aligned}$$

Simulation To verify the simulation PO SIM_{CM}, each action executed in a composed event $evt1 \parallel evt2$ must not contradict the corresponding actions in the abstract event $evt0$.

For a concrete composed event $evt1 \parallel evt2$ refining event $evt0$, the simulation PO requires that each concrete action $T_1(q_1, w_1, w_1') \wedge T_2(q_2, w_2, w_2')$ simulates the abstract ones $S_0(p_0, v_0, v_0')$.

From (1.9), the proof rule that needs to be verified is:

$$\begin{aligned} SIM_{evt0 \sqsubseteq (evt1 \parallel evt2)}: \quad & I_0(v_0) \wedge I_1(w_1) \wedge I_2(w_2) \wedge J_{CM}(v_0, w_1, w_2) \\ & \wedge H_1(q_1, w_1) \wedge H_2(q_2, w_2) \\ & \wedge W_1(p_0, w_1, w_2, q_1, q_2, w_1', w_2') \\ & \wedge W_2(v_0', w_1, w_2, q_1, q_2, w_1', w_2') \\ & \wedge T_1(q_1, w_1, w_1') \wedge T_2(q_2, w_2, w_2') \\ & \vdash S_0(p_0, v_0, v_0'). \end{aligned}$$

These are the required POs to verify composed machines. Next we show that composed machines are monotonic which allows further refinements of sub-components while preserving refinement of the composition.

2.3.4 Monotonicity of Shared Event Composition for Composed Machines

An important property of the shared event composition in Event-B is *monotonicity*. Here we prove by means of refinement POs that the composition is monotonic confirming the result described by Butler [53] using actions systems and CSP. Figure 2.2 shows abstract specification $M1$ composed with other specification $N1$, creating a composed model $M1 \parallel N1$. $M1$ is refined by $M2$ and $N1$ by $N2$ respectively:

- $M1$ is characterised by variables v_M, invariants $I_M(v_M)$ and event evt_{M1}.
- $M2$ is characterised by variables w_M, gluing invariants $J_M(v_M, w_M)$ and event evt_{M2}.
- $N1$ is characterised by variables v_N, invariants $I_N(v_N)$ and event evt_{N1}.
- $N2$ is characterised by variables w_N, gluing invariants $J_N(v_N, w_N)$ and event evt_{N2}.

Monotonicity allows us to say that $CM1$ is refined by $CM2$. In other words, once we compose specifications $M1$ and $N1$, discharge the required composed POs, $M1$ and $N1$ can be refined individually while the composition properties are preserved without the need to recompose refinements $M2$ and $N2$. We want to formally prove the monotonicity property through refinement

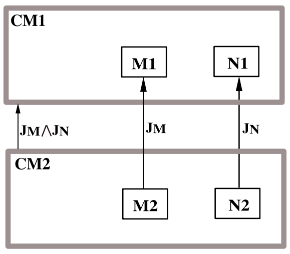


FIGURE 2.2: Refinement of composed machine $CM1 \mathrel{\widehat{=}} M1 \parallel N1$ by $CM2 \mathrel{\widehat{=}} M2 \parallel N2$

POs between composed machines (in Fig. 2.2 between $CM1$ and $CM2$). Therefore if the refinement POs hold between $CM1$ and $CM2$, we can say that $CM2$ refines $CM1$: $CM1 \sqsubseteq CM2$. An event evt_{M1} in machine $M1$ is represented as:

$$evt_{M1} \mathrel{\widehat{=}} \textbf{ANY } p_M \textbf{ WHERE } G_M(p_M, v_M) \textbf{THEN } S_M(p_M, v_M, v'_M) \textbf{ END}.$$

An event evt_{M2} in machine $M2$ refining abstract event evt_{M1} is represented as:

$$evt_{M2} \mathrel{\widehat{=}} \textbf{ANY } q_M \textbf{ WHERE } H_M(q_M, w_M) \textbf{THEN } T_M(q_M, w_M, w'_M) \textbf{ END}.$$

The gluing invariant of the refinement between $M1$ and $M2$ is expressed as $J_M(v_M, w_M)$ relating the states of $M1$ and $M2$: $M1 \sqsubseteq_{J_M} M2$. From (1.5) we can derive the refinement

PO between $M2$ and $M1$ for the concrete event evt_{M2} refining abstract event evt_{M1}.

$$\begin{aligned}
REF_{evt_{M1}\sqsubseteq evt_{M2}} : \quad & REF_{evt_{M1}\sqsubseteq evt_{M2}H} \vdash REF_{evt_{M1}\sqsubseteq evt_{M2}G} \\
& \equiv I_M(v_M) \wedge J_M(v_M, w_M) \\
& \quad \wedge G_M(p_M, v_M) \wedge H_M(q_M, w_M) \\
& \quad \wedge S_M(p_M, v_M, v'_M) \wedge T_M(q_M, w_M, w'_M) \\
& \quad \vdash \exists v'_M \cdot G_M(p_M, v_M) \wedge S_M(p_M, v_M, v'_M) \wedge J_M(v'_M, w'_M).
\end{aligned} \tag{2.17}$$

Similarly for machines $N1$ and $N2$, the gluing invariant is expressed as $J_N(v_N, w_N)$ relating the states of $N1$ and $N2$: $N1 \sqsubseteq_{J_N} N2$. Furthermore, the refinement PO for concrete event evt_{N2} refining abstract event evt_{N1} is expressed as:

$$\begin{aligned}
REF_{evt_{N1}\sqsubseteq evt_{N2}} : \quad & REF_{evt_{N1}\sqsubseteq evt_{N2}H} \vdash REF_{evt_{N1}\sqsubseteq evt_{N2}G} \\
& \equiv I_N(v_N) \wedge J_N(v_N, w_N) \\
& \quad \wedge G_N(p_N, v_N) \wedge H_N(q_N, w_N) \\
& \quad \wedge S_N(p_N, v_N, v'_N) \wedge T_N(q_N, w_N, w'_N) \\
& \quad \vdash \exists v'_N \cdot G_N(p_N, v_N) \wedge S_N(p_N, v_N, v'_N) \wedge J_N(v'_N, w'_N).
\end{aligned} \tag{2.18}$$

We refine an abstract event in $CM1$ by a concrete one in $CM2$ and verify that the refinement POs for each individual machine hold for the composition. Event evt_{M1} from machine $M1$ and event evt_{N1} from machine N1 are composed, resulting in the abstract composed event $evt_{M1} \parallel evt_{N1}$ in $CM1$ from Fig. 2.2. Such abstract composed event is represented as:

$$\begin{aligned}
evt_{M1} \parallel evt_{N1} \widehat{=} & \textbf{ANY } p_M, p_N \textbf{ WHERE } G_M(p_M, v_M) \wedge G_N(p_N, v_N) \\
& \textbf{THEN } S_M(p_M, v_M, v'_M) \parallel S_N(p_N, v_N, v'_N) \textbf{ END}.
\end{aligned}$$

A concrete composed event between $M2$ and $N2$ in $CM2$ ($evt_{M2} \parallel evt_{N2}$), refining the abstract event $evt_{M1} \parallel evt_{N1}$, is represented as:

$$\begin{aligned}
evt_{M2} \parallel evt_{N2} \widehat{=} & \textbf{ANY } q_M, q_N \textbf{ WHERE } H_M(q_M, w_M) \wedge H_N(q_N, w_N) \\
& \textbf{THEN } T_M(q_M, w_M, w'_M) \parallel T_N(q_N, w_N, w'_N) \textbf{ END}.
\end{aligned}$$

The gluing invariant relating the states of $CM1$ and $CM2$ is expressed as the conjunction of the gluing invariants between $M1/M2$ and $N1/N2$:

$$J_{CM}(v_M, v_N, w_M, w_N) = J_M(v_M, w_M) \wedge J_N(v_N, w_N) \tag{2.19}$$

Theorem 2.5. Let composed machine $CM1$ be defined by machines $M1$ and $N1$ and composed event $evt_{M1} \parallel evt_{N1}$. Then composed machine $CM2$ is a valid refinement of $CM1$ if the refinement proof obligations between machines $M2$ and $N2$ and machines $M1$ and $N1$ hold respectively for each concrete composed event $evt_{M2} \parallel evt_{N2}$ that refines abstract composed event $evt_{M1} \parallel evt_{N1}$.

From (2.15), the refinement PO between concrete composed event $evt_{M2} \parallel evt_{N2}$ and abstract composed event $evt_{M1} \parallel evt_{N1}$ is:

$$
\begin{aligned}
REF_{(evt_{M1}\parallel evt_{N1})\sqsubseteq(evt_{M2}\parallel evt_{N2})}: \quad & I_M(v_M) \land I_N(v_N) \land J_{CM}(v_M, v_N, w_M, w_N) \\
& \land H_M(q_M, w_M) \land H_N(q_N, w_N) \\
& \land T_M(q_M, w_M, w'_M) \land T_N(q_N, w_N, w'_N) \\
& \vdash \exists v'_M, v'_N \cdot G_M(p_M, v_M) \land G_N(p_N, v_N) \\
& \land S_M(p_M, v_M, v'_M) \land S_N(p_N, v_N, v'_N) \land J_{CM}(v'_M, v'_N, w'_M, w'_N).
\end{aligned}
$$

Assume: $REF_{evt_{M1}\sqsubseteq evt_{M2}}$ and $REF_{evt_{N1}\sqsubseteq evt_{N2}}$.
Prove: $REF_{(evt_{M1}\parallel evt_{N1})\sqsubseteq(evt_{M2}\parallel evt_{N2})}$.

Proof. Assume the hypotheses of $REF_{(evt_{M1}\parallel evt_{N1})\sqsubseteq(evt_{M2}\parallel evt_{N2})}$:

$$J_{CM}(v_M, v_N, w_M, w_N) \equiv J_M(v_M, w_M) \land J_N(v_N, w_N) \qquad \{\text{expanding } J_{CM} \text{ from } (2.19)\}$$

$$I_M(v_M) \land H_M(q_M, w_M) \land T_M(q_M, w_M, w'_M) \tag{2.20}$$

$$I_N(v_N) \land H_N(q_N, w_N) \land T_N(q_N, w_N, w'_N) \tag{2.21}$$

Prove:

$$
\begin{aligned}
& \exists v'_M, v'_N \cdot G_M(p_M, v_M) \land G_N(p_N, v_N) \land S_M(p_M, v_M, v'_M) \land S_N(p_N, v_N, v'_N) \\
& \land J_{CM}(v'_M, v'_N, w'_M, w'_N).
\end{aligned}
$$

The proof proceeds as follows:

$$
\begin{aligned}
& \exists v'_M, v'_N \cdot G_M(p_M, v_M) \land G_N(p_N, v_N) \\
& \quad \land S_M(p_M, v_M, v'_M) \land S_N(p_N, v_N, v'_N) \\
& \quad \land J_{CM}(v'_M, v'_N, w'_M, w'_N) \\
\equiv\ & \exists v'_M, v'_N \cdot G_M(p_M, v_M) \land G_N(p_N, v_N) \\
& \quad \land S_M(p_M, v_M, v'_M) \land S_N(p_N, v_N, v'_N) \\
& \quad \land J_M(v'_M, w'_M) \land J_N(v'_N, w'_N) && \{\text{expanding } J_{CM} \text{ from } (2.19)\} \\
\equiv\ & \exists v'_M \cdot G_M(p_M, v_M) \land S_M(p_M, v_M, v'_M) \land J_M(v'_M, w'_M) \\
& \quad \land \exists v'_N \cdot G_N(p_N, v_N) \land S_N(p_N, v_N, v'_N) \land J_N(v'_N, w'_N) && \{\text{disjoint } v'_M, v'_N\} \\
\Leftarrow\ & REF_{evt_{M1}\sqsubseteq evt_{M2}G} \\
& \quad \land REF_{evt_{N1}\sqsubseteq evt_{N2}G} && \{(2.17)+(2.20), (2.18)+(2.21)\}
\end{aligned}
$$

□

The refinement POs for composed machines is expressed as the conjunction of the refinement POs for the individual machines. Therefore the monotonicity property holds if the refinement POs of individual machines hold.

2.3.4.1 Monotonicity of Non-Composed Events for Composed Machines

We also need to prove the monotonicity for non-composed events that appear at both levels of abstraction. We shall prove it using machines $M1$ and $CM2$ as seen in Fig. 2.3 (similar for $N1$ and $CM2$).

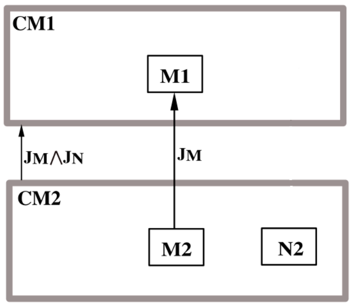


FIGURE 2.3: Refinement of composed machine $CM1 \mathrel{\widehat{=}} M1$ by $CM2 \mathrel{\widehat{=}} M2 \parallel N2$

Theorem 2.6. Let an event evt_{M1} in machine $M1$ be refined by a composed event $evt_{M2} \parallel evt_{N2}$ in composed machine $CM2$. Assuming that machine $M1$ is refined by machine $M2$ and $INV_{evt_{N2}}$ holds, then the monotonicity is preserved and event $M1$ is refined by the composed event $M2 \parallel N2$.

Assume: $REF_{evt_{M1} \sqsubseteq evt_{M2}}$ and $INV_{evt_{N2}}$.
Prove: $REF_{evt_{M1} \sqsubseteq (evt_{M2} \parallel evt_{N2})}$.

In this case, the gluing invariant described in (2.19) does not use neither the variables (v_N) neither the invariants (I_N). Therefore it can be simplified and rewritten as:

$$J_{CM}(v_M, w_M, w_N) = J_M(v_M, w_M) \wedge J_N(w_N) \tag{2.22}$$

From (2.15), the refinement PO between concrete composed event $evt_{M2} \parallel evt_{N2}$ and abstract event evt_{M1}:

$$\begin{aligned} REF_{evt_{M1} \sqsubseteq (evt_{M2} \parallel evt_{N2})}: \quad & I_M(v_M) \wedge J_{CM}(v_M, w_M, w_N) \\ & \wedge H_M(q_M, w_M) \wedge H_N(q_N, w_N) \\ & \wedge T_M(q_M, w_M, w'_M) \wedge T_N(q_N, w_N, w'_N) \\ & \vdash \exists v'_M \cdot G_M(p_M, v_M) \wedge S_M(p_M, v_M, v'_M) \wedge J_{CM}(v'_M, w'_M, w'_N). \end{aligned}$$

Proof. Assume the hypotheses of $REF_{evt_{M1} \sqsubseteq (evt_{M2} \parallel evt_{N2})}$:

$$\begin{aligned} & J_{CM}(v_M, w_M, w_N) \equiv J_M(v_M, w_M) \wedge J_N(w_N) \qquad \{\text{expanding } J_{CM} \text{ from } (2.22)\}. \\ & I_M(v_M) \wedge H_M(q_M, w_M) \wedge T_M(q_M, w_M, w'_M) \\ & H_N(q_N, w_N) \wedge T_N(q_N, w_N, w'_N) \end{aligned}$$

And assume $INV_{evt_{N2}}$:

$$\begin{aligned} INV_{evt_{N2}} : \quad & INV_{evt_{N2}H} \vdash INV_{evt_{N2}G} \\ \equiv \quad & J_N(w_N) \wedge H_N(q_N, w_N) \wedge T_N(q_N, w_N, w'_N) \\ & \vdash j_N(w'_N). \end{aligned} \tag{2.23}$$

The proof proceeds as follows:

$$\begin{aligned} & \exists v'_M \cdot G_M(p_M, v_M) \wedge S_M(p_M, v_M, v'_M) \wedge J_{CM}(v'_M, w'_M, w'_N) \\ \equiv\ & \exists v'_M \cdot G_M(p_M, v_M) \wedge S_M(p_M, v_M, v'_M) \\ & \wedge J_M(v'_M, w'_M) \wedge J_N(w'_N) && \{\text{expanding } J_{CM} \text{ from } (2.22)\} \\ \Leftarrow\ & REF_{evt\_M1 \sqsubseteq evt\_M2G} \\ & \wedge J_N(w'_N) && \{\ (2.17)\} \\ \Leftarrow\ & REF_{evt\_M1 \sqsubseteq evt\_M2G} \\ & \wedge INV_{evt\_N2G} && \{\ (2.23)\} \end{aligned}$$

□

2.3.4.2 New Events

New events must refine event *skip* and their state space include only new variables w; abstract variables v do not change state. Nevertheless new composed events must respect the refinement POs.

Theorem 2.7. Let evt_{M2} be a new (composed) event in $CM2$ refining skip. If we assume that the invariant proof obligation for event evt_{M2} holds, then the monotonicity property is preserved (i.e. $REF_{skip \sqsubseteq evt_{M2}}$ holds).

From (2.15), the refinement PO for new event evt_{M2} is necessary to be verified to ensure that monotonicity is preserved. It can be expressed as:

$$\begin{aligned} REF_{skip \sqsubseteq evt_{M2}} : \quad & REF_{skip \sqsubseteq evt_{M2}H} \vdash REF_{skip \sqsubseteq evt_{M2}G} \\ \equiv \quad & I_M(v_M) \wedge J_{CM}(v_M, w_M, w_N) \\ & \wedge H_M(q_M, w_M) \wedge T_M(q_M, w_M, w'_M) \\ & \vdash \exists v'_M \cdot G_M(p_M, v_M) \wedge S_M(p_M, v_M, v'_M) \wedge J_{CM}(v'_M, w'_M, w'_N). \end{aligned}$$

And assume $INV_{evt_{M2}}$:

$$\begin{aligned} INV_{evt_{M2}} : \quad & INV_{evt_{M2}H} \vdash INV_{evt_{M2}G} \\ \equiv \quad & J_M(v_M, w_M) \wedge H_M(q_M, w_M) \wedge T_M(q_M, w_M, w'_M) \\ & \vdash j_M(v_M, w'_M). \end{aligned} \tag{2.24}$$

Moreover, since evt_{M2} is a new event refining *skip* (event with guard always TRUE and without actions), then:

$$J_{CM}(v_M, w_M, w_N) = J_M(v_M, w_M) \tag{2.25}$$

$$G_M(p_M, v_M) = TRUE \tag{2.26}$$

$$S_M(p_M, v_M, v'_M) = \varnothing. \tag{2.27}$$

Assume: $INV_{evt_{M2}}$
Prove: $REF_{skip \sqsubseteq evt_{M2}}$.

The proof proceeds as follows:

Proof.

$$\begin{aligned}
& \exists v'_M \cdot G_M(p_M, v_M) \wedge S_M(p_M, v_M, v'_M) \wedge J_{CM}(v'_M, w'_M, w'_N) && \\
\equiv\ & \exists v'_M \cdot J_{CM}(v'_M, w'_M, w'_N) && \{(2.26) \text{ and } (2.27)\} \\
\equiv\ & \exists v'_M \cdot J_M(v'_M, w'_M) && \{\ (2.25)\} \\
\Leftarrow\ & INV_{evt\_M2G} && \{\ (2.24)\}
\end{aligned}$$

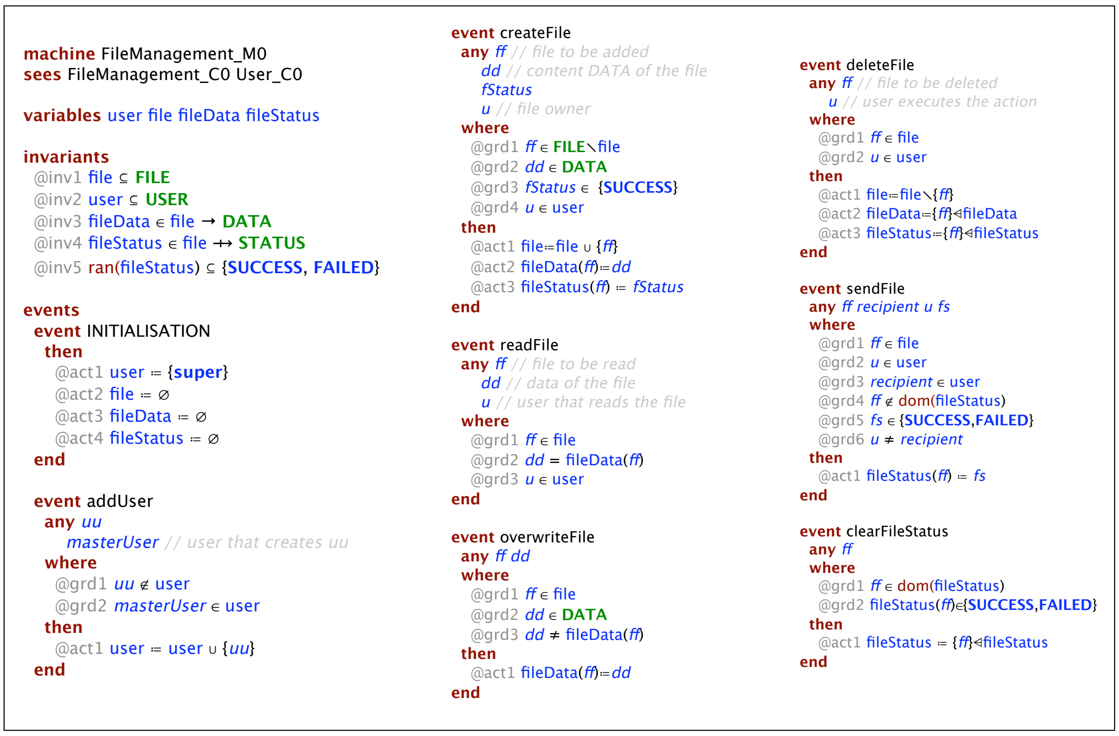
□

Next section presents the application of the shared event composition to a more complex case study whose architecture is a distributed system: file transfer system.

2.4 File Access Management case study

A distributed system is presented where two component specifications are composed in the style defined in Fig. 1.11. A specification of a file management system is developed: files containing *DATA* can be created, read, overwritten, deleted and sent to other users. Another separated specification deals with the access management of files in which each file has an owner. The owners are users with clearance level from 1 to 10 where 10 is the highest level. A *super* user exists with clearance level 10. Moreover, files have a classification level varying from 1 to 10. Permission is needed in order to read, modify or delete a file. When the permission is granted, the requested action can take place.

The first specification is defined as machine $FileManagement\_M0$ and variables *user*, *file*, *fileData* and *fileStatus* (defines the status of a file operation and can have the states *SUCCESS* or *FAILED*) as depicted in Fig. 2.4. After a file is created or sent, variable *fileStatus* is updated accordingly to the result of the operation. In order to allow a new operation in the same file, the state of that file must be reset in event *clearFileStatus*. The file operations are defined by events $createFile$, $readFile$, $overwriteFile$, $deleteFile$, $sendFile$ and $clearFileStatus$ as seen in Fig. 2.4. The access management specifica-

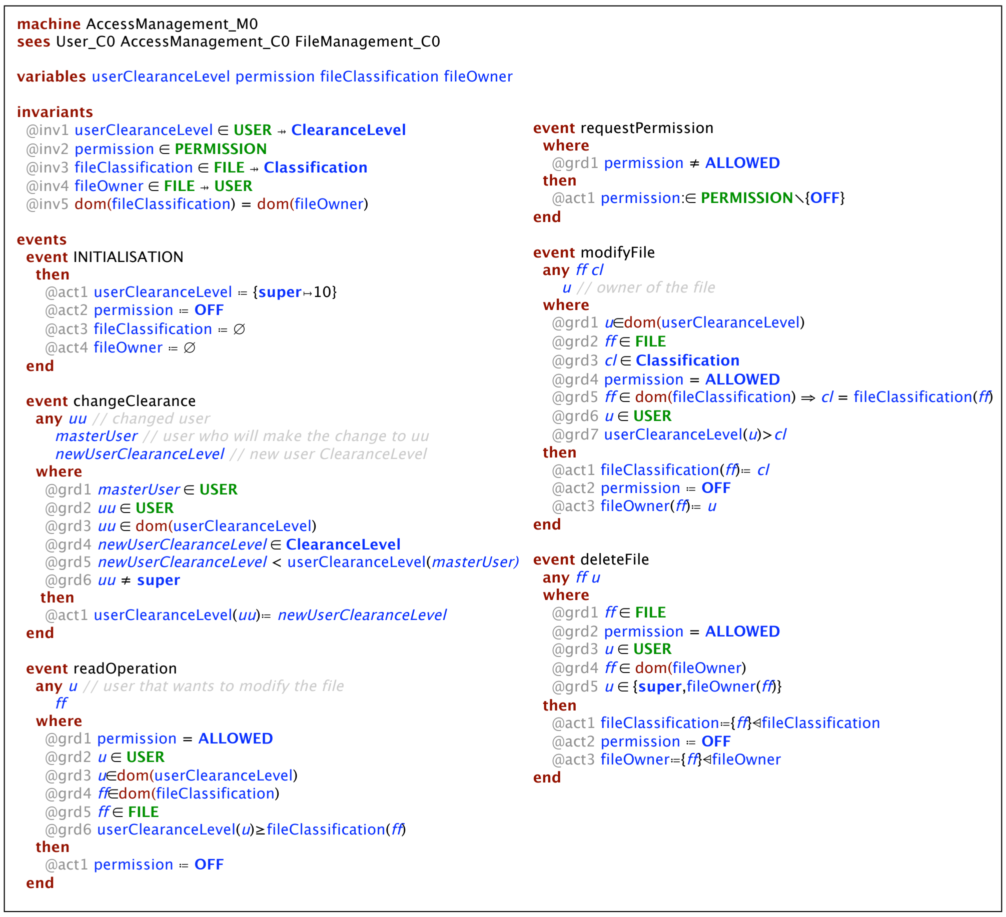


```
machine FileManagement_M0
sees FileManagement_C0 User_C0

variables user file fileData fileStatus

invariants
 @inv1 file ⊆ FILE
 @inv2 user ⊆ USER
 @inv3 fileData ∈ file → DATA
 @inv4 fileStatus ∈ file ⇸ STATUS
 @inv5 ran(fileStatus) ⊆ {SUCCESS, FAILED}

events
 event INITIALISATION
  then
   @act1 user ≔ {super}
   @act2 file ≔ ∅
   @act3 fileData ≔ ∅
   @act4 fileStatus ≔ ∅
 end

 event addUser
  any uu
     masterUser // user that creates uu
  where
   @grd1 uu ∉ user
   @grd2 masterUser ∈ user
  then
   @act1 user ≔ user ∪ {uu}
 end

event createFile
 any ff // file to be added
    dd // content DATA of the file
    fStatus
    u // file owner
 where
  @grd1 ff ∈ FILE∖file
  @grd2 dd ∈ DATA
  @grd3 fStatus ∈ {SUCCESS}
  @grd4 u ∈ user
 then
  @act1 file≔file ∪ {ff}
  @act2 fileData(ff)≔dd
  @act3 fileStatus(ff) ≔ fStatus
end

event readFile
 any ff // file to be read
    dd // data of the file
    u // user that reads the file
 where
  @grd1 ff ∈ file
  @grd2 dd = fileData(ff)
  @grd3 u ∈ user
end

event overwriteFile
 any ff dd
 where
  @grd1 ff ∈ file
  @grd2 dd ∈ DATA
  @grd3 dd ≠ fileData(ff)
 then
  @act1 fileData(ff)≔dd
end

event deleteFile
 any ff // file to be deleted
    u // user executes the action
 where
  @grd1 ff ∈ file
  @grd2 u ∈ user
 then
  @act1 file≔file∖{ff}
  @act2 fileData≔{ff}⩤fileData
  @act3 fileStatus≔{ff}⩤fileStatus
end

event sendFile
 any ff recipient u fs
 where
  @grd1 ff ∈ file
  @grd2 u ∈ user
  @grd3 recipient ∈ user
  @grd4 ff ∉ dom(fileStatus)
  @grd5 fs ∈ {SUCCESS,FAILED}
  @grd6 u ≠ recipient
 then
  @act1 fileStatus(ff) ≔ fs
end

event clearFileStatus
 any ff
 where
  @grd1 ff ∈ dom(fileStatus)
  @grd2 fileStatus(ff)∈{SUCCESS,FAILED}
 then
  @act1 fileStatus ≔ {ff}⩤fileStatus
end
```

FIGURE 2.4: Machine *FileManagement_M0*

tion is defined by machine *AccessManagement_M0* and variables *userClearanceLevel*, *permission*, *fileClassification* and *fileOwner* as seen in Fig. 2.5. A user can change the clearance of another user as long as the former has a clearance level superior to the latter as described in event *changeClearance* (guard *grd5*). For all the other operations, permission is needed given by the non-deterministic action in event *requestPermission*. With permission granted, a file can be read, modified or deleted. Moreover, only users with a clearance level superior to the file classification can modify a file (guard *grd7* in event *modifyFile*). To delete a file, described in event *deleteFile*, the user must be the owner of the file or the *super* user as described by guard *grd5*.

These two specifications were developed in two different machines as they deal with different contexts: machine *FileManagement_M0* handles the physical creation and modification of files and respective data. Machine *AccessManagement_M0* handles the conditions in which a reading and a modification can occur. By composing these two specifications and respective events, we explore the development of a composed specification that is constrained by the other. The composed machine *FileAccessManagement* can be seen in Fig. 2.6. Modifying, overwriting, sending or deleting a file must be authorised (request permission) and only a defined set of users are allowed to do it (in opposition to what happens in machine *FileManagement_M0*); events corresponding to the creation of users and change of clearance are synchronised and occur in parallel. The conjunction of the guards of each event (Def. (1.2)) restrains the conditions to enable the composed event. Nevertheless some events are not composed such as *requestPermission* or *clearFileStatus*. Moreover, additional invariants are added allowing the interaction of



```
machine AccessManagement_M0
sees User_C0 AccessManagement_C0 FileManagement_C0

variables userClearanceLevel permission fileClassification fileOwner

invariants
 @inv1 userClearanceLevel ∈ USER ⇸ ClearanceLevel
 @inv2 permission ∈ PERMISSION
 @inv3 fileClassification ∈ FILE ⇸ Classification
 @inv4 fileOwner ∈ FILE ⇸ USER
 @inv5 dom(fileClassification) = dom(fileOwner)

events
 event INITIALISATION
  then
   @act1 userClearanceLevel ≔ {super↦10}
   @act2 permission ≔ OFF
   @act3 fileClassification ≔ ∅
   @act4 fileOwner ≔ ∅
 end

 event changeClearance
  any uu // changed user
    masterUser // user who will make the change to uu
    newUserClearanceLevel // new user ClearanceLevel
  where
   @grd1 masterUser ∈ USER
   @grd2 uu ∈ USER
   @grd3 uu ∈ dom(userClearanceLevel)
   @grd4 newUserClearanceLevel ∈ ClearanceLevel
   @grd5 newUserClearanceLevel < userClearanceLevel(masterUser)
   @grd6 uu ≠ super
  then
   @act1 userClearanceLevel(uu)≔ newUserClearanceLevel
 end

 event readOperation
  any u // user that wants to modify the file
    ff
  where
   @grd1 permission = ALLOWED
   @grd2 u ∈ USER
   @grd3 u∈dom(userClearanceLevel)
   @grd4 ff∈dom(fileClassification)
   @grd5 ff ∈ FILE
   @grd6 userClearanceLevel(u)≥fileClassification(ff)
  then
   @act1 permission ≔ OFF
 end

event requestPermission
 where
  @grd1 permission ≠ ALLOWED
 then
  @act1 permission:∈ PERMISSION∖{OFF}
end

event modifyFile
 any ff cl
   u // owner of the file
 where
  @grd1 u∈dom(userClearanceLevel)
  @grd2 ff ∈ FILE
  @grd3 cl ∈ Classification
  @grd4 permission = ALLOWED
  @grd5 ff ∈ dom(fileClassification) ⇒ cl = fileClassification(ff)
  @grd6 u ∈ USER
  @grd7 userClearanceLevel(u)> cl
 then
  @act1 fileClassification(ff)≔ cl
  @act2 permission ≔ OFF
  @act3 fileOwner(ff)≔ u
end

event deleteFile
 any ff u
 where
  @grd1 ff ∈ FILE
  @grd2 permission = ALLOWED
  @grd3 u ∈ USER
  @grd4 ff ∈ dom(fileOwner)
  @grd5 u ∈ {super,fileOwner(ff)}
 then
  @act1 fileClassification≔{ff}⩤fileClassification
  @act2 permission ≔ OFF
  @act3 fileOwner≔{ff}⩤fileOwner
end
```

FIGURE 2.5: Machine *AccessManagement_M0*

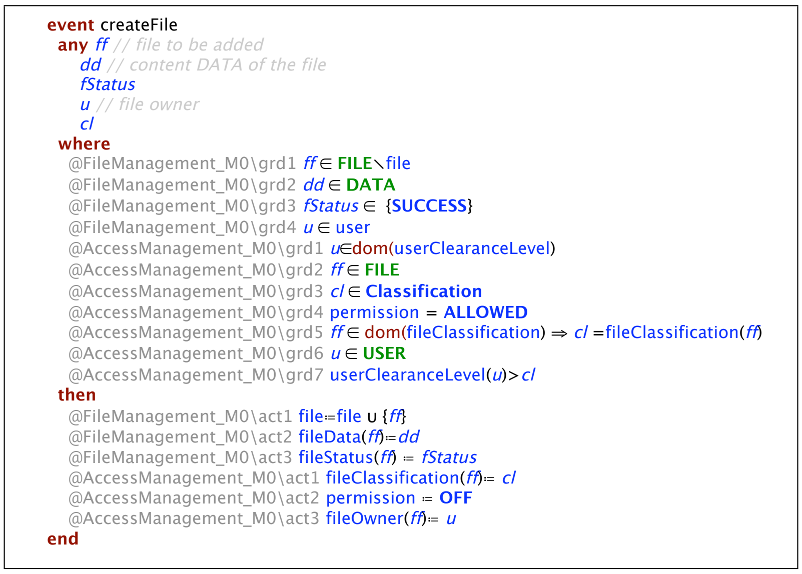
states but still without possibility to share variables. Among the added invariants, the most important is the one that requires the classification of a file to be lower than the clearance level of its owner (*@inv4*).

As aforementioned in a shared event composition, the composed events communicate through value passing. The value passing is allowed when composed events have parameters with the same name and compatible types (cf. (1.20)). For instance, the composed event *createFile* results from the composition of events *AccessManagement_M0.modifyFile* and *FileManagement_M0.createFile*. *AccessManagement_M0.modifyFile* has parameters *ff* of type *FILE*, *u* of type *USER* and *cl* of type *Classification*. *FileManagement_M0.createFile* has parameters *ff* of type *FILE*, *dd* of type *DATA*, *u* of type *USER* and *fStatus* of type *STATUS*. When these two events are synchronised, parameters *ff* and *u* are shared as seen in Fig. 2.7 (the labels of the guards and actions, starting with '@', define their source). Although not explicitly defined, parameter *ff* inputs an element of *FILE* (guards *FileManagement_M0\grd1* and *AccessManagement_M0\grd2*) that will be added to the variable *file* in action *FileManagement_M0\act1*. Similarly, parameter *u* behaves as an input parameter. The respective actions occur in parallel: when a file is created, its content is defined by parameter *dd* and the resulting state of the operation is updated by *fStatus*; also the file is classified according to the parameter *cl* and has an

COMPOSED MACHINE *FileAccessManagement*
INCLUDES
  AccessManagement_M0, *FileManagement_M0*
INVARIANTS
  @inv1: $dom(userClearanceLevel) = user$
  @inv2: $dom(fileClassification) = file$
  @inv3: $fileOwner \in file \rightarrow user$
  @inv4: $\forall f \cdot f \in file \Rightarrow userClearanceLevel(fileOwner(f)) > fileClassification(f)$
EVENTS
  addUser
    Combines Events *AccessManagement_M0.changeClearance* ‖ *FileManagement_M0.addUser*
  modifyUser
    Combines Events *AccessManagement_M0.changeClearance*
  createFile
    Combines Events *AccessManagement_M0.modifyFile* ‖ *FileManagement_M0.createFile*
  readFile
    Combines Events *AccessManagement_M0.readOperation* ‖ *FileManagement_M0.readFile*
  overwriteFile
    Combines Events *AccessManagement_M0.modifyFile* ‖ *FileManagement_M0.overwriteFile*
  deleteFile
    Combines Events *AccessManagement_M0.deleteFile* ‖ *FileManagement_M0.deleteFile*
  sendFile
    Combines Events *AccessManagement_M0.modifyFile* ‖ *FileManagement_M0.sendFile*
  requestPermission
    Combines Events *AccessManagement_M0.requestPermission*
  clearFileStatus
    Combines Events *FileManagement_M0.clearFileStatus*

FIGURE 2.6: Composed machine *FileAccessManagement*

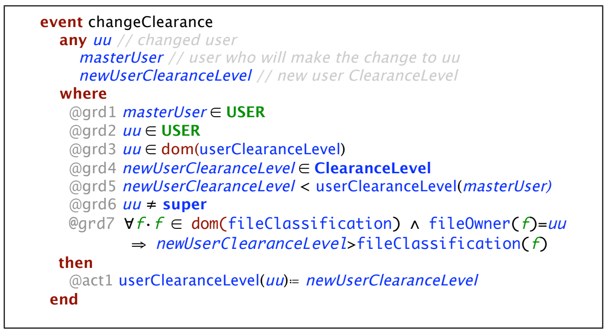
owner u. The others composed events behave similarly.



```
event createFile
 any ff // file to be added
   dd // content DATA of the file
   fStatus
   u // file owner
   cl
 where
  @FileManagement_M0\grd1 ff∈ FILE∖file
  @FileManagement_M0\grd2 dd∈ DATA
  @FileManagement_M0\grd3 fStatus∈ {SUCCESS}
  @FileManagement_M0\grd4 u∈ user
  @AccessManagement_M0\grd1 u∈dom(userClearanceLevel)
  @AccessManagement_M0\grd2 ff∈ FILE
  @AccessManagement_M0\grd3 cl∈ Classification
  @AccessManagement_M0\grd4 permission = ALLOWED
  @AccessManagement_M0\grd5 ff∈ dom(fileClassification) ⇒ cl =fileClassification(ff)
  @AccessManagement_M0\grd6 u∈ USER
  @AccessManagement_M0\grd7 userClearanceLevel(u)>cl
 then
  @FileManagement_M0\act1 file≔file ∪ {ff}
  @FileManagement_M0\act2 fileData(ff)≔dd
  @FileManagement_M0\act3 fileStatus(ff) ≔ fStatus
  @AccessManagement_M0\act1 fileClassification(ff)≔ cl
  @AccessManagement_M0\act2 permission ≔ OFF
  @AccessManagement_M0\act3 fileOwner(ff)≔ u
end
```

FIGURE 2.7: "Expanded" event *createFile* from composed machine *FileAccessManagement*

The composed machine needs to be verified to ensure that the properties of the model are preserved. The verification is accomplished by discharging the proof obligations as described in Sect. 2.3.3. Moreover, the additional invariants must also be preserved by all the events in the composed machine. After the generation of the proof obligations for the composed machine *FileAccessManagement* only one proof obligation is not automatically discharged: it is a composition gluing invariant preservation PO referring to *inv4* in event *modifyUser*. After analysing the event, it is easy to understand why the PO cannot be discharged: there is no information in the event that guarantees that the files owned by *uu* have a classification that is inferior to the new user's clearance. To discharge this PO, it is necessary to add a guard to the composed event *modifyFile* that guarantees that all the files owned by *uu* have a classification that is inferior than the new clearance. But the composition of machines is structural and therefore no guards can be added directly to the composed machine. Instead a new guard needs to be added to the original event *changeClearance* in the included machine *AccessManagement_M0* from where the composed event *modifyUser* comes from (cf. Fig. 2.6). Guard *grd8* is added in event *changeClearance* of machine *AccessManagement_M0* as seen in Fig. 2.8. After changing the event *changeClearance*, the proof obligations can be re-generated



```
event changeClearance
  any uu // changed user
    masterUser // user who will make the change to uu
    newUserClearanceLevel // new user ClearanceLevel
  where
   @grd1 masterUser ∈ USER
   @grd2 uu ∈ USER
   @grd3 uu ∈ dom(userClearanceLevel)
   @grd4 newUserClearanceLevel ∈ ClearanceLevel
   @grd5 newUserClearanceLevel < userClearanceLevel(masterUser)
   @grd6 uu ≠ super
   @grd7 ∀f· f ∈ dom(fileClassification) ∧ fileOwner(f)=uu
            ⇒ newUserClearanceLevel>fileClassification(f)
  then
   @act1 userClearanceLevel(uu)≔ newUserClearanceLevel
 end
```

FIGURE 2.8: Event *changeClearance* from machine *AccessManagement* with added guard *grd8*

for the composed machine and as expected, all the POs are automatically discharged. Moreover, no changes were made directly in the composed machine. In this manner, there is more flexibility in the interaction of specifications as the changes in the individual sub-components are directly reflected in the composed machines.

One of the properties of the shared event composition is monotonicity. Therefore sub-components can be further refined independently preserving the verified properties while composed. For instance, machine *AccessManagement_M0* can be refined by defining a more deterministic event *requestPermission* based on the kind of operation and the user that intends to execute the operation. For machine *FileManagement_M0*, the event *sendFile* can be further refined by introducing a queue where events would be stored before being processed (creating a new file own by the file recipient). The independent refinement of the sub-components results in a separation of behaviours and properties that can be verified without the interference of other sub-components.

2.5 Related Work

Composition allows the interaction of sub-components and usually occurs through variable sharing, event sharing or a combination of both. Back [23], Abadi and Lamport[1] studied the interaction of components through shared variable composition. Jones [187] also proposes a shared variable composition for VDM by restricting the behaviour of the environment and the operation itself in order to consider the composition valid using rely-guarantee conditions.

CSP [92] allows the specification of distributed systems from an event-based viewpoint. Processes and environment behaviours can be composed using the parallel composition operator $\parallel$. They interact by synchronisation of common events within the respective alphabets (*interaction*) and stop if any of the involved processes deadlocks. Another option is to have processes with different alphabets: *concurrency* exists when independent events occur in parallel. On the one hand, when the alphabets of processes P and Q do not have common events, $\alpha P \cap \alpha Q = \{\}$, then the alphabet is given as $\alpha(P \parallel Q) = \alpha P \cup \alpha Q$ and the traces of $P \parallel Q$ are pure interleavings between events of both processes: $traces(P \parallel Q) = \{s \mid \exists\, t : traces(P); u : traces(Q) \cdot s\ interleaves(t, u)\}$. On the other hand, when the alphabets of P and Q are exactly the same, $\alpha P = \alpha Q$, then $traces(P \parallel Q) = traces(P) \cap traces(Q)$. *Communication* is a special class of event described by a pair $c.v$ where c corresponds to the name of the channel and v corresponds to the value of the message which passes. Channels can be considered members of the alphabet of the process and used for communication in only one direction and between two processes [92]. If two processes P and Q are composed in parallel and both have a common channel c, interaction happens whenever both processes are ready to engage in the common channel. If P is ready for $c!v$ and process Q is ready for $c?x$, v can be passed from P to Q [41]: $(c!v \rightarrow P) \parallel (c?x \rightarrow Q_x) = c!.v \rightarrow (P \parallel Q_v)$. The result is an output channel and the process Q receives the value v. This can also be applied for channels with input-input behaviour. Our approach is similar to CSP concurrency where events from different machines can be composed and interact, similar to what happens between events of different processes.

In Z, composition can be achieved by combining schemas. Two signatures of different schemas can be combined if they are *type compatible*: each variable common to the two has the same type in both of them. The result is a larger signature which contains all the variables of each of them. The properties of each of the schemas can be connected through logical connectives such as $\vee$, $\wedge$, $\Rightarrow$ or $\equiv$ [173]. Still combining schemas, views [34, 99] allow the development of partial specifications that can interact through invariants that relate their state or by operations' synchronisation. Views are similar to our composition as it allows the exploration of sub-components interaction without variable sharing.

In Circus [186, 155] (that combines Z and CSP), processes may be defined explicitly or in terms of other processes (compound processes). Compound processes are defined using the CSP operators of sequence, external and internal choice, parallelism and interleaving, or their corresponding iterated operators, event hiding, or indexed operators, which are particular to Circus specifications. An action can be a schema, a guarded command, an invocation of another action, or a combination of these constructs using CSP operators. Communication is achieved by parallelism and interleaving of actions declaring a synchronisation channel set and two sets that partition all the variables. In the parallelism $A1[\![ns1|cs|ns2]\!]A2$, the actions $A1$ and $A2$ synchronise on the channels in set cs. Both $A1$ and $A2$ have access to the initial values of all variables in both $ns1$ and $ns2$. However, $A1$ and $A2$ may modify only the values of the variables in $ns1$ and $ns2$, respectively. The changes made by $A1$ in variables in $ns1$ are not seen by $A2$, and vice-versa. Oliveira *et al* [132] make use of the Circus communication system to specify a distributed fire protection system divided into fire detection and gas discharge covering two different separate zones. In our composition machine, the included machines communicate by synchronised events similar to channels in Circus. Similarly, the included machines can only modify their own variables but can read the other variables in a composed event.

