UNIVERSITY OF SOUTHAMPTON
Faculty of Engineering and Applied Science
Department of Electronics and Computer Science

A progress report submitted for continuation towards a PhD

Supervisor: Prof. Michael Butler
Advisor: Prof. Mike Poppleton
Examiner: Prof. Julian Rathke

Composition using Event-B Notation

by Renato Silva

October 24, 2008

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

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

A progress report submitted for continuation towards a PhD

by Renato Silva

Composition is the process on which it is possible to combine different sub-systems into a larger system. Known and studied in several areas, this has the advantage of reusability and combination of systems especially when it comes to distributed systems. While applying composition, properties must be maintained and proofs obligations need to be discharged in order to the final result to be considered valid. Our goal is to add this feature to the Rodin Platform (using Event-B notation) and study the concerns, properties, conditions, proof obligations, advantages and disadvantages when create/analysing system specifications. Since the composition maintains the monotonicity property of the systems, the sub-systems can be refined independently on a further stage, preserving composition properties.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Formal methods are mathematical based techniques (models) for the specification, development and verification (through formal proofs) of software and hardware systems [1]. When developing large, complex systems or dealing with critical projects, in our best understanding this methods should be applied. This will allow reasoning about the system, based on the requirements. This methodology it is also know as *Model Reasoning* which contrast with the *Test Reasoning* where the system is only tested after the implementation [2]. If there is a fault on an already working/implemented system that uses the latter approach, for instance a design fault, it might be quite late to fix the problem. The consequences usually are time-consuming and expensive (or even life-threatening because with this kind of tests, it is impossible to check all possible states of the system). To tackle this kind of problem, formal methods are used. In our case, we use the notation called *Event-B*, based on another notation called classical B [3]. Event-B has a platform tool called *Rodin* [4] which facilitates the work while modelling systems.

There are some useful techniques that can be applied to this task, like Refinement, Decomposition or Generic Instantiation [5]. The first one it is already used on the Rodin platform, but the last two are not. Refinement is the process that allows the inclusion of more details into a model in a stepwise fashion based on a previous simple model, also know as the abstract model. Decomposition is the process that allows the splitting of a system into sub-systems. After that, the sub-systems can be refined independently, adding more sub-details. Generic Instantiation is basically the reuse of properties of a system and to use those in another system with the help of some theorems to confirm the consistency of the properties and the system itself after the inclusion of the instantiation [5]. So our goal is:

- Develop the **Decomposition** techniques and tools over the Rodin platform.

We start by exploring existing relevant material on composition and decomposition.

Based on that study, we decide to start by developing **Composition** techniques (see chapter 3.2). Only after that, and after the understanding of that process, the decomposition problem will be 'attacked'. It was decided to follow this approach mainly because Rodin already has the refinement process. Using this feature, it is simpler to create a machine that is the composition of several machines and use the refinement process to prove that the composed/wrapped machine is a refinement of another abstract machine. Our work aim to answer the following questions:

- Understand the constraints and consequences on the new system when composing a machine.
- Choose the best way, from the user point of view, to interact with **Composition** technique.
- Understand how to add new functionalities to Rodin. The Rodin platform is based on Eclipse platform [6], which has a complex architecture that needs to be studied deeply in order to develop new functionalities. Using features of this platform, like allowing extensibility through the addiction of software packages called plug-ins [6], it is possible to develop *Composition*

On the one hand, the Composition technique can be very useful for distributed systems. If the intention is to create a large system, an approach would be to start creating small and simpler components and after some refinements, to compose those becoming the result of such operation, a larger and complex system. On the other hand, the decomposition allows to decrease the complexity of a model, by splitting it into sub-models which can be easily manage independently, but also maintaining the same properties that exist before the separation. So the outcome for this work would be:

- To use both techniques (composition and decomposition) in Rodin, giving more development options to the user while creating/analysing a model.
- While building this techniques, we intend to develop some properties and proofs obligations that are required to assure the validation of the entire system that is being modelled.
- Modelling distributed systems in a way that permit us to have independent components that can be joined, split or further developed/refined while keeping their singular properties.
- Try to tackle the criticism that affects the formal methods (specially on industry environments) by giving more options to the designers and developers of large, complex or critical systems, in particular, because of the popularity of composition and decomposition in several other areas. Demonstrating that this approach has

> advantages on the development of projects, in terms of costs, ease of use and accurate development is our weapon to fight the scepticism that surrounds formal methodologies.

A case study is used to help the understanding during the development of this process: is about a Railway system, in which the interaction between the trains, tracks and a communication layer [7] is modelled. This case study has already been developed using other formal methods: CSP and B notation [8] [9].

This document is organised as follows: chapter 2 describes the formal methods, highlighting Event-B notation. Chapter 3 discusses the background and related work, including issues like composition/decomposition, other works and a comparison between all the works. Chapter 4 describes the developed work. Chapter 5 summarises the conclusion and future work. The appendix describes the full specification of the case study applied for this work.

Chapter 2

Formal Methods

This chapter describe in more details what are formal methods, why to use those, classification and applicability. Event-B, a kind of formal method is introduced and explained, as well as the Rodin platform, where systems can be modelled (using Event-B syntax).

2.1 Requirements

The first step to create a system is to gather the necessary requirements in order to assess if it worths to go further with the project. This usually includes some studies about the environment and surroundings, prices evaluation, costs estimation, and based on the motivation for the project, take a decision. During that phase, some requirements documents are created to officialise the necessary information for the project to be started.

So this phase it is quite important on the project life cycle because it is from there, that will be developed [2]:

- Technical Specification
- Design

Although it is quite an important phase, many of the requirements usually lack necessary information or sometimes do not even exist. An error or omission in this document may lead to the repercussion of this error in the abstract model [10] - although the abstract model can reveal inconsistencies on the requirements. Poor quality requirement documents can lead to bigger problems on onward phases like changing specifications during the design or worst, while implementing. Abrial [2] suggests some steps to be followed in order to prevent this kind of issues, saying that a good Requirement Document (RD) should be structured in two parts:

- Explanatory Text: Comments containing the enough detail so the first time reader can understand the content.
- Reference Text: Usually short statements, numbered or labeled that should be easy to read but without too much detail. If it is not a first time reader, this content should be understood anyway.

2.1.1 Classic Software Development Life Cycle

Let's analise the classical software development lifecycle, as an example on how to fit the requirements, specification, design and other phases, by seeing the figure 2.1[1].

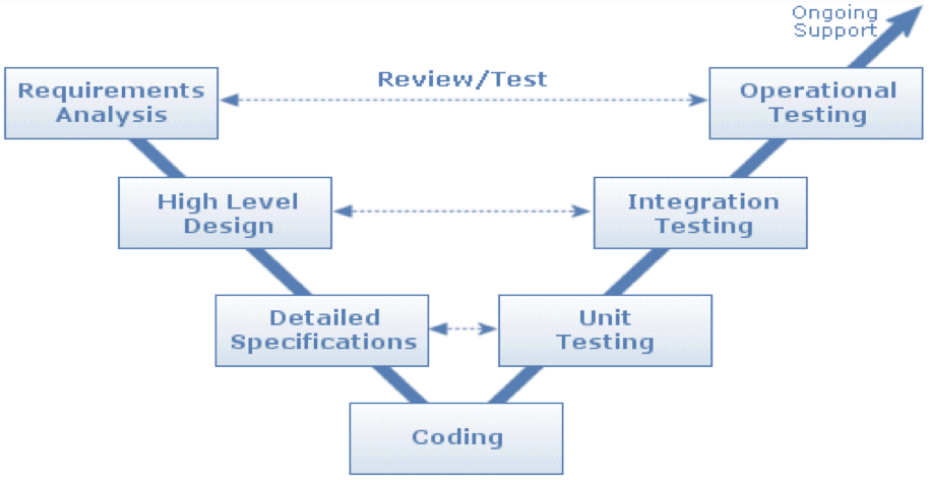


FIGURE 2.1: Classic Software Development Life Cycle

So the development starts with the gathering of requirements and after that phase it will be made the design of the system. After the design, it can be created the detailed specification of the project. This phase has many details because it is just before the implementation (`Coding`) phase. Usually after the coding, there is a team that is defined as the `Unit Testing` and as the name suggests, will test the software created by `Coding` team. In case of a flaw or bug on the software (test according to the specification), the software is returned to the `Coding` in order to correct it - called 'laboratory execution' [5] . This cycle will exist as long as all possible (detectable) bugs are found and corrected. Afterwards, it is made an `Integration Test` with all the software packages generated by all the different teams and packed into one system. This is tested according to the design and if passes it is made the `Operational Test`, where the software it is installed on the client site and checked if is working according to the requirements. After this installation, usually there is a maintenance team responsibly for the support after the installation phase. This is the ideal behaviour of this model.

[1] UK Software House - http://www.uksh.com/about/software-development-life-cycle.php

This model it is used for a long time, in many projects with several success cases (this model is based on the waterfall model [11]). But has some disadvantages/problems as you can see in [11]. Sometimes the `Unit Testing` phase tend to require a lot of time, not to mention that is expensive, giving rise to eventual delays. This happens when faults are found and the cycle between this team and the `Coding` last for a long period of time. And this problem can get worst when the detected bug source is not on the coding but instead it is an *specification* problem. In that case, it is necessary to change the specification or eventually the design. This process is time consuming, expensive and possibly life threatening. Although in this case, the `Unit Testing` discovered the bug, it could have passed the Testing phase unseen. That happens because it is impossible to test every single case to assure that the project is delivered without failures. This incompleteness is the consequence of lack of assurance, beforehand and independently of the tested object, on the expected results of a testing session [5]. The source of problem could have started with some failure on the specification, with some condition or state forgotten or not referred.

The solution could be to find a way to assure that the specification is correct - according to the requirements. And to assure the correctness, the better should be having some kind of proofs that analytically prove that the specification is according to the requirements.

2.2 Formal Methods Definition

Through the use of rigorous mathematical techniques, **Formal Methods** allows to reason about the specification and modelling a system that becomes correct by construction [2].

We can divide the application of formal methods into 3 steps [10] :

- Creation of requirement document
- Development of the Abstraction Model (first model representing the 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 to an Implementation. On a programming software project, there are already tools that automatically do this task.

If we return to the Classic Software Development Life Cycle, the orange arrows (Figure 2.2) represent the inclusion of the Formal Method on this life cycle.

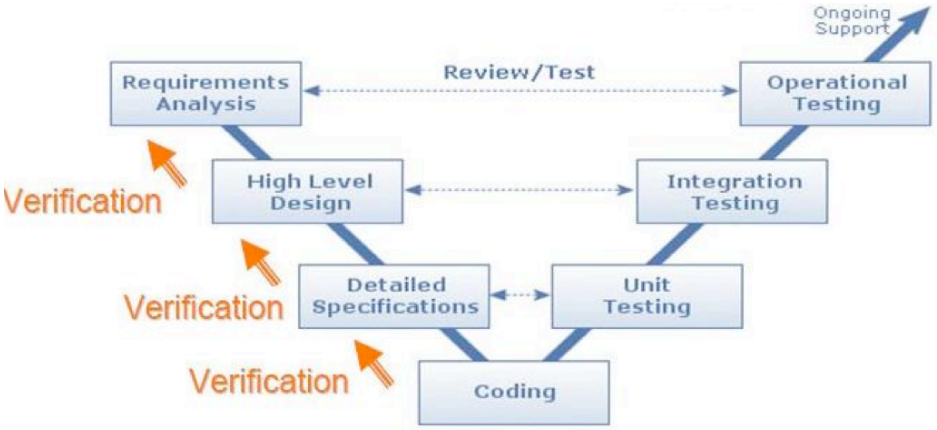


FIGURE 2.2: Classic Software Development Life Cycle with the inclusion of formal methods

2.2.1 Formal Methods Classification

Formal methods can differ in several aspects, like syntax (specification language), semantics or applications (where and when to use). So it is possible to make a classification for each notation and divide into similar categories. The criteria for this division can be very diversified, though. [12; 13] suggest a classification for the formal methods as follows:

- **Model based**: a system, described by state and operations, evolve through the execution of operations - changing the system's state. There is no explicit representation of concurrency and some functional requirements cannot be expressed (temporal requirements). Examples are: Z [14], B [15], VDM [16] or Event-B.

- **Logic based**: Logics are used to describe desirable properties of the system such as specification, temporal or probabilistic behaviour. The validity of this properties relies on the associated axiom system. The final executable specification can be used for simulation and prototype construction and correctness refinement steps are applied on the construction of such systems. Examples are: Hoare Logic [17], WP-Calculus [18], Modal Logic [19] or Temporal Logic [20]

- **Algebraic Approach**: Explicit definition of operations is given by describing the behaviour of different operations without any definition of states. Similar to model-based approach where the concurrency is not explicitly expressed. Examples are: OBJ [21] or LARCH [22].

- **Process Algebra Approach** Explicit representation of concurrent systems is allowed. The system behaviour is constrained by all observable communication between processes. Examples are: CSP (Communicating System Processes) [9],

CCS (Calculus of Communicating Systems) [23], ACP (Algebra of Communicating Processes) [24] or LOTOS (Language of Temporal Ordering Specification) [25].

- **Net based**: Graphical notation are popular because of the ease on specifying system without the need of a deeper understanding of the underlaying framework. Graphical languages are combined with formal semantics, bringing some advantages in system creation/development. Examples are: Petri Net [26], StateCharts [27] or UML-B [28].

2.2.2 State-Based Approach vs Event-Based Approach

Besides the previous classification, formal methods can be seen from a behaviour point of view where we can divide it in two categories [29; 30; 31]:

- State-Based: On a state-based approach, the system is described by a sequence of state changes. A state is a set of assignments to a set of components (frequently variables). Usually rooted in logic and close to how imperative programming languages deal with state. This approach forces a close examination on how the real system is represented in the model [29; 32].
- Event-Based: On an event-based approach, the system is described by a sequence of events/operations/actions changes. The specification is manipulated algebraically, while defining the actions [29].It is used to develop and integrate systems that are loosely coupled (ideal for large-scale distributed applications). Introduces freedom, flexibility and increases the complexity of designing and understanding of systems. The integrated systems can communicate by generating and receiving event notifications [31].

The choice of what kind of approach is better depends on the goal on the system. A state-based approach can change states through the execution of events and an event-based approach can use the system's state to enable the execution of the operations. Event-Based view is suitable for message-passing distributed systems while State-Based view is suitable for design of parallel algorithms [33]. Not always is possible to make a very clear distinction of this two situations: depending on the viewpoint a formal notation is seen, it can show both characteristics views. [30] introduces a framework to choose between both approaches and discusses the possible combinations and applicability while creating/developing a system specification.