CSP-OZ [167] (CSP combined with Object-Z) and TCOZ [142, 118, 117] (Timed Communicating Object-Z that is an integration of Object-Z and Timed CSP) use Object-Z data structure and the CSP structure for the control flow of a system. The Z mathematical toolkit is extended with object oriented structuring techniques. Timed CSP has strong process control modelling capabilities. The multi-threading and synchronisation primitives of CSP are extended with timing primitives. In CSP-OZ, classes in Object-Z and processes in CSP are given an identical failure divergence semantics (history of class objects corresponds to traces of processes in CSP) which allows the development of communication through synchronised operations with the same name in a similar style as it happens with channels in CSP: local parameters to operations can be passed via message passing [167, 75]. The sequencing of operation events is determined by the preconditions of the individual events, at each time the object participates in any event which is currently enabled [117]. Nevertheless such an approach is not well suited for considering multi-threading and real-time due to the restriction of operations to atomic events. TCOZ identifies operation schemas with terminating CSP processes that perform only state updates: rather than treating operation as atomic events, they are treated as sequences of abstract state-update events [117]. TCOZ specifications have the same strcuture as Object-Z ones except in the structure of the class definition that may include CSP channels and processes definitions. The Z operation schema is the only way to describe a state change in TCOZ and it is not responsible for communicating inputs and outputs. Composition occurs using communicating CSP-style channels between class objects: state, initialisation schemas are conjoined; operation schemas with the same name are also conjoined resulting in compositive objects. The behaviour of an (active) compositive is defined by the construct $MAIN$ along with the channel construct *chan*. Active objects are modelled as pure (non-terminating) CSP processes, using the basic infinite timed failures semantics [117]. For synchronisation, channel renaming may be required where input and output parameters can be passed similar to the original CSP. Intermediate channels can be introduced as internal interfaces between objects. The internal interfaces are protected from environment by hiding them [118]. Another approach for describing the semantics of TCOZ is given by Qin *et al* [142]: using unifying theories of programming (UTP), a unified semantic model for both channel and sensor/actuators based communications in TCOZ is defined. Unlike our approach, we do not blend different formalisms and define the corresponding semantics: we keep the Event-B semantics and inspired by the CSP, we build a correspondence between events' composition with possible value-passing communication and the synchronisation of processes using CSP channels. Just like in CSP, there is different semantics between input and output channels. For our composition, input and output parameters also have different semantics expressed by enabledness POs (cf. Definition 1.20 and Sect. 1.5.3.3). Note that these POs are currently not implemented in the Rodin platform.

In classical B the composition [4, 140, 158] uses keywords like *Includes* to extend a machine, not allowing writing access to variables in the included machine or keyword *Sees* used to complement machines. Although systems are developed in single machines in classical B, Bellegarde et at [32] suggest a composition by rearranging the separated machines and synchronising their operations under feasibility conditions. The behaviour of a component composition is seen as a labelled transition system using weakest preconditions, where a set of authorised transitions are defined. The objective is to verify the refinement of synchronised parallel composition between components but it is limited to finite state transitions and a finite number of components. This work differs from ours as it uses a labelled transition system while we use synchronisation and communication in the CSP style. Variable sharing is also possible unlike our shared event composition. Butler and Walden [52] discuss a combination of action systems and classical B by composing machines using parallel systems in an action systems style and preserving the invariants of the individual machines. This approach allows the classical B to derive parallel and distributed systems and since the parallel composition of action systems is monotonic, the sub-systems in a parallel composition may be refined independently. This work is closely related to our work as it follows a CSP style to compose actions with similar underlying semantics and notion of refinement. Combining state machine diagrams and classical B, Papatsaras and Stoddart [135] manually decompose a global a system into sub-components. The sub-components are then composed in classical B using the *Includes* keyword. Similarly to our approach, sub-components communicate via shared parameters. Since there is not a formal methodology to follow, the resulting composition needs to be proved to be a valid refinement of the global system which is not the case in our work, where we prove the monotonicity of our composition. Abrial et al [124, 15] propose a state-based decomposition for Event-B introducing the notion of shared variables and external events as described in Sect. 1.6.1. Although this approach and our work are both monotonic and sub-components can be refined independently, their respective nature is suitable for different kind of systems: parallel programs for shared variable and distributed systems for shared event [42]. Sorge et al [172] propose a feature composition in Event-B and define composition proof obligations to ensure its consistency. In the feature composition approach, exploration of specifications' composition with possible variable sharing (similar to the shared variable style) is allowed but no refinement is defined which differs from our work. Nevertheless similar to our work, sub-components POs are reused to avoid re-proving composition POs.

2.6 Conclusions

Based on the close relation between action systems and Event-B plus the correspondence between action systems and CSP [53], we define our Event-B composition with an event-based behaviour. Shared event composition is proved to be monotonic by means of proof obligations. Consequently sub-components can be further refined independently. Refinement in a "top-down" style for developing specifications is allowed including the generation of POs. During composition, sub-components interact through event parameters by value-passing. We extend Event-B to support shared event composition, allowing combination and reuse of existing sub-components through the introduction of *composed machines*. Required static checks are developed and POs are generated to validate the composition. Such an approach seems suitable for modelling distributed systems, where the system can be seen as a combination of interacting parts (sub-components). This work is the result of the exploration of specifications' composition in a shared event style. A methodology for the composition is defined including the verification of properties through the generation of proof obligations. We do not address the step corresponding to the translation of this composition exploration to an implementation and it is a study that needs to be carried out in the future. A tool was developed to support composition in the Rodin platform. Although we have defined the required POs for composition, currently they are not implemented in the tool. At the moment, the generation of a new machine (that is the expansion of the sub-components) is required to validate the composition. A file transfer case study defined as a distributed system is modelled using the composition tool. We intend to carry on developing the shared event composition approach by adding the enabledness POs when available for the Rodin platform. With the developed work, we have the necessary conditions to develop another reuse technique that can be seen as the inverse operation of composition: decomposition. This is further discussed in Chapter 4.

Chapter 3

Generic Instantiation

It is believed that reusability in formal development should reduce the time and cost of formal modelling within a production environment. Along with the ability to reuse formal models, it is desirable to avoid unnecessary re-proof when reusing models. Event-B supports generic developments through the context construct. However Event-B lacks the ability to instantiate and reuse generic developments in other formal developments. We propose a methodology to instantiate generic models and extend the instantiation to a chain of refinements. We define sufficient proof obligations to ensure that the proofs associated to a generic development remain valid in an instantiated development thus avoiding re-proofs. This chapter is based on the paper [163] that appeared in the ICFEM (International Conference in Formal Engineering Methods) 2009.

3.1 Introduction

Reusability has always been sought in several areas as a way to reduce time, cost and improve the productivity of developments [176]. Examples can be found in areas such as software, mathematics and even formal methods. Generic instantiation can be seen as a way of reusing components and solving difficulties raised by the construction of large and complex models [124, 15]. The goal is to reuse generic developments (single model or a chain of refinements) and create components with similar properties instead of starting from scratch. Reusability is applied through the use of a *pattern* as the basic structure and afterwards each new component is generated through parameterisation.

We propose a generic instantiation approach for Event-B by instantiating machines. The instances inherit properties from the generic development (pattern) and afterwards are *parameterised* by renaming/replacing those properties to more specific names according to the instance. Proof obligations are generated to ensure that assumptions used in the pattern are satisfied in the instantiation. In that sense our approach avoids re-proof of pattern proof obligations in the instantiation. The models are developed in the Rodin platform. A simple case study modelling a protocol communication is described to illustrate the use of instantiation.

Section 3.2 defines how generic instantiation is interpreted by us. In Sect. 3.3 instantiated machines are introduced. Section 3.4 gives an application of instantiation in combination with shared event composition. The application of instantiation to a chain of refinements is described in Sect. 3.5. Section 3.6 discusses an open question that arises when instantiating theorems and invariants in a pattern. Conclusions are drawn in Sect. 3.7.

3.2 Generic Instantiation

In order to explain our approach for Generic Instantiation we use a simple case study. A protocol is modelled between two entities, *Source* and *Destination* which communicate by sending messages through a channel. The content of the channel has a maximum dimension. To send a message it is necessary to add the content of the message to the channel. Based on the proposed requirements it is possible to create a context *ChannelParameters* to model the channel as seen in Fig. 3.1(b). The content of the message

```
machine Channel sees ChannelParameters

variables channel

invariants
  @inv1 channel ⊆ Message
  @inv3 finite(channel)
  @inv2 card(channel) ≤ max_size

events
  event INITIALISATION
    then
      @act1 channel ≔ ∅
  end

  event Send
    any m
    where
      @grd1 m ∈ Message
      @grd2 card(channel) < max_size
    then
      @act1 channel ≔ channel ∪ {m}
  end

  event Receive
    any m
    where
      @grd1 m ∈ channel
    then
      @act1 channel ≔ channel\{m}
  end
end
```

(a)

```
context ChannelParameters

constants max_size

sets Message

axioms
  @axm1 max_size ∈ ℕ
end
```

(b)

FIGURE 3.1: Machine *Channel* and respective context *ChannelParameters*

is of type *Message* and has a maximum dimension *max_size*. Figure 3.1(a) represents the machine side where a variable *channel* stores all the sent/received messages. The *channel* messages have type *Message* and the number of messages in the channel is limited. Messages are introduced in the *channel* to be sent as seen in event *Send*. The event *Receive* models the reception of the message in the destination by extracting the messages from the *channel*. Elements in *ChannelParameters* context are the parameters (type and constant) for the *Channel* machine.

Now suppose we wish to model a bi-directional communication between two entities using two channels. Both channels are similar so an option is to *instantiate machine Channel* twice to create two instances: one channel called *Request* and the other *Response*. The protocol, represented in Fig. 3.2 starts by a message being sent from the Source. After arriving at the Destination, the reception of the message is acknowledged in the Source. Then a response is sent from the Destination and after arriving at the Source, it is also acknowledged in the Destination.

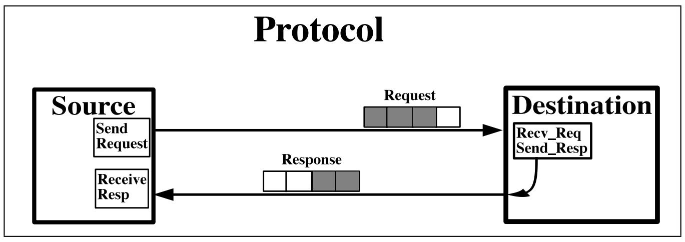


FIGURE 3.2: Protocol diagram

The instantiation of *Channel* is achieved by applying *machine instantiation*. An instance of the pattern *Channel* is created with more specific properties. A detailed description of the machine instantiation is described in Sect. 3.3. Moreover, a context containing the specific instances properties is required to model the protocol. In our case study we use the context *ProtocolTypes* in Fig. 3.3, where types *Request* and *Response* replace the more generic type *Message* and constants *qmax_size* and *pmax_size* replace *max_size*. This context must be provided by the modeller/developer.

```
context ProtocolTypes

constants qmax_size pmax_size

sets Request Response

axioms
  @axm1 qmax_size ∈ ℕ
  @axm2 pmax_size ∈ ℕ
end
```

FIGURE 3.3: ProtocolTypes Context

Abrial and Hallerstede [15] and Métayer et al [124] propose the use of generic instantiation for Event-B. It is suggested that the contexts of a development (equivalent to the pattern) can be merged and reused through instantiation in other developments. That proposal lacks a mechanism to apply the instantiation from the *pattern* to the instances. Therefore our work proposes a mechanism to instantiate machines and extend the instantiation to a refinement chain. The reusability of a development is expressed by instantiating a development (*pattern*) according to a more specific *problem*.

3.3 Generic Instantiation and Instantiated Machines

Inspired by the previous case study and having the ability to compose machines (Shared Event Composition plug-in [162]) and rename elements (Refactory plug-in [160] and Sect. 5.4) on the Rodin platform, we propose an approach to instantiate machines. As mentioned the context plays an important role while instantiating since this is where the specific properties of the instance are defined (parameterisation). The use of context is briefly discussed before *instantiated machines* are introduced.

3.3.1 Contexts

As aforementioned, contexts in Event-B are the static part of a model containing properties of the modelled system through the use of axioms and theorems. Having a closer look at the possible usage of contexts, there are two possible viewpoints:

Parameterisation : the context is seen only by one machine (or one chain of machine refinements) and defines specific properties for that machine (sets, constants, axioms, theorems). These properties are unique for that machine and any other machine would have different properties.

Sharing : a context is seen by several machines and there are some properties (sets, constants, axioms, theorems) shared by the machines. Therefore the context is used to share properties.

Several model developments mix the usage for the same context. For the ordinary modeller this distinction is not very clear and perhaps not so important. Our approach of generic instantiation reuses components and personalises each instance implying the use of ***Parameterisation***.

3.3.2 Example of Instantiated Machine

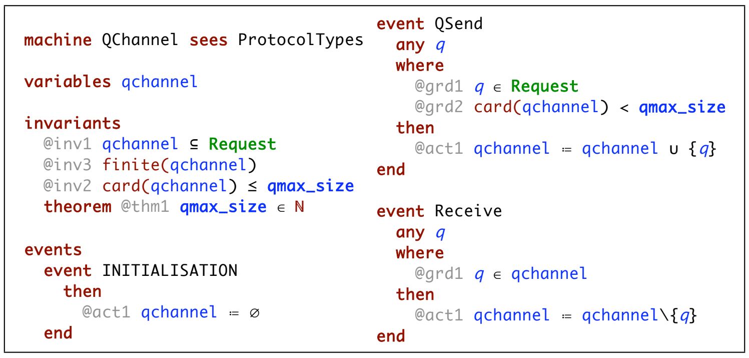
An *instantiated machine* instantiates a generic machine (pattern). If the generic machine sees a context, then the context elements (sets and constants) have to be replaced by instance elements. The instance elements must already exist in a context seen by the instantiated machine (in our case study, this corresponds to ProtocolTypes - see Fig. 3.3).

In the case study, the instantiated machine *QChannel*, that is an instance of the machine *Channel* for requests, is represented in Fig. 3.4. Note that *ChannelParameters* elements (sets and constants) are `replaced` because the replacement elements are already defined in *ProtocolTypes*. Machine elements (variables, parameters and events) are `renamed` since they did not exist before. The instantiated machine *PChannel*, an instance of *Channel* for responses, is similar.

```
INSTANTIATED MACHINE QChannel
INSTANTIATES Channel VIA ChannelParameters
SEES ProtocolTypes /* context containing the instance properties*/
REPLACE            /* replace parameters in ChannelParameters*/
    SETS Message := Request
    CONSTANTS max_size := qmax_size
RENAME          /* rename variables and events in machine Channel*/
    VARIABLES channel := qchannel
    EVENTS Send := QSend
            m := q   /*optional:rename parameter m in event Send*/
            Receive := Receive
            m := q /*optional:rename parameter m in event Receive*/
END
```

FIGURE 3.4: Instantiated Machine: *QChannel* instantiates *Channel*

Pattern Assumptions and Instance Theorems: Axioms in contexts are assumptions about a system and are used to help discharge proofs obligations. When instantiating, we need to show that assumptions in the pattern are satisfied by the replacement sets and constants. A possible solution is to convert the *pattern axioms* into *instantiated machine theorems* after the replacement is applied. A theorem has a proof obligation associated with it. By ensuring that a proof obligation related to each axiom is generated and discharged when instantiating a machine, we are confirming the correctness of the instantiation by satisfying the pattern assumptions (see Theorem *thm1* in Fig. 3.5). In this manner, the theorem is automatically generated in the instantiated machine and does not need to be manually added in the pattern context. "Expanded" machine *QChannel* can be seen in Fig. 3.5.



```
machine QChannel sees ProtocolTypes

variables qchannel

invariants
  @inv1 qchannel ⊆ Request
  @inv3 finite(qchannel)
  @inv2 card(qchannel) ≤ qmax_size
  theorem @thm1 qmax_size ∈ ℕ

events
  event INITIALISATION
    then
      @act1 qchannel ≔ ∅
  end

event QSend
  any q
  where
    @grd1 q ∈ Request
    @grd2 card(qchannel) < qmax_size
  then
    @act1 qchannel ≔ qchannel ∪ {q}
end

event Receive
  any q
  where
    @grd1 q ∈ qchannel
  then
    @act1 qchannel ≔ qchannel\{q}
end
```

FIGURE 3.5: Expanded version of instantiated machine QChannel

The instance *QChannel* sees the context *ProtocolTypes* (provided by the modeller/developer) that contains the context information for the instances. The type *Message* in context *ChannelParameters* is replaced by *Request* in *ProtocolTypes*, the constant *max_size* is replaced by *qmax_size*, the variable *channel* in *Channel* is renamed *qchannel* and event *Send* is renamed *QSend*. The axiom that exists in *ChannelParameters* is converted into a theorem in *QChannel* (but easily discharged by the axioms in *Proto-*

colTypes). We convert the axiom *axm1* from the generic context *ChannelParameters*:

$$@axm1\ max\_size \in \mathbb{N}$$

into the theorem *thm1* in the instance *QChannel*:

$$@thm1\ qmax\_size \in \mathbb{N}$$

This results from the replacement of the constant *max_size* by *qmax_size*. For a machine theorem, the respective proof obligation is [9]:

$$\boxed{\begin{array}{l} Axioms \\ Invariants \\ \vdash \\ Theorem \end{array}}$$

For theorem *thm1*, the proof obligation to be generated is the following:

$$\boxed{\begin{array}{l} qmax\_size \in \mathbb{N} \quad \texttt{/*axiom from } \mathit{ProtocolTypes}\texttt{*/} \\ pmax\_size \in \mathbb{N} \quad \texttt{/* axiom from } \mathit{ProtocolTypes}\texttt{*/} \\ qchannel \subseteq Request \ \texttt{/*invariant from } \mathit{QChannel}\texttt{*/} \\ \ldots \\ \vdash \\ qmax\_size \in \mathbb{N} \end{array}}$$

The first axiom of *ProtocolTypes* easily discharges this proof obligation. Note the expansion of *QChannel* is not required in practice. We use it to show the meaning of an *instantiated machine.*

3.3.3 Definition of Generic Instantiation of Machines

Based on the instantiated machine *QChannel*, a general definition for generic instantiation of machines can be drawn. Considering Context *Ctx* and machine *M* in Fig. 3.6 together as a *pattern*, we can create a generic *instantiatiated machine IM* as seen in Fig. 3.7.

(a)	(b)
CONTEXT Ctx **SETS** $S_1 ... S_m$ **CONSTANTS** $C_1 ... C_n$ **AXIOMS** $Ax_1 ... Ax_p$	**MACHINE** M **SEES** Ctx **VARIABLES** $v_1 ... v_q$ **EVENTS** $ev_1 ... ev_r$

FIGURE 3.6: Generic view of a context and a machine

The context D contains the replacement properties (sets $DS_1, \ldots, DS_m$ and constants $DC_1, \ldots, DC_n$) for the elements in context *Ctx*. The variables, events and parameters

```
INSTANTIATED MACHINE IM
INSTANTIATES M VIA Ctx
SEES D                /* context containing the instance properties */
REPLACE               /* replace parameters defined in context C */
    SETS $S_1 := DS_1, \ldots, S_m := DS_m$ /* Carrier Sets */
    CONSTANTS $C_1 := DC_1, \ldots, C_n := DC_n$ /* Constants */
RENAME                /*rename elements in machine M */
    VARIABLES $v_1 := nv_1, \ldots, v_q := nv_q$       /* optional */
    EVENTS $ev_1 := nev_1$       /* optional */
            $p_1 := np_1, \ldots, p_s := np_s$       /*  parameters:  optional */
            :
            $ev_r := nev_r$
END
```

Figure 3.7: An Instantiated Machine

are also renamed by new variables $nv_1, \ldots, nv_q$, new events $nev_1, \ldots, nev_r$ and new parameters $np_1, \ldots, np_s$. From the *pattern* we are able to create several instances that can be used in a more specific *problem*. During the creation of instances validity checks are required:

1. A static validation of replaced elements is required, e.g., a type must be replaced by a type and a constant with a constant.
2. All sets and constants should be replaced, i.e., no uninstantiated parameters.
3. Renaming the constants, variables, events must be injective (not introducing name clashes) in order to reuse all the existing proof obligations.
4. Replacing sets does not have to be injective. Different sets in the pattern can be replaced by the same instance set.
5. Only given sets (defined by the user) can be replaced. Built-in types such as integer numbers $\mathbb{Z}$ and boolean BOOL cannot be replaced.

3.3.4 Avoiding re-proofs

As described above, a proof obligation is a sequent of the form $H \vdash G$ (where H represents some hyphoteses and G represents a goal). Renaming variable (or replacing constant) v with w and type (carrier set) S to T results in instantiated POs as follows:

$$[v := w]\ (H \vdash G) \text{ (variable/constant instantiation)}$$
$$[S := T]\ (H \vdash G) \text{ (type instantiation)}$$

$H \vdash G$ is valid means that the proof has been constructed. We must ensure if $H \vdash G$ is valid, then any instantiation of $H \vdash G$ that avoids name clashes is also valid. Instantiation of variables and constants maintains validity since a sequent is implicitly universally quantified over its free variables and quantified variables may be renamed provided there are no name clashes.

Schmalz [156, 157], inspired by term rewriting in the Rodin platform, describe a theoretical foundation of term rewriting for logics of partial functions as well as the semantics of Event-B logic based on Isabelle/HOL [85, 130]. [157] describes the rewriting and instantiation of proof rules as redundant inference rules that may be derived from a given valid proof rule while preserving soundness. Existing generic proofs can be reused in the instances following some side-conditions as described below. A general inference rule is written as:

$$\frac{\vdash H_1 \quad \dots \quad H_n}{\vdash G} \; r \quad (\underline{x}\, fresh) \tag{3.1}$$

where $H_1 \dots H_n$, $n > 0$, are a sequence of (possible empty) sequents called *antecedents*, G is a sequent called *consequent*, has an optional name r and $\underline{x}$ are possible empty freshness conditions (the variables introduced as part of a proof rule step like $\forall goal$). Variables in $\underline{x}$ are pairwise distinct and do not occur free in G. Furthermore, by convention, type variables are considered free in the sequents in Event-B [156]. Two kind of substitution are considered:

Ordinary (bound variable) Substituition: σ_1 replaces ordinary variables $\underline{y}$ by variables $\underline{u}$ (called the right-hand side of σ_1). It is denoted as: $\sigma_1 = [\underline{y} := \underline{u}]$ where $\underline{y}$ is a sequence of pairwise distinct variables and $\underline{u}$ a sequence of variables of the same length and type as $\underline{y}$.

Type Substitution: σ_2 substitutes type variables $\underline{\mu}$ for type variables $\underline{\alpha}$. It is denoted as: $\sigma_2 = [\underline{\alpha} := \underline{\mu}]$ where $\underline{\alpha}$ is a sequence of pairwise distinct type variables and $\underline{\mu}$ a sequence of types having the same length as $\underline{\alpha}$.

The instantiation of inference rule r for ordinary and type substitution can be expressed as:

$$\frac{\vdash H_1\sigma \quad \dots \quad H_n\sigma}{\vdash G\sigma} \; r \quad (\underline{x}\sigma\, fresh) \tag{3.2}$$

where σ is a substitution over Σ (type signature containing the set of all types). In an ordinary substitution, the right-hand side of σ corresponds to ordinary variables. Moreover, the instantiation is possible if the following side-conditions on σ hold [156]:

- The variables in $\underline{x}\sigma$ are pairwise distinct. Moreover, for type substitution, variables in $\underline{x}$ cannot have the same name and different types.
- If a variable x is free in one of $H_1 \dots H_n$ and $x\sigma$ belongs to $\underline{x}\sigma$, then x belongs to $\underline{x}$.

POs in the generic model (pattern) are sequents of the form:

$$A, I, H \vdash_D G \tag{3.3}$$

where A represents the axioms, I represents the invariants and H represents the guards.

The substitution σ results in A being replaced by B, where B is the specific axioms and we have that

$$B \vdash_D A\sigma \tag{3.4}$$

From (3.3) and [157], we have that

$$A\sigma, I\sigma, H\sigma \vdash_D G\sigma \tag{3.5}$$

i.e, variable and type substitution preserves validity. Then from (3.4) and (3.5), we have:

$$B, I\sigma, H\sigma \vdash_D G\sigma \tag{3.6}$$

(3.3) is the form that a PO takes in the pattern machine, (3.6) is the form a PO takes in the specific machine and we have shown that (3.6) follows from (3.3).

3.4 Example of Instantiation and Composition

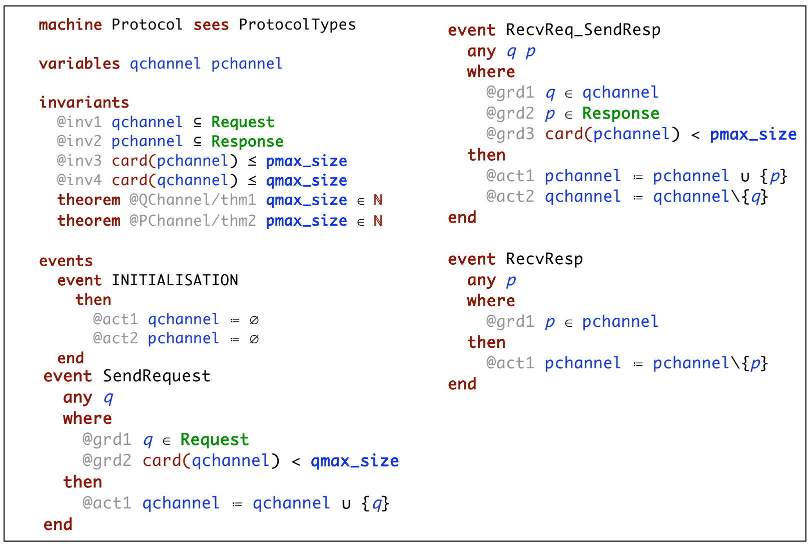
The creation of the instances is an intermediate step in the overall model development. In our case study we model a protocol between entities that sends and receives messages. By using the created instances and the shared event composition plug-in, we share events between *Request* and *Response* and model the protocol. A composed machine *Protocol* modelling this system can be seen in Fig. 3.8.

```
COMPOSED MACHINE Protocol
REFINES -
INCLUDES
    QChannel
    PChannel
EVENTS
    SendRequest
        Combines Events QChannel.QSend
    RecvReq_SendResp
        Combines Events QChannel.Receive || PChannel.Send
    RecvResp
        Combines Events combines PChannel.Receive
END
```

FIGURE 3.8: Composed Machine Protocol

As seen in Fig. 3.2, while composing the instance machines *QChannel* and *PChannel* we add the events that are unique for each entity (*SendRequest* and *RecvResp*). In Fig. 3.8, event *SendRequest* sends a message through the channel from Source to Destination.

RecvResp models the reception of the response in the *Source* after being sent by Destination. Moreover the event that relates the communication between the two entities is also modelled (*RecvReq_SendResp*). The request is received and acknowledged and the response to that request is sent in parallel (from this combined event, a possible refinement is processing the request message before sending the response). We opt not to refine an abstract machine in Fig. 3.8 (*REFINES* clause is empty: “-”) although it is possible. The composed machine *Protocol* corresponds to the expanded machine in Fig. 3.9.



```
machine Protocol sees ProtocolTypes

variables qchannel pchannel

invariants
  @inv1 qchannel ⊆ Request
  @inv2 pchannel ⊆ Response
  @inv3 card(pchannel) ≤ pmax_size
  @inv4 card(qchannel) ≤ qmax_size
  theorem @QChannel/thm1 qmax_size ∈ ℕ
  theorem @PChannel/thm2 pmax_size ∈ ℕ

events
  event INITIALISATION
    then
      @act1 qchannel ≔ ∅
      @act2 pchannel ≔ ∅
  end
 event SendRequest
  any q
  where
    @grd1 q ∈ Request
    @grd2 card(qchannel) < qmax_size
  then
    @act1 qchannel ≔ qchannel ∪ {q}
 end

event RecvReq_SendResp
  any q p
  where
    @grd1 q ∈ qchannel
    @grd2 p ∈ Response
    @grd3 card(pchannel) < pmax_size
  then
    @act1 pchannel ≔ pchannel ∪ {p}
    @act2 qchannel ≔ qchannel\{q}
end

event RecvResp
  any p
  where
    @grd1 p ∈ pchannel
  then
    @act1 pchannel ≔ pchannel\{p}
end
```

FIGURE 3.9: *“Expanded” machine Protocol*

The two instances of machine *Channel* model a bi-directional communication channel between two entities. This allows us to express the applicability of generic instantiation for modelling distributed systems. Nevertheless generic instantiation is not restricted to this kind of systems. When modelling a finite number of similar components with some specific individual properties, instantiated machines are a suitable option (as described in our case study in Chapter 6).

3.5 Generic Instantiation applied to a chain of refinements

The above sections describe generic instantiation applied to individual machines. Although it is already an interesting way of reusing, the instantiation of a chain of machines in a large model would be more interesting. In other words, we *instantiate a chain of refinements*. Suppose we have a development Dv containing several refinement levels ($Dv_1, Dv_2, \ldots, Dv_n$). The most concrete model Dv_n matches a generic model (pattern) P_1 that is part of a chain of refinements $P_1, P_2, \ldots, P_m$ as seen in Fig. 3.10. By applying generic instantiation we instantiate the pattern P_1 according to Dv_n. That instantiation is a refinement of Dv_n and it is called Dv_{n+m}_abs (the suffix *abs* stands for abstract). In

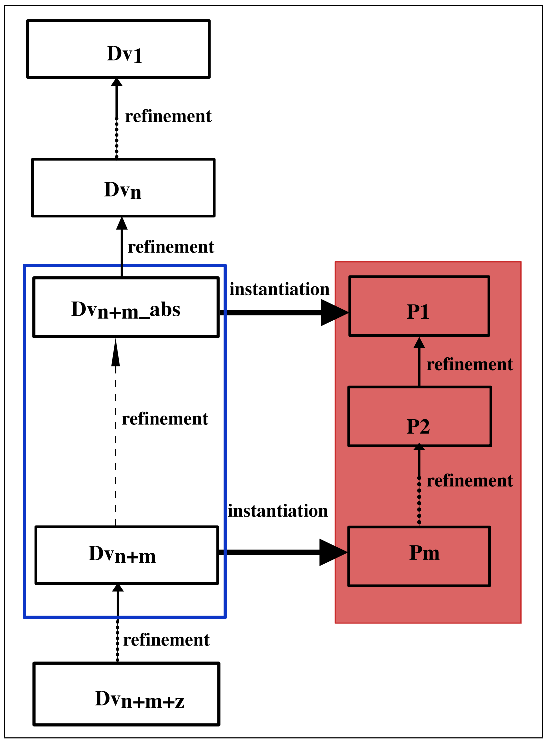


FIGURE 3.10: Instantiation of a generic chain of refinements

addition we can extend the instantiation to one of the refinement layers of the pattern and apply it to the development Dv. The outcome is a further refinement layer for Dv_n for free (Dv_{n+m}_abs corresponds to the instantiation of P_1 and Dv_{n+m} corresponds to the instantiation of P_m). The refinement between Dv_{n+m}_abs and Dv_{n+m} does not introduce refinement proof obligations since the proof obligations were already discharged in the pattern chain. This follows from the instantiated machines where the re-proof of pattern proof obligations is avoided. Afterwards Dv_{n+m} can be further refined to Dv_{n+m+z}. For a better understanding of this approach, we will refine our case study and apply an instantiation over the pattern chain.

3.5.1 Refinement of the Channel case study

We refine the *Channel* machine. For the first refinement, the requirement is to include buffers before and after adding a message to the channel. A second refinement specifies the type *Message*. In particular, *Message* will be divided in two parts: *header* and *body*. The *header* of the *Message* contains the destination identification and the *body* represents the content of the message (data). *header* and *body* are based on the records proposal for Event-B suggested by Evans and Butler [69] and also in work developed by Rezazadeh *et al.* [150].

The first refinement requires the introduction of two new variables *sendingBuffer* and *receivingBuffer* and a new event *addMessageBuffer* that loads the message to *sending-*

Buffer before being introduced in the channel in the *Send* event. The latter event reflects the introduction of the buffers. In the event *Receive*, messages in *channel* are extracted and loaded to *receivingBuffer* as seen in Fig. 3.11.

```
machine Channel_M1 refines Channel
sees ChannelParameters

variables channel sendingBuffer
          receivingBuffer

invariants
  @inv1 sendingBuffer ⊆ Message
  @inv2 receivingBuffer ⊆ Message

events
  event INITIALISATION
    then
      @act1 channel ≔ ∅
      @act2 sendingBuffer ≔ ∅
      @act3 receivingBuffer ≔ ∅
  end

  event addMessageBuffer
    any m
    where
      @grd1 m ∈ Message
      @grd2 m ∉ sendingBuffer
    then
      @act1 sendingBuffer≔sendingBuffer∪{m}
  end

event Send refines Send
  any m
  where
    @grd1 sendingBuffer ≠ ∅
    @grd2 m ∈ sendingBuffer
    @grd3 card(channel) < max_size
  then
    @act1 channel ≔ channel ∪ {m}
    @act2 sendingBuffer≔sendingBuffer∖{m}
end

event Receive refines Receive
  any m
  where
    @grd1 m ∈ channel
    @grd2 m ∉ receivingBuffer
  then
    @act1 channel ≔ channel∖{m}
    @act2 receivingBuffer≔receivingBuffer∪{m}
end
```

FIGURE 3.11: *Channel_M1*: refinement of *Channel*

The second refinement is a data refinement over the type *Message* by dividing it into *header* and *body*. The *header* contains the destination identification and the *body* contains the data of the message. Constants *header* and *body* are defined in the context *ChannelParameters_C2* as in Fig. 3.12.

```
context ChannelParameters_C2 extends ChannelParameters

constants header body

sets DATA DESTINATION

axioms
  @axm3 header ∈ Message → DESTINATION
  @axm4 body ∈ Message → DATA
end
```

FIGURE 3.12: Context *ChannelParameters_C2*

In Fig. 3.13 the machine *Channel_M2* data refines the variable *channel* and introduces a new event, *processMessage* that processes the received message after being retrieved from the receiving buffer. A variable *storeDATA* is also introduced to store the data that each destination receives.

3.5.2 Instantiation of a chain of refinements

We can consider the chain of refinements of *Channel* as a pattern. In that case, having all the proof obligations discharged we can reuse this pattern in a more specific development. The chain of refinements is seen as a single entity where it is possible to choose an *initial* and a *final* refinement level. Using our case study, we intend to instantiate and refine

```
machine Channel_M2 refines Channel_M1
sees ChannelParameters_C2

variables channel sendingBuffer
          receivingBuffer storeDATA

invariants
  @inv1 storeDATA ∈ DESTINATION ⇸ ℙ(DATA)

events
  event INITIALISATION
    then
      @act1 channel ≔ ∅
      @act2 sendingBuffer ≔ ∅
      @act3 receivingBuffer ≔ ∅
      @act4 storeDATA ≔ DESTINATION × {∅}
  end

  event addMessageBuffer
  refines addMessageBuffer
    any h b m
    where
      @grd1 header(m) = h
      @grd2 body(m) = b
      @grd3 m ∉ sendingBuffer
    then
      @act4 sendingBuffer≔sendingBuffer∪{m}
  end

event send refines Send
  any m
  where
    @grd1 sendingBuffer ≠ ∅
    @grd2 m ∈ sendingBuffer
    @grd3 card(channel) < max_size
  then
    @act1 channel ≔ channel ∪ {m}
    @act2 sendingBuffer≔sendingBuffer\{m}
end

event receive refines Receive
  any m
  where
    @grd1 m ∈ channel
    @grd2 m ∉ receivingBuffer
  then
    @act1 channel ≔ channel\{m}
    @act2 receivingBuffer≔receivingBuffer∪{m}
end

event processMessage
  any m dest d
  where
    @grd1 m ∈ receivingBuffer
    @grd3 header(m) = dest
    @grd4 d = body(m)
    @grd5 dest ∈ dom(storeDATA)
  then
    @act1 storeDATA(dest)≔storeDATA(dest)∪{d}
end
```

FIGURE 3.13: *Channel_M2*: refinement of *Channel_M1*

QChannel with the chain of refinements of machine *Channel*, selecting *Channel* and *Channel_M2* as our initial and final refinement levels respectively. In Fig. 3.14 the shaded chain of refinement is seen as a single entity. After the selection of the two refinement levels to be instantiated, *QChannel_M2_abs* and *QChannel_M2* are created. *QChannel_M2* is treated as a refinement of *QChannel_M2_abs* as a consequence of the instantiation. Subsequently, *QChannel_M2* can be further refined to *QChannel_Mz*.

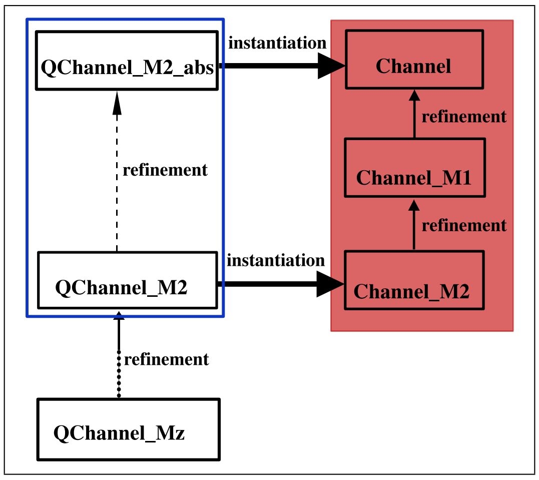


FIGURE 3.14: Instantiation of a chain of refinements: *Channel* to *Channel_M2*

The refinement relationship between *Channel* and *Channel_M2* is ensured by discharging all the proof obligations in the chain of refinement (all the proofs are discharged automatically in the Rodin platform). By instantiating *Channel* and *Channel_M2* implicitly we are also referring to *Channel_M1*. Some of the properties of *Channel_M2* are inherited from *Channel_M1* (for instance the buffers) but for the instantiation purpose it is not necessary to incorporate *Channel_M1* explicitly. The instantiation of a chain of refinements follows the instantiation of a single machine as seen in Fig. 3.15.

```
INSTANTIATED REFINEMENT QChannel_M2
INSTANTIATES Channel_M2 VIA ChannelParameters_C2
REFINES -
SEES ProtocolTypes_C2
REPLACE
      SETS Message := Request
      CONSTANTS max_size := qmax_size
                      header := qHeader
                      body := qBody
RENAME
      VARIABLES channel := qchannel
                     receivingBuffer := qReceivingBuffer
                     sendingBuffer := qSendingBuffer
      EVENTS Send := QSend
                m := q
                receive := Receive
                m := q
END
```

FIGURE 3.15: Instantiation of a chain of refinements

The initial refinement level corresponds to the most abstract machine of the pattern. Therefore it is not necessary to explicitly refer to it. The final refinement level is any of the other refinement levels in the chain. The replacement and renaming is applied to the occurrences in both instances whenever applicable. Once again it is not necessary to "expand" *QChannel_M2* but that can be seen in Fig. 3.16. In an instantiation of a chain of refinements, the pattern context is seen as a *flat context* comprising all the properties seen by the refinements until the selected final refinement level is reached. Therefore context *ProtocolTypes_C2* is the parameterisation context for *QChannel_M2* and extends *ProtocolTypes*, similarly to the relation between contexts *ChannelParameters_C2* and *ChannelParameters*. As before, axioms in *ProtocolTypes_C2* must be respected in the instance, so axioms are converted in theorems in *QChannel_M2*.

3.5.3 Definition of Generic Instantiation of Refinements

From the case study it is possible to draw a generic definition for the instantiation of a chain of refinements. If we consider a pattern that consists of a chain of refinements *M1, M2, . . . Mt*, we can create a generic *instantiated refinement IR* as seen in Fig. 3.17. The instantiated refinement *IR* instantiates one of the refinements of the pattern M_t via the parameterisation context Ctx_t. *IR* refines an abstract machine IR_0 and sees the context D_w containing the instance properties. The replacement and renaming are similar to the machine instantiation but apply to both M_1 and M_t. The initial level does not need to be explicitly defined since the most abstract level of the chain is automatically considered. Therefore M_1 is automatically defined as the initial level. In addition to the validity checks for instantiated machines, instantiated refinements require:

```
machine QChannel_M2 refines QChannel_M1
sees ProtocolTypes_C2

variables qchannel qReceivingBuffer
          qSendingBuffer qStoreDATA

invariants
  @inv1 qStoreDATA ∈ DESTINATION ⇸ ℙ(DATA)
  theorem @theo1 qHeader ∈ Request → DESTINATION
  theorem @theo2 qBody ∈ Request → DATA

events
  event INITIALISATION
    then
      @act1 qchannel ≔ ∅
      @act2 qSendingBuffer ≔ ∅
      @act3 qReceivingBuffer ≔ ∅
      @act4 qStoreDATA ≔ DESTINATION × {∅}
  end

  event AddMessageBuffer
  refines qAddMessageBuffer
    any h b m
    where
      @grd1 qHeader(m) = h
      @grd2 qBody(m) = b
      @grd3 m ∉ qSendingBuffer
    then
      @act1 qSendingBuffer ≔ qSendingBuffer∪{m}
  end

  event QSend refines QSend
    any q
    where
      @grd1 qSendingBuffer ≠ ∅
      @grd2 q ∈ qSendingBuffer
      @grd3 card(qchannel) < qmax_size
    then
      @act1 qchannel ≔ qchannel ∪ {q}
      @act2 qSendingBuffer≔qSendingBuffer\{q}
  end

  event Receive refines Receive
    any q
    where
      @grd1 q ∈ qchannel
      @grd2 q ∉ qReceivingBuffer
    then
      @act1 qchannel ≔ qchannel\{q}
      @act2 qReceivingBuffer≔qReceivingBuffer∪{q}
  end

  event processMessage
    any m dest d
    where
      @grd1 m ∈ qReceivingBuffer
      @grd2 qHeader(m) = dest
      @grd3 d = qBody(m)
      @grd4 qHeader(m) ∈ dom (qStoreDATA)
    then
      @act1 qStoreDATA(dest)≔qStoreDATA(dest)∪{d}
  end
```

(a)

```
context ProtocolTypes_C2 extends ProtocolTypes

constants qHeader qBody pHeader pBody

sets DATA DESTINATION

axioms
  @axm3 qHeader ∈ Request → DESTINATION
  @axm4 qBody ∈ Request → DATA
  @axm5 pHeader ∈ Response → DESTINATION
  @axm6 pBody ∈ Response → DATA
end
```

(b)

FIGURE 3.16: Expanded version of instantiated machine *QChannel_M2* (a) and context *ProtocolTypes_C2* (c)

1. A static validation for the existence of a chain of refinements for M $(M_1, M_2, \ldots, M_t)$.

2. The types and constants in the contexts seen by the initial and final level of refinement should be instantiated.

The instantiation of refinements reuses the pattern proof obligations in the sense that the instantiation renames and replaces elements in the model but does not change the model itself (nor the respective properties). The correctness of the refinement instantia-

```
INSTANTIATED REFINEMENT IR
INSTANTIATES $M_t$ VIA $Ctx_t$
REFINES $IR_0$      /* abstract machine */
SEES $D_w$          /* context containing the instance properties */
REPLACE            /* replace parameters defined in context C */
    SETS $S_1 := DS_1, \ldots, S_m := DS_m$ /* Carrier Sets */
    CONSTANTS $C_1 := DC_1, \ldots, C_n := DC_n$ /* Constants */
RENAME     /*rename variables, events and params in $M_1$ to $M_t$ */
    VARIABLES $v_1 := nv_1, \ldots, v_q := nv_q$
    EVENTS $ev_1 := nev_1$       /* optional */
              $p_1 := np_1, \ldots, p_s := np_s$      /* parameters :optional */
              ...
              $ev_r := nev_r$
END
```

FIGURE 3.17: An Instantiated Refinement

tion relies in reusing the pattern proof obligations and ensuring the assumptions in the context parameterisation are satisfied in the instantiation.

3.6 Instantiating Theorems and Invariants

Theorems in contexts and machines are assertions about characteristics and properties of the system. Theorems have associated proof obligations that are discharged based on the model assumptions (axioms and invariants) . Once the theorems are discharged, they can be used as hypotheses for discharging other proof obligations in the model, since they work as a consequence of the assumptions.

An interesting question arises when a pattern is instantiated and contains theorems and invariants. If a proof obligation of a theorem is discharged by creating an instance we would not want to re-prove the theorem proof. Regarding the invariants and respective proof obligations we would have a similar situation where we would not want to discharge proof obligations in the instance if they were already discharged in the pattern. Ideally we would like to add to the instance the assumptions and assertions given by the theorems and invariants without re-proving them. Although addressed here as an open question, this situation suggests a different kind of theorem that does not exist in Event-B, a *pre-proved theorem* to be used in the instance. A *pre-proved theorem* would be similar to a theorem but it would not have an associated proof obligation. The invariants imported from the pattern fall under the same category, where the respective proof obligations should not be re-generated. Informally the instances are just renaming and replacing elements without changing the semantics under the original pattern (if the validity checks are followed) so theorems and invariants would work as assumptions in the instantiated machine. The assumptions in the pattern (axioms) need to be satisfied by the instances through the generation of proof obligations but the same does not apply for invariants and theorems that are assertions in the pattern.