2.3 Advantages and Difficulties

Some of the advantages of the formal methods are [34]:

- Clear specifications (contracts)
- Rigorous validation (does the contract specify the right system?) and verification (does the finished product satisfy the contract?)
- Proves the correctness of the system at the specification phase, by reasoning about the requirements details.

On the other hand, there are some difficulties associated with this techniques [2]:

- Construction of the Abstract model, because in general engineers (namely software engineers) don't have the required background/education on modelling [10].
- To use formal methods, the development process has to change (as seen on Figure 2.2) which can be difficult. People need to change the way of working which requires time, good will and includes spending more money than the usual right at the beginning of the project (but usually pays off on the following stages of the project).
- Modelling a system is not the same as implementing one. In general, one starts by modelling the properties so that the final result is correct. And even in the beginning, one has to reason to assure correctness through the use of proofs.
- Lack of proper requirement documents makes this task harder.

2.4 Applications and Examples

Applications using formal methods include complex, critical (that have high human or economic consequences [35]), large scale or high-integrity systems where safety or security is important. Areas like avionics or trains are some of which this kind of issues are important and already use formal methods.

Examples of real applications (Industry) are [2; 10] :

- Paris Metro Line
- Roissy Airport Shuttle (France)

In our point of view, the use of Formal Methods should increase the understanding of the systems, revealing the possible flaws and improve the system. The proofs are just a formal way to reason and to assure the correctness of the system. Some formal notation give a big importance on the proving part, and somehow the user is in a situation where is more interested in proving than understanding all the details behind the specification. Event-B notation (section 2.5) results in focusing more on the system itself and not so much on proofs (advantage of having tool support).

2.5 Event-B

2.5.1 Definition

Event-B is a kind of formal method which combines mathematical techniques, set theory and first order logic. It is used as a notation and method for discrete systems modelling and it is an evolution of others formal method notations like B-Method (also know as classical B)[3], Z[14] and Actions Systems. It is considered an evolution because it simplifies the notation, becoming easy to teach, to learn and unlike the *siblings*, is more suitable for parallel, reactive or distributed system development. Another advantage is the modelling tool support (section 2.6) [36].

Event-B models are described in terms of two basic constructs: *contexts* and *machines*. Contexts are the static part of the model while machines are considered the dynamic part. Contexts can extend (or be extended by) other context and are referred (seen) by machines [37]. A more detailed overview of this two components is given as follows, since some of the properties are useful for the understanding on the described work.

2.5.2 Context

Context is the static part of the model. Which means that is used to store, for instance, the types (Carrier Sets) and constants used during the development of the specification. Table 2.1 shows the Context sections.

2.5.3 Machine

The machine file contains the dynamic part of the model. It describes the system state, the operations to interact with the environment (as well as 'internal operations' [38]) and the properties, conditions and constraints on the model. Table 2.2 gives a brief description of each of the sections.

2.5.3.1 Events - "`Events`"

Events are machine operations and it is the way the system interacts with the surrounding environment. An event consists on a set of guards that define if this operation should be enabled (optional), set of parameters (optional) and set of actions where variable assignments are made (optional). Event properties are described on the table 2.3.

In Event-B, an event can be represented in one of following forms[5]:

(1) $evt \mathrel{\widehat{=}}$ **BEGIN** $S(v)$ **END**

Property	**Description**
Context Extensions - "`Extends`"	The context does not have to be created from the scratch. Can extend some other context that already exists and inherit their properties.
Definition/Types - "`Sets`"	"`Sets`" defines the Carrier Sets that will be used while modelling. Event-B uses Sets to define data structure, so can also be seen has a data type. Properties of data types can be extended on the axioms and theorems section.
Constants - "`Constants`"	Just like the name suggests, constants are used as elements of an enumerated carrier set or for initialise variables. Frequently the constants are used on the axioms section for defining context properties.
Axioms - "`Axioms`"	Axioms define "rules" for the static part, using existing constants and carrier sets. It is applied first order logic and predicate. The properties of the static part are defined (and possibly initialisation states for variables).
Theorems - "`Theorems`"	Theorems are used to help proving properties of the system. Can be seen as similar to axioms, but unlike the latter, theorems must be proved in order to be considered valid. Also uses first order logic and predicate.

TABLE 2.1: Context Sections

(2) $evt \mathrel{\widehat{=}}$ **WHEN** $G(v)$ **THEN** $S(v)$ **END**

(3) $evt \mathrel{\widehat{=}}$ **ANY** t **WHERE** $P(t, v)$ **THEN** $S(t, v)$ **END**

where $G(...)$ is a predicate that denotes the *guard*, v denotes the machine *variables*, t denotes some *parameters* and $S(...)$ denotes a set of *actions*.

The consistency of a machine depends on their own events. Each *action* of an event whose *guard* is true, can modify the state of a *variable* as long as it preserves the machine invariant [39]. There is a "special" event that must exist in all valid machines: INITIALISATION. This event does not have any *guard* and defines the state in which the model starts (first variables assignment).

The relation between machines and context is shown on figure 2.3 [5].

2.5.4 Refinement

Refinement allows the construction of a model in a gradual way, making it more precise and closer to the implementation, thus, closer to the reality [5]. At same time, the overall correctness of the system should be preserved. So the new model is said to be the *concrete model*, while the existing system, on which the refinement was applied, is said to be the *abstract model*. Refinement can also be applied to a machine and the

Property	**Description**
Refine Abstract Machine - "`Refines`"	Possibility to create a machine based on a previous created (and discharged) abstract machine. This section specifies which abstract machine will be refined.
See Context - "`Sees`"	Allows the selection of contexts that are seen by this machine. There is no limitation on the number of context seen by the same machine.
Variables - "`Variables`"	Define the system state. Variables can define the system's properties on the Invariant section and are initialisated in the INITIALISATION event. After that, variable value changes only occur while executing an event.
Invariant - "`Invariant`"	Expresses the properties of the system, using variables, sets and constants through application of first order logic and predicates. Because this are "global" rules of the system, they must be preserved during the whole model processing: this applies especially for events and respective guards to assure a valid specification.
Machine Theorems - "`Theorems`"	Theorems in the *Machine* are very similar to theorems in *Context* (2.1). The main difference is that it is possible to create predicate clauses using variables of the systems (at the Context, variables are not "visible").
Variant - "`Variant`"	New events can be defined in a concrete machine. They must refine an implicit abstract event whose only action is `skip`. Variant is either a natural number or a finite set expression. Some of the new events can be selected to decrease a variant so they do not take control of the system forever [4].

TABLE 2.2: Machine Sections

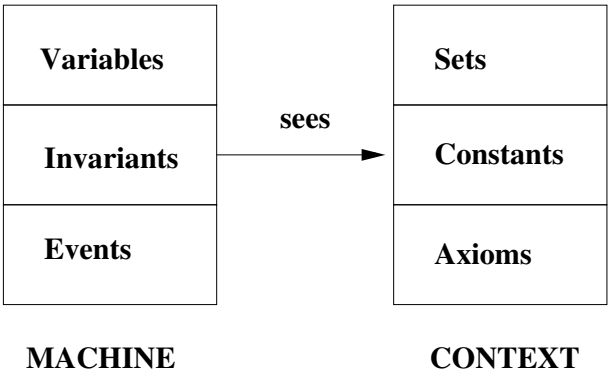


FIGURE 2.3: Machine and Context Relationship

Property	**Description**
Refines Event - "`refines`"	While refining, some events can be refinements of abstract events.
Parameters - "`any`"	Local variables that change/store the state of the machine variables. Can be: *Input*: this kind of parameter receives (it is read) a value. *Output*: this kind of parameter outputs (it is written) a value.
Guards - "`where`"	Guard is a conjunction of predicates that must be true in order to the event be enabled and executed.
Witnesses - "`with`"	When a concrete event refines an abstract one which is parameterized, then all abstract parameters must receive a value in the concrete event. Such values are called witnesses.
Actions - "`then`"	An action makes simultaneous assignments to different variables, changing the system state (it is not possible to assign values to parameters).