3.7 Conclusions

Reusability is of significant interest in the general software engineering research community. Reuse has its advantages and disadvantages discussed by Standish [176] and Cheng [55]. Reusing patterns in a style similar to design patterns is proposed in [63] using the KAOS specification language and temporal logic. KAOS goals are combined with existing patterns, that are already proved correct and complete and proofs can be reused. Sabatier [154] discusses the reuse of formal models as a detailed component specification or a high level requirement, and presents some real project examples. In classical B [158, 4], reuse is expressed using the keywords *INCLUDES* and *USES* where an existing machine can be used in other developments. Instantiation is a way of reusing.

Instantiation is well-established in areas such as mathematics and other formal methods like classical B or theorem provers such as Isabelle [137]. Blazy *et al.* [33] reuse Gang of Four (GoF) design pattern adapted to formal specifications (denominated specification patterns) for classical B. Several reuse mechanisms are suggested like instantiation, composition and extension. Proof obligations are also reused when the patterns are applied. Focusing on the instantiation, this is achieved by renaming sets (machine parameters), variables and operations. Unlike our work, this approach only defines patterns as a single abstract machine whereas we define the parameterisation in contexts and extend the pattern to a chain of refinements.

Abrial and Hallerstede [15] and Métayer et al [124] make use of generic instantiation for Event-B. The flattening of the context is proposed in a way that the contexts of the pattern are merged and the reuse by instantiating the flat context is suggested. Following and extending that approach, we:

- propose a methodology for the implementation of generic instantiation.
- define a generic instantiation mechanism for a machine as an *instantiated machine*.
- define a generic instantiation mechanism for a chain of refinement as an *instantiated refinement*.
- show that that generic proofs can be reused in specific instances under the conditions described in Sects. 3.3.3, 3.3.4 and 3.5.3.

The motivation for such implementation is concerned with reusability of components and existing developments. By creating an instance from a generic model, a new parameterised model is created based on the pattern with new specific properties.

Event-B supports generic developments but lacks the capacity to instantiate and reuse those generic developments. As a solution, generic instantiation is applied to patterns and as an outcome instantiated machines are created and parameterised. An *instantiated machine* instantiates a generic machine, is parameterised by a context and the pattern elements are renamed/replaced according to the instance. In a similar style, an *instantiated refinement* instantiates a chain of refinements reusing the pattern proof obligations assuming that the instantiated proof obligations are as valid as the pattern ones. By quantifying the variables, constants and types we want to ensure that pattern proof obligations remain valid when instantiating. Event-B is not a higher-order formalism: although it is possible to quantify over expression, it is not possible to directly quantify types. Nevertheless instantiation of sequences (hypotheses and goal) is possible as long as is done in an alpha-congruent manner. Therefore the generic proofs can be instantiated and be used in the instance since they will also hold. A renaming plug-in was developed supporting the renaming of Event-B elements and respective proofs. Optimisation at level of proof renaming will be investigated in the future as it may be a slow operation for large proof trees. A practical case that models a communication protocol between two entities illustrates the advantages of using generic instantiation and in particular how to use our approach in the Rodin platform. Although a simple case study, we believe that it can be applied to more complex cases.

Further study is required to determine if context instantiation similar to instantiated machines is a worthwhile approach while modelling (for instance, to instantiate sets into *implementable* types) Some methodological points will arise in a possible implementation of instantiated machines and refinements in the Rodin platform. As an example, Section 3.6 addresses the situation of instantiating theorems and invariants and is left as an open question. A future step for the instantiation of a chain of refinements is to study the possibility of selecting any of the refinement levels as the initial refinement level giving more freedom to the modeller. In a long term perspective, any refinement chain could be considered a pattern. Moreover a library of patterns could be provided when modelling: whenever a formal development fits in a pattern, instantiation could be applied taking advantage of the reusability of the model and respective proof obligations.

Chapter 4

Decomposition

In the previous chapters we defined mechanisms for reusability. Still following that line of work, we propose decomposition as another approach for reusing. Decomposition is motivated by the possibility of breaking a complex problem or system into parts that are easier to conceive, manage and maintain. The partition of a model into sub-components can also be seen as a design/architectural decision and the further development of the sub-components in parallel is possible. Two methods have been identified for Event-B model decomposition: *shared variable* and *shared event*. Besides alleviating the complexity for large systems and respective proofs, decomposition allows team development in parallel over the same model which is very attractive in the industrial environment. Moreover the proof obligations of the original (non-decomposed) model can be reused by the sub-components. This chapter describes the work on decomposition, which is one of the main goals of this thesis. Part of this work was accepted as a workshop paper in Workshop on Tool Building in Formal Methods as [165] in the conference ABZ 2010 and afterwards selected to be extended and appear in a special edition of the journal *Software: Practice and Experience* as [166]. This work was carried out in collaboration with Thai Son Hoang and Carine Pascal. Our contribution was the development of the shared event approach in terms of methodology and in terms of tool support. This is described in more detail in Chapter 5. The decomposition tool developed for the Rodin platform has been successfully used in several case studies such as a flash system development [62, 60], decomposition of a space craft system [73], development of a cruise control system, development of a pipeline system, among other works.

4.1 Introduction

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. The main purpose of the model decomposition is precisely to address such difficulty by cutting a large model into smaller components. The complexity of the whole model (also referred as original model) is decreased by studying, and thus refining, each part independently of the others [124]. Two methods have been identified for the Event-B decomposition: shared variable [15, 8] and shared event [45, 47]. Moreover the decomposition also partitions the POs which are expected to be easier to be discharged in the sub-components. From another point of view, shared event decomposition is the inverse operation of shared event composition described in Chapter 2. There it was proved that the shared approach is monotonic and therefore the resulting sub-components could be further refined. That proof is applied to decomposition in a similar fashion following the failure-divergence definition of CSP as described in Sect. 1.6.2. The properties of parallel composition in CSP are also the properties of shared event decomposition. The most relevant property is monotonicity: as long as the partition of events maintains the original events interface, the decomposition properties hold which allow the independent refinement of sub-components. For the shared variable composition, the monotonicity is proved in [8]. Therefore we can introduce team development: several developers share parts of the same model but work independently and in parallel. We propose a plug-in developed in the Rodin platform [151] that supports these two decomposition methods for Event-B.

Section 4.2 introduces decomposition using a simple example, describing how model properties are partition, proof obligations are split and the possibility of refining sub-components. The definition and validity of the decomposition is illustrated in Sect. 4.3. Section 4.4 describes the limitations of this approach. We conclude this chapter in Sect. 4.5 with a summary of this study, discussion about related work, applications and future work.

4.2 Decomposition Styles

The discussion about the two styles of decomposition was introduced in Sect. 1.7. The semantics of decomposition is the syntactic composition of M1 and M2 and the proof obligations for M are then derived via that syntactic composition. Consequently, machines $M1$ and $M2$ are constructed according to descriptions in Sect. 1.7.1 and Sect. 1.7.2. The definition of decomposition is described in Sect. 4.3. Here an example is presented to illustrate the use of both kind of decompositions. A simple communication process is modelled. The abstract model can be seen in Fig. 4.1.

The variable a is initialised with the constant *d0* and variable b is assigned a value non-deterministically. The initial model contains only the event *copy* that copies the value of a to variable b in one single step as described in Fig. 4.2(a). A refinement of *Communication* (*Communication_M1*) introduces a middleware entity that stores

```
machine Communication_M0

sees Communication_C0

variables a b

invariants
  @inv1 a ∈ DATA
  @inv2 b ∈ DATA

events
  event INITIALISATION
    then
      @act1 a ≔ d0
      @act2 b :∈ DATA
  end

  event copy
    then
      @act1 b ≔ a
  end
end
```

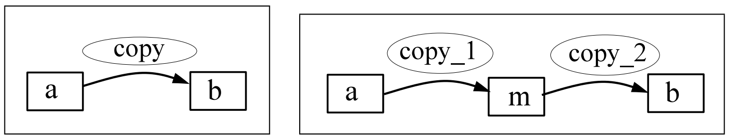
(a) Machine *Communication_M0*

```
context Communication_C0

constants d0

sets DATA

axioms
  @axm1 d0 ∈ DATA
end
```

(b) Context *Communication_C0*

FIGURE 4.1: Event-B model of the *Communication* example

temporarily the value of a before copying it to b as seen in Fig. 4.2(b). Variable m represents the middleware.



(a) Diagram of abstract machine *Communication_M0* (b) Diagram of *Communication_M1*, refinement of *Communication_M0*

FIGURE 4.2: Diagrams corresponding to the Simple Communication example

Butler [47, 46] suggests an event refinement diagram to decompose atomicity and we use it to show the refinement relationship between the events in *Communication_M0* and *Communication_M1* as seen in Fig. 4.3. In fact we are decomposing the initial single atomic operation into two steps using a middleware. The diagram is read from left to

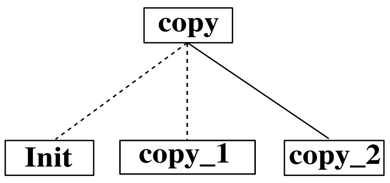


FIGURE 4.3: Event refinement diagram illustrating atomicity decomposition

right and that indicates its sequential control. In other words, the abstract event *copy* is refined by first executing the initialisation event (*Init*), then event *copy_1* and afterwards *copy_2*. In the same figure, the lines that link the events are relevant: a dashed line represents events that refine skip (such as *Init* and *copy_1(p)*). A solid line defines a refinement relation between events. Thus event *copy_2* must be proved to refine *copy*. The refinement *Communication_M1* can be seen in Fig. 4.4. Note that a control variable *ctrl* is introduced to ensure when the content of m can be copied to b. Invariant *inv3* ex-

```
machine Communication_M1 refines Communication_M0

sees Communication_C0

variables a b m ctrl

invariants
  @inv1 m ∈ DATA
  @inv2 ctrl ∈ BOOL
  @inv3 ctrl = TRUE ⇒ m = a

variant {ctrl,TRUE}

events
  event INITIALISATION
    then
      @act1 a ≔ d0
      @act2 b :∈ DATA
      @act3 m :∈ DATA
      @act4 ctrl ≔ FALSE
  end

  convergent event copy_1
    any p
    where
      @grd1 p = a
      @grd2 ctrl = FALSE
    then
      @act1 m ≔ p
      @act2 ctrl ≔ TRUE
  end

  event copy_2 refines copy
    where
      @grd1 ctrl = TRUE
    then
      @act1 b ≔ m
      @act2 ctrl ≔ FALSE
  end
end
```

FIGURE 4.4: Machine *Communication_M1* refinement of *Communication_M0*

presses that when variable *ctrl* is true, the value of the middleware m corresponds to the value of source a. This invariant can be seen as a requirement for the refinement between abstract event *Communication_M0.copy* and concrete event *Communication_M1.copy_2*. The convergent event *copy_1* requires a variant that guarantees that this event is not enabled forever. Such variant is expressed as $\{ctrl, TRUE\}$ which means that eventually the control variable *ctrl* will be TRUE and in that case *copy_1* event is not enabled.

Depending on the chosen decomposition style, a system can be decomposed into different number of sub-components as seen in the following sections. In the rest of this section, we give an informal introduction to the two decomposition styles using a running example. In Sect. 4.3 we give decomposition a precise definition and show that they represent valid refinements.

4.2.1 Shared Event Decomposition of *Communication*

From the modeller's point of view, the decomposition starts by defining which sub-components will be generated. The following step is to define the partition of variables over the sub-components. The rest of the model decomposition (events, parameters, invariants, contexts) is a consequence of the variables allocation as defined below. For the shared event decomposition, we decompose *Communication_M1* in three parts: *MA*, *MB* and *MM* as seen in Fig. 4.5.

Variable a is allocated to machine *MA*, variables m and *ctrl* to machine *MM* and variable b to machine *MB*. It follows that event *copy_1* is split between *MA* and *MM* and event *copy_2* is split between *MB* and *MM*. The resulting machines can be seen in Fig. 4.6. Next we describe the steps for a machine decomposition focusing on invariants, events,

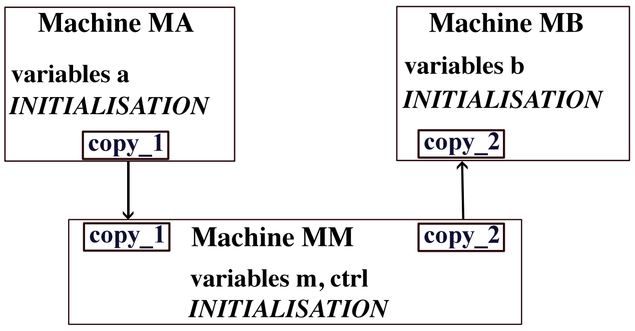
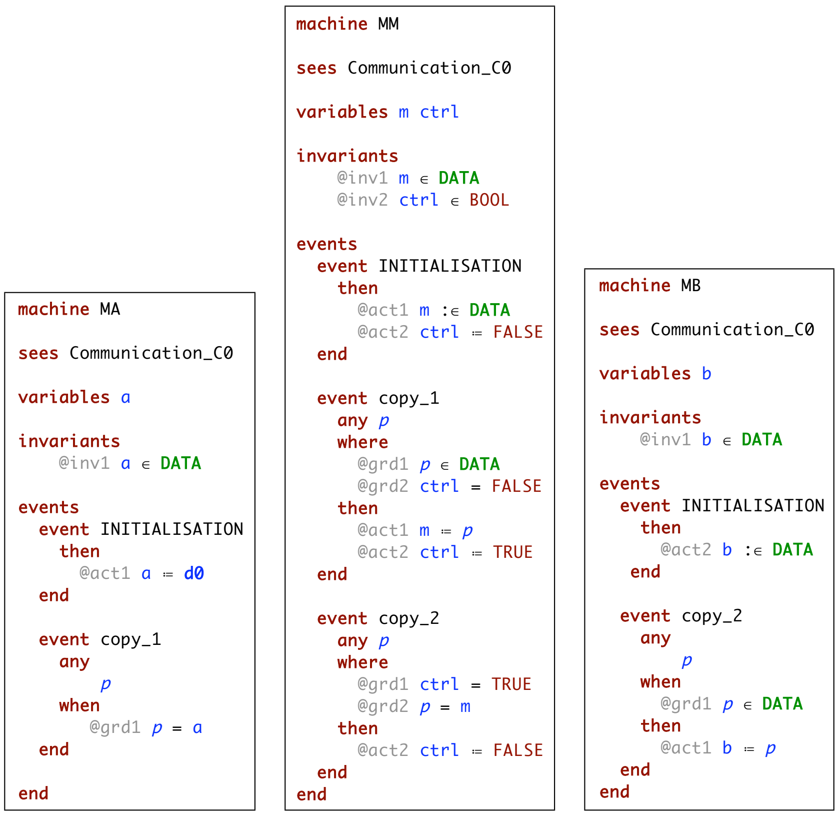


FIGURE 4.5: Decomposition of *Communication_M1* into machines *MA*,*MB* and *MM*



```
machine MA

sees Communication_C0

variables a

invariants
    @inv1 a ∈ DATA

events
  event INITIALISATION
    then
      @act1 a ≔ d0
  end

  event copy_1
    any
        p
    when
      @grd1 p = a
  end

end
```

(a) Machine *MA*

```
machine MM

sees Communication_C0

variables m ctrl

invariants
    @inv1 m ∈ DATA
    @inv2 ctrl ∈ BOOL

events
  event INITIALISATION
    then
      @act1 m :∈ DATA
      @act2 ctrl ≔ FALSE
  end

  event copy_1
    any p
    where
      @grd1 p ∈ DATA
      @grd2 ctrl = FALSE
    then
      @act1 m ≔ p
      @act2 ctrl ≔ TRUE
  end

  event copy_2
    any p
    where
      @grd1 ctrl = TRUE
      @grd2 p = m
    then
      @act2 ctrl ≔ FALSE
  end
end
```

(b) Machine *MM*

```
machine MB

sees Communication_C0

variables b

invariants
    @inv1 b ∈ DATA

events
  event INITIALISATION
    then
      @act2 b :∈ DATA
  end

  event copy_2
    any
        p
    when
      @grd1 p ∈ DATA
    then
      @act1 b ≔ p
  end
end
```

(c) Machine *MB*

FIGURE 4.6: Machines *MA*, *MM* and *MB*

variant and contexts. The initial partition of variables between the sub-components defines the rest of the decomposition as detailed below.

Invariants: The decomposition of the invariants depends on the scope of the variables. Therefore the minimal set of invariants must include the variable type definitions as illustrated by *inv1* and *inv2* in *Communication_M1* (Fig. 4.4) or *inv1* and *inv2* in *MM* (Fig. 4.6(b)). And these are the required invariants for a valid refinement. Additional ones depend on the user, as they may be useful in later refinements or to help in reusing the sub-components. An example of partition of invariants among the sub-components is *inv3* in *Communication_M1*:

$$ctrl = TRUE \Rightarrow m = a$$

This invariant contains three variables: *ctrl*, *m* and *a*. According to the defined decomposition, *ctrl* and *m* are variables of *MM* and *a* is a variable of *MA*. This suggests that *inv3* can be a constraint of the composition of the sub-components and not a constraint of the individual sub-components. As a result, invariant clause *inv3* in *Communication_M1* is not part of any of the sub-components. Alternatively when an invariant clause is demanded and uses variables placed outside the scope of a sub-component, a further refinement of the composed component might be required to make an explicit separation of the variables. If we consider again *inv3* and we would like to add this invariant to the sub-components, we would need to find a rewrite that invariant without including variables $ctrl$, m and a in the same predicate.

Events: The partition of the events depends on the partition of the variables. For instance, variables *m* and *ctrl* are part of *MM* so their initialisation is allocated to the same sub-component. Event *copy_1* in machine *Communication_M1* has a parameter *p*. When the decomposition occurs, that parameter is shared between the decomposed events. But the guards referring to that parameter are different in each decomposed event: in *MA* the guard is similar to *Communication_M1* ($p = a$); in *MM* only the type of the parameter p is defined ($p \in DATA$). The type of p is an implicit guard in the original event and during the decomposition, the type of p is made explicit:

$$p = a \Leftrightarrow p \in DATA \wedge p = a$$

The guards of a decomposed event inherits the guards on the composed event according to the variable partition. Variable a is not within the scope of machine *MM* so only the type of p is defined in the guard of *MM.copy_1*.

A different situation occurs for event *copy_2* in machine *Communication_M1*. Although the original event does not have parameters, the decomposed events have a new parameter *p*. Action *act1* in *Communication_M1.copy_2* refers to two variables (b and m) belonging to two different sub-components:

$$@act1\ b := m$$

This assignment needs to be rewritten in a way that these variables are not part of the same expression. A solution is to refine this event in a way that the guards and actions do not refer to variables allocated to different sub-components. Before the decomposition, we refine event *copy_2* by adding parameter p:

copy_2 $\widehat{=}$ **ANY** p **WHERE** $ctrl = \mathrm{TRUE} \wedge p = m$ **THEN** $b := p \parallel ctrl := \mathrm{FALSE}$ **END**.

Parameter p receives the value of variable m. Then the value of p is assigned to variable b. Whereas variable m is within the scope of *MM* only, the guard $p = m$ is added to *MM.copy_2* while *MB.copy_2* contains the guard $p \in DATA$ and the action $b := p$.

Non-shared parameters do not need to be maintained in the sub-components. Since parameters are local to events, only parameters explicitly used in guards or actions are included in the sub-events.

Variant: Variant is only necessary when new events are introduced in a refinement. Decomposed events in sub-components are inherit from the composed component so no new events are introduced meaning that variants are not required.

Contexts: The context *Communication_C0* used in the example is shared between all the machines. That context (and possible others) can be flattened into a single context and decomposed. The context decomposition results from the exclusion of elements (sets, constants, axioms) that are not used by the sub-component that sees that context. On the one hand, decomposing contexts can inadvertently remove relevant information. On the other hand, not decomposing it can add too many (not relevant and unnecessary) hypotheses which is not beneficial for the proofs: on the contrary, it might be harmful and complicate the discharge of proofs. Therefore, the context decomposition is optional as it varies with the system being modelled.

The events in the sub-components maintain the interface of the original events. By event interface we refer to the structure of the original event excluding elements referring to variables not in the scope of the sub-event.

4.2.1.1 Refinement of Sub-Components

An advantage of the decomposition is the possibility to further refine sub-components independently from the original component and other partitions. This advantage leads to the concept of team development over the same model by different modellers which it is an attractive option, in particular for the industry. In this section we introduce a database table as a refinement of the sub-component *MB* (resulting from the shared event decomposition of *Communication_M1*) to store the received values (*registries*). The database table contains three fields: *REGID*, *DATA* and *PRIORITY*. The new fields are introduced in the new context *MB_C0* as seen in Fig. 4.7. *REGID* is the identification field of all elements in the table of the database. It is defined as a constant and represented by a subset of natural numbers (axiom *axm1* in Fig. 4.7). The *PRIORITY* field corresponds to the priority that a registry is processed: *LOW*, *MEDIUM* or *HIGH*. The constants *id0* and *p0* initialise the database fields. New variables are introduced to represent the database registries: *idR* and *priority*. An auxiliary boolean variable

```
context MB_C0

constants REGID LOW MEDIUM HIGH id0 p0

sets PRIORITY

axioms
  @axm1 REGID ⊆ ℕ
  @axm2 partition(PRIORITY, {LOW}, {MEDIUM}, {HIGH})
  @axm3 id0 ⊆ REGID
  @axm4 p0 ∈ PRIORITY
end
```

FIGURE 4.7: Context *MB_C0* seen by refinement of *MB*

processQueue is used as a flag to enable the enqueueing of a registry in the database when a new value is copied to b as seen in Fig. 4.8.

```
machine MB_1 refines MB sees Communication_C0 MB_C0

variables b idR processQueue priority

invariants
  @inv1 idR ∈ REGID ⤀ DATA
  @inv2 processQueue ∈ BOOL
  @inv4 priority ∈ dom(idR) → PRIORITY

variant {processQueue,FALSE}

events
  event INITIALISATION
    then
      @act1 b :∈ DATA
      @act2 processQueue ≔ FALSE
      @act3 idR ≔ id0 × {d0}
      @act4 priority ≔ id0 × {p0}
  end
```

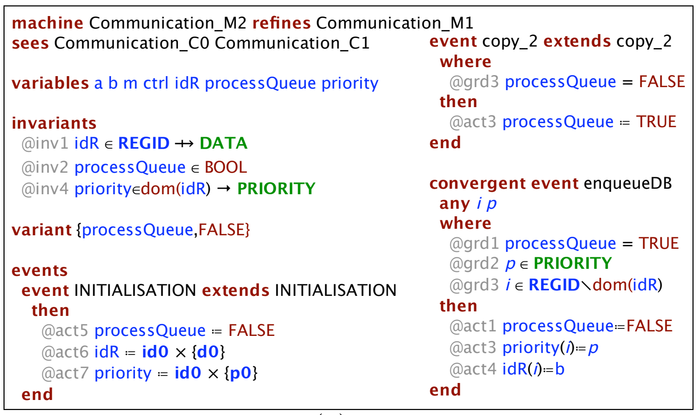
```
event copy_2 refines copy_2
  any p
  where
    @grd1 p ∈ DATA
    @grd2 processQueue = FALSE
  then
    @act1 b ≔ p
    @act2 processQueue ≔ TRUE
end

 convergent event enqueueDB
  any i p
  where
    @grd1 processQueue = TRUE
    @grd2 p ∈ PRIORITY
    @grd3 i ∈ REGID\dom(idR)
  then
    @act1 processQueue≔FALSE
    @act3 priority(i)≔p
    @act4 idR(i)≔b
end
```

FIGURE 4.8: Machine *MB_1* which is a refinement of *MB*

After event *copy_2* is executed, new event *enqueueDB* adds an element to the database. The added registry must have a fresh identification (not used before in the *idR* function) and the *priority* of the registry is defined non-deterministically in guard *grd2*. A variant is necessary for the new convergent event *enqueueDB* which is easily found by defining that eventually *processQueue* is *FALSE*. A possible refinement for the current model is to process the registries according to the priority. The priority field can also be defined more deterministically according to the message data. In a team development environment, the middleware could be refined while in parallel with other sub-components.

4.2.2 Shared Variable Decomposition of *Communication*

For the shared variable approach, we decide to do a further refinement. After copying the values, they are processed by being stored in a simple database similar to the one used in the shared event refinement described by context *Communication_C1* (equal to *MB_C0* in Fig. 4.7). A boolean variable *processQueue* and new event *enqueueDB* are also introduced as seen in Fig. 4.9. Concrete event *copy_2* *extends* the abstract *copy_2*, meaning that the concrete event is a copy of the abstract one plus additional concrete guards, actions, parameters.



```
machine Communication_M2 refines Communication_M1
sees Communication_C0 Communication_C1

variables a b m ctrl idR processQueue priority

invariants
 @inv1 idR ∈ REGID ⇸ DATA
 @inv2 processQueue ∈ BOOL
 @inv4 priority∈dom(idR) → PRIORITY

variant {processQueue,FALSE}

events
 event INITIALISATION extends INITIALISATION
  then
   @act5 processQueue ≔ FALSE
   @act6 idR ≔ id0 × {d0}
   @act7 priority ≔ id0 × {p0}
 end

event copy_2 extends copy_2
 where
  @grd3 processQueue = FALSE
 then
  @act3 processQueue ≔ TRUE
end

convergent event enqueueDB
 any i p
 where
  @grd1 processQueue = TRUE
  @grd2 p ∈ PRIORITY
  @grd3 i ∈ REGID∖dom(idR)
 then
  @act1 processQueue≔FALSE
  @act3 priority(i)≔p
  @act4 idR(i)≔b
end
```

(a)

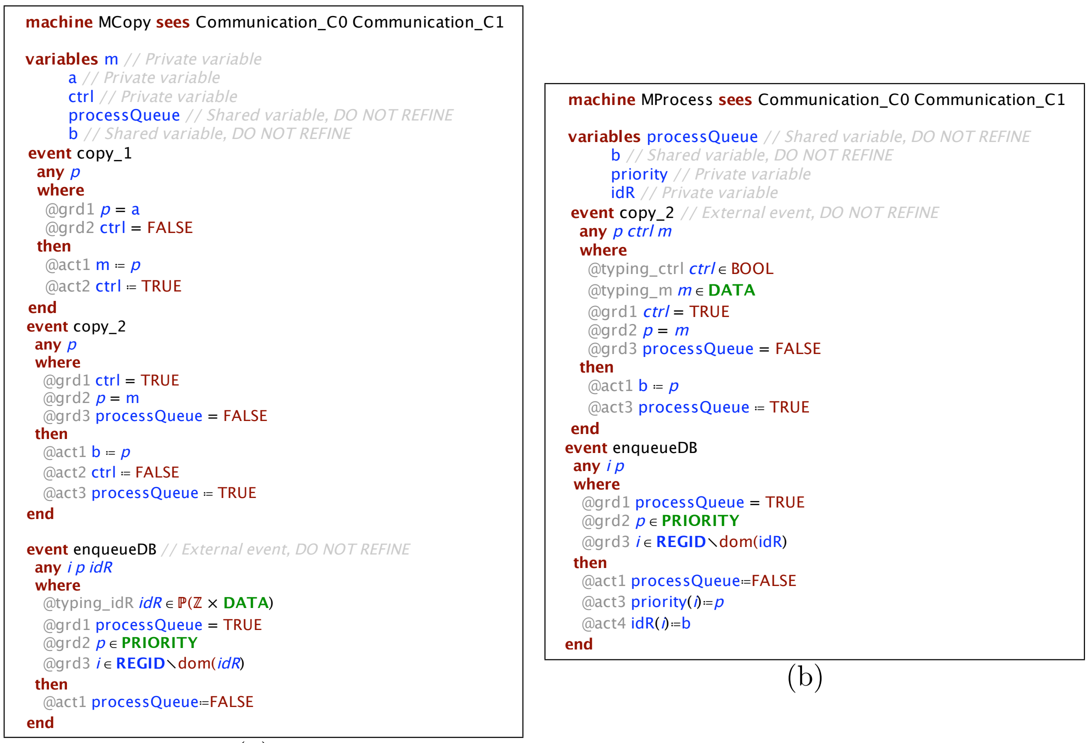
Figure 4.9: Excerpt of machine *Communication_M2*

Communication_M2 is shared variable decomposed by separating the copy of the values and the processing, described by machines *MCopy* and *MProcess* respectively. Events *copy_1* and *copy_2* are allocated to *MCopy* while event *enqueueDB* is allocated to *MProcess*. The variables separation depends on the event allocation leading to *private variables* (accessed by a single sub-component) or *shared variables* (accessed by multiple sub-components). The shared variables are used in events *copy_2* and *enqueueDB*: *processQueue* and *b*. All the other variables are private. The invariants splitting depends on the initial separation of variables, similar to the shared event approach.

The following step is to separate the *private events* and create the *external events*. Private events are allocated according to the user's choice. External events are based on the original events, preserving the shared variables and turning *private variables* into event *parameters*. If an original event depends on a shared variable, then an external event is created in the sub-components that use that variable. Events *copy_2* and *enqueueDB* use shared variables and consequently external events are required. An external event *copy_2* is created in *MProcess* using the shared variable *b*. The other variables used by the original *copy_2* become parameters in the external event as they are not in the scope of that sub-component (*ctrl* and *m*). Event *enqueueDB* is similarly built. The resulting machines can be seen in Fig. 4.10.

4.3 Definition and Validity of Decomposition

We want to formally prove that a machine M can be decomposed into machines $M1$ and $M2$. We shall prove through refinement POs that $M \sqsubseteq M1 \parallel M2$. The proofs are described in the following sections.



```
machine MCopy sees Communication_C0 Communication_C1

variables m // Private variable
       a // Private variable
       ctrl // Private variable
       processQueue // Shared variable, DO NOT REFINE
       b // Shared variable, DO NOT REFINE
event copy_1
 any p
 where
  @grd1 p = a
  @grd2 ctrl = FALSE
 then
  @act1 m ≔ p
  @act2 ctrl ≔ TRUE
end
event copy_2
 any p
 where
  @grd1 ctrl = TRUE
  @grd2 p = m
  @grd3 processQueue = FALSE
 then
  @act1 b ≔ p
  @act2 ctrl ≔ FALSE
  @act3 processQueue ≔ TRUE
end

event enqueueDB // External event, DO NOT REFINE
 any i p idR
 where
  @typing_idR idR ∈ ℙ(ℤ × DATA)
  @grd1 processQueue = TRUE
  @grd2 p ∈ PRIORITY
  @grd3 i ∈ REGID∖dom(idR)
 then
  @act1 processQueue≔FALSE
end
```

(a)

```
machine MProcess sees Communication_C0 Communication_C1

variables processQueue // Shared variable, DO NOT REFINE
       b // Shared variable, DO NOT REFINE
       priority // Private variable
       idR // Private variable
event copy_2 // External event, DO NOT REFINE
 any p ctrl m
 where
  @typing_ctrl ctrl ∈ BOOL
  @typing_m m ∈ DATA
  @grd1 ctrl = TRUE
  @grd2 p = m
  @grd3 processQueue = FALSE
 then
  @act1 b ≔ p
  @act3 processQueue ≔ TRUE
end
event enqueueDB
 any i p
 where
  @grd1 processQueue = TRUE
  @grd2 p ∈ PRIORITY
  @grd3 i ∈ REGID∖dom(idR)
 then
  @act1 processQueue≔FALSE
  @act3 priority(i)≔p
  @act4 idR(i)≔b
end
```

(b)

FIGURE 4.10: Excerpt of the output of shared variable decomposition of *Communication_M2*: *MCopy* and *MProcess*

4.3.1 Shared Event Style

Assume a machine M with two set of disjoint variables $v1, v2$. For the shared event decomposition, events can be categorised in 3 ways: $evt1(p1, v1)$, $evt2(p2, v2)$ and $evt3(p3, v1, v2)$. $evt1(p1, v1)$ is local to $M1$, $evt2(p2, v2)$ is local to $M2$ and $evt3(p3, v1, v2)$ is split into $evt3'(p3, v1)$ and $evt3''(p3, v2)$. The invariant of M is represented by $I(v1, v2)$ (for simplicity we exclude the use of context elements). Machine $M1$ is represented by variable $v1, w1$ and events $evt1(p1, v1)$ and $evt3'(p3, v1)$. The invariant of $M1$ is represented by $J1(v1, w1)$. Machine $M2$ is similar to $M1$ with variable $v2, w2$, events $evt2(p2, v2)$, $evt3''(p3, v2)$ and invariant $J2(v2, w2)$.

We want to prove that $M1$ and $M2$ when composed in parallel are a valid refinement of M. The refinement POs need to be verified for $M1 \parallel M2$ in order to ensure that is a concrete refinement of the abstraction M: $M \sqsubseteq M1 \parallel M2$. Events of the form $evt3$ have the following shape:

$$evt3 \mathrel{\widehat{=}} \textbf{ANY } p_3 \textbf{ WHERE } G_{31}(p_3, v_1) \wedge G_{32}(p_3, v_2) \textbf{ THEN } S_{31}(p_3, v_1, v_1') \parallel S_{32}(p_3, v_2, v_2') \textbf{ END}.$$

Definition 4.1. After the decomposition, event $evt3$ is decomposed into events $evt3'$

and $evt3''$ that are defined as:

$$evt3' \mathrel{\widehat{=}} \textbf{ANY } p_3 \textbf{ WHERE } G_{31}(p_3, v_1) \textbf{ THEN } S_{31}(p_3, v_1, v_1') \textbf{ END}$$
$$evt3'' \mathrel{\widehat{=}} \textbf{ANY } p_3 \textbf{ WHERE } G_{32}(p_3, v_2) \textbf{ THEN } S_{32}(p_3, v_2, v_2') \textbf{ END}.$$

in a way such that $evt3 \sqsubseteq evt3' || evt3''$.

Abstract machine $M(v_1, v_2)$ is decomposed into machines $M1(v_1)$ and $M2(v_2)$. When the events resulting from the shared event decomposition are composed, they are a valid refinement of the respective abstract event. Abstract event $evt3(p_3, v_1, v_2)$ is decomposed into events $evt3'(p_3, v_1)$ and $evt3''(p_3, v_2)$ as long as the set of variables v_1 and v_2 are disjoint. Moreover, the guards and actions of events $evt3'(p_3, v_1)$ and $evt3''(p_3, w_2)$ result from the original abstract event referring only to their respective set of variables. The set of parameters p_3 is the same for the three events. We consider that sub-components do not introduce additional invariants.

Theorem 4.1. If $M1$ and $M2$ are the resulting machines from shared event decomposition of M, then $M \sqsubseteq M1 \parallel M2$. In other words, $REF_{evti \sqsubseteq (evti' \parallel evti'')}$ holds, where $evti$ is an abstract event, $evti'$ is the resulting decomposed event in machine $M1$ and $evti''$ is the resulting decomposed event in machine $M2$.

From the refinement PO for machines (1.5), we need to prove that the resulting events after the decomposition refine the original abstract ones. Sub-components do not introduce additional invariants. Consequently $J_1(v_1, w_1) = J_2(v_2, w_2) = TRUE$. For events of form $evt1$ and $evt2$, the resulting events are exactly the same as the original ones. For events of form $evt3$:

$$\begin{aligned} REF_{evt3 \sqsubseteq (evt3' \parallel evt3'')}: \quad & I_1(v_1, v_2) \\ & \wedge J_1(v_1, w_1) \wedge J_2(v_2, w_2) \\ & \wedge H_{31}(p_3, v_1) \wedge H_{32}(p_3, v_2) \\ & \wedge T_{31}(p_3, v_1, v_1') \wedge T_{32}(p_3, v_2, v_2') \\ & \vdash \exists v_1', v_2' \cdot G_{31}(p_3, v_1) \wedge G_{32}(p_3, v_2) \\ & \wedge S_{31}(p_3, v_1, v_1') \wedge S_{32}(p_3, v_2, v_2') \\ & \wedge J_1(v_1', w_1') \wedge J_2(v_2', w_2'), \end{aligned} \tag{4.1}$$

$w1$ and $w2$ are concrete variables in the refinement.

Also assume:

$$\begin{aligned}
v_1 \cap v_2 = \varnothing & \\
J_1(v_1, w_1) = & \quad \text{TRUE} && (4.2)\\
J_2(v_2, w_2) = & \quad \text{TRUE} && (4.3)\\
H_{31}(p_3, v_1) = & \quad G_{31}(p_3, v_1) && (4.4)\\
H_{32}(p_3, v_2) = & \quad G_{32}(p_3, v_2) && (4.5)\\
T_{31}(p_3, v_1, v_1') = & \quad S_{31}(p_3, v_1, v_1') && (4.6)\\
T_{32}(p_3, v_2, v_2') = & \quad S_{32}(p_3, v_2, v_2') && (4.7)
\end{aligned}$$

Prove: $REF_{evt3 \sqsubseteq (evt3' \| evt3'')}$.

Proof. Assume the hypotheses of $REF_{evt1 \sqsubseteq (evt1' \| evt1'')}$:

$$\begin{aligned}
& I_1(v_1, v_2) \\
& J_1(v_1, w_1) \wedge J_2(v_2, w_2) \equiv \text{TRUE} && \{\text{From } (4.2) \text{ and } (4.3)\} && (4.8)\\
& H_{31}(p_3, v_1) \wedge H_{32}(p_3, v_2) \equiv G_{31}(p_3, v_1) \wedge G_{32}(p_3, v_2) && \{\text{From } (4.4) \text{ and } (4.5)\} && (4.9)\\
& T_{31}(p_3, v_1, v_1') \wedge T_{32}(p_3, v_2, v_2') \\
& \ \equiv S_{31}(p_3, v_1, v_1') \wedge S_{32}(p_3, v_2, v_2') && \{\text{From } (4.6) \text{ and } (4.7)\} && (4.10)
\end{aligned}$$

Prove:

$$\vdash \exists v_1', v_2' \cdot G_{31}(p_3, v_1) \wedge G_{32}(p_3, v_2) \wedge S_{31}(p_3, v_1, v_1') \wedge S_{32}(p_3, v_2, v_2') \wedge J_1(v_1', w_1') \wedge J_2(v_2', w_2').$$

The proof proceeds as follows:

$$\begin{aligned}
& \ \exists v_1', v_2' \cdot G_{31}(p_3, v_1) \wedge G_{32}(p_3, v_2) \\
& \ \ \wedge S_{31}(p_3, v_1, v_1') \wedge S_{32}(p_3, v_2, v_2') \\
& \ \ \wedge J_1(v_1', w_1') \wedge J_2(v_2', w_2') \\
\equiv & \ \exists v_1', v_2' \cdot G_{31}(p_3, v_1) \wedge G_{32}(p_3, v_2) \\
& \ \ \wedge S_{31}(p_3, v_1, v_1') \wedge S_{32}(p_3, v_2, v_2') \\
& \ \ \wedge \text{TRUE} && \{\text{From } (4.8)\} \\
\equiv & \ G_{31}(p_3, v_1) \wedge G_{32}(p_3, v_2) \\
& \ \ \wedge \exists v_1', v_2' \cdot S_{31}(p_3, v_1, v_1') \wedge S_{32}(p_3, v_2, v_2') && \{G_1 \text{ and } G_2 \text{ have free vars}\} \\
\equiv & \ \text{TRUE} \\
& \ \ \wedge \text{TRUE} && \{\text{From hypotheses } (4.9) \text{ and } (4.10)\}
\end{aligned}$$

□

$M1 \parallel M2$ is a valid refinement of M. In fact, they are syntactically the same apart from the invariants that can be lost during the decomposition: $M \neq M1 \parallel M2$ but $M \sqsubseteq M1 \parallel M2$.

4.3.2 Shared Variable Style

Abrial and Hallerstede [15] use refinement POs to prove that the shared variable decomposition is monotonic. In Fig. 4.11[1], machine M is decomposed into N and P (represented by the diagonal arrows) which are further refined by NR and PR respectively (vertical arrows). Afterwards, NR and PR are composed originating MR. To prove monotonicity, it is necessary to prove that MR is a valid refinement of M.

$$\begin{tikzcd}
\mathsf{N} & & \mathsf{P} \\
& \mathsf{M} \arrow[ul] \arrow[ur] & \\
\mathsf{NR} \arrow[uu] & & \mathsf{PR} \arrow[uu] \\
& \mathsf{MR} \arrow[ul] \arrow[ur] \arrow[uu, "?"'] &
\end{tikzcd}$$

FIGURE 4.11: Shared Variable Decomposition Diagram

Abrial [8] proves this property by means of state relations between the original machine and sub-components.

4.4 Limitations

The decomposition should have a final goal: a misleading decomposition may harm the development of a system instead of helping. For the shared variable decomposition the partition of events is always possible in the sense that it is always possible to generate sub-components. On the other hand, that decomposition might be less significant despite being possible: a further refinement may be more complex and not benefit the development. The point of decomposition (correct abstraction level) is important, since if it is done too early, the sub-component might be too abstract and will not be able to be refined (without knowing more about the other sub-systems). If the system is decomposed too late, it will not benefit from the approach anymore. For the shared event decomposition, the partition of variables is not always possible for all developments. An additional "preparation step" may be required to solve complex predicates (invariants, guards, axioms) or assignments (actions) involving variables allocated to different sub-components. This step can be achieved through refinement. Another limitation is that the overlapping of elements in the sub-components is not allowed which sometimes may be useful. Even in the shared variable approach, the overlapped (shared) elements cannot be further refined independently.

[1]Extracted from [15]

4.5 Conclusions

This chapter presents the decomposition of Event-B models and tool support in the Rodin platform. Decomposition can advantageously be used to decrease the complexity and increase the modularity of large systems, especially after several refinements. Main benefits are the distribution of POs over the sub-components which are expected to be easier to be discharged and the further refinement of independent sub-components in parallel introducing team development of a model. Our goal is to develop a robust tool to model distributed systems that can be used by academic institution and industrial companies.

The decomposition benefits has been exploited as seen in the literature: [30, 29] study the formal development of MAS (Multi-Agent Systems) which are complex distributed systems to be used for critical applications using abstraction and decomposition for classical B and Event-B. Lanoix [110] also studies MAS using shared variable decomposition to model a platoon of vehicles using Event-B. Butler [44] uses the shared event approach in classical B to decompose a railway system into three sub-components: Train, Track and Communication. The system is modelled and reasoned as a whole in an event-based approach, both the physical system and the desired control behaviour.Go and Shiratori [83] propose an automatic decomposition method using LOTOS [98]: the correctness is ensured if the combined behavior of decomposed sub-specifications is the same as the system's behavior before the decomposition. The method decomposes a process into two processes composed by the parallel operator and automatically generates an additional process that gives some information about the synchronization. The additional process corresponds to the middleware in a shared event decomposition in Event-B. Rezazadeh and Butler [148] use classical B to model a distributed monitoring and control system for vehicles entering and leaving a controlled area. After some refinements, the model is decomposed into asynchronous sub-systems. Rezazadeh [147] and Butler [149] introduce some guidelines for formal development of web-based applications (distributed systems that can be accessed using a Web browser) in B-method. That formal modelling considers only safety properties and a decomposition is suggested based on the CSP style message-passing channels. Iliasov [96] suggests a kind of decomposition based on modularization. The modules are introduced as a special case of shared variable decomposition by modelling sequential systems and Event-B is extended to model a system in the space domain. Separation logic [146, 144], an extension of the Hoare logic, supports reasoning about shared mutable data structures in a "bottom-up" approach where sub-components are put together and some composition properties can be proved. Such an approach is different from ours: we follow a "top-down" approach proving the global properties in the abstraction and decomposing only after proving the composition properties. Nevertheless Hoare and O'Hearn [93] combine the concurrent separation logic (CSL) and CSP aiming to reason about the communication between concurrent processes. In this work, trace semantics of parallel composition uses a composition operation on traces that partition channel ends between processes. The communication occurs via point to point channels with value passing messages not covering divergence nor refusals. The traces of parallel processes correspond to the separation conjunction of the processes traces: $traces(P \parallel Q) = traces(P) * traces(Q)$ where the alphabet of processes P and Q are disjoint. Comparing to our shared event decomposition also based in the CSP, value passing channels correspond to sub-components events that communicate via shared parameters.

There is a need for modularisation and reuse of sub-components in order to model large systems and manage better the respective POs. Event-B lacks a sub-component mechanism. Thus we propose to tackle that problem through the decomposition of a system by their events or variables. The shared variable (state-based) approach is suitable for designing parallel algorithms while the shared event (event-based) is suitable for message-passing distributed systems [45]. Following any of these two approaches, the parallel components of a distributed system can be refined and decomposed separately without making any assumptions about the rest of the system. The shared variable style relies on the work of Abrial and Hallerstede [15] where variables are shared and exists the notion of external events. Butler [45] suggests the shared event decomposition where events are partitioned through the sub-components and the interaction occurs via shared parameters. The work developed by Butler in [40] for action system is strongly related with the same approach for shared event decomposition in Event-B [45] as both approaches are state-based formalism combined with event-based CSP. The end-user chooses a decomposition style depending on specific systems and on its modelling preferences. The decomposition configuration is stored persistently for replaying/editing although further study is still required for this matter. We present an example of the different styles of decomposition. A tool was developed to model distributed systems in the Rodin platform that can be used by the industry (cf. Sect. 5.5). A visualisation view for decomposition seems intuitive and we intend to explore it using GMF [82].

Chapter 5

Tool Support

The adoption of a technology or even a theory in detriment of another can rely on the tool support [89]. The efficiency of the tool, how practical it is, the range of problems it can solve and user support are some of the important points when developing a tool. Formal methods are not different: tool support is important to add automation, efficiency and ease the task of developing formal models. Mathematical rigour enables modellers to analyse and verify models at any part of the program lifecycle: requirements engineering, specification, architecture, design, implementation, testing, maintenance, and evolution [188]. To a long time, formal methods has been primarily restricted to various research organisations. However, it is becoming apparent that formal methods is in the transition process from academic research to industrial application. Formal methods tools are also in the process of transition from academic toys to industrial-strength tools [58]. In this document we address this topic by giving tool support to the previous described techniques: composition, decomposition and generic instantiation.

5.1 Introduction

In the previous chapters, three techniques that help the modelling of complex systems were described. The semantics behind each one of the techniques and their advantages/disadvantages were explained with the usage of small examples. Nevertheless the broad usage of such techniques requires tool support to allow automation, to ease the user's effort of applying the techniques and to be more efficient. Moreover, the tool implementation ofter unveils constraints that are not taken into account while studying the techniques such as scaling, optimisation or miscellaneous other issues. A user friendly tool can be a powerful support to a defined theory and often is the reason why the theory may be adopted or not. Therefore we strive to have suitable tool support not only to more easily test the strength of the technique but also to allow others to quickly use it.

As aforementioned, the Rodin platform [151] is the result of an EU research project. It is a software toolset, based on modern software programming tools developed to use Event-B notation. It is open source, based on Eclipse Platform [66] and it works has a complement for rigorous modelling developments [49]. The aim is to benefit industry by permitting the integration of any necessary functionality in the same tool. Rodin contains a Static Checker that analyses Event-B components for syntactical errors (well-formedness and typing of models), a Proof Obligation Generator for generating PO and these obligations can be discharged by a theorem prover. An important Rodin feature is the high level of extensibility reflected by, for instance, the ability to contribute plug-ins. Plug-ins are components providing a certain type of service within the context of the Eclipse workbench. By components here we mean objects that may be configured into a system at system deployment time [66], such as the default theorem prover (B4free [21]) or model checking systems (ProB [141]). Three tools (plug-ins) resulted from the study of the three previous techniques: shared event composition plug-in, refactory plug-in and decomposition plug-in. These were developed for the Rodin platform although the methodology behind them could be implemented in other platforms and even for other formalisms.

This chapter is organised as follows: we described the tool support for shared event composition in Section 5.2. After the generic instantiation (Section 5.3) and refactory (Section 5.4) are outlined. Section 5.5 illustrates the decomposition tool before the conclusions in Section 5.6.

5.2 Shared Event Composition Plug-in

A plug-in for composed machines was developed to support the shared event composition. We extend the Rodin static checker to validate composed machines based on checks defined in Sect. 2.3.2. POs ought to be automatically generated over the composed machines. Currently this feature is not available but we will address this issue in the future. The current solution to address POs is to generate a standard machine from the composed machine. In Fig. 5.1(a), composed machine $cM2$ includes machines $M3$ and $M4$. $cM2$ is then "expanded" as a standard machine $M2'$ which itself refines abstract machine $M1$. The composition POs (including refinement) are generated in $M2'$. Generating a new machine allows the further development of the composed model. Moreover the inspection of the composed events is beneficial based on the experience of using the tool. In the future, we would like to still have the option to generate a new machine, but the POs should be discharged at the composed machine as depicted in Fig. 5.1(b).

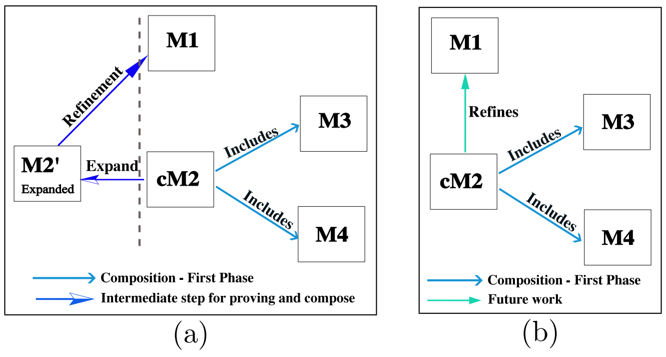
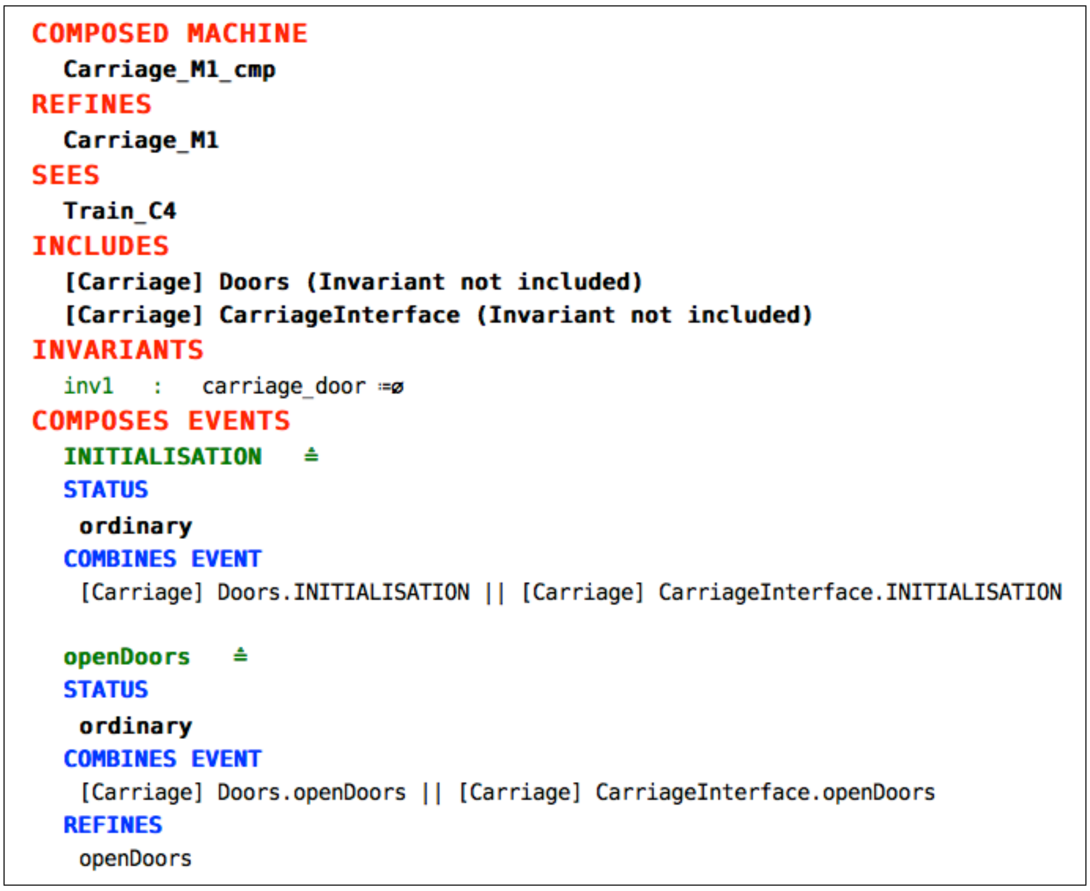


FIGURE 5.1: Composition structure: current(a) and future(b)

5.2.1 Composed Machines

The tool implementation follows the structure described in Sect. 2.3.1 as seen in Fig. 5.2. A new constructor is added to the Event-B syntax. This constructor, *composed machine*, allows standard machines to be *included*: they are structured and saved in a single file. The interaction between standard machines occur by defining which events are composed in parallel. Moreover additional invariants, can be added to composed machine. This is the only way to relate the state space of the included machines, since the machines remain independent of each other. Composed machines can refine standard machines. Consequently the abstract events must be refined by concrete ones to comply with the refinement proof obligations.

In Fig. 5.2, composed machine *Carriage_M1_cmp* (extracted from the case study in Chapter 6) refines machine *Carriage_M1* and sees context *Train_C4*. Moreover two machines are included: *Doors* and *CarriageInterface*. In other words, we want to express that $\mathit{Carriage\_M1} \sqsubseteq \mathit{Carriage\_M1\_cmp} \wedge \mathit{Carriage\_M1\_cmp} \equiv (\mathit{Doors} \parallel \mathit{CarriageInterface})$. Moreover invariant *inv1* is an additional invariant for this composed machine. Therefore $I_{CM}(v\_Doors, v\_CarriageInterface) = I(v\_Doors) \wedge I(v\_CarriageInterface) \wedge carriage\_door = \varnothing$, where $v\_Doors, v\_CarriageInterface$ are the variables of machine $Doors$ and $\_CarriageInterface$ respectively. The label in the front (*Invariant not included*) means that the invariant clauses of the individual machines will not be included if this composed machine is "expanded" as explained in Sect. 5.2. The interaction is achieved with the composition of events: the initialisation events are composed in parallel; also the composed event $openDoors$ result from the composition of event $Doors.openDoors$ and $CarriageInterface.openDoors$. This composed event refines the abstract event $Carriage\_M1.openDoors$. More composed events can be added in a similar fashion. As future work, proof obligations should be generated directly for composed machines.



```
COMPOSED MACHINE
  Carriage_M1_cmp
REFINES
  Carriage_M1
SEES
  Train_C4
INCLUDES
  [Carriage] Doors (Invariant not included)
  [Carriage] CarriageInterface (Invariant not included)
INVARIANTS
  inv1   :   carriage_door ≔ø
COMPOSES EVENTS
  INITIALISATION   ≙
  STATUS
   ordinary
  COMBINES EVENT
   [Carriage] Doors.INITIALISATION || [Carriage] CarriageInterface.INITIALISATION

  openDoors   ≙
  STATUS
   ordinary
  COMBINES EVENT
   [Carriage] Doors.openDoors || [Carriage] CarriageInterface.openDoors
  REFINES
   openDoors
```

FIGURE 5.2: Pretty print of the composed machine tool

5.3 Generic Instantiation Plug-in

Generic instantiation is a technique to help the development of models. In particular if a development matches or fits an existing pattern, that pattern (and possibly its refinement chain) can be instantiated and reused. This is particular interesting when we are targeting multiple instances of a generic pattern because we can benefit from the existing proofs of the pattern (that are expected to be discharged) and customise the instance according to a particular purpose.

In Sects. 3.3.3 and 3.5.3, we suggest a methodology for the implementation of the generic instantiation of Event-B machines through *instantiated machine* (Fig. 5.3) and *instantiated refinement* 5.4 files. The instantiation is a result of the mandatory replacement of pattern's sets and constants and the optional renaming of variables, parameters and events. The user needs to supply an instance context to be used for the replacement of sets and constants (context D in Fig. 5.3). Furthermore if the instance corresponds to a refinement of an existing development, the abstract machine (M_0) needs to be specified. Consequently, refinement proof obligations are generated and need to be discharged by the user.

For instantiated refinements, a pattern refinement chain is instantiated. We define as the

```
INSTANTIATED MACHINE IM
INSTANTIATES M VIA Ctx
REFINES $M_0$      /* abstract machine */
SEES D                /* context containing the instance properties */
REPLACE              /* replace parameters defined in context C */
    SETS $S_1 := DS_1, \ldots, S_m := DS_m$ /* Carrier Sets */
    CONSTANTS $C_1 := DC_1, \ldots, C_n := DC_n$ /* Constants */
RENAME              /*rename elements in machine M */
    VARIABLES $v_1 := nv_1, \ldots, v_q := nv_q$      /* optional */
    EVENTS $ev_1 := nev_1$        /* optional */
             $p_1 := np_1, \ldots, p_s := np_s$      /*  parameters:  optional */
             :
             $ev_r := nev_r$
END
```

Figure 5.3: An Instantiated Machine

```
INSTANTIATED REFINEMENT IR
INSTANTIATES $M_t$ VIA $Ctx_t$
REFINES $IR_0$       /* abstract machine */
SEES $D_w$            /* context containing the instance properties */
REPLACE             /* replace parameters defined in context C */
    SETS $S_1 := DS_1, \ldots, S_m := DS_m$ /* Carrier Sets */
    CONSTANTS $C_1 := DC_1, \ldots, C_n := DC_n$ /* Constants */
RENAME    /*rename variables, events and params in $M_1$ to $M_t$ */
    VARIABLES $v_1 := nv_1, \ldots, v_q := nv_q$
    EVENTS $ev_1 := nev_1$      /* optional */
             $p_1 := np_1, \ldots, p_s := np_s$      /*  parameters :optional */
               ...
             $ev_r := nev_r$
END
```

Figure 5.4: An Instantiated Refinement

starting point of the instantiation the most abstract machine of the refinement chain (in the future, the abstract machine selection might be more flexible). Besides the instance context (D_w in Fig. 5.4), the mandatory replacement of sets and constants and the definition of the abstract machine (IR_0), the modeller is given the choice to explicitly define the last refinement machine to be refined (machine M_t).

The output of the instantiation is a new Event-B machine/refinement chain similar to the pattern apart from the differences originated by the renaming and replacements according to instantiated machine/refinement. Moreover to reuse the pattern proofs, pattern axioms must be preserved in the instance and therefore theorems (refactored from the pattern axioms) are automatically generated in the instances.

Although the structure of instantiated machines and instantiated refinements are defined, we were not able to develop the tool support for instantiation due to time constraints. We also intend to build a library of patterns that could easily be instantiated. This library should be categorised according to the formal modelling pattern as suggested by Stepney [178]. Nevertheless the need to rename Event-B model elements, in particular when the renaming involved a refinement chain, was strong enough for tool support to be developed. That tool is intended to be used as part of generic instantiation tool support. The renaming refactory framework is described in more detail in the next section.

5.4 Renaming Refactory Framework

The instantiation implies the renaming/replacing of some properties in the pattern. A renaming supporting tool is required in a tool implementation of generic instantiation. Moreover one of the most recurring requirements from users of the Rodin platform is to have simple means for renaming modelling elements. Users want to have a unique operation that will rename an element both at its declaration and all its occurrences. A renaming operation entails that the renaming of an element does not modify its existing proof state (no loss of proof) [160]. These requirements fall in the more general context of refactoring. In software engineering, "refactoring source code" means improving the source code without changing its overall results, and is sometimes informally referred to as "cleaning it up". In the case of the Rodin platform, the refactoring framework is not intended to change the overall behaviour of the files/elements nor losing proofs. Note that this tool is also useful for the shared event composition (Sect. 5.2) where the occurrence of variables with the same name results in the renaming of at least one of them (shared variables are not allowed). This section describes the developed work for the renaming/refactory framework, giving an overview of the architecture and how the framework works. Initially the renaming framework was designed and developed by Stefan Hallerstede and Sonja Holl [94].

The basic requirement for the renaming framework is the ability to rename Event-B elements. Moreover renamings involving machine refinements or context extensions should propagate through all the occurrences of the elements even in different files keeping the consistency of the model. "Renaming" simply renames the free identifiers and by checking possible renaming clashes we ensure that we are not changing the meaning of the model (apart from the change of names or labels). As a consequence, the overall proof obligations state should not change after the renaming.

Figure 5.5[1] shows the renaming framework architecture. It is considered a framework because it is designed in a generic way allowing the incorporation of other languages (i.e. not restricted to Event-B).

The renaming framework is an Eclipse plug-in [66, 36]. The renaming operation starts at the RefactoryManager which loads the refactoring tree in the extension points. The refactoring tree corresponds to the structure of the language to be used: that structure is used to navigate and find occurrences in the files. Afterwards Operation Scheduler retrieves the related files, the symbols (possible name clashes) and the individual renaming operations (renaming operation is different for each element) to be applied. The *run()* method checks for possible clashes, returns a clash report and requests a confirmation before executing the renaming. Upon confirmation, the renaming is propagated in a "top-down" style (from the abstract to the concrete level) throughout the model and

[1]Extracted from [160] and designed by Stefan Hallerstede and Sonja Holl

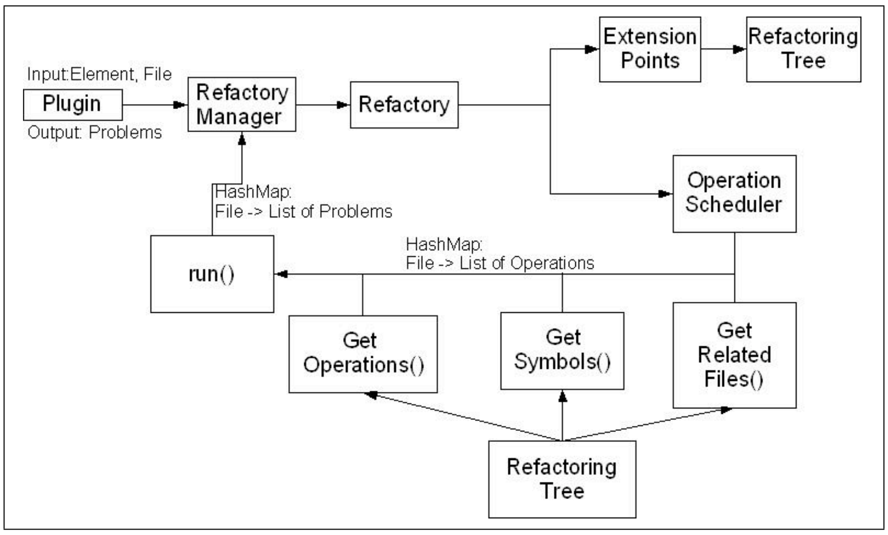


FIGURE 5.5: Renaming/Refactory Architecture

related files. The possible clashes are overestimated: if the files are somehow related (for instance, two machines share the same context but are not a refinement of each other) a clash can be reported. Currently the refactory plug-in also uses the Rodin Indexer plug-in [184] to accelerate the search of elements and find clashes.

The renaming can be applied to the following Event-B elements:

- variables
- carrier sets
- constants
- event parameters
- labelled elements like events, invariants, actions, guards, axioms
- machines
- contexts

Figure 5.6[2] represents the refactory trees when an invariant label is renamed.

The renaming operation creates a list of related files and proofs to be renamed according to the refactory tree in the extension points.

[2]Extracted from [160] and designed by Hallerstede and Sonja Holl

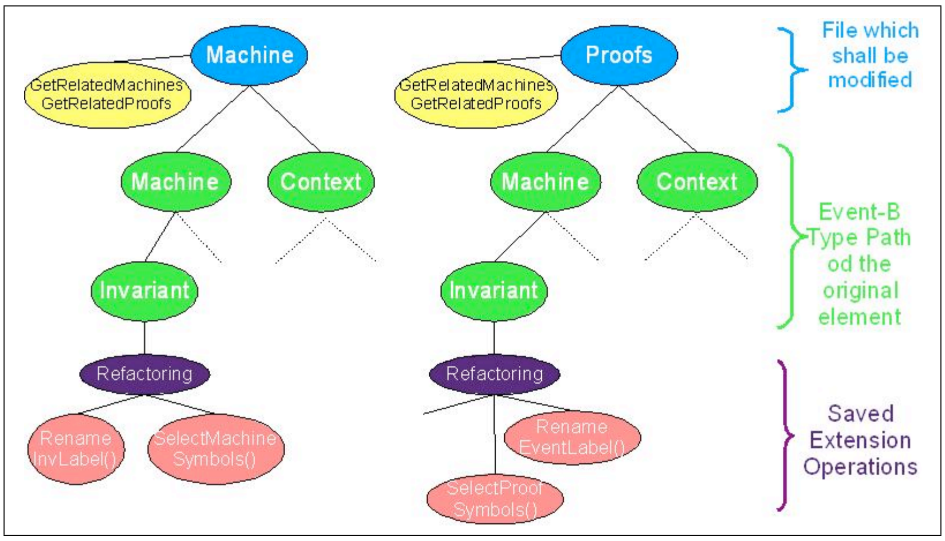


FIGURE 5.6: Refactoring Trees after processing the extension points

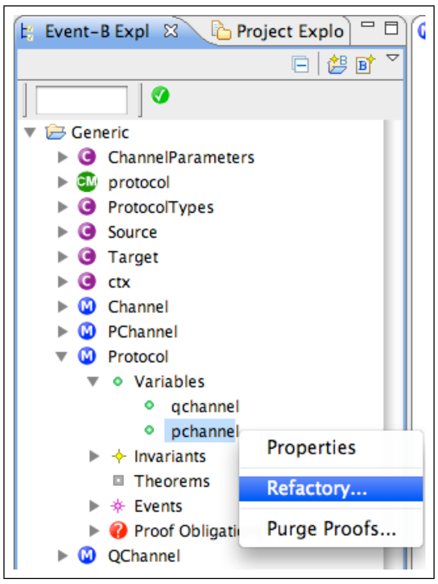
5.4.1 User interface

This section briefly describes how the renaming plug-in is used. After installing this plug-in (available under the main Rodin Update site `http://rodin-b-sharp.sourceforge.net/updates`) in the Event-B explorer perspective, the user selects the element to rename as seen in Fig. 5.7(a). After the introduction of the new name (Fig. 5.7(b)), a list of related files is created and the possible clashes are reported as seen in Fig. 5.7(c). Thereafter the user decides to proceed or not by confirming the renaming execution.

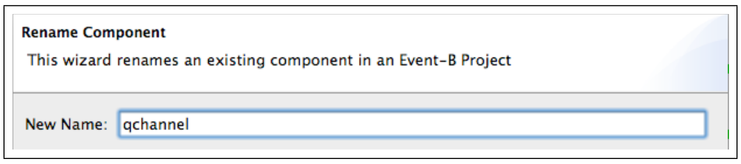
5.4.2 Renaming Proof Obligations

One of the initial requirements for the renaming plug-in is the renaming of proofs. The current version (v1.1.0) supports the renaming of proofs including the renaming of carrier sets (that was not possible in previous versions). In the Rodin platform, proof obligations for a model are divided in three different files: Proof Obligation file (bpo), Proof file (bpr) and Proof Status file (bps). The proof obligation (bpo) contains the proof obligations generated by the Proof Generator (Sect. 1.5.7) for a model. The proof file (bpr) contains the proof tree for each proof obligation including generated hypotheses to be discharged, applied proof rules (device used to construct proofs of sequents) and the elements (variables, carrier sets, constants, etc) that are part of the proof. Finally the proof status file (bps) contains the state of the proof obligation: not proved or discharged. Any change in the model regenerates a new bpo and bps file. The bpr files are heavier (proof tree needs to be reconstructed and loaded to memory) so the proof trees are reused whenever possible. For the renaming of proof obligation, three possible solutions arised:

- Adding new hypotheses: after the renaming execution terminates, the proof obli-



(a) Refactory menu

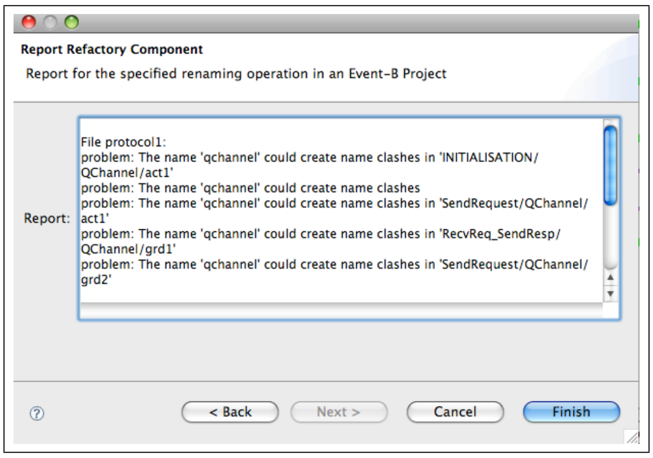


Rename Component

This wizard renames an existing component in an Event-B Project

New Name: qchannel

(b) New name wizard



Report Refactory Component

Report for the specified renaming operation in an Event-B Project

Report:

```
File protocol1:
problem: The name 'qchannel' could create name clashes in 'INITIALISATION/
QChannel/act1'
problem: The name 'qchannel' could create name clashes
problem: The name 'qchannel' could create name clashes in 'SendRequest/QChannel/
act1'
problem: The name 'qchannel' could create name clashes in 'RecvReq_SendResp/
QChannel/grd1'
problem: The name 'qchannel' could create name clashes in 'SendRequest/QChannel/
grd2'
```

(c) Report wizard

FIGURE 5.7: Refactory User Interface

gation files are updated by adding new hypotheses (something like $old\_name = new\_name$) to the proof trees (in bpr files). This approach has its advantages (fast, since proofs do not have to be replayed) and disadvantages (it is not really refactoring since you can still see the old variables in the refactored proof). Also it does not work for carrier sets (in Event-B, two carrier sets are always distinct).

- Renaming the proof obligation: the occurrences of the names to be changed (old_name) in the proof obligations are renamed to the new name (new_name). There are two possible implementations of this option:

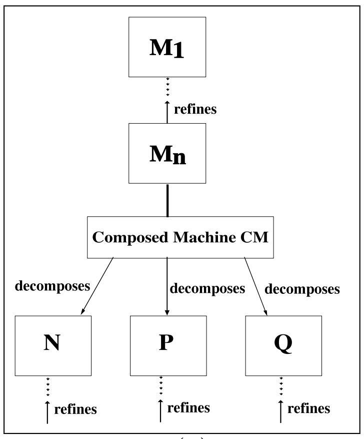
- Renaming proof tree: The occurrences of the *old_name* are replaced with *new_name* at the level of the proof tree. The reasoners (generates proof rules) for those proofs are rerun returning a new (renamed) version of the original proof rule. The reasoners input elements in the style *old_name* $\rightarrow$ *new_name*, which will be used to re-compute the reasoners recursively when replaying the proof tree [123].
- Renaming proof files: just as machine and context files, the structure of proof files is added to the plug-in extension-points (corresponding to bpr). The result is renaming the occurrences directly over bpr whenever necessary. The disadvantage of using this approach is the dependence on the proof file structure: changes in the proof file structure would require the change of the plug-in which is cumbersome.

We opt to use the second solution where proof trees are renamed. The leaves of the trees need to be explored to find and rename all the occurrences but in the end, the proof trees completely reflect the renamed element. The disadvantage is that the operation can take some time to finish when there are many proof trees with several long leaves (complex proofs). We intend to work on an optimisation of such renaming in the future.

5.5 Decomposition tool

Using the extensibility of the Rodin platform, a plug-in was developed for the semi-automatic decomposition of models. The tool allows shared event and shared variable decomposition. This work was developed in collaboration with Thai Son Hoang and Carine Pascal. With Michael Butler, we agreed that the correct methodology to support decomposition should be different for each style: for *shared variable decomposition*, the *events* to be allocated to each sub-component should be selected by the user; for the *shared event decomposition*, *variables* instead are selected by the user; as much as possible, the rest of the decomposition process should not require the user's input. Hoang started the development of the tool by creating the interface corresponding to the decomposition for the shared variable decomposition. Pascal continued that development by introducing the required validations, creation of external external and shared variables. Our contribution was the development of the shared event approach in terms of validations, splitting events, validating predicates and generating the sub-components. Moreover, we developed the persistency file where the decomposition configuration can be saved and re-run as many times as desired.

The decomposition originates sub-components according to the decomposition configuration (allocation of variables). That configuration is stored persistently in a composed machine (cf. Chapter 2 and [162]) for possible future reuse or editing as seen in Fig. 5.8.



(a)

COMPOSED MACHINE CM
REFINES M_n
INCLUDES
 N
 P
 Q
EVENTS
 evt_1 **REFINES** M_n.evt_1
  Combines Events N.evt_1 || P.evt_1
 ...
 evt_n **REFINES** M_n.evt_n
  Combines Events P.evt_n || Q.evt_n
END

(b)

FIGURE 5.8: Decomposition tool diagram for a machine M_n and composed machine CM

The input for the decomposition is a machine of a given Rodin project selected by the end-user. After the selection of decomposition configuration, the tool generates the sub-components automatically. The steps to be followed in order to decompose are (we decompose machine M_n in Fig. 5.8(a)):

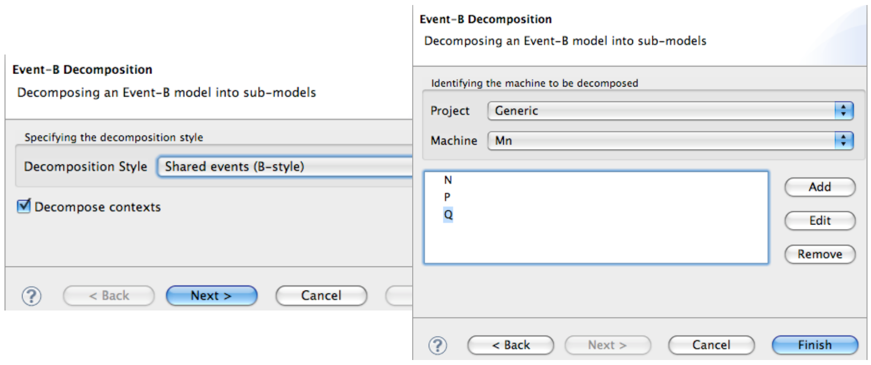
1. End-user selects a machine M_n to decompose.
2. End-user defines sub-components to be generated: N, P, Q ....
3. End-user selects the decomposition style to use:

   Shared Variable: end-user selects the events to be allocated to sub-components. The tool automatically decomposes the rest of the model according to the event partition (shared/private variables, external events).

   Shared Event: the end-user selects the variables to be allocated for each sub-component. The rest is done automatically.

4. The end-user can opt to decompose the seen contexts into the sub-components similarly to the machine decomposition.
5. Sub-components are fulfilled according to the decomposition configuration. Invariants depending on variables allocated to a single sub-component (private variables) are automatically added.
6. The decomposition configuration is stored as a composed machine.
7. Sub-components N, P, Q ... can be further refined.

For the shared event decomposition, a validation might be required to ensure that the selected machine (Machine M_n) does not have complex predicates or assignments in-


Event-B Decomposition
Decomposing an Event-B model into sub-models

Specifying the decomposition style
Decomposition Style: Shared events (B-style)
☑ Decompose contexts

Event-B Decomposition
Decomposing an Event-B model into sub-models

Identifying the machine to be decomposed
Project: Generic
Machine: Mn

N
P
Q

FIGURE 5.9: Graphical User Interface for the Decompositon tool

volving variables of different sub-components. That would be a hint that a further requirement is required in the model before the decomposition.

The decomposition configuration is performed through a wizard using the Rodin's Graphical User Interface as depicted in Fig. 5.9. The decomposition configuration is stored persistently for replaying/editing although further study is still required for this matter. A visualisation view for decomposition seems intuitive and we intend to explore it using GMF [82].

5.6 Conclusions

The progression and the maturity of formal methods shifted the way they are applied nowadays. In the 1980s the application of the Z notation to the IBM CICS transaction processing system was recognised as a major (award-winning) technical achievement, but it is significant that it used only very simple tools: syntax and type-checkers. In the 1990s, the Mondex project was largely a paper-and-pencil exercise, but it still achieved the highest level of certification [188]. Modelling and proving manually by hand it is still possible but slowly the need for less error-prone methodologies, in particular for repetitive tasks, requires the use of tools. Developing tools to support the formal methods process has been an activity that started with the first developments of formal methods technology. Both the underlying formal methods technology and formal methods tools have evolved substantially over the past four decades [58]. Today many people feel that it would be inconceivable not to use some kind of verification tool [188]. Consequently the tool development in formal methods and best practices to reach it are currently subject of study within our community [107].

In this document, we address this topic by envisaging tool support to the previous described techniques: composition, decomposition and generic instantiation. In all techniques we have suggested methodologies for the implementation of tools. Nevertheless due to time constraints, only composition and decomposition have more elaborated tool support. As a result of our study, we addressed the need of other tools (such as refactoring of Event-B elements throughout refinement chain while maintaining the validity of discharged proofs). Our goal was to develop prototype tools that with experience and application to more complex case studies, could become more mature and robust. Consequently further challenges regarding formal modelling to be found and tackled. And even as prototypes, the developed plug-ins have been used already to model different systems such as flash systems [62, 60], a spacecraft system [73] or cruise control system [190], among others with success. With the received feedback, the tools undertook several changes resulting in performance improvements, becoming more user friendly and sometimes having additional features. There are still plenty of tool challenges to be explored and developed as described in Chapter 7.

Chapter 6

Case Study

A case study involving the specification and refinement of an Event-B model is presented. This chapter describes how the techniques presented in the previous chapters may be used in practice. Throughout the case study, some design rules for Event-B are presented. These rules are specialisations of Event-B techniques already presented. These rules were suggested by the needs of the case study, but are general enough to be useful in other cases.

6.1 Introduction

Case studies can be described as a process or record of research in which detailed consideration is given to the development of a particular matter over a period of time. They have two main purposes: the explanation and description of the application of a particular technique (illustration purposes) and to validate the usefulness of the technique in a variety of systems (validation purpose). The described case study fulfils the first purpose: modelling a complex system from an abstraction to a more concrete model. Consequently the number of events, variables and proof obligations increase in a way that the model starts becoming hard to manage. Therefore a suitable solution at this stage is to use our decomposition technique. This procedure is repeatedly applied to the rest of the refinements. The application of decomposition in simple, abstract cases has very little or no real advantage. As aforementioned in Section. 4.4, the point of decomposition (correct abstraction level) is important, since if it is done too early, the sub-component might be too abstract and will not be able to be refined (without knowing more about the other sub-systems); if the system is decomposed too late, it will not benefit from the approach anymore. Therefore the application of decomposition only occurs after several refinements as expected.

The second purpose of case studies is usually achieved through the development of different models that represent different kind of systems. Their application allows the assessment of techniques, their suitability, advantages and disadvantages when applied in different manners. Besides the case study in this chapter, the presented techniques have already been used for different systems:

- Flash System Development [62, 60]: use of shared event composition and decomposition.
- Decomposition of a Spacecraft System [73]: use of shared event decomposition.
- Development of a Cruise Control System [190]: use of shared event composition and decomposition.
- Development of a Pipeline System [56, 12]: use of shared event composition and decomposition.
- Development of Parallel Programs [90]: use of shared variable decomposition over shared data accessed by different components.
- Development of a Multi-directional Communication Channel [163]: use of generic instantiation.

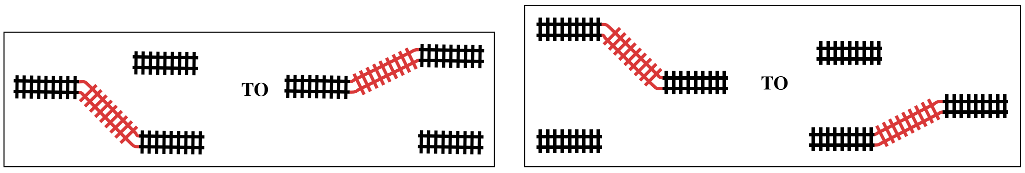
Here, a safety-critical metro system case study is developed. This version is a simplified version of a real system but tackles points where there the model becomes complex and where the presented techniques are suitable: stepwise incrementation of the complexity of the system being modelled, sub-components communication, stepwise addition of requirements at each refinement level, refinement of decomposed sub-components. We develop a metro system model introducing several details including notion of tracks, switches, several safety measures and doors functionality among others. If the presented techniques were not used, the metro system model would be extremely complex and hard to manage after the inclusion of all the requirements due to the high number of variables, events, properties to be added and proof obligations to be discharged. Decomposition and generic instantiation alleviate that issue by introducing modularity and reusing existing sub-components allowing further manageable refinements to be reached.

The metro doors requirements are based on real requirements. The case study is developed in the Rodin platform using the developed tools whenever possible. We use the shared event composition/decomposition and generic instantiation. The metro system can be seen as a distributed system. Nevertheless the modelling style suggested can be applied to a more general use.

6.2 Overview of the safety-critical metro system

The safety-critical metro system case study describes a formal approach for the development of embedded controllers for a metro system[1]. Butler [44] makes a description of embedded controllers for a railway using classical B. The railway system is based on the french train system and it was subject of study as part of the european project MATISSE [121]. Our starting point is based on that work but applied to a metro system. That work goes as far as our first decomposition originating three sub-components. We augment that work by refining each sub-component, introducing further details and more requirements to the model. Moreover in the end we instantiate emergency and service doors for the metro system.

The metro system is characterised by trains, tracks circuits (also called sections or CDV: *Circuit De Voie*, in French) and a communication entity 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 a hazard situation, trains receive a notification to brake. The track is responsible for controlling the sections, changing switch directions (switch is a special track that can be divergent or convergent as seen in Fig. 6.1) and sending signalling messages to the trains.



(a) Divergent Switch (b) Convergent Switch

FIGURE 6.1: Different types of Switches: divergent and convergent

Figure 6.2[2] shows a schematic representation of the metro system decomposed into three sub-components. Initially the metro system is modelled as a whole. Global properties are introduced and proved to be preserved throughout refinement steps. The abstract model is refined in three levels (*MetroSystem_M0* to *MetroSystem_M3*) before we apply the first decomposition. We follow a general top-down guideline to apply decomposition:

Stage 1 : Model system abstractly, expressing all the relevant global system properties.

Stage 2 : Refine the abstract model to fit the decomposition (preparation step).

Stage 3 : Apply decomposition.

Stage 4 : Develop independently the decomposed parts.

[1]A version of this model is available online at http://eprints.ecs.soton.ac.uk/23135/
[2]Image extracted from [44]

For instance, **Stage 1** is expressed by refinements *MetroSystem_M0* to *MetroSystem_M3*. *MetroSystem_M3* is also used as the preparation step before the decomposition corresponding to **Stage 2**. The model is decomposed into three parts: *Track*, *Train* and *Middleware* as described in **Stage 3**. This step allows further refinements of the individual sub-components corresponding to **Stage 4**. The following decompositions follow a similar pattern.

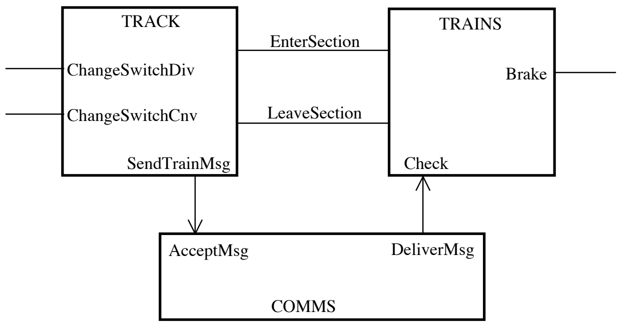


FIGURE 6.2: Components of metro system

An overview of the entire development can be seen in Fig. 6.3. After the first decomposition, sub-components can be further refined. Train global properties are introduced in *Train* leading to several refinements until *Train_M4* is reached. *Train_M4* is decomposed into *LeaderCarriage* and *Carriage*. We are interested in refining the sub-component corresponding to carriages in order to introduce doors requirements. These requirements are extracted from real requirements for metro carriage doors.*Carriage* is refined and decomposed until it fits in a generic model *GCDoor* corresponding to a *Generic Carriage Door* development as seen in Fig. 6.4. We then instantiate *GCDoor* into two instances: *EmergencyDoors* and *ServiceDoors* benefiting from the refinements in the pattern. We describe in more detail each of the development steps in the following sections.

6.3 Abstract Model: *MetroSystem_M0*

We model a system constituted by trains that circulate in tracks. The tracks are divided into smaller parts called sections. The most important (safety) global property introduced at this stage states that two trains cannot be in the same section at the same time (which would mean that the trains had clashed).

We need to ensure some properties regarding the routes (set of track sections):

- Route sections are all connected: sections should be all connect and cannot have empty spaces between them.

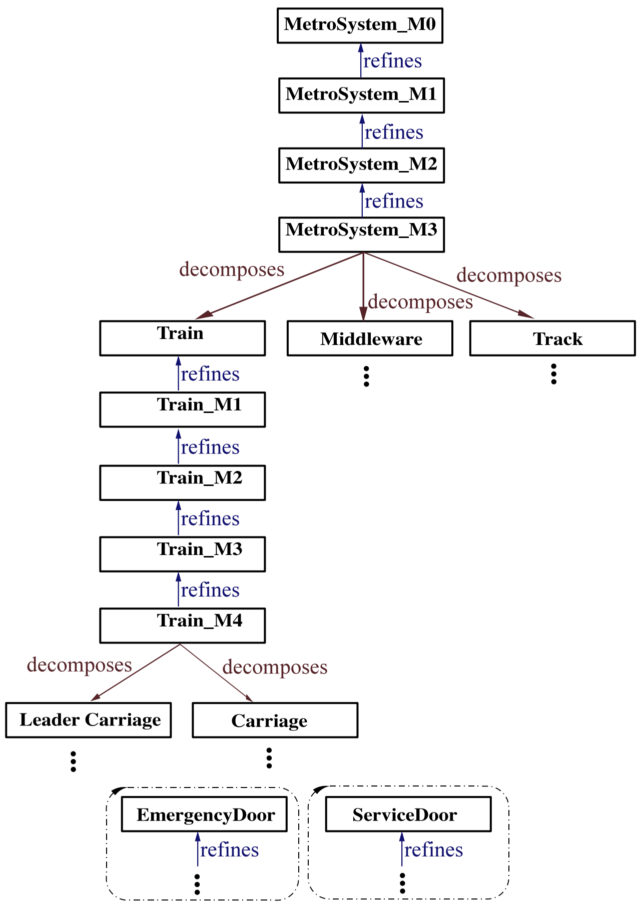


FIGURE 6.3: Overall view of the safety-critical metro system development

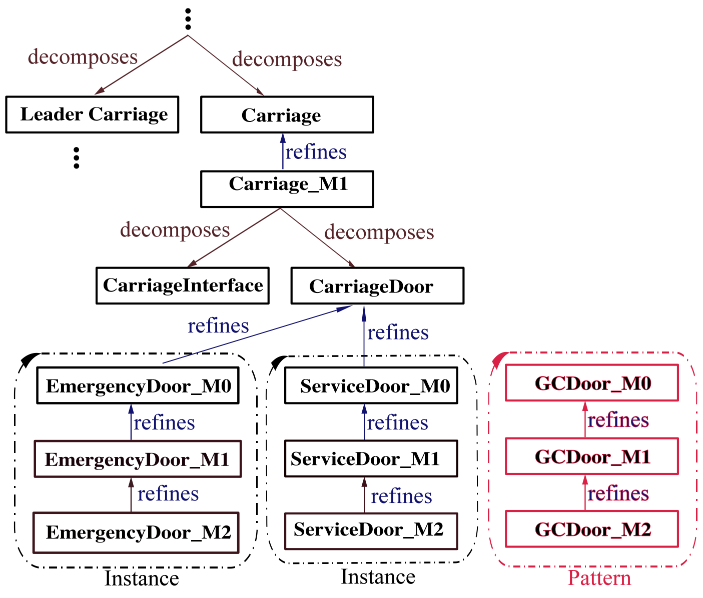


FIGURE 6.4: Carriage Refinement Diagram and Door Instantiation

- There are no loops in the route sections: sections cannot be connected to each other and cannot introduce loops.