TABLE 2.3: Properties of Events

respective context(s) separately. For instance, it is possible to add new sets, constants or axioms to an existing context as long as the abstract context properties are kept [40].

The refinement process needs to be validated (by generating proof obligations) in order to assure the correctness of the entire system. That is achieved by proving that the concrete events keep the behaviour of the respective abstract ones, that the new model does not introduce divergence and the invariant of the concrete model (gluing invariant 2.5.4.1) is preserved for every event enabled. For more details, see [5].

2.5.4.1 Gluing Invariant

When introducing new variables to the concrete model, it is possible to have a relation between the (new) concrete variables and the abstract ones: that relation is called the *Gluing Invariant*. It is similar to an abstract invariant, but besides depending on new variables, also depends on abstract variables. When applying a refinement, for instance, it can be intended to introduce a new variable w that represents a property in a way that is closer to the reality. If that property is already defined on the abstract model (although in a general, abstract fashion) using an abstract variable v, it is necessary to relate v and w in a way that "glues" the state of the concrete model to the abstract one, using the gluing invariant $J(v, w)$.

2.5.4.2 Proof Obligations for Refinement

The refinement proof obligations at Event-B can be classified into obligations for preserving *safety* and obligations for preserving *liveness*. The proof obligations that exist

for refinement are :

- Well-Definedness (WD): of invariant, event, guard, theorem or variant.
- Guard Strengthening (GRD) : relation between the abstract and concrete guard for each event.
- Action Simulation (SIM) : relation between abstract and refined event action.
- Decreasing of variant (VAR): when new events are added.

2.5.4.3 Safety and Liveness Properties

Lamport [41] defines informally, two general classes of system properties [42]:

- **Safety**: states that something (wrong or bad) will not happen.
- **Liveness**: states that something (good and desirable) must happen (will eventually happen).

When refining, it is intended to introduce more details to the model or make design decisions. This implies that the overall behaviour of the abstract model is kept and that the concrete model does not get onto two states [33]:

- Divergence: occurs when a system behaves chaotically. Happens whenever some events are aborted.
- Deadlock: occurs when no event is enabled and as a consequence, the system's state does not change (this state can be provoked voluntarily: after some state changes, it is intended to "freeze" the state of the system).

To keep the Liveness property while refining, two sub-properties must uphold [42]:

- Enableness: assures the abstract behaviour to be reflected on the concrete model. If an event is enabled on the abstract model, it should be enabled on the concrete model (meaning that the guard on the abstract event in conjunction with the concrete invariant, should imply the guard of the concrete event plus the new events' guards).
- Non-Divergence: Like mention above, divergence happens when events are aborted because a new event takes control of the system and becomes enabled forever. This sub-property assures that such situation does not happen (with the use of Variant as long as the invariant hold)

When refining, safety properties are kept by:

- Guard Conditions
- Invariant

When refining, liveness properties are kept by:

- Well-Definedness
- Abstract and Concrete Guard Condition(s) for each event
- Invariant (Gluing Invariant or Concrete Invariant)
- Variant (in case new events are added)

When refining, some other properties can be concluded:

- The non-determinism of individual actions can be reduced, as long as the concrete system as a whole preserves the abstract behaviour.
- It is possible to reduce the range of output values on an event that has output behaviour.
- The range of input values on an event that has input behaviour has to be preserved.
- In the overall, the external choice must be preserved although internally, some individual events may have a non-determinism reduction.

During the composition it is possible to refine a system, so this properties should be preserved in order to consider the final result valid. The Rodin platform assures already the safety property, but not completely the liveness property, since does not prevent deadlocks (support the enableness proof obligations). Since it is an important aspect of our work, we intend to study more what is involved in the assurance of those properties and try to implement it on the Rodin platform.

2.6 Rodin Platform

The RODIN (Rigorous Open Development Environment for Complex Systems) Platform [4] is the result of an EU research project. It is a software tool, based on modern software programming tools developed to use Event-B notation [43].

Based on the idea that a large, complex or critical project should be started by modelling the specification and reason about it, this tool was created to help the development of specifications. It has a bigger, ambitious purpose which is to decrease the gap between the industry and the criticism that affects the formal methods (especially on industrial environments). It should prove that it is a reliable tool and the modelling does not have to be a cumbersome, hard to achieve and that everyone with some programming and mathematics background can adjust itself to the concept of creating specification and to the tool. The main idea is to increase the understanding of the system that is being built, abstracting as much as possible from the generated proofs (the tool tries to solve as many as possible, based on the data contained on the model) that are not more than the formal proof (model is sound) that the created system correspond to the requirements [43].

2.6.1 Features

The Rodin has some features which makes it an unique tool for the development of models, helping on the understanding of the system as a whole. Some of those features are [39]:

- *Openness* , i.e., is an open source tool (based on Eclipse Platform (Java Development Tools of Eclipse), which is an open source platform for software development. [6]), allowing users to integrate their own tools and where the source code is available to everybody who is interested. It works has a complement for the rigorous modelling development [43]. The intention is to benefit the industry by permitting the integration of any functionality that is considered necessary, on the same software tool. At same time, the tool is not restricted to any concepts which possibly will increase the longevity of the platform.

- Contains a database (*repository*) where the persistence data of the model is stored. Does not have a fixed syntax for the modelling notation (not constrained to a syntax makes the tool very flexible).

- *Static Checker* which validates if the system properties are valid and in the case of problems, raise warning/errors. Although the platform does not have a fixed syntax, the notation used (Event-B) has, so it must be checked for eventual syntax errors.

- *Proof Generator* which generate the proofs to be discharged in order to consider the model valid and *Automatic Prover* which is a theorem prover that tries to solve as many proofs as possible automatically. The proofs that are not automatically discharged, have to be proved interactively.

- *Graphical User Interface* used to create/edit the model and reason about the system (interactive proofs).

- *Extensibility* related to Openness, allows the integration of features or functionalities to the tool (e.g. model checkers, theorem provers, animators, UML-B [28], Latex,etc), through the development of *plug-ins*. A plug-in in Eclipse is a component that provides a certain type of service within the context of the Eclipse workbench [44]. In other words, is a piece of software (Java) that follows a defined structure and can be embedded to the Eclipse (in our case Rodin), extending a new functionality to the platform.

The high level of extensibility is reflected by, for instance, the ability to extend the default theorem prover (B4free provers provided by ClearSy [45]), model checking (ProB provided by University of Düsserdorf [46]) or even animate models (Brama provided by ClearSy [47] 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 class are introduced and "merged" with Event-B notation [48]. On figure 2.4 can be seen a screenshot of the user interface for Rodin Platform.