These properties can be preserved if we represent the routes as a transitive closure relation. We use the no-loop property proposed by Abrial [9] applied to model a tree structured file system in Event-B [61]: a context is defined and this property is proved over track section relations and functions. The reason we choose this formulation, instead of transitive closure which is generally used is to make the model simpler and easier to prove. Context *TransitiveClosureCtx* containing the transitive closure property can be seen in Fig. 6.5.

```
context TransitiveClosureCtx

constants cdvrel // type of relation on sections
          tcl // transitive closure of an cdvrel
          cdvfn // type of function on sections */

sets CDV // Track Sections

axioms
  @axm1 cdvrel = CDV ↔ CDV
  @axm2 cdvfn = CDV ⇸ CDV
  @axm3 tcl ∈ cdvrel → cdvrel
  @axm4 ∀r·(r∈cdvrel ⇒ r ⊆ tcl(r)) // r included in tcl(r)
  @axm5 ∀r·(r∈cdvrel ⇒r;tcl(r) ⊆ tcl(r)) // unfolding included in tcl(r)
  @axm6 ∀r,t·(r∈cdvrel ∧ r⊆t ∧ r;t⊆t ⇒ tcl(r)⊆t) // tcl(r) is least
  theorem @thm1 cdvfn ⊆ cdvrel
  theorem @thm2 ∀r·r∈cdvrel ⇒ tcl(r) = r ∪ (r;tcl(r)) // tcl(r) is a fixed
point
  theorem @thm3 ∀t·t∈cdvfn∧(∀s·s⊆t∼[s]⇒s=∅)⇒tcl(t)∩(CDV ◁ id)=∅
theorem @thm4 tcl(∅) = ∅
end
```

FIGURE 6.5: Context *TransitiveClosureCtx*

Set CDV represents all the track sections in our model. Constant *tcl* which is a transitive closure, it is defined as a total function mapped from $CDV \leftrightarrow CDV$ to $CDV \leftrightarrow CDV$. Giving $r \in CDV \leftrightarrow CDV$, the transitive closure of r is the least x satisfying $x = r \cup r; x$ [61]. Difficult transitive closure proofs in machines are avoided by using theorems such as theorem $thm3$ shown in Fig. 6.5: for $s \subseteq CDV$ and t as a partial function $CDV \nrightarrow CDV$, $s \subseteq t^{-1}[s]$ means that s contains a loop in the t relationship. Hence, this states that the only such set that can exist is the empty set and thus the t structure cannot have loops. This theorem has been proved using the interactive prover of Rodin. The strategy to prove this theorem is to use proof by contradiction [61].

We define the environment of the case study (static part) with context *MetroSystem_C0* that extends *TransitiveClosureCtx* as seen in Fig. 6.6. Set $TRAIN$ represent all the trains in our model. Several track properties are described in the axioms:

- The constant *net* represents the total possible connectivity of sections (all possible routes subject to the switches positions) defined as relation $CDV \leftrightarrow CDV$ ($axm1$). No circularity is allowed as described by $axm2$. Moreover, the no loop property

```
context MetroSystem_C0 extends TransitiveClosureCtx

constants aig_cdv // Switches
          net // Total connectivity of sections */
          div_aig_cdv // divergent switches 1->2
          cnv_aig_cdv // convergent switches 2->1
          next0

sets TRAIN

axioms
  @axm1 net ∈ CDV ↔ CDV // net represents the connectivity between track sections /*
  @axm2 net ∩(CDV ◁ id)=∅ // no cdv is connected to itself
  @axm3 aig_cdv ⊆ CDV // aig_cdv is a subset of CDV representing cdv that are switches
  @axm4 div_aig_cdv  ⊆ aig_cdv // div_aig_cdv ⊆ aig_cdv
  @axm5 cnv_aig_cdv ⊆ aig_cdv
  @axm6 div_aig_cdv ∩ cnv_aig_cdv = ∅
  @axm7 finite(net) // explicite declaration to simplify the proving
  @axm8 (aig_cdv × aig_cdv) ∩ net = ∅ // switches are not directly connected
  @axm9 ∀cc·(cc ∈ (CDV\aig_cdv) ⇒ card(net[{cc}]) ≤1 ∧ card(net~[{cc}])≤1) // non
switch cdv has at most one successor and at most one predecessor
  @axm10 ∀cc·( cc ∈ aig_cdv ⇒ ( (card(net[{cc}])≤2 ∧ card(net~[{cc}])≤1) ∨ (
card(net[{cc}]) ≤1 ∧ card( net~[{cc}])≤2 ))) // switch cdv has at most two predecessors
and one successor or one predecessor and two successors
  @axm11 tcl(net)∩id=∅ // No-loop property
  theorem @thm1 tcl(net) = net ∪ (net;tcl(net))// the transitive closure of net is
equal to net ∪ net;tcl(net)
end
```

Figure 6.6: Context *MetroSystem_C0*

for *net* is expressed by axiom *axm*11. Theorems *thm*1 states that *net* preserves transitive closure.

- Switches (*aiguillages* in French) are sections (*axm*3) that cannot be connected to each others (*axm*6). They are represented by *aig_cdv* divided into two kinds: *div_aig_cdv* for divergence switches and *cnv_aig_cdv* for convergent switches. Moreover switches have at most two predecessors and one successor or one predecessor and two successors (*axm*10).
- Non-switches have at most one successor and at most one predecessor (*axm*9).

Besides the global property described before defined by invariant *inv*13 in Fig. 6.7(a), some other properties of the system are added:

1. The trains (variable *trns*) circulate in tracks. The current route based on current positions of switches is defined by *next*: a partial injection $CDV \rightarrowtail\!\!\!\!\!\!+ \; CDV$. *next* is a subset of *net* (*inv*1) preserving the transitive closure property as described by theorem *thm*1,*thm*2 and does not have loops (*thm*3). Sections occupied by trains are represented by variable *occp*. These sections also preserve the transitive closure property as seen by *thm*4.
2. A train occupies at least one section and the section corresponding to the beginning and end of the train is represented by variables *occpA* and *occpZ* respectively. Note that *next* does not indicate the direction that a train is moving in: the direction can be *occpA* to *occpZ* or *occpZ* to *occpA*. These two variables point to the same section if the train only occupies one section (*inv*11).

The system proceeds as follows: trains modelled in the system circulate by entering and leaving sections (events *enterCDV* and *leaveCDV* in Fig. 6.7(b)), ensuring that the next section is not occupied (*grd*9 in *enterCDV*) and updating all the sections occupied by the train (*act*1 and *act*2 in both events). At this abstract level, event *modifyTrain* modifies a train defining the set of occupied sections for a train *t*. A train changes speed, brakes or stops braking in events *changeSpeed*, *brake* and *stopBraking*. When event *brake* occurs, train *t* is added to a set of braking trains (variable *braking*). Variable *next* represents the current connectivity of the trail based on the positions of switches. The current connectivity can be updated by changing convergent/divergent switches in events *switchChangeDiv* and *switchChangeCnv* as seen in Fig. 6.7(b).

6.4 First Refinement: *MetroSystem_M1*

MetroSystem_M1 refines *MetroSystem_M0*, incorporating the communication layer and an emergency button for each train. The communication work as follows: a message is sent from the tracks, stored in a buffer and read in the recipient train. The properties to be preserved for this refinement are:

1. Messages are exchanged between trains and tracks. If a train intends to move to an occupied section, track sends a message negating the access to that section and the train should brake.
2. As part of the safety requirements, all trains have an emergency button.
3. While the emergency button is enabled, the train continues braking and cannot speed up.

Now the system proceeds as follows: trains that enter and leave sections must take into account the messages sent by the tracks. Therefore events corresponding to enter and leaving section need to be strengthened to preserve this property. The requirement concerning the space required for the train to halt is a simplification of a real metro system and could require adjustments to replicate the real behaviour (for instance the occupied sections of a train could be defined as the sum of the sections directly occupied by the train and the sections indirectly occupied by the same train that correspond to the sections required for the train to halt). Nevertheless in real systems, trains can have in-built a way to detect the required space to break. For instance in Communication Based Train Control (CBTC [97, 72]) systems, that is called the *stopping distance downstream*.

The messages are represented by variables *tmsgs* that stores the messages (buffer) sent from the tracks and *permit* that receives the message in the train, expressing property 1. At this level, the messages are just boolean values assessing if a train can move to the

```
machine MetroSystem_M0 sees MetroSystem_C0

variables next // Currrent connectivity based on switch positions
          trns // Set of trains on network
          occp // Occupancy function for section
          occpA // Initial cdv occupied by train
          occpZ // Final   cdv occupied by train
          braking speed

invariants
  @inv1 next ⊆ net
  @inv2 next ∈ CDV ⤔ CDV
  @inv3 trns ⊆ TRAIN
  @inv4 occp ∈ CDV ↔ trns
  @inv5 occpA ∈ trns → CDV
  @inv6 ∀tt·(tt∈trns ⇒ occpA(tt) ∈ occp∼[{tt}])
  @inv7 occpZ ∈ trns → CDV
  @inv8 ∀tt·(tt∈trns ⇒ occpZ(tt) ∈ occp∼[{tt}])
  @inv9 braking ⊆ trns
  @inv10 speed ∈ trns → ℕ
  @inv11 ∀tt·tt∈trns ∧ card(occp∼[{tt}])>1 ⇒ occpA(tt) ≠ occpZ(tt)
  @inv12 finite(occp∼)
  @inv13 ∀t1,t2·t1∈trns ∧ t2∈trns ∧ t1≠t2 ⇒ occp∼[{t1}]∩occp∼[{t2}]=∅
   theorem @thm1 next ∈ cdvfn
  theorem @thm2 tcl(next) = next ∪ (next;tcl(next)) // tcl(next) is a fixed
point
  theorem @thm3 (∀s·s⊆next∼[s]⇒s=∅)⇒tcl(next)∩(CDV ◁ id)=∅ // next has no
loops
  theorem @thm4 ∀tt,s·tt∈trns ∧ s ⊆ next▷occp∼[{tt}] ⇒ tcl(s) = s ∪
(s;tcl(s))
```

(a) Variables, invariants in *MetroSystem_M0*

```
event enterCDV
  any t1 c1 c2
  where
    @grd1 t1 ∈ trns
    @grd2 c1 ∈ CDV
    @grd3 c2 ∈ CDV
    @grd4 speed(t1)>0
    @grd5 c1 = occpZ(t1)
    @grd6 c1∈dom(next)
    @grd7 c2 = next(occpZ(t1))
    @grd8 ∀tt·tt∈trns ∧ card((occp ∪ {c2 ↦ t1})∼[{tt}])>1
          ⇒ (occpZ⩤{t1 ↦ c2})(tt) ≠ occpA(tt)
    @grd9 c2 ∉ dom(occp)
  then
    @act1 occpZ(t1) ≔ c2
    @act2 occp≔occp ∪ { c2 ↦ t1}
end

event leaveCDV
  any t1 c1 c2
  where
    @grd1 t1 ∈ trns
    @grd2 c1 ∈ CDV
    @grd3 c2 ∈ CDV
    @grd4 speed(t1)>0
    @grd5 c1∈dom(next)
    @grd6 c1=occpA(t1)
    @grd7 c2=next(c1)
    @grd8 occpA(t1)≠occpZ(t1)
    @grd9 c2 ∈ (occp\{c1↦t1})∼[{t1}]
    @grd10 ∀tt·tt∈trns ∧ card(((occp \ {c1 ↦ t1}))∼[{tt}])>1
           ⇒ (occpA⩤{t1 ↦ c2})(tt)≠occpZ(tt)
  then
    @act1 occpA(t1)≔c2
    @act2 occp ≔ occp\{c1↦t1}
end

event changeSpeed
  any t1 s1
  where
    @grd1 t1 ∈ trns
    @grd2 s1 ∈ ℕ
    @grd3 t1∈ braking ⇒ s1<speed(t1)
  then
    @act1 speed(t1) ≔ s1
end
```

```
event brake
  any t1
  where
    @grd1 t1 ∈ TRAIN
    @grd2 t1∈trns\braking
  then
    @act1 braking≔braking∪ {t1}
end

event stopBraking
  any t1
  where
    @grd1 t1 ∈ TRAIN
    @grd2 t1∈braking
  then
    @act1 braking≔braking\{t1}
end

event switchChangeDiv
  any ac c1 c2
  where
    @grd1 ac ∈ div_aig_cdv
    @grd2 c1 ∈ CDV
    @grd3 c2 ∈ CDV
    @grd8 c2 ∉ ran (next)
    @grd4 (ac ↦ c1) ∈ next
    @grd5 (ac ↦ c2) ∈ net
    @grd6 c1 ≠ c2
    @grd7 ac ∉ dom(occp)
  then
    @act1 next ≔ next ⩤ {ac ↦ c2}
end

event switchChangeCnv
  any ac c1 c2
  where
    @grd1 ac ∈ cnv_aig_cdv
    @grd2 c1 ∈ CDV
    @grd3 c2 ∈ CDV
    @grd8 c2 ∉ dom (next)
    @grd4 (c1 ↦ ac) ∈ next
    @grd5 (c2 ↦ ac) ∈ net
    @grd6 c1 ≠ c2
    @grd7 ac ∉ dom (occp)
  then
    @act1 next ≔ ({c1}⩤next) ∪ {c2 ↦ ac}
end
```

```
event addTrain
  any t oc
  where
    @grd1 t ∈ TRAIN\trns
    @grd2 oc ∈ CDV
    @grd3 oc ∉ dom(occp)
  then
    @act1 trns≔trns ∪{t}
    @act2 speed(t)≔0
    @act3 occpA(t) ≔ oc
    @act4 occpZ(t) ≔ oc
    @act5 occp ≔ occp ∪ {oc↦t}
end

event modifyTrain
  any t ocA oc
  where
    @grd1 ocA∈dom(next)
    @grd2 t ∈ trns
    @grd3 oc ⊆ CDV
    @grd4 ocA ∈ oc
    @grd5 oc ∩ dom(occp)=∅
    @grd6 finite(oc)
    @grd7 occpZ(t)∈dom(next)
    @grd8 card(oc)=0 ⇒ocA = occpZ(t)
    @grd9 card(oc)≥1
          ⇒ occpZ(t) ≠ ocA ∧ next(occpZ(t))∈oc
    @grd10 next(ocA)∉oc
  then
    @act1 occpA(t) ≔ ocA
    @act2 occp ≔ occp ∪ (oc×{t})
 end
```

(b) Events of *MetroSystem_M0*

Figure 6.7: Variables, invariant and events of *MetroSystem_M0*

following section (check if the section is free): if TRUE the train can move; if FALSE the next section is occupied and the train should brake. New event *sendTrainMsg* models the message sending. The reception of messages is modelled in event *recvTrainMsg* where the message is stored in *permit* before *tmsgs* is reset. The guards of event *brake* are strengthened to allow a train to brake when $permit(t)$ = FALSE or when the emergency button is activated (guard *grd*3 in Fig. 6.8(b)). Property 2 is expressed by adding variable *emergency_button*. The activation/deactivation of the emergency button occurs in the new event *toggleEmergencyButton*. Property 3 is expressed by guard *grd*3 in event *stopBraking*: a train can only stop braking if the emergency button is not enabled.

```
machine MetroSystem_M1 refines MetroSystem_M0  sees MetroSystem_C0

variables next trns occp occpA occpZ
          braking speed
          tmsgs permit emergency_button

invariants
  @inv1 tmsgs ∈ trns ⟶ ℙ(BOOL)
  @inv2 permit ∈ trns ⟶ BOOL
  @inv3 emergency_button ∈ trns ⟶ BOOL
```

(a) Variables and invariants in *MetroSystem_M1*

```
event brake refines brake
  any t1
  where
    @grd1 t1 ∈ TRAIN
    @grd2 t1∈trns\braking
    @grd3 permit(t1) = FALSE
          ∨ emergency_button(t1)=TRUE
  then
    @act1 braking≔braking ∪ {t1}
end

event stopBraking refines stopBraking
  any t1
  where
    @grd1 t1 ∈ TRAIN
    @grd2 t1∈braking
    @grd3 emergency_button(t1) = FALSE
  then
    @act1 braking≔braking\{t1}
end
```

```
event sendTrainMsg
    any t1
    where
      @grd1 t1 ∈ trns
      @grd2 tmsgs(t1) = ∅
    then
      @act1 tmsgs(t1)≔ {bool(
            occpZ(t1)∈dom(next)
            ∧next(occpZ(t1)) ∉ dom(occp))}
  end

event recvTrainMsg
    any t1 bb
    where
      @grd1 t1 ∈ trns
      @grd2 bb ∈ tmsgs(t1)
    then
      @act1 permit(t1) ≔ bb
      @act2 tmsgs(t1) ≔ ∅
  end
```

```
event toggleEmergencyButton
  any t value
  where
    @guard t ∈ trns
    @guard1 value ∈ BOOL
  then
    @act1 emergency_button(t)≔ value
end
```

(b) Some events of *MetroSystem_M1*

FIGURE 6.8: Excerpt of *MetroSystem_M1*

6.5 Second Refinement: *MetroSystem_M2*

In this refinement, we introduce train doors and platforms where the trains can stop to load/unload. When stopped, a train can open its doors. The properties to be preserved are:

1. If a train door is opened, then the train is stopped. In contrast, if the train is moving, then its doors are closed.

2. If a train door is opened, that either means that the train is in a platform or there was an emergency and the train had to stop suddenly.

3. A train door cannot be allocated to different trains.

We consider that platforms are represented by single sections. A train is in a platform if one of the occupied sections correspond to a platform. Doors are introduced as illustrated in Fig. 6.9(a) by sets $DOOR$ and their states are represented by $DOOR\_STATE$. Variables $door$ and $door\_state$ represent the train doors and their current states as seen in Fig. 6.9(b): all trains have allocated a subset of doors ($inv2$). Several invariants are introduced to preserve the desired properties: property 1 is defined by invariants $inv4$ and $inv5$; property 2 is defined by invariant $inv7$; property 3 is stated by $inv3$; theorem $thm1$ is used for proving purposes (if no doors are open, then all doors are closed).

To preserve $inv5$, the guards of $changeSpeed$ (in Fig. 6.8(b)) are strengthened by $grd4$ ensuring that whilst the train is moving, the train doors are closed. Also events that model entering and leaving sections are affected, with the introduction of a similar guard ($grd11$ in $leaveCDV$). Adding/removing train doors is modelled in events $addDoorTrain$ and $removeDoorTrain$ respectively: to add/remove a door, the respective train must be stopped. If the train is stopped and either one of the occupied sections corresponds to a platform or the emergency button is activated (guard $grd3$), doors can be opened as seen in event $openDoor$. For safety reasons, event $toggleEmergencyButton$ is strengthened by guard $grd3$ to activate the emergency button whenever doors are open and the train is not in a platform.

6.6 Third Refinement and First Decomposition: *MetroSystem_M3*

This refinement does not introduce new details to the model. It corresponds to the preparation step before the decomposition. We want to implement a three way shared event decomposition and therefore we need to separate the variables that will be allocated to each sub-component. In particular for exchanged messages between the sub-components, the protocol will work as follows: messages are sent from *Track* and stored in the *Middleware*. After receiving the message, the *Middleware* forwards it to the corresponding *Train*. *Train* reads the message and processes it according to the content. This protocol allows a separation between *Train* and *Track* with the *Middleware* working as a bridge between these two sub-components.

The decomposition follows the steps described in Sect. 5.5. Variables are distributed according to Fig. 6.10. To avoid constraints during the decomposition process, predicates and assignments containing variables that belong to different sub-components are rearranged in this refinement step.

```
context MetroSystem_C1 extends MetroSystem_C0

constants OPEN CLOSED PLATFORM

sets DOOR_STATE DOOR

axioms
  @axm1 partition(DOOR_STATE, {OPEN}, {CLOSED})
  @axm2 PLATFORM ⊆ CDV
end
```

(a) Context *MetroSystem_C1*

```
machine MetroSystem_M2 refines MetroSystem_M1  sees MetroSystem_C1

variables next trns occp occpA occpZ
          braking speed tmsgs permit
          door door_state emergency_button

invariants
  @inv1 door_state ∈ DOOR ⟶ DOOR_STATE
  @inv2 door ∈ trns ⟶ ℙ(DOOR)
  @inv3 ∀t1,t2·t1 ∈ dom(door) ∧ t2 ∈ dom(door) ∧ t1 ≠t2
        ⇒ door(t1) ∩ door(t2) = ∅
  @inv4 ∀t·t ∈ dom(door) ⇒(∃d·d⊆door(t) ∧ door_state[d]={OPEN}
        ⇒ speed(t)=0)
  @inv5 ∀t·t ∈ dom(door) ∧ speed(t) > 0
        ⇒ door(t) ⊆ door_state~[{CLOSED}]
  @inv6 ∀t,d·t ∈ dom(door) ∧ d ∈ door(t) ∧ PLATFORM ∩ occp~[{t}]≠∅
        ⇒ door_state(d) ∈ {OPEN, CLOSED}
  @inv7 ∀t·t ∈ dom(door) ∧ door(t) ∩ door_state~[{OPEN}] ≠ ∅
        ⇒ PLATFORM ∩ occp~[{t}]≠∅ ∨ emergency_button(t) = TRUE
  theorem @thm1 ∀t·t ∈ dom(door) ∧ door(t) ∩ door_state~[{OPEN}] =∅
          ⇒ door(t)⊆door_state~[{CLOSED}]
```

(b) Variables, invariants in *MetroSystem_M2*

```
event toggleEmergencyButton
refines toggleEmergencyButton
  any t value
  where
    @grd1 t ∈ dom(door)
    @grd2 value ∈ BOOL
    @grd3 door(t) ∩ door_state~[{OPEN}] ≠ ∅
          ∧ PLATFORM ∩ occp~[{t}]=∅
          ⇒ value = TRUE
  then
    @act1 emergency_button(t)≔ value
end

event openDoor
  any t ds
  where
    @grd1 t ∈ dom(door)
    @grd2 speed(t) = 0
    @grd3 occp~[{t}] ∩ PLATFORM ≠ ∅
          ∨ emergency_button(t) = TRUE
    @grd4 ds ⊆ door(t)
    @grd5 ∃d·d∈ds⇒door_state(d)=CLOSED
    @grd6 ds≠∅
  then
    @act1 door_state≔ door_state  (ds×{OPEN})
end

event closeDoor
  any t ds
  where
    @grd1 t ∈ dom(door)
    @grd2 speed(t) = 0
    @grd3 ds ⊆ door(t)
    @grd4 door_state[ds]={OPEN}
    @grd5 ds≠∅
  then
    @act1 door_state≔ door_state  (ds×{CLOSED})
end
```

```
event addDoorTrain
  any t d
  where
    @grd1 t ∈ trns
    @grd2 d ⊆ DOOR
    @grd3 ∀tr·tr∈dom(door) ∧ tr≠t
          ∧ door(tr)≠∅⇒d∩door(tr)=∅
    @grd5 speed(t)=0
    @grd7 d∩door(t)=∅
  then
    @act1 door(t)≔door(t)∪d
    @act2 door_state≔
          door_state(d×{CLOSED})
end

event removeDoorTrain
  any t d
  where
    @grd1 t ∈ dom(door)
    @grd2 d ⊆ DOOR
    @grd3 d ⊆ door(t)
    @grd4 door_state[d]={CLOSED}
    @grd5 speed(t)=0
  then
    @act1 door(t) ≔ door(t)\d
end
```

```
event leaveCDV refines leaveCDV
  any t1 c1 c2
  where
    @grd1 t1 ∈ dom(door)
    @grd2 c1 ∈ CDV
    @grd3 c2 ∈ CDV
    @grd4 speed(t1)>0
    @grd5 c1∈dom(next)
    @grd6 c1=occpA(t1)
    @grd7 c2=next(c1)
    @grd8 occpA(t1)≠occpZ(t1)
    @grd9 c2 ∈ (occp\{c1↦t1})~[{t1}]
    @grd10 ∀tt·tt∈trns
    ∧ card(((occp \ {c1 ↦ t1}))~[{tt}])>1
    ⇒ (occpA{t1 ↦ c2})(tt)≠occpZ(tt)
    @grd11 door(t1)∩door_state~[{OPEN}]=∅
    @grd12 permit(t1)=TRUE
  then
    @act1 occpA(t1)≔c2
    @act2 occp ≔ (occp\{c1↦t1})
end

event changeSpeed refines changeSpeed
  any t1 s1
  where
    @grd1 t1 ∈ dom(door)
    @grd2 s1 ∈ N
    @grd3 t1∈ braking ⇒ s1<speed(t1)
    @grd4 door(t1)∩door_state~[{OPEN}]=∅
  then
    @act1 speed(t1) ≔ s1
end
```

(c) Some events of *MetroSystem_M2*

Figure 6.9: Excerpt of *MetroSystem_M2*

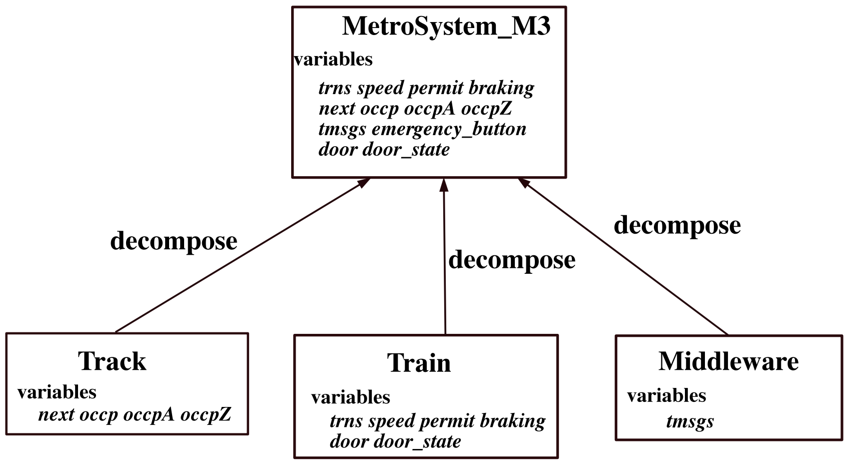


FIGURE 6.10: *MetroSystem_M3* (shared event) decomposed into *Track*, *Train* and *Middleware*

Some guards need to be rewritten in the refined events. For instance, guard $grd10$ in event $leaveCDV$ needs to be rewritten in order not to include both variables $trns$ (sub-component $Train$) and $occp$ (sub-component $Track$). Therefore it is changed from:

$$\forall tt \cdot tt \in \mathbf{trns} \land card((occp \cup \{c2 \mapsto t1\})^{-1}[\{tt\}]) > 1 \Rightarrow (occpZ \mathbin{\lhd\!\!\!-} \{t1 \mapsto c2\})(tt) \neq occpA(tt)$$

to:

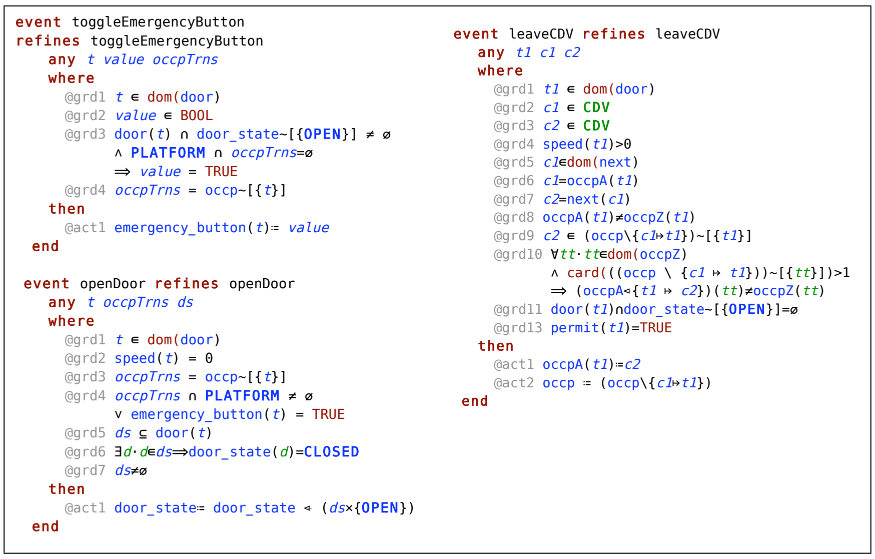
$$\forall tt \cdot tt \in \mathbf{dom(occpZ)} \land card((occp \cup \{c2 \mapsto t1\})^{-1}[\{tt\}]) > 1 \Rightarrow (occpZ \mathbin{\lhd\!\!\!-} \{t1 \mapsto c2\})(tt) \neq occpA(tt) \text{ (Fig. 6.11).}$$

Both predicates represent the same property since $trns$ corresponds to the domain of variable *occpZ* (see $inv7$ in Fig. 6.7(a)). In Fig. 6.11, the original guard $grd3$ in $toggleEmergencyButton$ is rewritten to separate variables *occp* and $door$. In this case, an additional parameter *occpTrns* representing the variable *occp* is added ($grd4$). This additional parameter will represent the value passing between the resulting decomposed events: parameter *occpTrns* is written the value of *occp* and afterwards it is read in guard $grd3$. Similarly guard $grd4$ in event *openDoor* must not include variables *occp* and *emergency_button* and consequently parameter *occpTrns* is added.

Sub-components $Train$, $Track$ and $Middleware$ are described in the following sections. The composed machine corresponding to the defined decomposition can be seen in Fig. 6.12 where it is illustrated how the original events are decomposed.

6.6.1 Machine *Track*

Machine *Track* contains the properties concerning the sections in the metro system. Events corresponding to entering, leaving tracks and changing switch positions are part of this sub-component resulting from the variables allocation for this sub-component: $next$, *occp*, *occpA* and *occpZ*. Event *sendTrainMsg* is also added since the messages are



```
event toggleEmergencyButton
refines toggleEmergencyButton
    any t value occpTrns
    where
      @grd1 t ∈ dom(door)
      @grd2 value ∈ BOOL
      @grd3 door(t) ∩ door_state~[{OPEN}] ≠ ∅
            ∧ PLATFORM ∩ occpTrns=∅
            ⇒ value = TRUE
      @grd4 occpTrns = occp~[{t}]
    then
      @act1 emergency_button(t)≔ value
  end

 event openDoor refines openDoor
    any t occpTrns ds
    where
      @grd1 t ∈ dom(door)
      @grd2 speed(t) = 0
      @grd3 occpTrns = occp~[{t}]
      @grd4 occpTrns ∩ PLATFORM ≠ ∅
            ∨ emergency_button(t) = TRUE
      @grd5 ds ⊆ door(t)
      @grd6 ∃d·d∈ds⇒door_state(d)=CLOSED
      @grd7 ds≠∅
    then
      @act1 door_state≔ door_state ⩤ (ds×{OPEN})
  end
```

```
event leaveCDV refines leaveCDV
   any t1 c1 c2
   where
     @grd1 t1 ∈ dom(door)
     @grd2 c1 ∈ CDV
     @grd3 c2 ∈ CDV
     @grd4 speed(t1)>0
     @grd5 c1∈dom(next)
     @grd6 c1=occpA(t1)
     @grd7 c2=next(c1)
     @grd8 occpA(t1)≠occpZ(t1)
     @grd9 c2 ∈ (occp\{c1↦t1})~[{t1}]
     @grd10 ∀tt·tt∈dom(occpZ)
            ∧ card(((occp \ {c1 ↦ t1}))~[{tt}])>1
            ⇒ (occpA⩤{t1 ↦ c2})(tt)≠occpZ(tt)
     @grd11 door(t1)∩door_state~[{OPEN}]=∅
     @grd13 permit(t1)=TRUE
   then
     @act1 occpA(t1)≔c2
     @act2 occp ≔ (occp\{c1↦t1})
 end
```

Figure 6.11: Preparation step before decomposition of *MetroSystem_M3*

sent from the tracks as seen in Fig. 6.13. The original events *toggleEmergencyButton* and *openDoor* require *occp* in their guards. Consequently part of these original events are included in this sub-component.

Note that the invariants defining the variables may change: in *MetroSystem_M1* variable *occp* is defined as $occp \in CDV \leftrightarrow trns$ (*inv4* in Fig. 6.7(a)); in *Track* is $occp \in CDV \leftrightarrow TRAIN$ (which is the same as theorem $typing\_occp : occp \in \mathbb{P}(CDV \times TRAIN)$ in Fig. 6.13). This is a consequence of the variable partition since *trns* is not part of *Track* and therefore the *occp* relation is updated with *trns*'s type: $TRAIN$ (cf. *inv*3 in Fig. 6.7(a)). Variables *occpA* and *occpZ* are subject to the same procedure where the original invariant is a total function $trns \rightarrow CDV$ and in the sub-component both become $\mathbb{P}(TRAIN \times CDV)$. The sub-components invariants are derived from the different initial abstract models (cf. their labels in Fig. 6.13). Invariants that only restrain the sub-component variables are automatically included although additional ones can be added manually.

6.6.2 Machine *Train*

Machine *Train* models the trains in the metro system. Trains entering/leaving a section, modelled by events *enterCDV* and *leaveCDV* are part of this sub-component, in spite of the decomposed events do not execute any actions (see Fig. 6.14(b)). The interaction with sub-component *Track* occurs through parameters $t1$, $c1$ and $c2$ (see events *Track.leaveCDV* in Fig. 6.13). Variables *door* and *door_state* are part of this sub-component and consequently the events that modify these variables: *openDoor* and *closeDoor*. Moreover, since the emergency button is part of a train, the respective variable *emergencyButton* (and the modification event *toggleEmergencyButton*) is also included in this sub-component. Event *recvTrainMsg* receives messages sent to the

```
COMPOSED MACHINE MetroSystem_M3_cmp
REFINES MetroSystem_M3
INCLUDES
    Track   Train   Middleware
EVENTS
    addTrain refines addTrain
        Combines Events Train.addTrain || Middleware.addTrain ||Track.addTrain
    modifyTrain refines modifyTrain
        Combines Events Train.modifyTrain ||Track.modifyTrain
    sendTrainMsg refines sendTrainMsg
        Combines Events Track.sendTrainMsg || Middleware.sendTrainMsg
    recvTrainMsg refines recvTrainMsg
        Combines Events Train.recvTrainMsg || Middleware.recvTrainMsg
    changeSpeed refines changeSpeed
        Combines Events Train. changeSpeed
    brake refines brake
        Combines Events Train.brake
    stopBraking refines stopBraking
        Combines Events Train.stopBraking
    enterCDV refines enterCDV
        Combines Events Train.enterCDV || Track.enterCDV
    leaveCDV refines leaveCDV
        Combines Events Train.leaveCDV || Track.leaveCDV
    openDoor refines openDoor
        Combines Events Train.openDoor || Track.openDoor
    closeDoor refines closeDoor
        Combines Events Train.closeDoor
    toggleEmergencyButton refines toggleEmergencyButton
        Combines Events Train.toggleEmergencyButton || Track.toggleEmergencyButton
    addDoorTrain refines addDoorTrain
        Combines Events Train.addDoorTrain
    removeDoorTrain refines removeDoorTrain
        Combines Events Train.removeDoorTrain
    switchChangeDiv refines switchChangeDiv
        Combines Events Track.switchChangeDiv
    switchChangeCnv refines switchChangeCnv
        Combines Events Track.switchChangeCnv
END
```

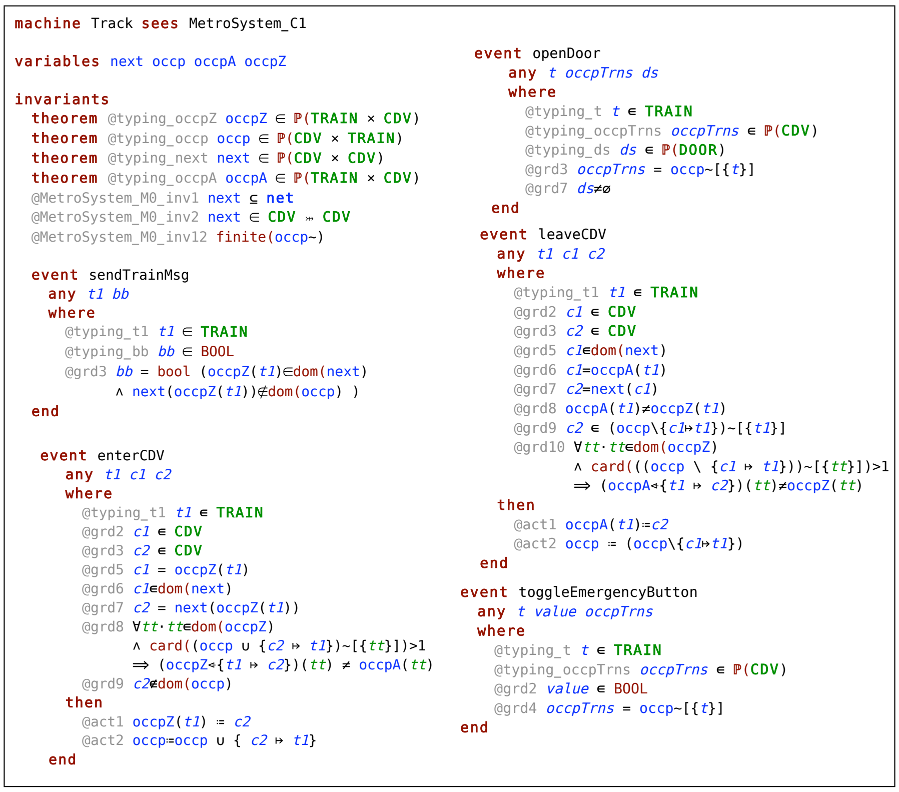
FIGURE 6.12: Composed machine tool view corresponding to *MetroSystem_M3* decomposition

trains and the content is stored in the variable *permit*. Although variable *permit* is set based on the content of the messages exchanged between *Train* and *Track*, that variable is read by trains. This is the reason why it is allocated to this sub-component. The events that change the speed of the train are also included in this sub-component: *brake*, *stopBraking*, *changeSpeed* due to variables *speed* and *braking* as depicted in Fig. 6.14.

6.6.3 Machine *Middleware*

Finally the communication layer in modelled by *Middleware* as seen in Fig. 6.15. *Middleware* bridges *Track* and *Trains*, by receiving messages ($sendTrainMsg$) from the tracks and delivering to the trains ($recvTrainMsg$). Variable *tmsgs* is used as a buffer.

Benefiting from the monotonicity of the shared event approach, the resulting subcomponents can be further refined. Following Fig. 6.3, *Train* is refined as described



```
machine Track sees MetroSystem_C1

variables next occp occpA occpZ

invariants
  theorem @typing_occpZ occpZ ∈ ℙ(TRAIN × CDV)
  theorem @typing_occp occp ∈ ℙ(CDV × TRAIN)
  theorem @typing_next next ∈ ℙ(CDV × CDV)
  theorem @typing_occpA occpA ∈ ℙ(TRAIN × CDV)
  @MetroSystem_M0_inv1 next ⊆ net
  @MetroSystem_M0_inv2 next ∈ CDV ⤖ CDV
  @MetroSystem_M0_inv12 finite(occp~)

  event sendTrainMsg
    any t1 bb
    where
      @typing_t1 t1 ∈ TRAIN
      @typing_bb bb ∈ BOOL
      @grd3 bb = bool (occpZ(t1)∈dom(next)
            ∧ next(occpZ(t1))∉dom(occp) )
  end

   event enterCDV
     any t1 c1 c2
     where
       @typing_t1 t1 ∈ TRAIN
       @grd2 c1 ∈ CDV
       @grd3 c2 ∈ CDV
       @grd5 c1 = occpZ(t1)
       @grd6 c1∈dom(next)
       @grd7 c2 = next(occpZ(t1))
       @grd8 ∀tt·tt∈dom(occpZ)
             ∧ card((occp ∪ {c2 ↦ t1})~[{tt}])>1
             ⇒ (occpZ⩤{t1 ↦ c2})(tt) ≠ occpA(tt)
       @grd9 c2∉dom(occp)
     then
       @act1 occpZ(t1) ≔ c2
       @act2 occp≔occp ∪ { c2 ↦ t1}
    end

event openDoor
    any t occpTrns ds
    where
      @typing_t t ∈ TRAIN
      @typing_occpTrns occpTrns ∈ ℙ(CDV)
      @typing_ds ds ∈ ℙ(DOOR)
      @grd3 occpTrns = occp~[{t}]
      @grd7 ds≠∅
  end
 event leaveCDV
   any t1 c1 c2
   where
     @typing_t1 t1 ∈ TRAIN
     @grd2 c1 ∈ CDV
     @grd3 c2 ∈ CDV
     @grd5 c1∈dom(next)
     @grd6 c1=occpA(t1)
     @grd7 c2=next(c1)
     @grd8 occpA(t1)≠occpZ(t1)
     @grd9 c2 ∈ (occp\{c1↦t1})~[{t1}]
     @grd10 ∀tt·tt∈dom(occpZ)
           ∧ card(((occp \ {c1 ↦ t1}))~[{tt}])>1
           ⇒ (occpA⩤{t1 ↦ c2})(tt)≠occpZ(tt)
   then
     @act1 occpA(t1)≔c2
     @act2 occp ≔ (occp\{c1↦t1})
 end
event toggleEmergencyButton
  any t value occpTrns
  where
    @typing_t t ∈ TRAIN
    @typing_occpTrns occpTrns ∈ ℙ(CDV)
    @grd2 value ∈ BOOL
    @grd4 occpTrns = occp~[{t}]
end
```

FIGURE 6.13: Excerpt of *Track*

in the following section.

6.7 Refinement of *Train*: *Train_M1*

In Train_M1, carriages are introduced as parts of a train. Each carriage has an individual alarm that when activated, triggers the train alarm (enables the emergency button of the train). Each train has a limited number of carriages. Each carriage has a set of doors and the sum of carriage doors corresponds to the doors of a train. The properties to be preserved are:

1. There is a limit to the number ($MAX\_NUMBER\_CARRIAGE$) of carriages per train.
2. Whenever a carriage alarm is activated, then the emergency button of that same train is activated.
3. The sum of carriage doors corresponds to the doors of a train.

The definition of these requirements require the introduction of some static elements like a carrier set *CARRIAGE*, constants *MAX_NUMBER_CARRIAGE* and *DOOR_CARRIAGE* (function between *DOOR* and *CARRIAGE*). The latter is defined as a constant because the number of doors in a carriage does not change. Context

```
machine Train sees MetroSystem_C1

variables trns speed permit braking emergency_button door_state door

invariants
  theorem @typing_trns trns ∈ ℙ(TRAIN)
  theorem @typing_door_state door_state ∈ ℙ(DOOR × DOOR_STATE)
  theorem @typing_braking braking ∈ ℙ(TRAIN)
  theorem @typing_speed speed ∈ ℙ(TRAIN × ℤ)
  theorem @typing_permit permit ∈ ℙ(TRAIN × BOOL)
  theorem @typing_door door ∈ ℙ(TRAIN × ℙ(DOOR))
  theorem @typing_emergency_button emergency_button ∈ ℙ(TRAIN × BOOL)
  @MetroSystem_M0_inv3 trns ⊆ TRAIN
  @MetroSystem_M0_inv9 braking ⊆ trns
  @MetroSystem_M0_inv10 speed ∈ trns ⟶ ℕ
  @MetroSystem_M1_inv2 permit ∈ trns ⟶ BOOL
  @MetroSystem_M1_inv7 emergency_button ∈ trns ⟶ BOOL
  @MetroSystem_M2_inv1 door_state ∈ DOOR ⟶ DOOR_STATE
  @MetroSystem_M2_inv2 door ∈ trns ⟶ ℙ(DOOR)
  @MetroSystem_M2_inv3 ∀t1,t2·t1 ∈ dom(door) ∧ t2 ∈ dom(door) ∧ t1 ≠t2 ⇒ door(t1) ∩ door(t2) = ∅
  @MetroSystem_M2_inv4 ∀t·t ∈ dom(door) ⇒(∃d·d⊆door(t) ∧ door_state[d]={OPEN} ⇒ speed(t)=0)
  @MetroSystem_M2_inv5 ∀t·t ∈ dom(door) ∧ speed(t) > 0 ⇒ door(t) ⊆ door_state∼[{CLOSED}]
  theorem @MetroSystem_M2_thm1 ∀t·t ∈ dom(door) ∧ door(t) ∩ door_state∼[{OPEN}] =∅
                                ⇒ door(t)⊆door_state∼[{CLOSED}]
```

(a) Variables and invariants in *Train*

```
event recvTrainMsg
  any t1 bb
  where
    @typing_t1 t1 ∈ TRAIN
    @typing_bb bb ∈ BOOL
  then
    @act2 permit(t1)≔bb
end

event changeSpeed
  any t1 s1
  where
    @typing_t1 t1 ∈ TRAIN
    @typing_s1 s1 ∈ ℤ
    @grd1 s1 ∈ ℕ
    @grd2 t1 ∈ dom(door)
    @grd3 t1 ∈ braking ⇒ s1 < speed (t1)
    @grd4 door(t1) ∩ door_state∼[{OPEN}] =∅
  then
    @act1 speed (t1) ≔ s1
end

event brake
  any t1
  where
    @typing_t1 t1 ∈ TRAIN
    @grd1 t1 ∈ trns\braking
    @grd2 t1 ∈ dom(emergency_button)
    @grd3 permit(t1) = FALSE
          ∨ emergency_button(t1)=TRUE
  then
    @act1 braking ≔ braking ∪ {t1}
end
```

```
event openDoor
  any t occpTrns ds
  where
    @typing_t t ∈ TRAIN
    @typing_occpTrns occpTrns ∈ ℙ(CDV)
    @typing_ds ds ∈ ℙ(DOOR)
    @grd1 t ∈ dom(door)
    @grd2 speed(t) = 0
    @grd4 occpTrns ∩ PLATFORM ≠ ∅
          ∨ emergency_button(t) = TRUE
    @grd5 ds ⊆ door(t)
    @grd6 ∃d·d∈ds⇒door_state(d)=CLOSED
    @grd7 ds≠∅
  then
    @act1 door_state≔ door_state ⩤ (ds×{OPEN})
end

event closeDoor
  any t ds
  where
    @typing_t t ∈ TRAIN
    @typing_ds ds ∈ ℙ(DOOR)
    @grd1 t ∈ dom(door)
    @grd2 speed(t) = 0
    @grd3 ds ⊆ door(t)
    @grd4 door_state[ds]={OPEN}
    @grd5 ds≠∅
  then
    @act1 door_state≔ door_state ⩤ (ds×{CLOSED})
end

event toggleEmergencyButton
  any t value occpTrns
  where
    @typing_t t ∈ TRAIN
    @typing_occpTrns occpTrns ∈ ℙ(CDV)
    @grd1 t ∈ dom(door)
    @grd2 value ∈ BOOL
    @grd3 door(t) ∩ door_state∼[{OPEN}] ≠ ∅
          ∧ PLATFORM ∩ occpTrns=∅
          ⇒ value = TRUE
  then
    @act1 emergency_button(t)≔ value
end
```

```
event addDoorTrain
   any t d
   where
     @typing_d d ∈ ℙ(DOOR)
     @typing_t t ∈ TRAIN
     @grd1 t ∈ trns
     @grd2 d ⊆ DOOR
     @grd3 ∀tr·tr∈dom(door) ∧ tr≠t
           ∧ door(tr)≠∅ ⇒ d∩door(tr)=∅
     @grd5 speed(t)=0
     @grd7 d∩door(t)=∅
   then
     @act1 door(t)≔door(t)∪d
     @act2 door_state≔door_state⩤(d×{CLOSED})
  end

  event removeDoorTrain
   any t d
   where
     @typing_d d ∈ ℙ(DOOR)
     @typing_t t ∈ TRAIN
     @grd1 t ∈ dom(door)
     @grd2 d ⊆ DOOR
     @grd3 d ⊆ door(t)
     @grd4 door_state[d]={CLOSED}
     @grd5 speed(t)=0
   then
     @act1 door(t) ≔ door(t)\d
  end
event leaveCDV
   any t1 c1 c2
   where
     @typing_t1 t1 ∈ TRAIN
     @grd1 t1 ∈ dom(door)
     @grd2 c1 ∈ CDV
     @grd3 c2 ∈ CDV
     @grd4 speed(t1)>0
     @grd11 door(t1)∩door_state∼[{OPEN}]=∅
     @grd12 permit(t1)=TRUE
  end
```

(b) Some events of *Train*

FIGURE 6.14: Excerpt of *Train*

Train_C2 is depicted in Fig. 6.16(a). Several variables are added such as *train_carriage* relating carriages with trains and *carriage_alarm* that is a total function between *CARRIAGE* and BOOL, illustrated in Fig. 6.16(b). Property 1 is expressed by invariant *inv6* stating that trains have a maximum of *MAX_NUMBER_CARRIAGE* carriages. Property 2 is defined in *inv7* as seen in Fig. 6.16(b). Events *activateEmergencyCarriageButton* and *deactivateEmergencyTrainButton* refine abstract event *toggleEmergencyButton*: the first event enables a carriage alarm and consequently enables the emergency button of the train; the later occurs when the emergency button of a train is active

```
machine Middleware sees MetroSystem_C1

variables tmsgs

invariants
  theorem @typing_tmsgs tmsgs ∈ ℙ(TRAIN × ℙ(BOOL))

events
  event INITIALISATION
    then
      @act1 tmsgs ≔ ∅
  end

  event sendTrainMsg
    any t1 bb
    where
      @typing_t1 t1 ∈ TRAIN
      @typing_bb bb ∈ BOOL
      @grd1 t1 ∈ dom(tmsgs)
      @grd2 tmsgs(t1)=∅
    then
      @act1 tmsgs(t1) ≔ {bb}
  end

  event recvTrainMsg
    any t1 bb
    where
      @typing_t1 t1 ∈ TRAIN
      @typing_bb bb ∈ BOOL
      @grd1 t1 ∈ dom(tmsgs)
      @grd2 bb ∈ tmsgs(t1)
    then
      @act1 tmsgs(t1)≔∅
  end

  event addTrain
    any t oc
    where
      @typing_t t ∈ TRAIN
      @grd1 oc ∈ CDV
    then
      @act6 tmsgs(t)≔∅
  end
```

FIGURE 6.15: Machine *Middleware*

and corresponds to the deactivation of the last enabled carriage alarm which results in deactivating the emergency button; a new event *deactivateEmergencyCarriageButton* is added to model the deactivation of a carriage alarm when there is still another alarm enabled for the same train (guards $grd4$ and $grd5$). The allocation and removal of carriages (events *allocateCarriageTrain* and *removeCarriageTrain*) refine *addDoorTrain* and *removeDoorTrain* respectively. In these two events, the parameter d representing a set of doors, is replaced in the witness section by the doors of the added/removed carriage: $d = DOOR\_CARRIAGE^{-1}[\{c\}]$. We continue the refinement of *Train* in the following section.

6.8 Second Refinement of *Train*: *Train_M2*

In this refinement of *Train*, carriages requirements are added. We specify *carriage doors* instead of the more abstract *train doors*. As a consequence, variable *doors* is data refined and disappears. Each train contains two cabin carriages (type A) and two ordinary carriages (type B) allocated as follows: A+B+B+A. Only one of the two cabin carriages is set to be the *leader carriage* controlling the set of carriages and the moving direction. Trains have states defining if they are in *maintenance* or if they are being driven *manually* or *automatically*. More safety requirements are introduced: if the speed of a train exceeds the safety maximum speed, the emergency brake for that train must be activated. The abstract event representing the change of speed is refined by several concrete events and includes the behaviour of the system when a train is above the maximum speed. The properties to be preserved in this refinement are:

1. If a train is not in maintenance, then it must have the correct number of carriages and the leader carriage must be defined already. Consequently, this is a condition to be verified before the train can change speed.

```
context Train_C1 extends MetroSystem_C1

constants MAX_NUMBER_CARRIAGE
          DOOR_CARRIAGE

sets CARRIAGE

axioms
  @axm1 MAX_NUMBER_CARRIAGE ∈ ℕ1
  @axm2 DOOR_CARRIAGE ∈ DOOR→CARRIAGE
  @axm3 ∀c·c∈ran(DOOR_CARRIAGE)
                  ⇒DOOR_CARRIAGE∼[{c}]≠∅
end
```

(a) Context *Train_C1*

```
machine Train_M1 refines Train  sees Train_C1

variables trns speed permit braking door_state door emergency_button
          train_carriage carriage_alarm

invariants
  @inv1 finite(trns)
  @inv2 carriage_alarm ∈ CARRIAGE → BOOL
  @inv3 train_carriage ∈ CARRIAGE ⇸ trns
  @inv4 finite(train_carriage)
  @inv5 finite(dom(train_carriage))
  @inv6 ∀t·t ∈ trns ⇒ card(train_carriage∼[{t}])≤MAX_NUMBER_CARRIAGE
  @inv7 ∃c·(c ∈ dom(train_carriage) ∧ carriage_alarm(c) = TRUE
  ⇔ c ∈ dom(train_carriage) ∧ emergency_button(train_carriage(c))= TRUE)
  @inv8 ∀t·t∈dom(door) ⇒ door(t)=DOOR_CARRIAGE∼[train_carriage∼[{t}]]
```

(b) Variables and Invariants of *Train_M1*

```
event activateEmergencyCarriageButton
refines toggleEmergencyButton
  any c occpTrns
  where
    @grd1 occpTrns ∈ ℙ(CDV)
    @grd2 c ∈ dom(train_carriage)
    @grd3 carriage_alarm(c) = FALSE
  with
    @value value = TRUE
    @t t = train_carriage(c)
  then
    @act1 carriage_alarm(c) ≔ TRUE
    @act2 emergency_button(train_carriage(c)) ≔ TRUE
end

event deactivateEmergencyCarriageButton
  any c
  where
    @grd1 c ∈ dom(train_carriage)
    @grd2 emergency_button(train_carriage(c)) = TRUE
    @grd3 carriage_alarm(c) = TRUE
    @grd4 {c} ≠ (dom(carriage_alarm ▷ {TRUE})
              ∩ train_carriage∼[{train_carriage(c)}])
    @grd5 card(train_carriage▷{train_carriage(c)})>1
  then
    @act1 carriage_alarm(c)≔ FALSE
end

event deactivateEmergencyTrainButton
refines toggleEmergencyButton
  any c occpTrns
  where
    @grd1 occpTrns ∈ ℙ(CDV)
    @grd2 c ∈ dom(train_carriage)
    @grd3 emergency_button(train_carriage(c)) = TRUE
    @grd4 carriage_alarm(c) = TRUE
    @grd5 {c} = (dom(carriage_alarm ▷ {TRUE})
               ∩ train_carriage∼[{train_carriage(c)}])
    @grd6 door(train_carriage(c))∩door_state∼[{OPEN}] = ∅
  with
    @value value = FALSE
    @t t = train_carriage(c)
  then
    @act1 carriage_alarm(c)≔ FALSE
    @act2 emergency_button(train_carriage(c)) ≔ FALSE
end
```

```
event alocateCarriageTrain refines addDoorTrain
  any c t
  where
    @grd1 c ∈ CARRIAGE\dom(train_carriage)
    @grd2 carriage_alarm[{c}]= {FALSE}
    @grd3 ∀tr·tr∈dom(door) ∧ tr≠t ∧ door(tr)≠∅
          ⇒ DOOR_CARRIAGE∼[{c}]∩door(tr)=∅
    @grd4 t ∈ trns
    @grd5 emergency_button(t) = FALSE
    @grd6 finite(train_carriage∼[{t}])
    @grd7 card(dom(train_carriage ▷ {t}))
                             <MAX_NUMBER_CARRIAGE
    @grd8 speed(t)=0
    @grd9 DOOR_CARRIAGE∼[{c}] ∩ door(t)=∅
  with
    @d d=(DOOR_CARRIAGE∼[{c}])
  then
    @act1 train_carriage(c)≔ t
    @act2 door(t)≔door(t) ∪ DOOR_CARRIAGE∼[{c}]
    @act3 door_state≔
      door_state⩤(DOOR_CARRIAGE∼[{c}]×{CLOSED})
end

event removeCarriageTrain refines removeDoorTrain
  any c t
  where
    @grd1 t ∈ dom(door)
    @grd2 c↦t ∈ train_carriage
    @grd3 carriage_alarm(c) = FALSE
    @grd4 emergency_button(t) = FALSE
    @grd5 speed(t)=0
    @grd6 DOOR_CARRIAGE∼[{c}]⊆door(t)
    @grd7 DOOR_CARRIAGE∼[{c}]≠∅
    @grd8 door_state[DOOR_CARRIAGE∼[{c}]]={CLOSED}
  with
    @d d = (DOOR_CARRIAGE∼[{c}])
  then
    @act1 train_carriage ≔ {c}⩤train_carriage
    @act2 door(t)≔door(t)\DOOR_CARRIAGE∼[{c}]
end
```

(c) Some events of *Train_M1*

FIGURE 6.16: Excerpt of machine *Train_M1*

2. If a train is in maintenance, then it must be stopped.

3. If the speed of a train exceeds the maximum speed, the emergency brake must be activated.

Figure 6.17(a) illustrates two new carrier sets: $SIDE$ corresponding to which side a carriage door or a platform is located (constants $LEFT$ or $RIGHT$) and $TRAIN\_STATE$ that defines the state of a train ($MAINTENANCE$, $MANUAL$ or $AUTOMATIC$). There are some new constants added as well: $CABIN\_CARRIAGE$ defined as a subset of $CARRIAGE$, $NUMBER\_CABIN\_CARRIAGE$ defining the number of cabin carriages allowed per train, $DOOR\_SIDE$ defined as a total function between $DOOR$ and $SIDE$ representing which side a door is located, MAX_SPEED defining the upper speed limit for running a train before the activation of the emergency brake and $PLATFORM\_SIDE$ defining the side of a platform.

Figure 6.17 shows *Train_M2* where several new variables are introduced: $leader\_carriage$ defining the leader carriage for a train ($inv6$), $trns\_state$ defining the state of a train ($inv8$), $emergency\_brake$ that defines which trains have the emergency brake activated ($inv11$) and $carriage\_door\_state$ defining the state of the carriage doors ($inv15$). Moreover $door\_train\_carriage$ defines the train doors based on the carriages ($inv2$, $inv3$ and $inv4$) and each door belongs to at most one train ($inv4$) although a train can have several doors ($inv2$). This variable refines $door$ that disappears in this refinement level, plus some gluing invariants: $inv1$, $inv5$ and theorem $thm2$ state that the range of $door$ for a train t is the same as the range of $door\_train\_carriage$ as long as t has doors.

Property 1 is expressed by $inv9$. Property 2 is expressed by $inv10$ and property 3 by $inv12$. $inv13$ and $inv14$ state that the doors in the domain of $door\_state$ are the same as the ones in carriage_door_state and therefore their state must match. Theorem $thm1$ relates the carriages doors with variables $door\_train\_carriage$ and $train\_carriage$. Theorem $thm3$ states that the domain of $carriage\_door\_state$ is a subset of the domain of $door\_state$ since both variables refer to the same set of doors.

New events are added defining the allocating of a leader carriage to a train (event $allocateLeaderCabinCarriageTrain$ in Fig. 6.17(c)). This event is enabled only if the train is in maintenance ($grd5$), already has the required number of carriages ($grd6$) but does not have a leader carriage yet ($grd7$). To deallocate the leader carriage in event $deallocateLeaderCabinCarriageTrain$, the train must be in maintenance. A train change state in event $modifyTrain$: to change to $MAINTENANCE$, the train must be stopped ($grd2$); for the other states, the number of cabin carriages must be $NUMBER\_CABIN\_CARRIAGE$ and a leading carriage have to be allocated already ($grd3$). Abstract event $changeSpeed$ is refined by four events: two to increase the speed ($increaseSpeed$ and $increaseMaxSpeed$ in Fig. 6.17(c)) and two to reduce the speed ($reduceSpeed$ and $reduceMaxSpeed$). If the speed of a train is increasing in a way that is superior to MAX_SPEED, event $increaseMaxSpeed$ is enabled and if it occurs, the emergency_brake is activated. If the current speed of a train is superior to MAX_SPEED but the new speed is decreasing in a way that is inferior to the maximum speed then the $emergency\_brake$ can be deactivated (event $reduceMaxSpeed$).

```
context Train_C2 extends Train_C1

constants CABIN_CARRIAGE NUMBER_CABIN_CARRIAGE
          LEFT RIGHT DOOR_SIDE PLATFORM_SIDE
          MAINTENANCE MANUAL AUTOMATIC MAX_SPEED

sets SIDE TRAIN_STATE

axioms
  @axm1 CABIN_CARRIAGE ⊆ CARRIAGE
  @axm2 NUMBER_CABIN_CARRIAGE ∈ ℕ1
  @axm3 DOOR_SIDE ∈ DOOR → SIDE
  @axm4 partition(SIDE, {LEFT}, {RIGHT})
  @axm5 partition(TRAIN_STATE, {MAINTENANCE},
        {MANUAL},{AUTOMATIC})
  @axm6 MAX_SPEED ∈ ℕ1
  @axm7 PLATFORM_SIDE ∈ PLATFORM → SIDE
  @axm8 finite(CABIN_CARRIAGE)
  @axm9 PLATFORM ≠∅
  @axm10 CABIN_CARRIAGE≠∅
  @axm11 CABIN_CARRIAGE⊆ ran(DOOR_CARRIAGE)
end
```

(a) Context *Train_C2*

```
variables trns speed permit braking door_state emergency_button train_carriage carriage_alarm
leader_carriage trns_state emergency_brake carriage_door_state door_train_carriage

invariants
  @inv1 ∀t·t∈ dom(door_train_carriage) ⇒ t ∈ dom(door) ∧ door(t) = door_train_carriage[{t}] ∧ door(t)≠∅
  @inv2 door_train_carriage ∈ trns ↔ DOOR
  @inv3 door_train_carriage = (DOOR_CARRIAGE;train_carriage)∼
  @inv4 door_train_carriage∼∈ DOOR ⇸ trns
  @inv5 ∀t·t∈ dom(door) ∧ door(t)≠∅ ⇒ door(t) = door_train_carriage[{t}]
  @inv6 leader_carriage ∈ trns ⇸ CABIN_CARRIAGE
  @inv7 finite(leader_carriage)
  @inv8 trns_state ∈ trns → TRAIN_STATE
  @inv9 ∀t,c· t∈ran(train_carriage) ∧ trns_state(t)≠MAINTENANCE ∧ c = train_carriage∼[{t}]
        ∧ finite(CABIN_CARRIAGE) ∧ t ∈ dom(leader_carriage)
        ⇒ card(c∩CABIN_CARRIAGE)=NUMBER_CABIN_CARRIAGE ∧ leader_carriage(t) ∈ c
  @inv10 ∀t·t∈trns ∧ trns_state(t)=MAINTENANCE ⇒ speed(t)=0
  @inv11 emergency_brake ⊆trns
  @inv12 ∀t·((t∈trns ∧ speed(t)>MAX_SPEED) ⇒ t ∈ emergency_brake)
  @inv13 carriage_door_state ∈ DOOR_CARRIAGE → DOOR_STATE
  @inv14 ∀d·d ∈ dom(door_state) ∧ door_state(d)=OPEN ⇒ carriage_door_state(d↦DOOR_CARRIAGE(d))=OPEN
  @inv15 ∀d·d∈dom(door_state)∧door_state(d)=CLOSED ⇒ carriage_door_state(d↦DOOR_CARRIAGE(d))=CLOSED
   theorem @thm1 ∀c·c∈ran(DOOR_CARRIAGE) ∧ c∈dom(train_carriage)
                ⇒ DOOR_CARRIAGE∼[{c}]⊆door_train_carriage[{train_carriage(c)}]
  theorem @thm2 ∀c·c ∈ dom(train_carriage) ∧ door(train_carriage(c)) ∩ door_state∼[{OPEN}]=∅
            ∧ door(train_carriage(c))≠∅ ⇒ DOOR_CARRIAGE∼[{c}]⊆door(train_carriage(c))
            ∧ DOOR_CARRIAGE∼[{c}] ∩ door_state∼[{OPEN}]=∅
  theorem @thm3 dom(dom(carriage_door_state)) ⊆ dom(door_state)
```

(b) Variables and Invariants

```
event increaseMaxSpeed refines changeSpeed
  any t1 s1
  where
    @grd1 s1 ∈ ℕ
    @grd2 t1 ∈ dom(door_train_carriage)\braking
    @grd3 trns_state(t1) ≠ MAINTENANCE
    @grd4 s1 > MAX_SPEED
    @grd5 speed(t1)<s1
    @grd6 t1 ∉ emergency_brake
    @grd7 speed(t1)≤ MAX_SPEED
    @grd8 door_train_carriage[{t1}]
          ∩ door_state∼[{OPEN}] =∅
    @grd9 door_train_carriage[{t1}]≠∅
    @grd10 permit(t1)=TRUE
  then
    @act1 speed (t1) ≔ s1
    @act2 emergency_brake ≔emergency_brake ∪ {t1}
end

event modifyTrain refines modifyTrain
  any t state
  where
    @grd1 t ∈ trns
    @grd2 state = MAINTENANCE ⇒ speed(t)=0
    @grd3 card(train_carriage∼[{t}]∩CABIN_CARRIAGE)
           =NUMBER_CABIN_CARRIAGE
           ∧ t ∈ dom(leader_carriage)
          ∧ leader_carriage(t) ∈ train_carriage∼[{t}]
    @grd4 state ∈ TRAIN_STATE
    @grd5 state ≠ trns_state(t)
  then
    @act1 trns_state(t)≔state
end

event allocateLeaderCabinCarriageTrain
  any c
  where
    @grd1 c ∈ dom(train_carriage)
    @grd2 finite(train_carriage∼[{train_carriage(c)}])
    @grd3 c ∈ CABIN_CARRIAGE
    @grd4 c ∈ dom(train_carriage ▷ {train_carriage(c)})
    @grd5 trns_state(train_carriage(c))=MAINTENANCE
    @grd6 card(dom(train_carriage ▷ {train_carriage(c)}))
          =MAX_NUMBER_CARRIAGE
    @grd7 train_carriage(c) ∉ dom(leader_carriage)
  then
    @act1 leader_carriage(train_carriage(c)) ≔ c
end

event deallocateLeaderCabinCarriageTrain
  any t
  where
    @grd1 t ∈ dom(leader_carriage)
    @grd2 finite(train_carriage∼[{t}])
    @grd3 trns_state(t)=MAINTENANCE
    @grd4 card(dom(train_carriage ▷ {t}))
          =MAX_NUMBER_CARRIAGE
  then
    @act1 leader_carriage ≔ {t}⩤leader_carriage
end
```

(c) Some events of *Train_M2*

FIGURE 6.17: Excerpt of machine *Train_M2*

6.9 Third Refinement of *Train*: *Train_M3*

As a continuation of the refinement of the train doors by carriage, we data refine variable *door_state*. The opening doors event needs to be strengthened to specify which doors to open when a train is stopped in a platform. Figure 6.18 shows an excerpt of *Train_M3*. Some additional properties related to the allocation of the leader carriage are defined: when a train has already allocated a leader carriage, then it has the correct number of carriages ($inv2$) and the leader carriage belongs to the set of carriage of that train ($inv3$). These two invariants could have been included in the previous refinement. Nevertheless due to the high number of proof obligations already existing in the previous refinement, they were added later. Variable *door_state* disappears being refined by *door_carriage_state* and gluing invariants $inv1$ and $thm2$. Theorem $thm1$ is added to help with the proofs: the carriage doors of a train t are the same as the doors defined by the constant $DOOR\_CARRIAGE$ restricted to the carriages. Some existing events are strengthened in this refinement to be consistent with the invariants as illustrated in Fig. 6.18(b). Due to $inv2$, event $allocateLeaderCabinCarriageTrain$ needs to be strengthened by adding guard $grd8$: this event is only enabled if the number of carriages for that train is equal to $NUMBER\_CABIN\_CARRIAGE$. Also events $allocateCarriageTrain$ and $removeCarriageTrain$ require an additional guard ($grd4$ and $grd11$ respectively) stating that the events are only enabled if train t does not have a leader carriage yet. Therefore we reinforce some ordering in the events: first carriages are allocated/removed; after the leader carriage can be allocated. Refined event $openDoors$ is strengthened with the inclusion of guard $grd8$: the set of carriage doors ds that are opened are located in the same side as the $platform$.

6.10 Fourth Refinement of *Train* and Second Decompositon: *Train_M4*

The fourth refinement of *Train* corresponds to the preparation step before the decomposition. Context *Train_C4*, illustrated in Fig. 6.19(a), introduces an enumerated carrier set $TRAIN\_MOVING\_STATE$ defining the moving state of a train: $MOVING$, NOT_READY (not ready to move) and $NEUTRAL$ (not moving but ready to move). We use additional control variables to help in the separation of aspects resulting in adding variables *ready_train* and *train_doors_closed*. Both are total functions between $trns$ and BOOL ($inv1$ and $inv2$ in Fig. 6.19(b)). *ready_train* defines trains that are ready to move or moving (which therefore have a leader carriage and the correct number of carriages to move ($inv3$)); *train_doors_closed* defines trains that have all their doors closed ($inv4$). These variables are somehow redundant and are mainly added as a preparation for the shared event decomposition: they will be allocated to *LeaderCarriage* and represent a combination of states defined by *Carriage* variables. They also simplify

```
machine Train_M3 refines Train_M2  sees Train_C2

variables trns speed permit braking emergency_button train_carriage carriage_alarm leader_carriage
          trns_state emergency_brake carriage_door_state door_train_carriage

invariants
  @inv1 ∀d,ds·d ∈ dom(door_state) ∧ ds ∈ DOOR_STATE ∧ carriage_door_state(d↦DOOR_CARRIAGE(d))=ds ⇔ door_state(d)=ds
  @inv2 ∀t·t∈trns ∧ t ∈ dom(leader_carriage) ∧ card(train_carriage∼[{t}])=MAX_NUMBER_CARRIAGE
        ∧ card(train_carriage∼[{t}]∩CABIN_CARRIAGE)=NUMBER_CABIN_CARRIAGE
  @inv3 ∀t·t∈trns ∧ t ∈ dom(leader_carriage) ⇒ leader_carriage(t) ∈ train_carriage∼[{t}]
  theorem @thm1 ∀t·t∈dom(door_train_carriage) ⇒ door_train_carriage[{t}]=DOOR_CARRIAGE∼[train_carriage∼[{t}]]
  theorem @thm2 ∀d,ds·d ⊆ dom(door_state) ∧ ds ∈ DOOR_STATE ∧ carriage_door_state[d×DOOR_CARRIAGE[d]]={ds}
                ⇔door_state[d]={ds}
```

(a) Variables and invariants

```
event allocateLeaderCabinCarriageTrain
refines allocateLeaderCabinCarriageTrain
  any c
  where
    @grd1 c ∈ dom(train_carriage)
    @grd2 finite(train_carriage∼[{train_carriage(c)}])
    @grd3 c ∈ CABIN_CARRIAGE
    @grd4 c ∈ dom(train_carriage ▷ {train_carriage(c)})
    @grd5 trns_state(train_carriage(c))=MAINTENANCE
    @grd6 card(train_carriage∼[{train_carriage(c)}])
          =MAX_NUMBER_CARRIAGE
    @grd7 train_carriage(c) ∉ dom(leader_carriage)
    @grd8 card(train_carriage∼[{train_carriage(c)}]∩CABIN_CARRIAGE)
          =NUMBER_CABIN_CARRIAGE
  then
    @act1 leader_carriage(train_carriage(c)) ≔ c
end

event allocateCarriageTrain refines allocateCarriageTrain
  any c t
  where
    @grd1 c ∈ CARRIAGE\dom(train_carriage)
    @grd2 carriage_alarm[{c}]= {FALSE}
    @grd3 ∀tr·tr ∈ dom(door_train_carriage) ∧ tr≠t
          ⇒ DOOR_CARRIAGE∼[{c}]∩door_train_carriage[{tr}]=∅
    @grd4 t ∈ trns\dom(leader_carriage)
    @grd5 emergency_button(t) = FALSE
    @grd6 finite(train_carriage∼[{t}])
    @grd7 card(dom(train_carriage ▷ {t}))<MAX_NUMBER_CARRIAGE
    @grd8 speed(t)=0
    @grd9 DOOR_CARRIAGE∼[{c}] ∩ door_train_carriage[{t}]=∅
    @grd10 trns_state(t)=MAINTENANCE
  then
    @act1 train_carriage(c)≔ t
    @act2 door_train_carriage ≔ door_train_carriage
                                ∪ ({t} × DOOR_CARRIAGE∼[{c}])
    @act3 carriage_door_state≔ carriage_door_state
                             ⩤ ((DOOR_CARRIAGE▷{c})×{CLOSED})
end
```

```
event openDoors refines openDoors
  any t occpTrns platform ds
  where
    @grd1 t ∈ TRAIN
    @grd2 occpTrns ∈ ℙ(CDV)
    @grd3 platform ∈ PLATFORM
    @grd4 platform ∈ (occpTrns ∩ PLATFORM)
    @grd5 t ∈ dom((DOOR_CARRIAGE;train_carriage)∼)
    @grd6 speed(t) = 0
    @grd7 ({platform} ≠ ∅) ∨ emergency_button(t) = TRUE
    @grd8 DOOR_SIDE[ds]={PLATFORM_SIDE(platform)}
    @grd9 ds ⊆ DOOR_CARRIAGE∼[train_carriage∼[{t}]]
    @grd10 ∀d·d∈ds
            ⇒carriage_door_state[{d}◁DOOR_CARRIAGE]={CLOSED}
    @grd11 ds≠∅
  then
    @act1 carriage_door_state≔ carriage_door_state
                             ⩤ ((ds◁DOOR_CARRIAGE)×{OPEN})
end

event removeCarriageTrain refines removeCarriageTrain
  any c t
  where
    @grd1 t ∈ dom(door_train_carriage)
    @grd2 c↦t ∈ train_carriage
    @grd3 carriage_alarm(c) = FALSE
    @grd4 emergency_button(t) = FALSE
    @grd5 trns_state(t)=MAINTENANCE
    @grd6 speed(t)=0
    @grd7 carriage_door_state[DOOR_CARRIAGE▷{c}]={CLOSED}
    @grd8 ∀d·d∈DOOR_CARRIAGE∼[{c}]
          ⇒ t = door_train_carriage∼(d)
    @grd9 c ∈ ran(DOOR_CARRIAGE)
    @grd10 DOOR_CARRIAGE∼[{c}]⊆door_train_carriage[{t}]
    @grd11 t ∉ dom(leader_carriage)
  then
    @act1 train_carriage ≔ {c}⩤train_carriage
    @act2 door_train_carriage ≔ door_train_carriage
                                ⩥DOOR_CARRIAGE∼[{c}]
```

(b) Refinement of some events in *Train_M3*

FIGURE 6.18: Excerpt of machine *Train_M3*

the event splitting by replacing predicates that contain variables related to carriages. For instance, in Fig. 6.19(c) guard $grd8$ of event $increaseMaxSpeed$ replaces guard $grd8$ in the abstract event (Fig. 6.17(c)): this event does not need to refer to variable $door\_train\_carriage$ since it is only required to ensure that all the train doors are closed when a train increases its speed ($train\_doors\_closed(t1) = \text{TRUE}$). The consequence of adding these variables is that they need to be consistent throughout the events. For instance, $act2$ needs to be added to the actions of $deallocateLeaderCabinCarriageTrain$ when a leader carriage is deallocated from a train which implies that the train is no longer ready to move (Fig. 6.19(c)). Therefore these control variables should be added with care in particular when it is intended to further refine the resulting sub-events after an event decomposition. Invariants $inv5$ and $inv6$ are gluing invariants resulting from the added variables: the first states that if a train has its doors opened, then the train must be stopped; the second states that if a train is ready, then the set of carriages for that train is not empty. All other events are updated reflecting the introduction of the new variables.

```
context Train_C4 extends Train_C2

constants MOVING NOT_READY NEUTRAL

sets TRAIN_MOVING_STATE

axioms
  @axm1 partition(TRAIN_MOVING_STATE, {MOVING}, {NOT_READY}, {NEUTRAL})
end
```

(a) Context *Train_C4*

```
machine Train_M4 refines Train_M3  sees Train_C4

variables trns speed permit braking emergency_button train_carriage
carriage_alarm leader_carriage trns_state emergency_brake
carriage_door_state door_train_carriage ready_train train_doors_closed

invariants
  @inv1 ready_train ∈ trns ⟶ BOOL
  @inv2 train_doors_closed ∈ trns ⟶ BOOL
  @inv3 ∀t·t∈dom(ready_train) ∧ ready_train(t) = TRUE ⇒ t∈trns
         ∧ card(train_carriage∼[{t}])=MAX_NUMBER_CARRIAGE
         ∧ card(train_carriage∼[{t}]∩CABIN_CARRIAGE)
           =NUMBER_CABIN_CARRIAGE
         ∧ t ∈ dom(leader_carriage)
  @inv4 ∀t·t∈dom(train_doors_closed)
        ∧ train_doors_closed(t) = TRUE
        ⇒ (∀d·d ∈door_train_carriage[{t}]
            ⇒ carriage_door_state(d↦DOOR_CARRIAGE(d))≠OPEN)
  @inv5 ∀t·t∈dom(train_doors_closed)
        ∧ train_doors_closed(t) = FALSE ⇒ speed(t) = 0
  @inv6 ∀t·t∈dom(ready_train) ∧ ready_train(t) = TRUE
        ⇒ DOOR_CARRIAGE▷train_carriage∼[{t}]≠∅
```

(b) Variables and invariants

```
event increaseMaxSpeed refines increaseMaxSpeed
    any t1 s1
    where
      @grd1 s1 ∈ N
      @grd2 t1 ∈ trns
      @grd3 t1 ∉ braking
      @grd4 trns_state(t1) ≠ MAINTENANCE
      @grd5 s1 > MAX_SPEED
      @grd6 speed(t1)<s1
      @grd7 t1 ∉ emergency_brake
      @grd8 speed(t1)≤ MAX_SPEED
      @grd9 train_doors_closed(t1) = TRUE
      @grd10 permit(t1)=TRUE
      @grd11 speed(t1)>0
      @grd12 ready_train(t1) = TRUE
    then
      @act1 speed (t1) ≔ s1
      @act2 emergency_brake ≔ emergency_brake ∪ {t1}
  end
```

```
event deallocateLeaderCabinCarriageTrain
refines deallocateLeaderCabinCarriageTrain
  any t lc
  where
    @grd1 t ∈ dom(leader_carriage)
    @grd2 finite(train_carriage∼[{t}])
    @grd3 trns_state(t)=MAINTENANCE
    @grd4 card(dom(train_carriage ▷ {t}))=MAX_NUMBER_CARRIAGE
    @grd5 lc = leader_carriage
  then
    @act1 leader_carriage ≔ {t}⩤leader_carriage
    @act2 ready_train(t) ≔ FALSE
end
```

(c) Refinement of some events in *Train_M4*

FIGURE 6.19: Excerpt of machine *Train_M4*

Now we are ready to proceed to the next decomposition as described in Fig. 6.3. We want to separate the aspects related to carriages from the aspects related to leader carriages:

Leader Carriage: Allocates the leader carriage, controls the speed of the train, modifies the state of the train, receives the messages sent from the central, handles the emergency button of the train.

Carriage: Add and removes carriages, opens and closes carriage doors, handles the carriage alarm.

The decomposition is summarised in Table 6.1 (equivalent to view of Fig. 6.12 with the addition of the variable partition):

	LeaderCarriage	Carriage
Variables	$trns, permit, braking, emergency\_button$ $trns\_state, speed, emergency\_brake$ $ready\_train, train\_doors\_closed$	$carriage\_alarm, leader\_carriage$ $carriage\_door\_state, door\_train\_carriage$ $train\_carriage$
Events	$openDoors, closeDoors$ $activateEmergencyCarriageButton$ $deactivateEmergencyCarriageButton$ $deactivateEmergencyTrainButton$ $allocateLeaderCabinCarriageTrain$ $deallocateLeaderCabinCarriageTrain$ $allocateCarriageTrain$ $modifyTrain, removeCarriageTrain$ $increaseSpeed, increaseMaxSpeed$ $reduceSpeed, reduceMaxSpeed$ $recvTrainMsg, brake, stopBraking$ $addTrain, enterCDV, leaveCDV$	$openDoors, closeDoors$ $activateEmergencyCarriageButton$ $deactivateEmergencyCarriageButton$ $deactivateEmergencyTrainButton$ $allocateLeaderCabinCarriageTrain$ $deallocateLeaderCabinCarriageTrain$ $allocateCarriageTrain$ $modifyTrain, removeCarriageTrain$

TABLE 6.1: Decomposition summary of *Train_M4*

6.10.1 Machine *LeaderCarriage*

Machine *LeaderCarriage* contains the variables that are not related to the carriages (Fig. 6.20(a)). Some events are only included in this sub-component: events dealing with the speed changes, entering and leaving sections, receiving messages and adding trains. All the other events are shared between the two sub-components.

6.10.2 Machine *Carriage*

The variables related to carriages are included in sub-component *Carriage* (Fig. 6.20(b)). All the events of *Carriage* result from splitting the original events as described in Table. 6.1. We are interested in adding more details about the carriage doors, therefore we further refine *Carriage*.

6.10.3 Refinement of *Carriage* and Decomposition: *Carriage_M1*

This refinement is a preparation step before the next decomposition. We intend to use an existing generic development of carriage doors as a pattern and apply a *generic instantiation* to our model. We use the shared event decomposition to adjust our current model to fit the first machine of the pattern. *Carriage_M1* refines *Carriage* and after is decomposed in a way that one of the resulting sub-components fits the generic model of carriage doors. The generic model is described in Sect. 6.11.

Two variables are introduced in this refinement, representing the carriage doors ($carriage\_door$) and their respective state ($carriage\_ds$) as seen in Fig. 6.21(a). The last variable is used

```
machine LeaderCarriage sees LeaderCarriage_C0

variables trns speed permit braking emergency_button trns_state
          emergency_brake ready_train train_doors_closed

invariants
  theorem @typing_train_doors_closed train_doors_closed ∈ ℙ(TRAIN × BOOL)
  @Train_MetroSystem_M0_inv3 trns ⊆ TRAIN
  @Train_MetroSystem_M0_inv9 braking ⊆ trns
  @Train_MetroSystem_M0_inv10 speed ∈ trns ⟶ ℕ
  @Train_MetroSystem_M1_inv2 permit ∈ trns ⟶ BOOL
  @Train_MetroSystem_M1_inv3 emergency_button ∈ trns ⟶ BOOL
  @Train_M1_inv1 finite(trns)
  @Train_M2_inv8 trns_state ∈ trns ⟶ TRAIN_STATE
  @Train_M2_inv10 ∀t·t∈trns ∧ trns_state(t)=MAINTENANCE ⇒ speed(t)=0
  @Train_M2_inv11 emergency_brake ⊆trns
  @Train_M2_inv12 ∀t·((t∈trns ∧ speed(t)>MAX_SPEED) ⇒ t ∈ emergency_brake)
  @Train_M4_inv14 ready_train ∈ trns ⟶ BOOL
  @Train_M4_inv16 train_doors_closed ∈ trns ⟶ BOOL
  @Train_M4_inv18 ∀t·t∈dom(train_doors_closed) ∧ train_doors_closed(t) = FALSE
                  ⇒ speed(t) = 0
  theorem @WD_Train_M4_inv6 ∀t·t∈dom(ready_train)⇒ready_train∈TRAIN ⇸ BOOL
```

(a) sub-component *LeaderCarriage*

```
machine Carriage sees Carriage_C0

variables train_carriage carriage_alarm leader_carriage carriage_door_state
          door_train_carriage

invariants
  theorem @typing_leader_carriage leader_carriage ∈ ℙ(TRAIN × CARRIAGE)
  theorem @typing_door_train_carriage door_train_carriage ∈ ℙ(TRAIN × DOOR)
  theorem @typing_train_carriage train_carriage ∈ ℙ(CARRIAGE × TRAIN)
  theorem @typing_carriage_alarm carriage_alarm ∈ ℙ(CARRIAGE × BOOL)
  @Train_M1_inv2 carriage_alarm ∈ CARRIAGE ⟶ BOOL
  @Train_M1_inv4 finite(train_carriage)
  @Train_M1_inv5 finite(dom(train_carriage))
  @Train_M2_inv3 door_train_carriage = (DOOR_CARRIAGE;train_carriage)~
  @Train_M2_inv7 finite(leader_carriage)
  @Train_M2_inv13 carriage_door_state ∈ DOOR_CARRIAGE ⟶ DOOR_STATE
  theorem @Train_M2_thm1 ∀c·c∈ran(DOOR_CARRIAGE) ∧ c∈dom(train_carriage)
        ⇒ DOOR_CARRIAGE~[{c}]⊆door_train_carriage[{train_carriage(c)}]
  theorem @Train_M3_thm1 ∀t·t∈dom(door_train_carriage)
        ⇒ door_train_carriage[{t}]=DOOR_CARRIAGE~[train_carriage~[{t}]]
```

(b) sub-component *Carriage*

FIGURE 6.20: Variables and invariants of *LeaderCarriage* and *Carriage*

to data refine *carriage_door_state* that disappears. The gluing invariant for this data refinement is expressed by *inv*4: the state of all the doors in *carriage_ds* match the state of the same door in *carriage_door_state*. As a result, some events need to be refined to fit the new variables. For instance, in Fig. 6.21(b), *act*1 in event *openDoors* updates variable *carriage_ds* instead of the abstract variable *carriage_door_state*. Also when carriage doors are allocated, both new variables are assigned as seen in actions *act*3 and *act*4 of event *allocateCarriageTrain* (similar for *removeCarriageTrain*).

Comparing with the generic model of carriage doors, the relevant events to fit the instantiation are *openDoors*, *closeDoors*, *allocateCarriageTrain* and *removeCarriageTrain*. Not by coincidence, these events manipulate variables *carriage_ds* and *carriage_door* that will instantiate generic variables *generic_door_state* and *generic_door* respectively. The decomposition summary is described in Table 6.2.

6.10.4 Machine *CarriageInterface*

Machine *CarriageInterface* contains the variables that are not related to the carriage doors. This machine handles the activation/deactivation of the carriage alarm, the deac-

```
machine Carriage_M1 refines Carriage sees Carriage_C0

variables carriage_alarm leader_carriage train_carriage carriage_door carriage_ds door_train_carriage

invariants
  @inv1 carriage_door ⊆ DOOR
  @inv2 carriage_ds ∈ carriage_door → DOOR_STATE
  @inv3 ∀c·c∈dom(train_carriage) ⇒ DOOR_CARRIAGE~[{c}]⊆carriage_door
  @inv4 ∀d,c·d↦c∈dom(carriage_door_state) ∧ d ∈ dom(carriage_ds) ∧ d∈ran(door_train_carriage)
        ⇒carriage_ds(d)=carriage_door_state(d↦c)
  @inv5 door_train_carriage~∈DOOR ⇸ TRAIN
  @inv6 ∀d·d∈ran(door_train_carriage) ⇒ d ∈ carriage_door
```

(a) Variables and invariants

```
event openDoors refines openDoors
  any t occpTrns platform ds
  where
    @typing_platform platform ∈ CDV
    @typing_ds ds ∈ ℙ(DOOR)
    @grd1 t ∈ TRAIN
    @grd2 occpTrns ∈ ℙ(CDV)
    @grd3 platform ∈ PLATFORM
    @grd4 platform ∈ (occpTrns ∩ PLATFORM)
    @grd5 t ∈ dom((DOOR_CARRIAGE;train_carriage)~)
    @grd6 DOOR_SIDE[ds]={PLATFORM_SIDE(platform)}
    @grd7 ds ⊆ DOOR_CARRIAGE~[train_carriage~[{t}]]
    @grd8 ds ⊆ dom(carriage_ds)
    @grd9 carriage_ds[ds]={CLOSED}
  then
    @act1 carriage_ds≔carriage_ds⩤ (ds×{OPEN})
end

event closeDoors refines closeDoors
  any t ds closed cds
  where
    @typing_closed closed ∈ BOOL
    @typing_ds ds ∈ ℙ(DOOR)
    @grd1 t ∈ TRAIN
    @grd2 t ∈ dom(((train_carriage~);(DOOR_CARRIAGE~)))
    @grd3 ds ⊆ ((train_carriage~);(DOOR_CARRIAGE~))[{t}]
    @grd4 cds = carriage_ds
    @grd5 (∃d·d∈DOOR_CARRIAGE~[train_carriage~[{t}]]\ds
           ∧ cds(d)≠CLOSED) ⇔ closed = FALSE
    @grd6 ds ⊆ dom(carriage_ds)
    @grd7 carriage_ds[ds]={OPEN}
  then
    @act2 carriage_ds≔carriage_ds ⩤ (ds×{CLOSED})
end
```

```
event allocateCarriageTrain refines allocateCarriageTrain
  any c t ds
  where
    @typing_t t ∈ TRAIN
    @typing_c c ∈ CARRIAGE
    @grd1 c ∈ CARRIAGE\dom(train_carriage)
    @grd2 carriage_alarm[{c}]= {FALSE}
    @grd3 t ∈ dom(door_train_carriage)
    @grd4 ∀tr·tr ∈ dom(door_train_carriage) ∧ tr≠t
          ⇒ DOOR_CARRIAGE~[{c}]∩door_train_carriage[{tr}]=∅
    @grd5 finite(train_carriage~[{t}])
    @grd6 card(dom(train_carriage ▷ {t}))<MAX_NUMBER_CARRIAGI
    @grd7 DOOR_CARRIAGE~[{c}] ∩ door_train_carriage[{t}]=∅
    @grd8 t∉dom(leader_carriage)
    @grd9 ds = DOOR_CARRIAGE~[{c}]
    @grd10 ds∩dom(carriage_ds)=∅
  then
    @act1 train_carriage(c)≔ t
    @act2 door_train_carriage ≔ door_train_carriage
          ∪ ({t} × DOOR_CARRIAGE~[{c}])
    @act3 carriage_door ≔ carriage_door ∪ ds
    @act4 carriage_ds ≔ carriage_ds ∪ (ds×{CLOSED})
end

event removeCarriageTrain refines removeCarriageTrain
  any c t ds
  where
    @typing_t t ∈ TRAIN
    @typing_c c ∈ CARRIAGE
    @grd1 t ∈ dom(door_train_carriage)
    @grd2 c↦t ∈ train_carriage
    @grd3 carriage_alarm(c) = FALSE
    @grd16 t ∈ dom(door_train_carriage)
    @grd10 ∀d·d∈DOOR_CARRIAGE~[{c}]
           ⇒ t = door_train_carriage~(d)
    @grd11 c ∈ ran(DOOR_CARRIAGE)
    @grd12 t ∉ dom(leader_carriage)
    @grd13 ds = DOOR_CARRIAGE~[{c}]
    @grd14 ds⊆carriage_door
    @grd15 carriage_ds[DOOR_CARRIAGE~[{c}]] = {CLOSED}
  then
    @act1 train_carriage ≔ {c}⩤train_carriage
    @act2 door_train_carriage ≔
              door_train_carriage ⩥DOOR_CARRIAGE~[{c}]
    @act3 carriage_door ≔ carriage_door \ ds
    @act4 carriage_ds ≔ ds⩤carriage_ds
end
```

(b) Refinement of some events in *Carriage_M1*

FIGURE 6.21: Excerpt of machine *Carriage_M1*

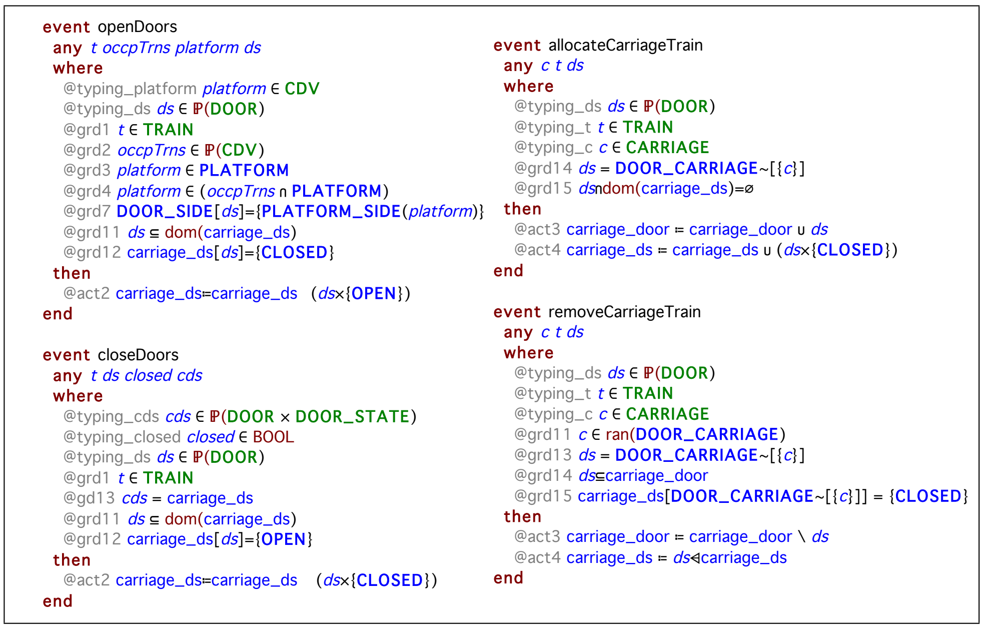
tivation of the emergency button and the allocation/deallocation of the leader cabin carriage. Events *openDoors*, *closeDoors*, *allocateCarriageTrain* and *removeCarriageTrain* are shared with *CarriageDoor*.