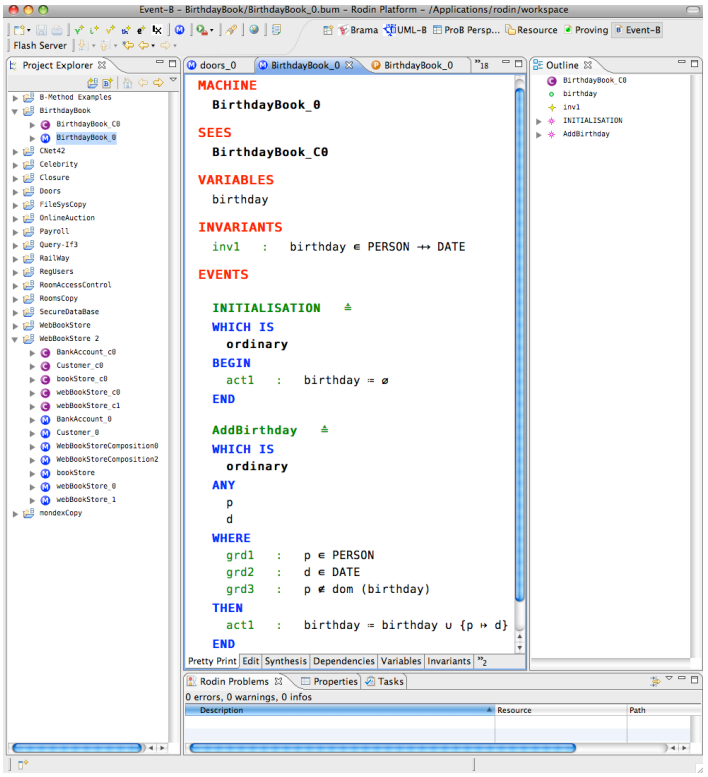


FIGURE 2.4: The Event-B Perspective

2.7 Plug-in Development for Rodin

We intend to add a new functionality to the Rodin platform, so by taking advantage of the Rodin's extensibility, we decide to develop a plug-in to achieve our goal. It was necessary to study and understand how a plug-in works in Eclipse, including their architecture, features, concepts, integration and execution.

Although a very powerful and easy to use tool, Eclipse has a complex architecture behind which requires spending a certain amount of time just to understand how package dependencies, rules on how to implement a plug-in and the interaction between different plug-ins work. The default Rodin platform features are also plug-ins that were added to the main component, the Rodin Database (Rodin Core). Although it required a long time to understand the technology and the implementation behind Eclipse, the Rodin creators consider nowadays that it worthed the trouble and the final result it is considered a success.

Chapter 3

Background in Composition and Decomposition

This chapter introduces the previous work related to composition/decomposition. They are well known (theorically speaking) techniques used in several areas like mathematics, logic, programming and even on other formal notations.

Compositional reasoning for state-based and also for process algebra system is presented. Composition and decomposition and how they are intended to be used, is described. After that, we present some approaches using different formal methods and focus on the differences/similarities between the already existing work and our approach. There exist many formal notations that could be used in this section, but we will discuss the ones that inspired Event-B or have a similar approach, focusing on the composition methodology for each one:

- Event-B
- B-Method / Classical B
- Action Systems
- CSP
- Z Notation
- VDM Notation

3.1 Compositional reasoning

When using the composition technique, it is intended to combine components and respective properties. In order to assure that the composition of parallel processes is valid,

some reasoning about the model is required. The main problem in model checking that prevents it from being used for verification of large systems is the *state explosion problem*. The primary cause of this problem is the parallel composition of interacting processes, where the number of states in the global model is exponential in the number of component processes [49]. The state explosion can be alleviated using *compositional reasoning*: verification of each component of the system in isolation, allowing global properties to be inferred about the entire system. Some approaches that try to solve this problem are presented here, focusing in particular on state-based and process algebra systems [49].

3.1.1 Compositional reasoning in state-based system

A possible way to model check state-based systems would be to compute the image of all the states that the system can achieve. Build this correspond to construct the global transition relation of the system. But sometimes that number is too large (or even infinite), so on a large scale is an approach that becomes unpractical. The most suitable approach would be to check the model without constructing the global transition relation explicitly. Using **Partitioned Transition Relations** it is possible to do that [49]:

- Disjunctive Partitioning: used for asynchronous systems by writing the global transition relation as a disjunction of the transition relations for the individual components of the systems. This technique allows to compute relational products for much larger asynchronous systems.
- Conjunctive Partitioning: used for synchronous systems. Because most of the systems often depend on a small number of variables, it is possible to optimize the computation of a relational product by using *early variable elimination* for variables in each transition. Although making a locally optimal choice does not guarantee an optimal solution, the minimum sum cost function seems to provide a good performance on most examples.

Other approach is to use **Lazy Parallel Composition**, where the global transition relation is never constructed as well. Instead, a restricted transition relation for all processes is created where 'important' states match with the global transition relation but it may behave in a different way for other states. The advantage is that in many cases it is possible to construct a significantly smaller restricted transition relation [49].

The **Interface Processes** approach is based on the fact that the state explosion problem is usually more severe for loosely coupled processes which communicate using a small number of shared variables. Using the *cone of influence reduction* [49] for each process (consider only variables that are somehow related or relevant for that process) will reduce the number of variables for each process. The method considers the set of variables used in the interface between two components and minimizes the system by eliminating

events that do not relate to the communication variables. The properties that refer to the interface variables are preserved, but the model becomes smaller [49].

A compositional proof system for shared variable concurrency is proposed in [52]. The first compositional characterisation of this kind of concurrency is called Rely/Guarantee (R/G) and was conceived by Jones [53]. So [52] demonstrate that R/G style proofs can be embedded in this approach, that makes direct use of history variables: auxiliary variables that record the sequence of state-changes and use the strongest postconditions assertions style. A very similar approach, but known as **Assume-guarantee reasoning** is a technique that verifies each component separately. Properties of the environment are *assumed* and if that is *guaranteed* by the other components, it is possible to conclude that the verified properties are true on the entire system, without constructing the global state graph. The assume-guarantee rule is sound [49; 50]. Assumptions have traditionally been defined manually, which has limited the practical impact of such reasoning. Over the last decade, researchers have focused on the automated generation of assumptions for assume-guarantee reasoning [51]. Some of that work can be seen in [51] where is made a small survey about automated assumption generation.

The **open system** approach verifies the correctness of components in isolation, before they are part of any system. Proofs are harder since is made assumptions describing a set of possible environments instead of a completely specified context. The advantages are that correctness proofs of a complete system can rely on components specification and that is possible to embed parts of a correctness proof into components, making these available each time a component is used to build a system (reuse of proofs without the need for proving them again) [50].

A semantic approach using mathematical theory of state-based reasoning is presented in [54], in particular for synchronous communication. Some advantages of such approach are highlighting the very concept of compositional state-based reasoning without any syntactic overhead and serves as basis for the encoding of the program semantics and corresponding proof rules inside tools that support program verification. The reasoning is done solely through specifications of their parts, without any reliance on their implementation mechanism.

3.1.2 Compositional reasoning in process algebra system

One form of compositional reasoning for CSP is described in [55], whereby refinement properties of a composite system can be inferred from (separately-proven) refinement properties of its components. Such rules are typically used for reasoning compositionally about systems where each component is specified independently of its environment, i.e. where the same specification would be appropriate whatever the context of the component in the wider system [56].

Assumption-Commitment is a further study on the Rely/Guarantee proof system presented by Jones [53]. This approach is proposed in [56] for CSP model checking, especially in the context of refinement-style model checking. In this case, the specifications include separate, explicit descriptions of both the environment in which components are supposed to operate correctly and the desired behaviour of the component in such an environment. A similar approach, but based on a predicate transformer called the *weakest guarantee* and a corresponding binary relation *guarantees* is proposed in [57].