6.10.5 Machine *CarriageDoor*

CarriageDoors contains the variables related to carriage doors and the events resulting from splitting the original events as described in Table 6.2. The resulting sub-events can be seen in Fig. 6.22.

	CarriageInterface	CarriageDoor
Variables	$carriage\_alarm, leader\_carriage$ $train\_carriage, door\_train\_carriage$	$carriage\_doors, carriage\_ds$
Events	$openDoors, closeDoors$ $allocateCarriageTrain$ $removeCarriageTrain$ $activateEmergencyCarriageButton$ $deactivateEmergencyCarriageButton$ $deactivateEmergencyTrainButton$ $allocateLeaderCabinCarriageTrain$ $deallocateLeaderCabinCarriageTrain$ $modifyTrain$	$openDoors, closeDoors$ $allocateCarriageTrain$ $removeCarriageTrain$

TABLE 6.2: Decomposition summary of *Carriage_M1*



```
event openDoors
 any t occpTrns platform ds
 where
  @typing_platform platform ∈ CDV
  @typing_ds ds ∈ ℙ(DOOR)
  @grd1 t ∈ TRAIN
  @grd2 occpTrns ∈ ℙ(CDV)
  @grd3 platform ∈ PLATFORM
  @grd4 platform ∈ (occpTrns ∩ PLATFORM)
  @grd7 DOOR_SIDE[ds]={PLATFORM_SIDE(platform)}
  @grd11 ds ⊆ dom(carriage_ds)
  @grd12 carriage_ds[ds]={CLOSED}
 then
  @act2 carriage_ds≔carriage_ds  (ds×{OPEN})
end

event closeDoors
 any t ds closed cds
 where
  @typing_cds cds ∈ ℙ(DOOR × DOOR_STATE)
  @typing_closed closed ∈ BOOL
  @typing_ds ds ∈ ℙ(DOOR)
  @grd1 t ∈ TRAIN
  @gd13 cds = carriage_ds
  @grd11 ds ⊆ dom(carriage_ds)
  @grd12 carriage_ds[ds]={OPEN}
 then
  @act2 carriage_ds≔carriage_ds  (ds×{CLOSED})
end

event allocateCarriageTrain
 any c t ds
 where
  @typing_ds ds ∈ ℙ(DOOR)
  @typing_t t ∈ TRAIN
  @typing_c c ∈ CARRIAGE
  @grd14 ds = DOOR_CARRIAGE∼[{c}]
  @grd15 ds∩dom(carriage_ds)=∅
 then
  @act3 carriage_door ≔ carriage_door ∪ ds
  @act4 carriage_ds ≔ carriage_ds ∪ (ds×{CLOSED})
end

event removeCarriageTrain
 any c t ds
 where
  @typing_ds ds ∈ ℙ(DOOR)
  @typing_t t ∈ TRAIN
  @typing_c c ∈ CARRIAGE
  @grd11 c ∈ ran(DOOR_CARRIAGE)
  @grd13 ds = DOOR_CARRIAGE∼[{c}]
  @grd14 ds⊆carriage_door
  @grd15 carriage_ds[DOOR_CARRIAGE∼[{c}]] = {CLOSED}
 then
  @act3 carriage_door ≔ carriage_door \ ds
  @act4 carriage_ds ≔ ds⩤carriage_ds
end
```

FIGURE 6.22: Events of sub-component *CarriageDoors*

There are two kind of carriage doors: *emergency doors* and *service doors*. We intend to instantiate twice the generic doors development, one per kind of door (the developments are similar for both kind of doors). Specific details for each kind of door are added as additional refinements later on. We describe the generic model and afterwards the instantiation.

6.11 Generic Model: *GCDoor*

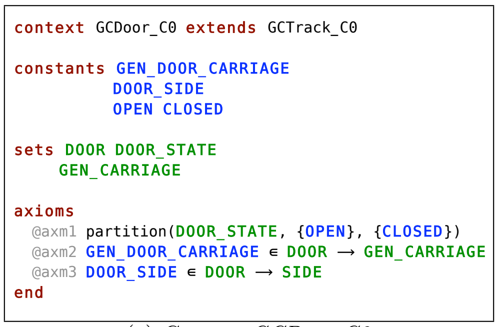
The generic model for the carriage doors is based in three refinements: *GCDoor_M0*, *GCDoor_M1* and *GCDoor_M2*. In each refinement step, more requirements and details are introduced.

6.11.1 Abstract machine *GCDoor_M0*

We start by adding the carriage doors and respective states. Four events model carriage doors. The properties to be preserved are:

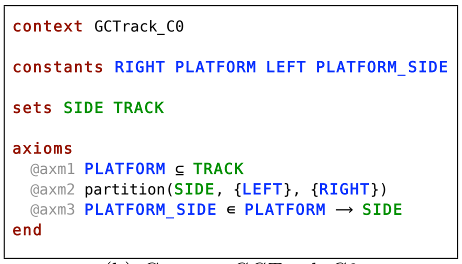
1. Doors can be added or removed.
2. Doors can be in an opening or closing state. Doors can only be open if the train is in a platform.
3. When adding/removing doors, they are closed by default for safety reasons.

The static part of the generic development is initially divided in two parts: context *GCDoor_C0* for the doors and context *GCTrack_C0* for the tracks as seen in Fig. 6.23.



```
context GCDoor_C0 extends GCTrack_C0

constants GEN_DOOR_CARRIAGE
          DOOR_SIDE
          OPEN CLOSED

sets DOOR DOOR_STATE
     GEN_CARRIAGE

axioms
  @axm1 partition(DOOR_STATE, {OPEN}, {CLOSED})
  @axm2 GEN_DOOR_CARRIAGE ∈ DOOR → GEN_CARRIAGE
  @axm3 DOOR_SIDE ∈ DOOR → SIDE
end
```

(a) Context *GCDoor_C0*



```
context GCTrack_C0

constants RIGHT PLATFORM LEFT PLATFORM_SIDE

sets SIDE TRACK

axioms
  @axm1 PLATFORM ⊆ TRACK
  @axm2 partition(SIDE, {LEFT}, {RIGHT})
  @axm3 PLATFORM_SIDE ∈ PLATFORM → SIDE
end
```

(b) Context *GCTrack_C0*

FIGURE 6.23: Generic contexts

Context *GCDoor_C0* contains sets $DOOR$, $DOOR\_STATE$ and $GEN\_DOOR\_CARRIAGE$, representing carriage doors, defining if a door is opened or closed and defining the carriages to which a door belongs to, respectively. Context *GCTrack_C0* contains sets $SIDE$ and $TRACK$, defining the side ($LEFT$ or $RIGHT$) of a door or platform and each section of the track, respectively. Machine *GCDoor_M0* contains variables *generic_door* and *generic_door_state*. The invariants of this abstraction are quite weak since we just add the type variables as can be seen in Fig. 6.24(a).

Property 1 is expressed by events $addDoor$ and $removeDoor$. Property 2 is expressed by variable *generic_door_state* and events $openDoors$ and $closeDoors$. Event $openDoors$ is only enabled if the set of doors ds is closed and if the parameter $occpTrns$, corresponding to the sections occupied by the carriage, intersects a platform. Doors are removed in event $removeDoor$, if they are $CLOSED$ confirming property 3. Next section describes the refinement of this machine.

```
machine GCDoor_M0 sees GCDoor_C0

variables generic_door generic_door_state

invariants
  @inv1 generic_door ⊆ DOOR
  @inv2 generic_door_state ∈ generic_door → DOOR_STATE

event openDoors
    any ds platform occpTrns
    where
      @grd ds ⊆ DOOR
      @grd1 ds ⊆ dom(generic_door_state)
      @grd2 generic_door_state[ds]={CLOSED}
      @grd3 platform ∈ PLATFORM
      @grd4 platform ∈ (occpTrns ∩ PLATFORM)
      @grd5 ds ≠∅
      @grd6 DOOR_SIDE[ds]={PLATFORM_SIDE(platform)}
    then
      @act1 generic_door_state≔generic_door_state  (ds×{OPEN})
  end
```

(a) Variables, invariants and event *openDoors*

```
event closeDoors
  any ds
  where
    @grd ds ⊆ DOOR
    @grd1 ds ⊆ dom(generic_door_state)
    @grd2 generic_door_state[ds]={OPEN}
    @grd3 ds ≠∅
  then
    @act1 generic_door_state≔generic_door_state
                                           (ds×{CLOSED})
end

event addDoor
  any ds c
  where
    @grd1 ds ∩ generic_door = ∅
    @grd2 ds ≠ ∅
    @grd3 ds = GEN_DOOR_CARRIAGE∼[{c}]
  then
    @act1 generic_door ≔ generic_door ∪ ds
    @act2 generic_door_state ≔ generic_door_state
                                           ∪ (ds×{CLOSED})
end

event removeDoor
  any ds c
  where
    @grd1 ds ⊆ generic_door
    @grd2 ds ≠ ∅
    @grd3 generic_door_state[ds]={CLOSED}
    @grd4 ds = GEN_DOOR_CARRIAGE∼[{c}]
  then
    @act1 generic_door ≔ generic_door \ ds
    @act2 generic_door_state ≔
                      ds⩤generic_door_state
end
```

(b) Some events in *GCDoors_M0*

FIGURE 6.24: Machine *GCDoors_M0*

6.11.2 Second refinement of *GCDoor*: *GCDoor_M1*

In this refinement more details are introduced about the possible behaviour of the doors. The properties to be preserved are:

1. The actions involving the doors may result from *commands* sent from the central door control. These commands have a type ($OPEN\_RIGHT\_DOORS$, $OPEN\_LEFT\_DOORS$, $CLOSE\_RIGHT\_DOORS$, $CLOSE\_LEFT\_DOORS$, $ISOLATE\_DOORS$, $REMOVE\_ISOLATION\_DOORS$), a state ($START$, $FAIL$, $SUCCESS$ and $EXECUTED$) and a target (set of doors).

2. After the doors are closed, they must be locked for the train to move.

3. If a door is open, then an opening device was used: $MANUAL\_PLATFORM$ if opened manually in a platform, $MANUAL\_INTERNAL$ if opened inside the carriage manually and $AUTOMATIC\_CENTRAL\_DOOR$ if opened automatically from the central control.

4. Doors can get obstructed when closed automatically (people/object obstruction). If an obstruction is detected then it should be tried to close the doors again.

The context used in this refinement ($GCDoor\_C1$) extends the existing one as seen in Fig. 6.25(a). Abstract events are refined to include the properties defined above. Some

```
context GCDoor_C1 extends GCDoor_C0

constants MANUAL_PLATFORM MANUAL_INTERNAL AUTOMATIC_CENTRAL_DOOR START FAIL SUCCESS EXECUTED
OPEN_RIGHT_DOORS OPEN_LEFT_DOORS CLOSE_RIGHT_DOORS CLOSE_LEFT_DOORS ISOLATE_DOORS REMOVE_ISOLATION_DOORS

sets OPENING_DEVICE COMMAND_STATE COMMAND_TYPE COMMAND

axioms
  @axm1 partition(OPENING_DEVICE, {MANUAL_PLATFORM}, {MANUAL_INTERNAL}, {AUTOMATIC_CENTRAL_DOOR})
  @axm2 partition(COMMAND_STATE, {START}, {FAIL}, {SUCCESS},{EXECUTED})
  @axm3 partition(COMMAND_TYPE, {OPEN_RIGHT_DOORS}, {OPEN_LEFT_DOORS}, {CLOSE_RIGHT_DOORS},
                               {CLOSE_LEFT_DOORS}, {ISOLATE_DOORS}, {REMOVE_ISOLATION_DOORS})
end
```

(a) Context *GCDoors_C1*

```
machine GCDoor_M1 refines GCDoor_M0  sees GCDoor_C1

variables generic_door generic_door_state locked_doors door_opening_device obstructed_door command
          command_doors command_type command_state

invariants
  @inv1 locked_doors ⊆ DOOR
  @inv2 ∀d·d∈locked_doors ∧ d ∈ dom(generic_door_state) ⇒ generic_door_state(d)∉{OPEN}
  @inv3 door_opening_device ∈ generic_door ⇸  OPENING_DEVICE
  @inv4 ∀d·d∈generic_door ∧ generic_door_state(d)=OPEN ⇒d∈dom(door_opening_device)
  @inv5 obstructed_door ⊆ dom(generic_door_state)
  @inv6 command ⊆ COMMAND
  @inv7 command_type ∈ command → COMMAND_TYPE
  @inv8 command_state ∈ command → COMMAND_STATE
  @inv9 command_doors ∈ command → ℙ(generic_door)
  @inv10 ∀dos·dos∈ran(command_doors) ⇒ dos ≠∅
  @inv11 ∀d,opDev·d ∈ generic_door ∧ opDev ∈ OPENING_DEVICE ∧ (d↦opDev)∈door_opening_device
         ∧ opDev=AUTOMATIC_CENTRAL_DOOR (∃cmd·cmd∈command ∧ d ∈ command_doors(cmd))
```

(b) Variables, invariants

FIGURE 6.25: Excerpt of machine *GCDoors_M1*

new invariants are added as seen in Fig. 6.25(b). Property 1 is defined by new variables *command*, *command_type*, *command_state* and *command_doors* (see invariants *inv*6 to *inv*9). Property 2 is defined by invariant *inv*2 (if a door is locked, then the door is not opened) and events *lockDoor*/*unlockDoor*. Property 3 is defined by variables *door_opening_device*, *inv*3 and *inv*11 (if a door is opened automatically, then a command has been issued to do so). Property 4 is defined by variable *obstructed_door*, *inv*5 and events *doorIsObstructed* and *closeObstructedDoor*. The system works as follows: doors can be opened/closed manually or automatically. To open/close a door automatically, a command must be issued from the central door control defining which doors are affected (for instance, to open a door automatically, event *commandOpenDoors* needs to occur). A command starts with state *START* which can lead to a successful result (*SUCCESS*) or failure (*FAIL*). Either way, it finishes with state *EXECUTED*. Abstract event *otherCommandDoors* refers to commands not defined in this refinement. If a door gets obstructed when being closed automatically (event *doorIsObstructed*) then event *closeObstructedDoor* models a successful attempt to close an obstructed door. Otherwise, it needs to be closed manually.

The system works as follows: doors can be opened/closed manually or automatically. If it is done automatically, a command sent from the central door control is issued defining which doors are affected (for instance, event *commandOpenDoors*, illustrated in Fig. 6.26, issues a command to open a set of doors automatically). Event *otherCommandDoors* is left abstract the enough in order to refer to commands not defined in this refinement. If a door gets obstructed when closing automatically (event *doorIsObstructed*) then

```
event commandOpenDoors
    any doors cmd cmd_type
    where
      @grd doors ⊆ generic_door
      @grd1 generic_door_state[doors]={CLOSED}
      @grd2 cmd_type
          ∈ {OPEN_RIGHT_DOORS,OPEN_LEFT_DOORS}
      @grd3 cmd ∈ COMMAND\command
      @grd4 doors ≠∅
    then
      @act1 command_state(cmd)≔START
      @act2 command_doors(cmd)≔doors
      @act3 command ≔ command ∪ {cmd}
      @act4 command_type(cmd)≔cmd_type
  end

  event otherCommandDoors
    any doors cmd cmd_type
    where
      @grd doors ⊆ generic_door
      @grd1 cmd_type ∈ COMMAND_TYPE
      @grd3 cmd ∈ COMMAND\command
      @grd4 doors ≠∅
    then
      @act1 command_state(cmd)≔START
      @act2 command_doors(cmd)≔doors
      @act3 command ≔ command ∪ {cmd}
      @act4 command_type(cmd)≔cmd_type
  end
```

```
event doorIsObstructed
    any ds cmd
    where
      @grd ds ⊆  DOOR\(locked_doors ∪ obstructed_door)
      @grd1 ds ⊆ dom(generic_door_state)
      @grd2 cmd ∈ command
      @grd3 command_type(cmd)
          ∈ {CLOSE_RIGHT_DOORS,CLOSE_LEFT_DOORS}
      @grd4 command_state(cmd)∈{START,FAIL}
      @grd5 ds ⊆ command_doors(cmd)
      @grd6 ds ≠∅
      @grd7 generic_door_state[ds]={OPEN}
    then
      @act1 obstructed_door ≔ obstructed_door ∪ ds
      @act2 command_state(cmd)≔FAIL
  end
```

```
event openDoorAutomatically
refines openDoors
 any ds cmd
 where
  @grd ds ⊆ generic_door\locked_doors
  @grd1 ds ⊆ dom(generic_door_state)
  @grd2 generic_door_state[ds]={CLOSED}
  @grd3 cmd ∈ command
  @grd4 command_type(cmd) ∈
    {OPEN_RIGHT_DOORS,OPEN_LEFT_DOORS}
  @grd5 command_state(cmd)=START
  @grd6 ds ⊆ command_doors(cmd)
  @grd7 ds ≠∅
 then
  @act1 generic_door_state≔
          generic_door_state ⩤ (ds×{OPEN})
  @act2 door_opening_device ≔ door_opening_device
                  ⩤ (ds×{AUTOMATIC_CENTRAL_DOOR})
 end
event lockDoor
  any d
  where
    @grd d ∈ generic_door\locked_doors
    @grd1 generic_door_state(d)=CLOSED
  then
    @act1 locked_doors≔locked_doors ∪ {d}
end

event unlockDoor
  any d
  where
    @grd1 d ∈ generic_door
    @grd2 d ∈ locked_doors
  then
    @act1 locked_doors≔locked_doors \ {d}
end


event closeObstructedDoor
refines closeDoors
 any ds cmd st
 where
    @grd ds ⊆ obstructed_door
    @grd1 ds ⊆ dom(generic_door_state)
    @grd2 cmd ∈ command
    @grd3 command_type(cmd)∈
    {CLOSE_RIGHT_DOORS,CLOSE_LEFT_DOORS}
    @grd4 command_state(cmd)=FAIL
    @grd5 ds ⊆ command_doors(cmd)
    @grd6 ds ≠∅
    @grd7 generic_door_state[ds]={OPEN}
    @grd8 st ∈ {SUCCESS,FAIL}
    @grd9 st = SUCCESS ⇔ command_doors(cmd)\ds=∅
      ∨ generic_door_state[command_doors(cmd)\ds]
      ={CLOSED}
  then
    @act1 generic_door_state≔
          generic_door_state⩤(ds×{CLOSED})
    @act2 obstructed_door ≔ obstructed_door \ ds
    @act3 command_state(cmd)≔st
end
```

FIGURE 6.26: Some events in *GCDoors_M1*

event *closeObstructedDoor* models a successful attempt to close an obstructed door. Otherwise, it needs to be closed manually.

6.12 Third refinement of *GCDoor*: *GCDoor_M2*

In the third refinement, malfunctioning doors can be isolated and in that case, they ignore the commands issued by the central command. Isolated doors can be either opened or closed. After the execution of a command, the corresponding state is updated according to the success/failure of the command. The properties to be preserved are:

1. Doors can be isolated (independently of the respective door state) in case of malfunction or safety reasons.

2. If a command is successful, it means that the command already occurred.

3. Two commands cannot have the same door as target except if the command has already been executed.

4. If a door is obstructed, then it must be in a state corresponding to $OPEN$.

The properties to be preserved are mainly defined as invariants. Property 1 is defined by new variable $isolated\_door$, $inv1$, $inv6$ and events $commandIsolationDoors$, $isolateDoor$ and $removeIsolatedDoor$ as seen in Fig. 6.27(b). Property 2 is defined by several invariants depending on the command: $inv2$ for opening doors, $inv3$ for closing doors, $inv4$ to isolate doors, $inv5$ to lift the isolation from a door. Property 3 is defined by $inv7$ and the last property by $inv8$.

An excerpt of *GCDoors_M2* is depicted in Fig. 6.27. New event $commandIsolationDoors$ models a command to add/remove doors from isolation refining the abstract event $otherCommandDoors$. After this command is issued, the actual execution (or not) of the command dictates the command state at refined event *updateIsolationCmdState*. A command log is created corresponding to the end of the command's task in event *executeLogCmdState*. Other commands could be added in a similar manner but we restrict to these commands for now. The state update of other commands (opening and closing doors) follows the same behaviour as the isolation one.

This model has three refinement layers with all the proof obligations discharged. We instantiate this model, benefiting from the discharged proof obligations and refinements to model emergency and service doors.

6.13 Instantiation of Generic Carriage Door

We use the $GCDoor$ development as a pattern to model emergency and service doors. The instantiation is similar for both kind of doors: specific details for each type of door are added later. We abstract ourselves from these details and focus in the instantiation of one of the doors: *emergency doors*.

The pattern context is defined by contexts $GCDoor\_C0$ (and context $GCTrack\_C0$) in Fig. 6.23 and $GCDoor\_C1$ in Fig. 6.25(a). The parameterisation context seen by the instance results from the context seen by the abstract machine $CarriageDoors$ as illustrated in Fig. 6.28(a). *CarriageDoors_C0* does not contain all the sets and constants that need to be instantiated. Therefore *CarriageDoors_C1* is created based on the pattern context *GCDoor_C1* (Fig. 6.28(b)).

```
machine GCDoor_M2 refines GCDoor_M1  sees GCDoor_C1

variables generic_door generic_door_state isolated_door locked_doors door_opening_device obstructed_door
          command command_doors command_type command_state

invariants
  @inv1 isolated_door ⊆ DOOR
  @inv2 ∀cmd,d· cmd ∈ command ∧ command_type(cmd)∈{OPEN_RIGHT_DOORS,OPEN_LEFT_DOORS}
        ∧d ∈ DOOR∧d ∈ command_doors(cmd)∧command_state(cmd)=SUCCESS ∧ d ∉ isolated_door⇒ generic_door_state(d)=OPEN
  @inv3 ∀cmd,d· cmd ∈ command ∧ command_type(cmd)∈{CLOSE_RIGHT_DOORS,CLOSE_LEFT_DOORS}
        ∧ d ∈ DOOR ∧ d ∈ command_doors(cmd)∧command_state(cmd)=SUCCESS∧d ∉ isolated_door⇒ generic_door_state(d)=CLOSED
  @inv4 ∀cmd,d· cmd ∈ command ∧ command_type(cmd)=ISOLATE_DOORS ∧ d ∈ DOOR
        ∧ d ∈ command_doors(cmd) ∧ command_state(cmd)=SUCCESS ⇒ d∈ isolated_door
  @inv5 ∀cmd,d· cmd ∈ command ∧ command_type(cmd)=REMOVE_ISOLATION_DOORS
        ∧ d ∈ DOOR ∧ d ∈ command_doors(cmd) ∧ command_state(cmd)=SUCCESS ⇒ d∉ isolated_door
  @inv6 ∀d· d∈isolated_door ∧ d ∈ dom(generic_door_state)⇒  generic_door_state(d)∈{OPEN, CLOSED}
  @inv7 ∀cmd1,cmd2· cmd1∈command ∧ cmd2∈command ∧ cmd1≠cmd2
        ∧ command_state(cmd1)≠EXECUTED ∧ command_state(cmd2)≠EXECUTED ⇒command_doors(cmd1)∩command_doors(cmd2)=∅
  @inv8 ∀d· d∈obstructed_door ⇒ generic_door_state(d)=OPEN
```

(a) Variables, invariants

```
event commandIsolationDoors refines otherCommandDoors
   any doors cmd cmd_type
   where
     @grd doors ⊆ generic_door
     @grd1 cmd_type ∈ {ISOLATE_DOORS,REMOVE_ISOLATION_DOORS}
     @grd2 cmd ∈ COMMAND\command
     @grd3 ∀cmd1· cmd1∈command
              ∧ command_state(cmd1)≠EXECUTED
              ⇒doors∩command_doors(cmd1)=∅
     @grd4 doors ≠∅
     @grd5 cmd_type = ISOLATE_DOORS ⇔ (doors∩isolated_door = ∅ )
     @grd6 cmd_type = REMOVE_ISOLATION_DOORS ⇔ isolated_door≠∅
            ∧ doors∩isolated_door≠∅
   then
     @act1 command_state(cmd)≔START
     @act2 command_doors(cmd)≔doors
     @act3 command ≔ command ∪ {cmd}
     @act4 command_type(cmd)≔cmd_type
 end

 event updateIsolationCmdState refines updateCmdState
   any state cmd
   where
     @grd cmd ∈ command
     @grd1 state ∈ COMMAND_STATE\{START,EXECUTED}
     @grd2 command_state(cmd)=START
     @grd3 command_type(cmd)
             ∈ {ISOLATE_DOORS,REMOVE_ISOLATION_DOORS}
     @grd4 (command_type(cmd) = ISOLATE_DOORS
           ∧ (∃d·d∈command_doors(cmd) ∧ d ∉isolated_door))
           ∨ (command_type(cmd) = REMOVE_ISOLATION_DOORS
           ∧ (∃d·d∈command_doors(cmd) ∧ d ∈isolated_door))
           ⇔ state = FAIL
   then
     @act1 command_state(cmd)≔state
 end
```

```
event executedLogCmdState refines updateCmdState
   any cmd
   where
     @guard3 cmd ∈ command
     @guard1 command_state(cmd)∈{FAIL,SUCCESS}
   with
     @state state = EXECUTED
   then
     @act1 command_state(cmd)≔EXECUTED
 end

 event isolateDoor
   any d cmd
   where
     @grd d ∈ generic_door\isolated_door
     @grd1 cmd ∈ command
     @grd2 command_state(cmd)=START
     @grd3 d ∈ command_doors(cmd)
     @grd4 command_type(cmd) = ISOLATE_DOORS
     @grd5 generic_door_state(d)∈{OPEN, CLOSED}
   then
     @act1 isolated_door≔ isolated_door ∪ {d}
 end

 event removeIsolatedDoor
   any d cmd
   where
     @grd d ∈ isolated_door
     @grd1 cmd ∈ command
     @grd3 d ∈ command_doors(cmd)
     @grd4 command_type(cmd) = REMOVE_ISOLATION_DOORS
     @grd2 command_state(cmd)=START
     @grd5 generic_door_state(d)∈{OPEN, CLOSED}
   then
     @act1 isolated_door≔ isolated_door \ {d}
 end
```

(b) Some events in *GCDoor_M2*

FIGURE 6.27: Excerpt of machine *GCDoor_M2*

Following the steps suggested in Sect. 3.5.2, we create the instantiation refinement for emergency carriage doors as seen in Fig. 6.29. As expected, the generic sets and constants are replaced by the instance sets existing in contexts *CarriageDoors_C0* and *CarriageDoors_C1*. Moreover, generic variables are renamed to fit the instance and be a refinement of abstract machine *CarriageDoors*. The same happens to generic events *addDoor* and *removeDoor*.

Comparing the abstract machine of the pattern *GCDoor_M0* and the last refinement of our initial development *CarriageDoors*, we realise that they are similar but not a perfect match. *CarriageDoors* events contains some additional parameters and guards resulting from the previous refinements. For instance, event *closeDoors* in *CarriageDoors* (Fig. 6.30(b)) contains an additional parameter *cds* compared to event *closeDoors* in

```
context CarriageDoor_C0

constants PLATFORM DOOR_SIDE PLATFORM_SIDE CLOSED OPEN
          DOOR_CARRIAGE

sets DOOR DOOR_STATE CDV SIDE CARRIAGE

axioms
  @MetroSystem_C1_axm1 partition(DOOR_STATE, {OPEN}, {CLOSED})
  @MetroSystem_C1_axm2 PLATFORM ⊆ CDV
  @Train_C1_axm2 DOOR_CARRIAGE ∈ DOOR → CARRIAGE
  @Train_C1_axm3 ∀c·c∈ran(DOOR_CARRIAGE)⇒DOOR_CARRIAGE∼[{c}]≠∅
  @Train_C2_axm4 DOOR_SIDE ∈ DOOR → SIDE
  @Train_C2_axm5 PLATFORM_SIDE ∈ PLATFORM → SIDE
  @Train_C2_axm6 PLATFORM ≠∅
end
```

(a) Context *CarriageDoors_C0*

```
context CarriageDoor_C1 extends CarriageDoor_C0

constants MANUAL_PLATFORM MANUAL_INTERNAL AUTOMATIC_CENTRAL_DOOR
START FAIL SUCCESS EXECUTED OPEN_RIGHT_DOORS OPEN_LEFT_DOORS
CLOSE_RIGHT_DOORS CLOSE_LEFT_DOORS ISOLATE_DOORS REMOVE_ISOLATION_DOORS

sets OPEN_DEV COMD_ST COMD_TYPE COMD

axioms
 @axm1 partition(OPEN_DEV, {MANUAL_PLATFORM}, {MANUAL_INTERNAL},
                                                {AUTOMATIC_CENTRAL_DOOR})
  @axm2 partition(COMD_ST, {START}, {FAIL}, {SUCCESS},{EXECUTED})
  @axm3 partition(COMD_TYPE,{OPEN_RIGHT_DOORS},{OPEN_LEFT_DOORS},
                  {CLOSE_RIGHT_DOORS}, {CLOSE_LEFT_DOORS},{ISOLATE_DOORS},
                 {REMOVE_ISOLATION_DOORS})
end
```

(b) Context *CarriageDoors_C1*

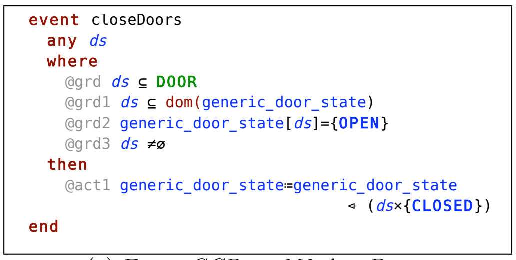
FIGURE 6.28: Parameterisation context *CarriageDoors_C0* plus additional context *CarriageDoors_C1*

```
INSTANTIATED REFINEMENT IEmergencyDoor_M2
INSTANTIATES GCDoors_M2 VIA GCDoor_C0 GCDoor_C1
REFINES CarriageDoors     /* abstract machine */
SEES CarriageDoors_C0 CarriageDoors_C1     /* instance contexts */
REPLACE
    SETS GEN_CARRIAGE := CARRIAGE   DOOR := DOOR
         DOOR_STATE := DOOR_STATE   SIDE := SIDE
         OPENING_DEVICE := OPEN_DEV   COMMAND_STATE := COMD_ST
         COMMAND := COMD   COMMAND_TYPE := COMD_TYPE
    CONSTANTS GEN_DOOR_CARRIAGE := DOOR_CARRIAGE
              OPEN := OPEN   PLATFORM := PLATFORM
              CLOSED := CLOSED   PLATFORM_SIDE := PLATFORM_SIDE
              ...
RENAME   /*rename variables, events and params*/
    VARIABLES generic_doors := carriage_doors   generic_door_state := carriage_ds
    EVENTS addDoor := allocateCarriageTrain   removeDoor := removeCarriageTrain
END
```

FIGURE 6.29: Instantiated Refinement *IEmergencyDoor_M2*

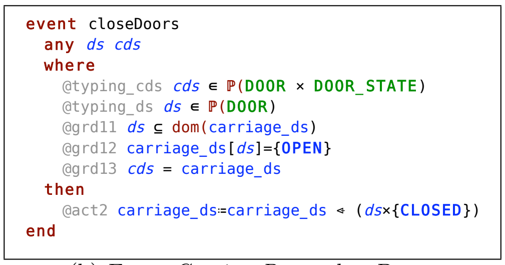
GCDoor_M0 (Fig. 6.30(a)). Some customisation is tolerable in the generic event to ensure that the instantiation of *GCDoor_M0.closeDoors* refines *CarriageDoors.closeDoors* by adding a parameter that match *cds* and respective guard *grd*13.

The customisation can be realised by a (shared event) composition of event *GCDoor_M0.closeDoors* with another event that introduces the additional parameter *cds* and guard $cds = carriage\_ds$. The monotonicity of the shared event composition allows the composed pattern to be instantiated as initially desired. Another option is to introduce an additional step: the last machine of the refinement chain before the



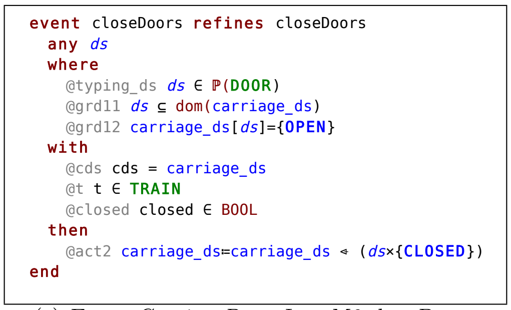
```
event closeDoors
  any ds
  where
    @grd ds ⊆ DOOR
    @grd1 ds ⊆ dom(generic_door_state)
    @grd2 generic_door_state[ds]={OPEN}
    @grd3 ds ≠∅
  then
    @act1 generic_door_state≔generic_door_state
                                  ◅ (ds×{CLOSED})
end
```

(a) Event *GCDoor_M0.closeDoors*



```
event closeDoors
  any ds cds
  where
    @typing_cds cds ∈ ℙ(DOOR × DOOR_STATE)
    @typing_ds ds ∈ ℙ(DOOR)
    @grd11 ds ⊆ dom(carriage_ds)
    @grd12 carriage_ds[ds]={OPEN}
    @grd13 cds = carriage_ds
  then
    @act2 carriage_ds≔carriage_ds ◅ (ds×{CLOSED})
end
```

(b) Event *CarriageDoors.closeDoors*



```
event closeDoors refines closeDoors
  any ds
  where
    @typing_ds ds ∈ ℙ(DOOR)
    @grd11 ds ⊆ dom(carriage_ds)
    @grd12 carriage_ds[ds]={OPEN}
  with
    @cds cds = carriage_ds
    @t t ∈ TRAIN
    @closed closed ∈ BOOL
  then
    @act2 carriage_ds≔carriage_ds ◅ (ds×{CLOSED})
end
```

(c) Event *CarriageDoorsInst_M0.closeDoors*

FIGURE 6.30: Event *closeDoors* in the pattern and instance; they differ in the parameters, guards and witnesses

instantiation (in our case study, machine *CarriageDoors*) is refined. The resulting refinement machine (*CarriageDoorsInst_M*0) refines the first instantiation machine (i.e. $CarriageDoors \sqsubseteq CarriageDoorsInst\_M0 \sqsubseteq EmergencyDoors\_M0$) "customising" the instantiation. Therefore the additional parameters (and respective guards) can disappear by means of witnesses as can be seen in Fig. 6.30(c). Ideally we aim to have a syntactic match (after instantiation) between the pattern and the initial instantiantion. Nevertheless a valid refinement is enough to apply the instantiation.

An instance machine *EmergencyDoor_M2* (Fig. 6.31) is similar to *GCDoor_M*2 apart from the replacements and renaming applied in *IEmergencyDoor_M2* (cf. Figs. 6.27, Fig. 6.29 and Fig. 6.31). That machine can be further refined (and decomposed) introducing the specific details related to emergency doors. The instantiation of the service doors follows the same steps.

Statistics: In Table 6.3, we describe the statistics of the development in terms of variables, events and proof obligations (and how many POs were automatically discharged by the theorem prover of the Rodin platform) for each refinement step. Almost 3/4 of the proof obligations are automatically discharged.

This case study was carried out under the following conditions:

- Rodin v2.1
- Shared Event Composition plug-in v1.3.1

```
machine EmergencyDoors_M2 refines EmergencyDoors_M1  sees CarriageDoors_C1

variables carriage_door carriage_ds isolated_door locked_doors door_opening_device obstructed_door
command command_doors command_type command_state

invariants
 @inv1 isolated_door ⊆ DOOR
 @inv2 ∀cmd,d· cmd ∈ command ∧ command_type(cmd)∈{OPEN_RIGHT_DOORS,OPEN_LEFT_DOORS}
         ∧ d ∈ DOOR ∧ d ∈ command_doors(cmd) ∧ command_state(cmd)=SUCCESS
         ∧ d ∉ isolated_door⇒ carriage_ds(d)=OPEN
 @inv3 ∀cmd,d· cmd ∈ command ∧ command_type(cmd)∈{CLOSE_RIGHT_DOORS,CLOSE_LEFT_DOORS}
         ∧ d ∈ DOOR ∧ d ∈ command_doors(cmd) ∧ command_state(cmd)=SUCCESS
         ∧ d ∉ isolated_door⇒ carriage_ds(d)=CLOSED
 @inv4 ∀d· d∈isolated_door ∧ d ∈ dom(carriage_ds)⇒  carriage_ds(d)∈{OPEN, CLOSED}
 @inv5 ∀cmd1,cmd2 · cmd1∈command ∧ cmd2∈command ∧ cmd1≠cmd2
         ∧ command_state(cmd1)≠EXECUTED
         ∧ command_state(cmd2)≠EXECUTED ⇒command_doors(cmd1)∩command_doors(cmd2)=∅
 @inv6 ∀cmd,d· cmd ∈ command ∧ command_type(cmd)=ISOLATE_DOORS ∧ d ∈ DOOR
         ∧ d ∈ command_doors(cmd) ∧ command_state(cmd)=SUCCESS ⇒ d∈ isolated_door
 @inv7 ∀cmd,d· cmd ∈ command ∧ command_type(cmd)=REMOVE_ISOLATION_DOORS
         ∧ d ∈ DOOR ∧ d ∈ command_doors(cmd) ∧ command_state(cmd)=SUCCESS ⇒ d∉ isolated_door
 @inv8 ∀d· d∈obstructed_door ⇒ carriage_ds(d)=OPEN
```

(a) Variables, invariants

```
event commandIsolationDoors refines otherCommandDoors
 any doors cmd cmd_type
 where
  @guard doors ⊆ carriage_door
  @guard1 cmd_type
        ∈ {ISOLATE_DOORS,REMOVE_ISOLATION_DOORS}
  @guard3 cmd ∈ COMD\command
  @guard4 ∀cmd1 · cmd1∈command
            ∧ command_state(cmd1)≠EXECUTED
            ⇒doors∩command_doors(cmd1)=∅
  @grd4 doors ≠∅
  @grd5 cmd_type = ISOLATE_DOORS ⇔ (doors∩isolated_door = ∅ )
  @grd6 cmd_type = REMOVE_ISOLATION_DOORS ⇔ isolated_door≠∅
          ∧ doors∩isolated_door≠∅
 then
  @act1 command_state(cmd)≔START
  @act2 command_doors(cmd)≔doors
  @act3 command ≔ command ∪ {cmd}
  @act4 command_type(cmd)≔cmd_type
end

event updateIsolationCmdState refines updateCmdState
 any state cmd
 where
  @guard3 cmd ∈ command
  @guard state ∈ COMD_ST\{START,EXECUTED}
  @guard1 command_state(cmd)=START
  @guard5 command_type(cmd)
              ∈ {ISOLATE_DOORS,REMOVE_ISOLATION_DOORS}
  @grd3 (command_type(cmd) = ISOLATE_DOORS
          ∧ (∃d · d∈command_doors(cmd) ∧ d ∉isolated_door))
          ∨ (command_type(cmd) = REMOVE_ISOLATION_DOORS
          ∧ (∃d · d∈command_doors(cmd) ∧ d ∈isolated_door))
          ⇔ state = FAIL
 then
  @act1 command_state(cmd)≔state
end
```

```
event executedLogCmdState refines updateCmdState
 any cmd
 where
  @guard3 cmd ∈ command
  @guard1 command_state(cmd)∈{FAIL,SUCCESS}
 with
  @state state = EXECUTED
 then
  @act1 command_state(cmd)≔EXECUTED
end

event isolateDoor
 any d cmd
 where
  @guard d ∈ carriage_door\isolated_door
  @guard1 cmd ∈ command
  @guard2 command_state(cmd)=START
  @guard3 d ∈ command_doors(cmd)
  @guard4 command_type(cmd) = ISOLATE_DOORS
  @guard5 carriage_ds(d)∈{OPEN, CLOSED}
 then
  @act1 isolated_door≔ isolated_door ∪ {d}
end

event removeIsolatedDoor
 any d cmd
 where
  @guard d ∈ isolated_door
  @guard1 cmd ∈ command
  @guard3 d ∈ command_doors(cmd)
  @guard4 command_type(cmd) = REMOVE_ISOLATION_DOORS
  @guard2 command_state(cmd)=START
  @guard5 carriage_ds(d)∈{OPEN, CLOSED}
 then
  @act1 isolated_door≔ isolated_door \ {d}
end
```

(b) Some events in *EmergencyDoor_M2*

FIGURE 6.31: Excerpt of instantiated machine *EmergencyDoor_M2*

- Model Decomposition plug-in v1.2.1
- Instantiation was done manually (currently tool support is not available).
- ProB v2.1.2
- Camille Text Editor 2.0.1

Although we are interested mainly interested in safety properties, the model checker ProB [141] proved to be very useful as a complementary tool during the development of this case study. In some stages of the development, all the proof obligations were

	Variables	Events	ProofObligations/Auto
TransitiveClosureCtx	–	–	10/10
MetroSystem_C0	–	–	5/3
MetroSystem_C1	–	–	0/0
MetroSystem_M0	7	10	75/64
MetroSystem_M1	10	13	17/17
MetroSystem_M2	12	17	78/57
MetroSystem_M3	12	17	24/22
Track	4	10	0/0
Train	7	14	0/0
Middleware	1	4	0/0
Train_M1	9	16	74/52
Train_M2	13	21	155/79
Train_M3	12	21	65/24
Train_M4	14	21	119/89
LeaderCarriage	9	21	0/0
Carriage	5	11	0/0
Carriage_M1	6	11	28/21
CarriageInterface	4	11	0/0
CarriageDoors	2	5	0/0
CarriageDoorsInst_M0	2	5	2/1
GCDoor_M0	2	5	6/6
GCDoor_M1	9	15	81/80
GCDoor_M2	10	22	170/153
Total			909/678(74.6%)

TABLE 6.3: Statistics of the metro system case study

discharged but with ProB we discovered that the system was deadlocked due to some missing detail. In large developments, these situations possibly occur more frequently. Therefore we suggest discharging the proof obligations to ensure the safety properties are preserved and run the ProB model checker to confirm that the system actually is free from deadlocks.

6.14 Discussion: Conclusions and Lessons Learned

We modelled a metro system case study, starting by proving its global properties through several refinement steps. Afterwards, due to an architectural decision and to alleviate the problem of modelling and handling a large system in one single machine, the system is decomposed in three sub-components. We further refine one of the resulting sub-components (*Train*), introducing several details in four refinements levels. Then again, due to the number of proof obligations, to achieve separation of aspects and to ease the further developments, we decompose it into two sub-components: *LeaderCarriage* and *Carriage*. Since we are interested in modelling carriage doors, sub-component *Carriage* is refined and afterwards decomposed originating sub-component *CarriageDoors*. Benefiting from an existing generic development for carriage doors *GCDoor*, we consider this development as a pattern and instantiate two kind of carriage doors: *service* and *emergency* doors. Although the instantiation is similar for both types of doors, the resulting instances can be further refined independently. Using generic instantiation, we avoid having to prove the proof obligations regarding the pattern *GCDoor*: *GCDoor_M0*, *GCDoor_M1* and *GCDoor_M2* (in the overall 257 POs). This figure only considers the instantiation of emergency doors (the instantiation of service doors would imply twice the number of POs).

From the experience of other developments involving a large number of refinements levels or refinements with large models, the development tools reach a point where it is not possible to edit the model due to the high amount of resources required to do it (or it is done very slowly). The decomposition is a possible solution that alleviates this issue by splitting the model into more tool manageable dimensions. Following a top-down approach, developed models become more complex in each refinement step. Nevertheless by applying decomposition, we alleviate the consequences of such complexity by separating concerns (architecture approach), decreasing the number of events and variables per sub-component which results in models that are more manageable from a tool point of view. Moreover, for each refinement, the properties (added as requirements) are preserved. Using generic instantiation, we avoid proving the pattern proof obligations *GCDoor*. Therefore we reach our goal of reusing existing developments as much as possible and discharge as little proof obligations as possible. Even the interactive proofs were relatively easy to discharge once the correct tactic was discovered. This task would be more difficult without the decomposition due to the elevated number of hypotheses to considered for each PO. Nevertheless we believe that the effort of discharging proof obligations could be minimised by having a way to reuse tactics. In particular when the same steps are followed to discharge similar POs.

In a combination of refinement and instantiation, we learned that the abstract machine and the abstract pattern do not necessarily match perfectly. In particular, some extra guards and parameters may exist resulting from previous refinements in the instance. Nevertheless the generic model can still be reused. We can (shared event) compose the pattern with another machine in a way that the resulting events include the additional parameters and guards to guarantee a valid refinement. Another interesting conclusion is that throughout an instantiation, it is possible not to use all the generic events. A subset of generic events can be instantiated in opposition to instantiate all. This a consequence of the event refinements that only depend on abstract and concrete events. Nevertheless this only applies for safety properties. If we are interested in liveness properties, the exclusion of a generic event may result in a system deadlock.

With this case study we aim to illustrate the application of decomposition and generic instantiation as techniques to help the development of formal models. Following these techniques, the development is structured in a way that simplifies the model by separating concerns and aspects and decreases the number of proof obligations to be discharged. 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 [114] describes the reuse of formal methods when analysing the requirements and designing the software between two spacecrafts' formal models. Stepney *et al.* [177, 178] 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 (called meaning preservation refactoring steps in Z promotion) to fit into a specific pattern. [177] suggests template support and architecture patterns to be supported by tools, something that currently does not happen. We have a similar viewpoint and we would like to address this issue in the future. Templates could be customised according to the modeller's needs and selected from an existing list, perhaps categorised as suggested in [177].

Butler [44] uses the shared event approach in classical B to decompose a railway system into three sub-components: Train, Track and Communication. The system is modelled and reasoned as a whole in an event-based approach, both the physical system and the desired control behaviour. Our case study follows a similar methodology applied to a metro system following the same shared event style. Moreover we introduce more requirements regarding the trains and the carriage doors, expressed through the use of decomposition and generic instantiation.

Chapter 7

Conclusions and Future Work

In this chapter we wrap up the contributions of this thesis and outline our objectives for the future. We aim to introduce reusability and modularity mechanisms when developing system specifications in particular large systems that become cumbersome to manage when scaling. For that we propose the use of composition, decomposition and generic instantiation to facilitate the development of large systems. We use the Event-B formal notation and the Rodin platform for the development of these techniques and respective tool support. We separate the conclusions and future work into three main topics giving more detail about each as follows.

7.1 Composition

Based on the close relation between action systems and Event-B plus the correspondence between action systems and CSP [53], we define our Event-B composition with an event-based behaviour. Shared event composition is proved to be monotonic by means of proof obligations. Consequently sub-components can be further refined independently. Refinement in a "top-down" style for developing specifications is allowed including the generation of POs. During composition, sub-components interact through event parameters by value-passing. We extend Event-B to support shared event composition, allowing combination and reuse of existing sub-components through the introduction of *composed machines*. Required static checks are defined and POs are generated to validate the composition. Such an approach seems suitable for modelling distributed systems, where the system can be seen as a combination of sub-components.

Currently we have developed a plug-in that allows shared event composition using Event-B in the Rodin platform. Some of the proofs to be generated are also generated in the included machines. By identifying the similarities between proofs, we have established that we can reuse proof obligations and reduce the effort of discharging proof obligations that are already done in the included machines. The shared event composition tool generates a new (composed) machine to ensure the validity of the composition using the already existing validation scheme for machines (generation of proof obligations). In the future the composed machine generation should be optional since this validation should be done directly over the composition file. Although the shared variable composition was not in the initial plans for this thesis, the close relation with our work suggested a deeper understanding of that style. During that study, we discovered a close relation between the rely/guarantee composition for VDM and the shared variable composition for Event-B that is also mentioned in Hoang and Abrial's work [90]. It should be possible to create a correspondence between these two approaches and we intend to investigate this in the future. Schneider *et al* [159] define a CSP semantics for Event-B as described in Sect. 1.5.6. Following that work, we define as future work the derivation the CSP semantics for Event-B machines to define the composition of machines in terms of traces, failures, divergences and infinite traces. A paper was accepted for the B workshop running in parallel with FM 2011 (International Symposium on Formal Methods) [161] and another to FMCO 2010 (International Symposia on Formal Methods for Components and Objects) based on shared event composition [164] and we gave a presentation about this work in the Rodin Workshop 2009.

To summarize, we list the future work for composition below:

- Generation of proof obligations for shared event composition directly rather than indirectly by expanding machine compositions.
- Further investigation on reuse of proofs obligations in the Rodin platform.
- Can rely/guarantee for VDM be applied to (Abrial) shared variable composition for Event-B?
- Adding enabledness POs when available for the Rodin platform.
- Derivation of CSP semantics for Event-B machines described by Schneider *et al* [159] to define the composition of machines in terms of traces, failures, divergences and infinite traces.

7.2 Generic Instantiation

The generic instantiation work was a result of the achievements towards composition and decomposition. The possibility to have patterns that can be reused in another developments seems very attractive while creating specifications in particular in a top-bottom style. Event-B supports generic developments but lacks the capacity to instantiate and reuse those generic developments. As a solution, generic instantiation is applied to patterns and as an outcome *instantiated machines* are created and parameterised. An instantiated machine instantiates a generic machine, is parameterised by a context and the pattern elements are renamed/replaced according to the instance. In a similar style, an *instantiated refinement* instantiates a chain of refinements reusing the pattern proof obligations assuming that the instantiated proof obligations are as valid as the pattern ones. By quantifying the variables, constants and types we ensure that pattern proof obligations remain valid when instantiating. A renaming plug-in was developed supporting the renaming of Event-B elements and respective proofs. Optimisation at level of proof renaming will be investigated in the future as it may become a slow operation for large proof trees. A paper was accepted at ICFEM 2009 (International Conference on Formal Engineering Methods) [163] describing this work. In the future, we intend to have tool support for generic instantiation as described in Chapter 3. With larger and relevant cases studies we should improve the tool and publish a paper with the results and conclusions. Moreover a library of patterns could be provided when modelling, divided according to the categories are suggested in [178].

To summarize, we list the future work for generic instantiation below:

- Optimisation of proof renaming
- Tool development in collaboration with ETH Zurich.
- Application of a large case study to and test the scalability and improve the tool.
- Definition of a categorised pattern library and customisable templates.
- Writing and submitting a paper as a continuation of the initial study describing the tool support and conclusions of application of a case study.

7.3 Decomposition

There is a need for modularisation and reuse of sub-components in order to model large systems and manage better the respective POs. Event-B lacks a sub-component mechanism. Thus we propose to tackle that problem through the decomposition of a system by their events or variables. The shared variable (state-based) approach is suitable for designing parallel algorithms while the shared event (event-based) is suitable for message-passing distributed systems [45]. Following any of these two approaches, the parallel components of a distributed system can be refined and decomposed separately without making any assumptions about the rest of the system. The shared variable style relies on the work of Abrial and Hallerstede [15] where variables are shared and exists the notion of external events. Butler [45] suggests the shared event decomposition where events are partition through the sub-components and the interaction occurs via shared parameters. The work developed by Butler in [40] for action system is strongly related with the same approach for shared event decomposition in Event-B [45] as both approaches are state-based formalism combined with event-based CSP.

We have collaborated in the tool support development for the decomposition technique in the Rodin platform, being responsable for the shared event approach development. The tool allows the semi-automatic decomposition in a shared variable and shared event. An initial study of such work has been accepted as a workshop paper for the ABZ 2010 conference [165] and an extended version of that paper was published in the journal Software: Practise and Experience expressing our results [166]. With the application of more case studies, we should have more results and conclusions that can be published. As described in Chapter 4, the decomposition tool has been widely used with positive feedback. Some improvements on the tool have been suggested and we intend to carry them out in the future. A large case study based on some real requirements is described in Chapter 6 and shows a practical implementation of the technique, the possible combination with generic instantiation and respective tool support. To summarize, we list the future work for decomposition below:

- Tool development and improvements.
- Extension of the composed machine plug-in to support the shared variable composition in order to store the decomposition configuration.
- Application of a large case study to test scalability.
- Writing and submitting a paper with the results of the application of decomposition in scalable systems.

7.4 Future Work

In the previous sections we described the future related to each technique that we studied in this document. Although they are powerful techniques that help the formal modelling of large and/or complex systems, there is still plenty to be studied and researched when it comes to the reusability of models. In particular, there is a need to decrease the user's effort when developing models, in particular, for the reuse of proofs. From our own experience, discussions with other formal developers and even for the industrial companies that use formal methods, often the time spent discharging proofs is greater than the time spent modelling the system itself. The same model can be developed in many different manners but the generated number of proofs and the ease to discharge them vary. Therefore although the outcome may be the same, the properties to be proved may be harder to achieve. This situation somehow suggests the need for guidelines on how to achieve the same goal in a simpler, cleaner and possible easier in terms of discharging proofs. These guidelines could be arranged by some modelling patterns that tend to occur even when different kind of systems are being developed. We intend to research these modelling patterns and come up with practical guidelines that can be easily applied to existing developments (in a way, it is a different continuation path from generic instantiation).

The reuse of proofs is another topic that also requires further study. Currently the Rodin platform allows the reuse of existing proofs to be applied to other proofs but these usually fail if the structure of the proof is slightly different. We need more powerful proof patterns that can be applied that are less sensitive to the structure. The modelling tactics often are repeated throughout a development and consequently results in similar kind of proof obligations to be discharged. Ideally once these kind of proofs are discharged, that proof tactic should easily be applied to the rest of the family of proof obligations. What happens at the moment is cumbersome where the user has to redo and re-apply the proofs steps instead of tackling more interesting and challenging proofs. This topic is quite broad but we intend to investigate to come up with more possible reuse of proofs techniques.

More recently, code generation [67] has been proposed for Event-B following the path of classical B. But this approach is more flexible as it allows the user to define *tasking machines* [68] to define how the model can be implemented. Tasking machines can be periodic, triggered by an event or happen only once (one shot). Moreover the implementation allows the definition of tasks that model or simulate the environment (*Environ machine*) and the definition of data that can be accessed by different threads (*Shared machine*). The communication and implementation of these different machines use the shared event composition to structure them in one single place. Another powerful option is to define the implementation data structure according to the target language to be implemented. As future work, we intend to extend the existing theory plug-in [115] to allow the mapping of new (or existing) theories into executable code. The intention is to let the user define new Event-B data structures (since they are not fixed) and map to the corresponding implementation for the specific target language. At the moment only Ada and C are supported, although in the future other target languages will supported.

Bibliography

[1] Martín Abadi and Leslie Lamport. Composing Specifications. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems - Models, Formalisms, Correctness*, volume 430, pages 1–41, Berlin, Germany, 1989. Springer-Verlag.

[2] Martín Abadi and Leslie Lamport. Decomposing Specifications of Concurrent Systems. In *PROCOMET '94: Proceedings of the IFIP TC2/WG2.1/WG2.2/WG2.3 Working Conference on Programming Concepts, Methods and Calculi*, pages 327–340, Amsterdam, The Netherlands, The Netherlands, 1994. North-Holland Publishing Co.

[3] Jean-Raymond Abrial. Extending B without Changing it (for Developing Distributed Systems). In *Proceedings of 1st Conference on the B method*, pages 169–191, Nantes, France, November 1996. Springer-Verlag.

[4] Jean-Raymond Abrial. *The B-Book: Assigning programs to meanings.* Cambridge University Press, 1996.

[5] Jean-Raymond Abrial. Formal Methods in Industry: Achievements, Problems, Future. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 761–768, New York, NY, USA, 2006. ACM.

[6] Jean-Raymond Abrial. A System Development Process with Event-B and the Rodin Platform. In *Formal Methods and Software Engineering*, volume 4789, pages 1–3. Springer Berlin / Heidelberg, 2007.

[7] Jean-Raymond Abrial. Formal Methods: Theory Becoming Practice. *J. UCS*, 13(5):619–628, May 2007.

[8] Jean-Raymond Abrial. Event Model Decomposition. Technical report, ETH Zurich, 2009 (Unpublished).

[9] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.

[10] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *International Journal on Software Tools for Technology Transfer (STTT)*, April 2010.

[11] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Michael Leuschel, Matthias Schmalz, and Laurent Voisin. Proposals for Mathematical Extensions for Event-B. http://deploy-eprints.ecs.soton.ac.uk/216/ (Unpublished), 2009.

[12] Jean-Raymond Abrial, Michael Butler, Rajev Joshi, Elena Troubitsyna, and Jim C. P. Woodcock. 09381 Extended Abstracts Collection – Refinement Based Methods for the Construction of Dependable Systems. In Jean-Raymond Abrial, Michael Butler, Rajeev Joshi, Elena Troubitsyna, and Jim C. P. Woodcock, editors, *Refinement Based Methods for the Construction of Dependable Systems*, number 09381 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2010. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[13] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, and Laurent Voisin. An Open Extensible Tool Environment for Event-B. In *ICFEM*, pages 588–605, 2006.

[14] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. Refinement and Reachability in Event_B. *ZB 2005: Formal Specification and Development in Z and B*, pages 222–241, 2005.

[15] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundam. Inf.*, 77(1-2):1–28, 2007.

[16] Jean-Raymond Abrial, Christophe Metayer, and Laurent Voisin. Rodin Manual and Language Definition. http://deploy-eprints.ecs.soton.ac.uk/11/, 2007.

[17] Jean-Raymond Abrial and Louis Mussat. Introducing Dynamic Constraints in B. In *B '98: Proceedings of the Second International B Conference on Recent Advances in the Development and Use of the B Method*, pages 83–128, London, UK, 1998. Springer-Verlag.

[18] R.D. Arthan. Undefinedness in Z: Issues for Specification and Proof. In *CADE-13 Workshop on Mechanization of Partial Functions. Available on the Web as ftp://ftp.cs.bham.ac.uk/pub/authors/M.Kerber/96-CADE-WS/Arthan.ps.gz*, pages 3–12. Springer, 1996.

[19] Atelier B Web Page. http://www.atelierb.eu/, September 2008. Online; accessed 27-July-2010.

[20] B-Core, The B-Technology Company: B-Toolkit. http://www.b-core.com/btoolkit.html, September 2008. Online; accessed 27-July-2010.

[21] B4Free. B4free. http://www.b4free.com, September 2008. Online; accessed 27-July-2010.

[22] Ralph-Johan Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of *Mathematical Center Tracts*. Mathematical Centre, Amsterdam, The Netherlands, 1980.

[23] Ralph-Johan Back. Refinement Calculus, part II: Parallel and Reactive Programs. In *REX workshop: Proceedings on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, pages 67–93, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[24] Ralph-Johan Back. Refinement of Parallel and Reactive Programs. Technical report, California Institute of Technology, Pasadena, CA, USA, 1992.

[25] Ralph-Johan Back and Michael Butler. Fusion and Simultaneous Execution in the Refinement Calculus. *Acta Informatica*, 35(11):921–949, 1998.

[26] Ralph-Johan Back and Reino Kurki-Suonio. Decentralization of Process Nets with Centralized Control. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 131–142, New York, NY, USA, 1983. ACM.

[27] Ralph-Johan Back and Joakim von Wright. *Trace refinement of action systems*, pages 367–384. Springer Berlin / Heidelberg, 1994.

[28] Jos C. M. Baeten and Jan A. Bergstra. Real Time Process Algebra. *Formal Asp. Comput.*, 3(2):142–188, 1991.

[29] Elisabeth Ball. *An Incremental Process for the Development of Multi-Agent Systems in Event-B.* PhD thesis, Southampon University, 2008.

[30] Elisabeth Ball and Michael Butler. Using Decomposition to Model Multi-agent Interaction Protocols in Event-B. In *FM'06 Doctoral Symposium*. Springer, 2006.

[31] Hubert Baumeister. Using Algebraic Specification Languages for Model-Oriented Specifications. Technical Report MPI-I-96-2-003, Max-Planck-Institut für Informatik, Saarbrücken, February 1996.

[32] Françoise Bellegarde, Jacques Julliand, and Olga Kouchnarenko. Synchronized Parallel Composition of Event Systems in B. In *ZB '02: Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, pages 436–457, London, UK, 2002. Springer-Verlag.

[33] Sandrine Blazy, Frédéric Gervais, and Régine Laleau. Reuse of Specification Patterns with the B Method. In Springer-Verlag SEP, editor, *ZB 2003: Formal Specification and Development in Z and B Lecture Notes in Computer Science*,

volume 2651 of *Lecture Notes in Computer Science*, pages 40–57, Turku, Finland, June 2003.

[34] E. A. Boiten, J. Derrick, H. Bowman, and M. Steen. Coupling schemas: data refinement and view(point) composition. In D.J. Duke and A.S. Evans, editors, *2nd BCS-FACS Northern Formal Methods Workshop*, Workshops in Computing, page 18. Springer-Verlag, July 1997.

[35] Tommaso Bolognesi. A Conceptual Framework for State-Based and Event-Based Formal Behavioural Specification Languages. In *ICECCS '04: Proceedings of the Ninth IEEE International Conference on Engineering Complex Computer Systems Navigating Complexity in the e-Engineering Age (ICECCS'04)*, pages 107–116, Washington, DC, USA, 2004. IEEE Computer Society.

[36] Azad Bolour. Eclipse plug-in architecture. http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html, September 2008. Online; accessed 27-July-2010.

[37] J. P. Bowen. *Formal Specification and Documentation using Z: A Case Study Approach*. International Thomson Computer Press, 1996.

[38] Jonathan P. Bowen and Michael G. Hinchey. Seven More Myths of Formal Methods. *IEEE Software*, 12:34–41, 1995.

[39] Brama: Graphical tool of modeling applied to the B formal method. http://www.brama.fr/, September 2008. Online; accessed 27-July-2010.

[40] Michael Butler. Refinement and Decomposition of Value-Passing Action Systems. In *CONCUR '93: Proceedings of the 4th International Conference on Concurrency Theory*, pages 217–232, London,UK, 1993.

[41] Michael Butler. Stepwise Refinement of Communicating Systems. *Science of Computer Programming*, 27(2):139–173, September 1996.

[42] Michael Butler. An Approach to the Design of Distributed Systems with B AMN. In *Proc. 10th Int. Conf. of Z Users: The Z Formal Specification Notation (ZUM), LNCS 1212*, pages 221–241, 1997.

[43] Michael Butler. csp2B: A Practical Approach to Combining CSP and B. *Formal Aspects of Computing*, 12:182–196, 2000.

[44] Michael Butler. A System-based Approach to the Formal Development of Embedded Controllers for a Railway. *Design Automation for Embedded Systems*, 6:355–366, 2002.

[45] Michael Butler. Synchronisation-Based Decomposition for Event-B. In *RODIN Deliverable D19 Intermediate report on methodology*, pages 47–57, 2006.

[46] Michael Butler. Incremental Design of Distributed Systems with Event-B. *Marktoberdorf Summer School 2008 Lecture Notes*, November 2008.

[47] Michael Butler. Decomposition Structures for Event-B. *Integrated Formal Methods iFM2009*, pages 20–38, February 2009.

[48] Michael Butler. External and Internal Choice with Event-B Groups in Event-B. *Formal Aspects of Computing*, (To be published), 2012.

[49] Michael Butler and Stefan Hallerstede. The Rodin Formal Modelling Tool. *BCS-FACS Christmas 2007 Meeting - Formal Methods In Industry, London.*, pages 1–5, December 2007.

[50] Michael Butler and Michael Leuschel. Combining CSP and B for Specification and Property Verification. In John Fitzgerald, Ian Hayes, and Andrzej Tarlecki, editors, *Formal Methods 2005*, number 3582 in LNCS, pages 221–236. Springer, January 2005.

[51] Michael Butler and Issam Maamria. Mathematical Extension in Event-B through the Rodin Theory Component. http://deploy-eprints.ecs.soton.ac.uk/251/ (Unpublished), 2010.

[52] Michael Butler and Marina Waldén. Distributed System Development in B. Technical Report TUCS-TR-53, Turku Centre for Computer Science, 14, 1996.

[53] Michael J. Butler. *A CSP Approach to Action Systems.* PhD thesis, Oxford University, 1992.

[54] Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock. A Refinement Strategy for Circus. *Formal Aspects of Computing*, 15:146–181, 2003. 10.1007/s00165-003-0006-5.

[55] Jingwen Cheng. A Reusability-Based Software Development Environment. *SIGSOFT Softw. Eng. Notes*, 19(2):57–62, 1994.

[56] John Colley and Michael Butler. On Proving with Event-B that a Pipelined Processor Model Implements its ISA Specification. In *Dagstuhl Seminar on Refinement Based Methods for the Construction of Dependable Systems*, 2009.

[57] Community Z Tools. Community Z Tools. http://czt.sourceforge.net/, January 2010.

[58] Dan Craigen. Tool Support for Formal Methods. In *Proceedings of the 13th international conference on Software engineering*, ICSE '91, pages 184–185, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[59] CSK Systems. VDMTools. http://www.vdmtools.jp/en/, January 2011.

[60] Kriangsak Damchoom. *An Incremental Refinement Approach to a Development of a Flash-Based File System in Event-B*. PhD thesis, School of Electronics and Computer Science, University of Southampton, 2010.

[61] Kriangsak Damchoom, Michael Butler, and Jean-Raymond Abrial. Modelling and Proof of a Tree-Structured File System in Event-B and Rodin. In *Proceedings of the 10th International Conference on Formal Methods and Software Engineering*, ICFEM '08, pages 25–44, Berlin, Heidelberg, 2008. Springer-Verlag.

[62] Kriangsak Damchoom and Michael J. Butler. Applying Event and Machine Decomposition to a Flash-Based Filestore in Event-B. In *SBMF*, pages 134–152, 2009.

[63] Robert Darimont and Axel van Lamsweerde. Formal Refinement Patterns for Goal-Driven Requirements Elaboration. In *SIGSOFT '96: Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 179–190, New York, NY, USA, 1996. ACM.

[64] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, EnglewoodCliffs, New Jersey, 1976.

[65] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.

[66] Eclipse. Eclipse homepage. http://www.eclipse.org, September 2008. Online; accessed 27-July-2010.

[67] Andy Edmunds. Code generation. http://wiki.event-b.org/index.php/Code_Generation, January 2012.

[68] Andy Edmunds. Tasking Event-B Overview. http://wiki.event-b.org/index.php/Tasking_Event-B_Overview, January 2012.

[69] Neil Evans and Michael Butler. A Proposal for Records in Event-B. In Tobias Nipkow, Jayadev Misra, and Emil Sekerinski, editors, *Formal Methods 2006*, volume LNCS 4085, pages 221–235. Springer, 2006.

[70] Neil Evans and Neil Grant. Towards the Formal Verification of a Java Processor in Event-B. *Electronic Notes in Theoretical Computer Science*, 201:45–67, March 2008.

[71] Event-B. Event-B.org. http://www.event-b.org, September 2008. Online; accessed 27-July-2010.

[72] Jerome Falampin, Michael Butler, and John Fitzgerald. Deploy deliverable d16 d2.1 pilot deployment in transportation (wp2). http://www.deploy-project.eu/pdf/D16_final6.pdf, September 2009.

[73] Asieh Salehi Fathabadi, Abdolbaghi Rezazadeh, and Michael Butler. Applying Atomicity and Model Decomposition to a Space Craft System in Event-B. In *THIRD NASA FORMAL METHODS SYMPOSIUM*, February 2011.

[74] Ludger Fiege, Gero Mühl, and Felix C. Gärtner. Modular event-based systems. *Knowl. Eng. Rev.*, 17(4):359–388, 2002.

[75] Clemens Fischer. CSP-OZ: a Combination of Object-Z and CSP. In *Proceedings of the IFIP TC6 WG6.1 international workshop on Formal methods for open object-based distributed systems*, pages 423–438, London, UK, UK, 1997. Chapman & Hall, Ltd.

[76] Clemens Fischer. How to Combine Z with Process Algebra. In *Proceedings of the 11th International Conference of Z Users on The Z Formal Specification Notation*, pages 5–23, London, UK, 1998. Springer-Verlag.

[77] John S. Fitzgerald. Triumphs and Challenges for Model-Oriented Formal Methods: The VDM++ Experience (Abstract). In *ISOLA '06: Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (isola 2006)*, pages 1–4. IEEE Computer Society, November 2006.

[78] John S. Fitzgerald and Cliff B. Jones. The connection between two ways of reasoning about partial functions. *Inf. Process. Lett.*, 107(3-4):128–132, 2008.

[79] Flowgate Consulting. Fastest. http://www.flowgate.net/?lang=en&seccion=herramientas#, January 2011.

[80] Formal Systems (Europe) Ltd. Formal Systems. http://www.fsel.com/software.html, January 2011.

[81] Dimitra Giannakopoulou and Corina S. Păsăreanu. Special Issue on Learning Techniques for Compositional Reasoning. *Form. Methods Syst. Des.*, 32(3):173–174, 2008.

[82] GMF. Graphical Modeling Framework. http://www.eclipse.org/modeling/gmf/, September 2008. Online; accessed 27-July-2010.

[83] Kentaro Go and Norio Shiratori. A Decomposition of a Formal Specification: An Improved Constraint-Oriented Method. *IEEE Transactions on Software Engineering*, 25(2):258–273, 1999.

[84] Joseph Goguen and Joseph Tardo. An introduction to OBJ: a language for writing and testing software specifications. In Marvin K. Zelkowitz, editor, *Specification of Reliable Software*, pages 170–189. IEEE Press, 1979, Reprinted in Software Specification Techniques, Nehan Gehani and Andrew McGettrick, Eds., Addison-Wesley, 1985, pages 391–420. No pdf file.

[85] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[86] John V. Guttag and James J. Horning. *Larch: languages and tools for formal specification*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[87] Stefan Hallerstede. Justifications for the Event-B Modelling Notation. In *The 7th International B Conference*, volume 4355 of *Lecture Notes in Computer Science*, pages 49–63. Springer Berlin / Heidelberg, Besançon , FRANCE, January 2007.

[88] Stefan Hallerstede. On the Purpose of Event-B Proof Obligations. In *ABZ2008*, June 2008.

[89] Karl Hess. The Importance of Tools. http://www.fff.org/freedom/0493c.asp, April 1993.

[90] Thai Hoang and Jean-Raymond Abrial. Event-B Decomposition for Parallel Programs. *Abstract State Machines, Alloy, B and Z*, pages 319–333, 2010.

[91] Charles Antony Richard Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[92] Charles Antony Richard Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science, 1985.

[93] Tony Hoare and Peter O'Hearn. Separation Logic Semantics for Communicating Processes. *Electron. Notes Theor. Comput. Sci.*, 212:3–25, April 2008.

[94] Sonja Holl. Refactoring of B Models, Bachelor Thesis. http://www.stups.uni-duesseldorf.de/thesis_detail.php?id=9, August 2008. Online; accessed 27-July-2010.

[95] J.J.M. Hooman, S. Ramesh, and W.P. de Roever. A Compositional Semantics for Statecharts. In J.-J.Ch. Meyer J.W. Klop and J.J.M.M. Rutten, editors, *Liber Amicorum: J.W. de Bakker, 25 jaar semantiek*, pages 275–287, Amsterdam: CWI, 1989.

[96] Alexei Iliasov, Elena Troubitsyna, Linas Laibinis, Alexander Romanovsky, Kimmo Varpaaniemi, Dubravka Ilic, and Timo Latvala. Supporting Reuse in Event B Development: Modularisation Approach. In *16th International Symposium on Formal Methods*, June 2009.

[97] Transportation Systems Design Inc. Communications based train control. http://www.tsd.org/cbtc/, January 2012.

[98] ISO. LOTOS A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Information Systems Processing - Open Systems Interconnection, 1987.

[99] Daniel Jackson. Structuring Z specifications with views. *ACM Trans. Softw. Eng. Methodol.*, 4(4):365–389, 1995.

[100] Jonathan Jacky. *The way of Z: practical programming with formal methods.* Cambridge University Press, New York, NY, USA, 1996. No pdf file.

[101] Lu Jian. Introducing data decomposition into VDM for tractable development of programs. *SIGPLAN Not.*, 30(9):41–50, 1995.

[102] Lu Jian. Developing Parallel Object-Oriented Programs in the Framework of VDM. *Ann. Software Eng.*, 2:199–211, 1996.

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

[104] Cliff B. Jones. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[105] Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice Hall International, second edition, 1990.

[106] Cliff B. Jones. Wanted: a compositional approach to concurrency. In *Programming methodology*, pages 5–15. Springer-Verlag New York, Inc., New York, NY, USA, 2003.

[107] P. Kefalas, G. Eleftherakis, and A. Sotiriadou. Developing Tools for Formal Methods. In *In 9th Panhellenic Conference on Informatics, Thessaloniki*, pages 625–639, 2003.

[108] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[109] Leslie Lamport. TLZ. *Z Users Conference*, pages 267–268, 1994.

[110] Arnaud Lanoix. Event-B Specification of a Situated Multi-Agent System: Study of a Platoon of Vehicles. In *TASE '08: Proceedings of the 2008 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering*, pages 297–304, Washington, DC, USA, 2008. IEEE Computer Society.

[111] Yves Ledru and Pierre-Yves Schobbens. Applying VDM to Large Developments. In *Conference proceedings on Formal methods in software development*, pages 55–58, New York, NY, USA, 1990. ACM.

[112] Xiaodong Liu, Zhiqiang Chen, Hongji Yang, Hussein Zedan, and William C. Chu. A Design Framework for System Re-Engineering. In *APSEC '97: Proceedings of the Fourth Asia-Pacific Software Engineering and International Computer Science Conference*, page 342, Washington, DC, USA, 1997. IEEE Computer Society.

[113] Xiaodong Liu, Hongji Yang, and Hussein Zedan. Formal Methods for the Re-Engineering of Computing Systems: A Comparison. In *COMPSAC '97: Proceedings of the 21st International Computer Software and Applications Conference*, page 409, Washington, DC, USA, 1997. IEEE Computer Society.

[114] Robyn R. Lutz. Reuse of a Formal Model for Requirements Validation. In *In Fourth NASA Langley Formal Methods Workshop. NASA*, 1997.

[115] Issam Maamria. Theory plug-in. http://wiki.event-b.org/index.php/Theory_Plug-in, August 2011.

[116] Issam Maamria and Michael Butler. Rewriting and Well-Definedness within a Proof System. In *Partiality and Recursion in Interactive Theorem Provers PAR-10*, July 2010.

[117] Brendan Mahony and Jin Song Dong. Overview of the Semantics of TCOZ. In K. Araki, A. Galloway, and K. Taguchi, editors, *IFM'99: Integrated Formal Methods*, pages 66–85. Springer-Verlag, 1999.

[118] Brendan Mahony and Jin Song Dong. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, February 2000.

[119] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992. No pdf file.

[120] Andrew Martin. Relating Z and First-Order Logic. In *Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume II*, FM '99, pages 1266–1280, London, UK, 1999. Springer-Verlag.

[121] MATISSE. Methodologies and Technologies for Industrial Strength Systems Engineering. http://cordis.europa.eu/search/index.cfm?fuseaction=proj.document&PJ_RCN=5253649, February 2003.

[122] Farhad Mehta. A Practical Approach to Partiality — A Proof Based Approach. In *Proceedings of the 10th International Conference on Formal Methods and Software Engineering*, ICFEM '08, pages 238–257, Berlin, Heidelberg, 2008. Springer-Verlag.

[123] Farhad Mehta. *Proofs for the Working Engineer*. PhD thesis, ETH ZURICH, 2008.

[124] Christophe Métayer, Jean-Raymond Abrial, and Laurent Voisin. Event-B Language. Technical report, Deliverable 3.2, EU Project IST-511599 - RODIN, May 2005.

[125] Christophe Métayer and Laurent Voisin. The Event-B Mathematical Language. Technical report, ClearSy and ETH Zurich, October 2007.

[126] Robin Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[127] Andrew P. Moore. The Specification and Verified Decomposition of System Requirements Using CSP. *IEEE Trans. Softw. Eng.*, 16(9):932–948, 1990.

[128] Carroll Morgan. The specification statement. *ACM Trans. Program. Lang. Syst.*, 10(3):403–419, 1988.

[129] Carroll Morgan. Of wp and CSP. In *Beauty is our business: a birthday salute to Edsger W. Dijkstra*, pages 319–326. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[130] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[131] Ernst-Rüdiger Olderog and Heike Wehrheim. Specification and (property) inheritance in CSP-OZ. *Sci. Comput. Program.*, 55(1-3):227–257, 2005.

[132] M V M Oliveira, A L C Cavalcanti, and J C P Woodcock. Refining Industrial Scale Systems in Circus. In I.R. East, J. Martin, P.H. Welch, D. Duce, and M. Green, editors, *Communicating Process Architectures 2004*, volume 62 of *Concurrent Systems Engineering Series*, pages 281–309. IOS Press, September 2004.

[133] Mehmet Ali Orgun and Wanli Ma. An Overview of Temporal and Modal Logic Programming. In D M Gabbay and H J Ohlbach, editors, *ICTL '94: Proceedings of the First International Conference on Temporal Logic*, pages 445–479, Berlin Heidelberg, 1994. Springer-Verlag.

[134] Overture Community. Overture: Tools for Formal Modelling in VDM. http://www.overturetool.org/, January 2011.

[135] Antonis Papatsaras and Bill Stoddart. Global and Communicating State Machine Models in Event Driven B: A Simple Railway Case Study. In *ZB 2002:Formal Specification and Development in Z and B*, volume 2272, pages 77–100. Springer Berlin / Heidelberg, 2002.

[136] Atanas N. Parashkevov and Jay Yantchev. ARC - A Tool for Efficient Refinement and Equivalence Checking for CSP. In *IEEE International Conference on Algorithms and Architectures for Parallel Processing ICA3PP '96*, pages 68–75, 1996.

[137] Lawrence C. Paulson. *Isabelle: a Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer – Berlin, 1994.

[138] Michael Poppleton. The Composition of Event-B Models. In *ABZ2008: Int. Conference on ASM, B and Z*, volume 5238, pages 209–222. Springer LNCS, September 2008.

[139] Michael Poppleton, Bernd Fischer, Christopher Franklin, Ali Gondal, ColinSnook, and Jennifer Sorge. Towards Reuse with "Feature-Oriented Event-B". In *McG-PLE: Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering*, pages 1–6. Department of Informatics and Mathematics University of Passau, Germany, October 2008.

[140] Marie-Laure Potet and Yann Rouzaud. Composition and Refinement in the B-Method. In *B '98: Proceedings of the Second International B Conference on Recent Advances in the Development and Use of the B Method*, pages 46–65, London, UK, 1998. Springer-Verlag.

[141] ProB. ProB. http://www.stups.uni-duesseldorf.de/ProB/overview.php, September 2008. Online; accessed 27-July-2010.

[142] S. Qin, J. S. Dong, and W. N. Chin. A Semantic Foundation for TCOZ in Unifying Theories of Programming. In *FME 2003 : Formal Methods : International Symposium of Formal Methods Europe, 8-14 September 2003, Pisa, Italy: proceedings.*, number 2805 in Lecture notes in computer science, pages 321–340. Springer, Berlin, September 2003.

[143] Wolfgang Reisig. *Petri Nets: an introduction.* Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[144] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[145] John C. Reynolds. Toward a Grainless Semantics for Shared-Variable Concurrency. In *Proc. FSTTCS'04, volume 3328 of LNCS*, pages 35–48. Springer-Verlag, 2004.

[146] John C. Reynolds. An Overview of Separation Logic. In Bertrand Meyer and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, chapter An Overview of Separation Logic, pages 460–469. Springer-Verlag, Berlin, Heidelberg, 2008.

[147] Abdolbaghi Rezazadeh. *Formal Patterns for Web-based Systems Design.* PhD thesis, Southampton University, 2007.

[148] Abdolbaghi Rezazadeh and Michael Butler. Event-Based Modelling and Refinement of Distributed Monitoring and Control Systems. In *Refinement of Critical Systems (RCS'03)*, 2003.

[149] Abdolbaghi Rezazadeh and Michael Butler. Some Guidelines for Formal Development of Web-based Applications in B-Method. In *4th International Conference of B and Z Users (ZB 2005)*, 2005.

[150] Abdolbaghi Rezazadeh, Neil Evans, and Michael Butler. Redevelopment of an Industrial Case Study Using Event-B and Rodin. In *BCS-FACS Christmas 2007 Meeting - Formal Methods In Industry*, December 2007.

[151] Rodin. RODIN project Homepage. http://rodin.cs.ncl.ac.uk, September 2008. Online; accessed 27-July-2010.

[152] A. W. Roscoe and Geoff Barrett. Unbounded Nondeterminism in CSP. In *Proceedings of the 5th International Conference on Mathematical Foundations of Programming Semantics*, pages 160–193, London, UK, 1990. Springer-Verlag.

[153] A. William Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

[154] Denis Sabatier. Reusing Formal Models. In *IFIP Congress Topical Sessions*, pages 613–620, 2004.

[155] Augusto Sampaio, Jim Woodcock, and Ana Cavalcanti. Refinement in Circus. In Lars-Henrik Eriksson and Peter Lindsay, editors, *FME 2002:Formal Methods—Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin / Heidelberg, 2002.

[156] Matthias Schmalz. The Logic of Event-B. Information Security Technical Reports 698: ftp://ftp.inf.ethz.ch/pub/publications/tech-reports/6xx/698.pdf, ETH Zürich, November 2010.

[157] Matthias Schmalz. Term Rewriting in Logics of Partial Functions. *Proceedings of ICFEM 2011*, 2011.

[158] Steve Schneider. *The B method: an introduction*. Palgrave, 2001.

[159] Steve Schneider, Helen Treharne, and Heike Wehrheim. A CSP Account of Event-B Refinement. In *Refine*, pages 139–154, 2011.

[160] Renato Silva. Renaming Framework. http://wiki.event-b.org/index.php/Refactoring_Framework, July 2009. Online; accessed 27-July-2010.

[161] Renato Silva. Towards the Composition of Specifications in Event-B. *Electronic Notes in Theoretical Computer Science*, 280(0):81–93, February 2011. Proceedings of the B 2011 Workshop, a satellite event of the 17th International Symposium on Formal Methods (FM 2011).

[162] Renato Silva and Michael Butler. Parallel Composition Using Event-B. http://wiki.event-b.org/index.php/Parallel_Composition_using_Event-B, July 2009. Online; accessed 27-July-2010.

[163] Renato Silva and Michael Butler. Supporting Reuse of Event-B Developments through Generic Instantiation. In Karin Breitman and Ana Cavalcanti, editors,

Formal Methods and Software Engineering, volume 5885 of *Lecture Notes in Computer Science*, pages 466–484. Springer Berlin / Heidelberg, Rio de Janeiro, Brazil, December 2009.

[164] Renato Silva and Michael Butler. Shared Event Composition/Decomposition in Event-B. In *FMCO Formal Methods for Components and Objects*, November 2010.

[165] Renato Silva, Carine Pascal, T. S. Hoang, and Michael Butler. Decomposition Tool for Event-B. Technical report, University of Southampton, October 2009.

[166] Renato Silva, Carine Pascal, Thai Son Hoang, and Michael Butler. Decomposition Tool for Event-B. *Software: Practice and Experience*, 41(2):199–208, February 2011.

[167] Graeme Smith. A Semantic Integration of Object-Z and CSP for the Specification of Concurrent Systems. In *Proceedings of FME 1997, volume 1313 of LNCS*, pages 62–81. Springer-Verlag, 1997.

[168] Graeme Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[169] Colin Snook. Combining UML and B. In *Forum on specification & design languages*, pages 24–27, Marseille, September 2002.

[170] Colin Snook and Michael Butler. UML-B and Event-B: an integration of languages and tools. In *The IASTED International Conference on Software Engineering - SE2008*, February 2008.

[171] Colin Snook and Kim Sandstrom. Using UML-B and U2B for formal refinement of digital components. In *Forum on specification & design languages*, 2003.

[172] Jennifer Sorge, Mike Poppleton, and Michael Butler. A Basis for Feature-Oriented Modelling in Event-B. In *ABZ2010*, February 2010.

[173] J. Mike Spivey. *The Z Notation: a Reference Manual.* Prentice-Hall, Inc., 1989.

[174] J. Mike Spivey. *The fuzz Manual.* The Spivey Partnership, Oxford, 1995.

[175] J. Mike Spivey. The fuzz type-checker for Z. http://spivey.oriel.ox.ac.uk/mike/fuzz/, January 2011.

[176] Thomas A. Standish. An Essay on Software Reuse. *IEEE Trans. Software Eng.*, 10(5):494–497, 1984.

[177] Susan Stepney, Fiona Polack, and Ian Toyn. An outline pattern language for Z: five illustrations and two tables. In *Proceedings of the 3rd international conference on Formal specification and development in Z and B*, ZB'03, pages 2–19, Berlin, Heidelberg, 2003. Springer-Verlag.

[178] Susan Stepney, Fiona Polack, and Ian Toyn. Patterns to Guide Practical Refactoring: examples targetting promotion in Z. In Didier Bert, Jonathan P. Bowen, Steve King, and Marina Walden, editors, *ZB2003: Third International Conference of B and Z Users, Turku, Finland*, volume 2651 of *LNCS*, pages 20–39. Springer, 2003.

[179] Kenji Taguchi and Keijiro Araki. The state-based CCS semantics for concurrent Z specification. In *Formal Engineering Methods., 1997. Proceedings., First IEEE International Conference on*, pages 283 –292, November 1997.

[180] Z Word Tools. Z Word Tools. http://zwordtools.sourceforge.net/, January 2011.

[181] Helen Treharne and Steve Schneider. Using a Process Algebra to Control B Operations. In *IFM '99: Proceedings of the 1st International Conference on Integrated Formal Methods*, pages 437–456, London, UK, 1999. Springer-Verlag.

[182] Formal Modelling with UML. http://users.ecs.soton.ac.uk/cfs/umlb.html, September 2008. Online; accessed 27-July-2010.

[183] Logic Viktor Vafeiadis. *A Marriage of Rely/Guarantee and Separation.* Springer Berlin / Heidelberg, 2007.

[184] Laurent Voisin and Nicolas Beauger. Rodin Index Design. http://wiki.event-b.org/index.php/Rodin_Index_Design, September 2008. Online; accessed 27-July-2010.

[185] Jim Woodcock and Ana Cavalcanti. A Concurrent Language for Refinement. In A. Butterfield and C. Pahl, editors, *IWFM'01: 5th Irish Workshop in Formal Methods*, BCS Electronic Workshops in Computing, Dublin, Ireland, July 2001.

[186] Jim Woodcock and Ana Cavalcanti. Circus: a Concurrent Refinement Language. Technical report, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD UK, July 2001.

[187] Jim Woodcock and B. Dickinson. Using VDM with Rely and Guarantee-Conditions. In *Proceedings of the 2nd VDM-Europe Symposium on VDM—The Way Ahead*, pages 434–458, New York, NY, USA, 1988. Springer-Verlag New York, Inc.

[188] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41:19:1–19:36, October 2009.

[189] Jim Woodcock and Carroll Morgan. Refinement of State-Based Concurrent Systems. In *VDM '90: Proceedings of the Third International Symposium of VDM*

Europe on VDM and Z - Formal Methods in Software Development, pages 340–351, London, UK, 1990. Springer-Verlag.

[190] Sanaz Yeganefard, Michael Butler, and Abdolbaghi Rezazadeh. Evaluation of a Guideline by Formal Modelling of Cruise Control System in Event-B. In *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215*, pages 182–191, April 2010.

[191] Pamela Zave and Michael Jackson. Conjunction as Composition. *ACM Trans. Softw. Eng. Methodol.*, 2(4):379–411, 1993.