A study, although applied to a combination of probabilistic finite state-behaviour and non-determinism is also presented in [58], using CCS. Also it is a good source for a survey about axiomatic theories of process algebra. Both a system and its desired external behaviour can be expressed as process terms. The correctness of the system can be verified by proving that these two terms are equivalent.

[59] introduces a compositional proof system applied to (a slightly modified version of) CCS, as a model of concurrency. To prove a property of a parallel composition, first it is proved that the corresponding properties hold of each component and then it is inferred in the proof system that the global property of the composition also holds. It is proposed a method of combining model checking with theorem provers, when the verification of the components is accomplished by model checking. One of the most important issues of this area is to know if the proof system is complete in general or for any particular class of CCS processes.

3.2 Composition/Decomposition in Event-B

On the one hand, composition is a technique used to aggregate sub-systems and generate larger systems. The motivation for the use of such techniques is the reusability of sub-components and the possibility of interaction between systems. In a distributed application, this insight is even more important since the intention is to have independent systems interacting with each other, and at same time keeping all the individual properties.

On the other hand, we have decomposition, which is the process on which a system is split in two or more parts. It is done when a system becomes too complex to be managed. So it is divided in sub-systems keeping the manageability/tractability. An interesting property to be explored is the independent refinement of each sub-system. This process must satisfy the constraint that the re-composition of (refined) components should be easy; in other words, the result of re-composition can be obtained directly without the decomposition [5]. The usability of this technique grows when it comes to distributed systems as well. Having a large system can become unpractical so dividing in separated sub-systems can only benefit each of the sub-systems while at the same time, it will

benefit the system has a whole. This task is only feasible only when the constituents have a certain level of modularity (i.e independence or non-interaction) [60].

We intend to use the knowledge acquired with the study and development of the composition to achieve the decomposition. This includes understanding the rules, the proof obligations, the constraints and the result of case studies and applying them on the development of the decomposition technique on Event-B. Unlike composition, that already have the necessary artifacts to be applied on classical B, decomposition can only be achieved manually. The goal is to use the tool support that Event-B has and extend it to include a way to decompose in a more automated way. This study will include the "how to decompose", which rules/proofs to be maintained/generated in order to consider this operation valid and based on previous developed work and studies, choose a suitable approach.

The composition technique itself is already complex (since it involves different systems), which raise more complex proof obligations. At same time, it is intended that the user keep the understanding of the system and that the proofs generated don't disrupt this concept, becoming too burdened in discharging the proofs than on improving the knowledge of the model.

In this chapter we will introduce what we intend to achieve with the composition, the properties to be kept, the proofs to be generated in order to consider a composition operation valid. Our approach can be seen as an event-based view because the interaction between the sub-systems is made through synchronised events (selected events from each sub-system that are merged). So one of the restrictions is that there is **no common state variable between the composed sub-systems**. The merged events can pass values (parameters) while synchronised. The next section discusses from the simple parallel composition (without parameters) to value-passing composition and defines the conditions to a composition be considered valid.

3.2.1 Parallel Composition

For Event-B, there are two approaches for the composition operation: *shared variable* and *shared event*. There will be made a brief description of the first and a more details explanation of the second, since this document is based in that approach.

3.2.1.1 Shared Variable Composition

This approach, also know as *type A* style, because of the author, Abrial, is one way to accomplish composition. [61; 5] propose a (de)composition that has a state-based view, since consist in variable sharing. When dividing a system in sub-components,

the splitting is done in terms of (external) variables. If a variable is shared by sub-components, it is necessary to introduce some external events (containing only external variables) on the sub-components, that simulate in each sub-component how the external variables are handled in the other. This approach also allows the independent refinement of each sub-component, as long as the common shared variables are refined in the same way, which is a constraint that does not exist in our approach. The re-composition of the (refined) sub-components is possible and this result should be proved as a refinement of the original system [61].

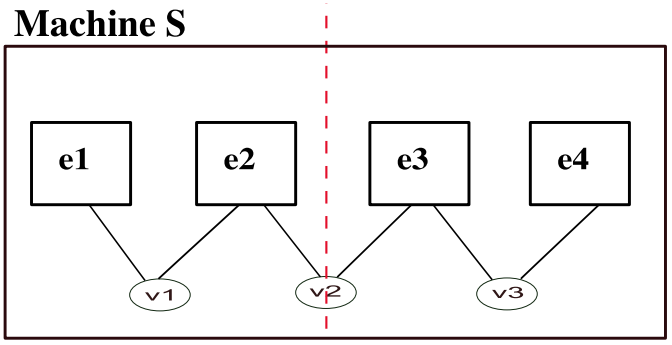


FIGURE 3.1: Shared Variable Decomposition

3.2.1.2 Shared Event Composition

Shared Event Composition is also know as *type B* style, because of the author, Butler, and it is the one we follow for the document. The composition is made in terms of shared events without variable sharing.

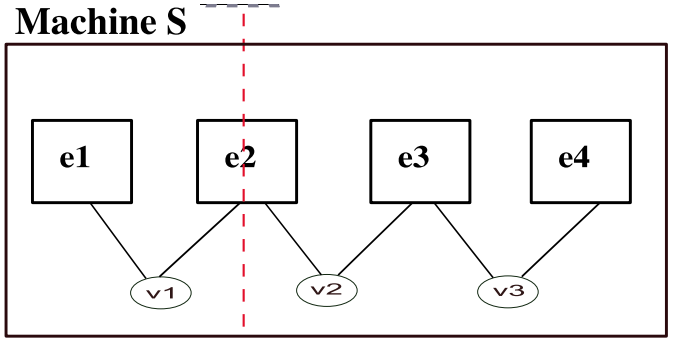


FIGURE 3.2: Shared Event Decomposition

In Event-B, there is no operator for parallel composition unlike in some other formal methods (CSP or Action Systems). [38] introduces an operator for composing machines ($\|$) and we follow our study based on this paper.

So if we have two events $evt1$ and $evt2$ that belong to different machines, as described below:

- $evt1 \mathrel{\widehat{=}}$ **WHEN** $G(m)$ **THEN** $S(m)$ **END**
- $evt2 \mathrel{\widehat{=}}$ **WHEN** $H(n)$ **THEN** $T(n)$ **END**

then the parallel composition of this two events, evt1 $\parallel$ evt2, can be expressed as:

- $evt1 \parallel evt2 \mathrel{\widehat{=}}$ **WHEN** $G(m) \wedge H(n)$ **THEN** $S(m) \parallel T(n)$ **END**

where m and n are sets of independent (no common) variables, $G(m)$ and $H(n)$ are guards and $S(m)$ and $T(n)$ are actions from the $evt1$ and $evt2$ respectively. So when both events are synchronised, they are composed, generating a new event whose guard is the conjunction of the original guards and the actions are statements executed in parallel. So the variables m, n are updated while the (possible) other variables of each system keep the same value.

Parallel Composition with Value-Passing The composition can be more complex, including parameters on the merged events. Parameters can vary on their scope, being internal (just visible for the source machine) or external (interaction with other machines) and this property will limit their use when composing, since only external parameters can be used for the event interaction.

When composing two events, if both of events have parameters, it is possible to pass one parameter (event "sender" sends the output parameter) to the other event (event "receiver" of the input parameter). Having events $evt3$ and $evt4$:

- $evt3 \mathrel{\widehat{=}}$ **ANY** $t?, x$ **WHERE** $G(t?, x, m)$ **THEN** $S(t?, x, m)$ **END**
- $evt4 \mathrel{\widehat{=}}$ **ANY** $t!, y$ **WHERE** $H(t!, y, n)$ **THEN** $T(t!, y, n)$ **END**

the composition operation ($evt3 \parallel evt4$) can be expressed as:

- $evt3 \parallel evt4 \mathrel{\widehat{=}}$
  ANY $t!, x, y$ **WHERE** $G(t!, x, m) \wedge H(t!, y, n)$ **THEN** $S(t!, x, m) \parallel T(t!, y, n)$ **END**

where t, x, y are set of parameters from each of the events $evt3$ and $evt4$. We use "!" for representing a parameter that has an output behaviour and "?" for representing an input parameter. Because there are common parameters between both events (t), they are composed as seem above. Note that in this case, $evt3$ has $t?$ as input parameter and $evt4$ has $t!$ as output parameter and the result is $t!$ itself an output parameter. This can be interpreted like a way of modelling message broadcasting, when composing input - output pairs of events. Two conditions must be validated for this kind of composition [62]:

1. The types of the common parameters must match, or at least be related. Meaning that if t? has type T1 and t! has type T2, then $T1 \cap T2 \neq \varnothing$.

2. $Inv4 \Rightarrow [evt4](G(t?, x, m))$

$Inv4$ is the invariant for the machine containing $evt4$. Condition 2 expresses that the result of the $evt4$ (the output value) will always be accepted by the input event $evt3$ (because it is accepted by the respective guard G).

It is also possible to compose input-input pairs of events:

- $evt5 \mathrel{\widehat{=}}$ **ANY** $t?, x$ **WHERE** $G(t?, x, m)$ **THEN** $S(t?, x, m)$ **END**
- $evt6 \mathrel{\widehat{=}}$ **ANY** $t?, y$ **WHERE** $H(t?, y, n)$ **THEN** $T(t?, y, n)$ **END**

and the composition $evt5 \parallel evt6$ results in:

- $evt5 \parallel evt6 \mathrel{\widehat{=}}$
  ANY $t?, x, y$ **WHERE** $G(t?, x, m) \wedge H(t?, y, n)$ **THEN** $S(t?, x, m) \parallel T(t?, y, n)$ **END**

The composition between input-input pairs of events result in an input event as can be seen above. The last possible choice (join output-output pair of events) is not permitted since this could result in the model reaching a deadlock state. This situation could happen if the merged events do not return the same values for the common output parameter.

It was shown the composition between only two joint events but this is not a restriction. It can be merged more than two events as long as the original events come from different machines and respect the presented conditions.

3.3 B-Method

Like alluded before, Classical B ([15]) can be seen as a parent of Event-B, a formal approach for the specification and development of computer software systems [3]. Using classical B, a system has a state and through operations, that state can change. The properties that should be preserved during the operations are 'invariants'. Just like Event-B, B-Method can be seen as both state-based view (explicit notion of "state" expressed by variables) or event-based view (operations can happen nondeterministically). Unlike Event-B, whose events (equivalent to operations on classical B) are atomic, in B-Method there are pre and post conditions for the operations. The creation of models

on top-bottom style is similar to Event-B, where the beginning of the model is very simple and through stepwise refinement, it is added more details and complexity. Besides *abstract machines* and *refinements* like Event-B, there is a third component in classical B:

- Implementations: corresponds to a special kind of refinement machine from which code can be produced, respecting the original abstract specification. There are different ways of generating the code and also can be used different tools for that like B-Toolkit [63] or Atelier B [64] [3].

Since B-Methods focus on software systems, the final result - implementation model - although similar to another refinement step, includes programming constructors and has some restrictions on the syntax to the used.

The B-Method already includes a syntax for the composition. There are some keywords that can be used to compose models as can be seen in [65; 3]:

- **Includes**: links abstract machines or refinements to abstract machines (similar to schema inclusion in Z). Allows the extension of abstract machines. If machine $M2$ includes machine $M1$, all the information (and state) of $M1$ is part of $M2$. $M1$ and $M2$ are independently defined - no related information. There is no restriction on the number of included machines and it is even possible to include machines that include own subsidiary machines. But the same included machine, can only be included by one machine each time. *Includes* is transitive, meaning that if $M3$ includes $M2, M1$ becomes visible to $M3$.
- **Imports**: links implementations to abstract machines, allowing the creation of software layers. When used, values of variables are only accessible via operations, in order to preserve the invariant of the imported machines. There is no restriction on the number of imported machines, but can only be imported by one implementation.
- **Sees**: allows sharing of sets, definitions and variables in a restricted way: no variable can be modified by the seeing component. Despite that, can be consulted (directly or via operation calls). Other property is that variables of the seen machine are not visible in the invariant of the host, so cannot be used to represent abstract variables. There is no transitivity between machines like in *Includes*. A machine can be seen by as many machines as desired.
- **Uses**: introduces a form of sharing between abstract machines. Allows the extension of abstract machines in multiple ways, but only on abstract machines. Works as a read only access and can be considered a generalisation of the *Sees* relationship. The only difference is that the state of the machine that is used, can be

referred on the host machine. If $M2$ *uses* $M1$, than $M2$ can express relationships about its own state and $M1$.

3.3.1 Parallel Composition for Classical B

Classical B has already an operator for parallel composition: $\|$. Used with the **Includes** allows the composition of machines, through an event-based interaction using operations. [3] has a more detailed discussion about it. Here we will present a brief description of the use of parallel composition (parallel operations using multiple inclusion) in classical B and compare with our approach.

Since classical B uses pre-condition (unlike Event-B), combining operations result in the conjunction of the preconditions (and/or the conjunction of the guards), and the body of the parallel combination will be the parallel combination of all the bodies. This can be expressed with the following:

- PRE $P1$ THEN $S1$ END $\|$ PRE $P2$ THEN $S2$ END
- $=$ PRE $P1 \wedge P2$ THEN $S1 \| S2$ END

where $P1, P2$ are pre-conditions and $S1, S2$ are operations statements.

The composition of events is also similar to Event-B, and it is possible to compose operation with both input behaviour or input/output behaviour. For the same reason as Event-B, it is not possible to compose operations that have both output behaviour [33].

3.4 Communicating Sequential Processes - CSP

CSP[9] is a process algebra formal method developed to tackle issues related to parallel processing and interaction between systems[66], inspired by imperative language of guarded commands from Dijkstra[67]. The behaviour of the system is described through processes. 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 [68]. The processes are constrained in the way in which they can engage in the events of its alphabet, using CSP process term language [8].

So 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 '$[]$'. $P \, [] \, Q$

represents the process that offers the choice to the environment between behaving as P or as Q. There is also a nondeterministic-choice operator '$\sqcap$'. $P \sqcap Q$ represents the process that internally chooses between behaving as P or Q, without any environment control.

3.4.1 Parallel Composition

The parallel composition of two processes P and Q is expressed as $P \parallel Q$. 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 [68]:

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

An event common to P and Q becomes a single event in $P \parallel Q$. A recursive definition is written $(\mu X \cdot F(X))$, where $F(X)$ is some expression containing X [62]. A composition between CSP processes $N1$ and $N2$ that have a common event c, can be expressed using the algebraic laws of CSP as follows:

- $N1 \mathrel{\widehat{=}} (\mu X \cdot a \rightarrow c \rightarrow X)$, $N2 \mathrel{\widehat{=}} (\mu X \cdot b \rightarrow c \rightarrow X)$
- $N1 \parallel N2 = (\mu X \cdot a \rightarrow b \rightarrow c \rightarrow X \,[\!]\, b \rightarrow a \rightarrow c \rightarrow X)$

meaning that the events a or b can be executed in either order and then both processes synchronise on the event c [68].

3.4.1.1 Hiding Operator

It is possible to hide processes from the environment (especially when composing). This can be done using the operator '\'. If $C \subseteq \alpha P$, then $P \backslash C$ describes the process that behaves as P but without the events in C. Using the algebraic laws, hiding can be represented as [68]:

- $(a \rightarrow P) \backslash C = a \rightarrow (P \backslash C)$ if $a \notin C$
- $(c \rightarrow P) \backslash C = (P \backslash C)$ if $c \in C$

3.4.1.2 Traces

A trace of the behaviour of a process is a finite sequence of symbols that represent events that were engaged by this process until a certain time. It will be a list of visible events executed by a process and we ignore the possibility of two events occur simultaneously: independent of the order, both of them would be seen and executed. The traces model does not distinguish between internal or external choice, nor model divergence. The representation of a trace can be written as follows:

- $\langle x, y \rangle$

where x and y and two events in which x is followed by y [9].

3.4.2 Parallel Composition with Value Passing

In CSP exists a special class of event known as *communication*. It is an event described by a pair $c.v$, where c is the name of the channel on which the communication occur and v is the content of the communication or the value of the message to be communicated. The set of all messages which a process P can communicate on channel c is defined as:

- $\alpha c(P) = \{v | c.v \in \alpha P\}$

Channels can have two types: *input* and *output*.

A process ready to input (receive) any value x on the channel c, and then behave like $P(x)$, is defined as:

- $(c?x \rightarrow P(x)) = (y : \{y \mid channel(y) = c\} \rightarrow P(message(y)))$

A process that outputs (send) a value v on the channel c and then behaves like P is defined as:

- $(c!v \rightarrow P(x)) = (c.v \rightarrow P(x))$

So it is possible to have interaction between processes, through the use of input or output channels. If an output channel is in parallel with an input channel with same name, the passing of values it is possible. Channels can be considered members of the alphabet of the process and used for communication in only one direction and between two processes only [9]. (Note that this is different from our approach, where it is possible

to have interaction between more than two events at same time and also not restricted to the name of the event itself.)

If two processes P and Q are composed in parallel, and both have a common channel c, interaction will happen whenever both processes are ready to engage on the common channel. If P is ready for $c!v$ (output channel) and process Q is ready for $c?x$ (output channel), v can be passed from P to Q, which can be represented by the following algebraic law [62]:

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

Like expected the result is an output channel and the process Q receives the value v (instead of x before the composition). Just like in our approach in Event-B and in Action System, it is required to confirm that the output value is accepted by the input channel. This can be also applied for channels with input-input behaviour.

3.4.3 CSP Semantics

The semantic model for CSP can be expressed through *traces* model in which a process behaviour is modelled by a non-empty, prefix-closed set of event-traces. The semantics of a CSP process P, with alphabet A, is modelled by a set of failures, $F[\![P]\!]$, and a set of divergences, $D[\![P]\!]$. A failure is a pair of the form (s, X), where $s \in A^*$ (the 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 $F[\![P]\!]$, then after engaging in the sequence of events s, a process may refuse all events in X. A divergence is simply a finite event-trace and $s \in D[\![P]\!]$ means that, after engaging in s, process P may diverge.

3.4.3.1 HIDE function

The semantics for CSP define the hiding of events, which are considered internal behaviour, thus, not visible by the environment. Correspond to the introduction of new events in Event-B. The representation of such process can be expressed using a function, HIDE, of $[\![P]\!]$ and C:

- $[\![P \backslash C]\!] \mathrel{\widehat{=}} HIDE([\![P]\!], C)[\![$

where P is a CSP process, $[\![P]\!]$ is short for failures-divergences semantics of P and C are the processes that are not visible to the environment [68; 62; 9].

3.4.3.2 PAR function

It is possible to define the semantics of parallel CSP processes as a function. So the process $P \parallel Q$ can be expressed semantically as:

- $[\![P \parallel C]\!] \mathrel{\widehat{=}} PAR([\![P]\!], [\![Q]\!])$

The laws that govern the behaviour of $(P \parallel Q)$ are exceptionally simple and regular. We will introduce a few of those laws although there are more properties as defined in [9]:

- 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.

3.5 Action Systems

Action System provide 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, although those actions do not have to terminate themselves. An approach to parallel and distributed systems was introduced by Back and Kurki-Suonio [69]. Further work allowed action system to be used on parallel and distributed systems in a stepwise manner [70], which is the approach that we are more interested in. The latter can be achieved through synchronized value-passing and there are already rules and definitions that allow the refinement and decomposition of such systems [62; 68].

A basic action system $P = (A, v, P_i, P_a)$ consists on a list of (alphabet) labels A, a list of variables v, a set of labelled statements (actions) $P_a = \{P_\alpha \mid \alpha \in A\}$ and a set of initialisation statements P_i. 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 [62].

[62] exposes a a composition using action systems from an event-based point of view (based on CSP). The interaction between systems is through common labelled actions. The next sections describe the parallel composition (including parallel composition with value passing).

3.5.1 Internal Actions and Hiding Operator

Internal actions are actions that are introduced after a refinement and do not have a correspondence to another action on the abstract model (refine the *skip* action). Can be considered similar to new events on an Event-B notation. So an action system that has internal actions can be represented as:

- $P = (A, v, P_i, P_A, P_H)$

where P_H correspond to all the internal actions of the action system P [62].

Based on CSP, action system has an operator that allows the internalisation of a set of actions [62]. This allows a set of actions $C \in alphabet(P)$ to be hidden from the environment and be considered internal. So an action system, having internal actions, $P = (A, v, P_i, P_A, P_H)$ has a set of internal actions $C \subseteq A$ where:

- $P \backslash C \mathrel{\widehat{=}} (A - C, v, P_i, P_{A-C}, P_H \cup \{var\ x?, y!\ \cdot P_c\ | c \in C\})$

where $x?$ are the input parameters used by the input actions of P and $y!$ are the output parameters used by the the output actions of P.

3.5.2 Parallel Composition

A type of composition, using shared variables is possible and described on [71]. Using *Renaming* for local variables that have the same name, *Hiding* to hide global variables, making them locals, it is possible to compose a system, allowing to clearly state which variables are used by which action. Other approach, proposed by Butler in [68] and based on CSP, uses shared actions where only the occurrence of actions is observable [62; 9]. Since our approach is related with the latter, we shall describe it in more detail.

Using the operator $\|$, the hiding operator ($'\backslash'$) and assuring that the actions systems have no shared state-variables, it is possible to represent the composition between two action systems P and Q, P $\|$ Q (from an event-based point of view). Common label actions are synchronised and composed in parallel while the rest of the independent actions are kept as they are: independent. So composing action systems can be represented as follows:

- $P = (A, v, P_i, P_A, P_H)$ and $Q = (B, w, Q_i, Q_B, Q_G)$
- $P \parallel Q \mathrel{\widehat{=}} (A \cup B, (v, w), P_i \parallel Q_i, par(P_A, Q_B), P_H \cup Q_G)$

The alphabet of the $P \parallel Q$ is the disjunction of both alphabets, the variables are merged and the initialisation actions of both action systems are executed in parallel. The internal actions of $P \parallel Q$ are the disjunction of both internal actions. $par(P_A, Q_B)$ represents the actions of the composed system, which contains the independent actions of P, the independent actions of Q and the actions that are common to both systems and that when synchronised, make the interaction between systems:

- $par(P_A, Q_B) \mathrel{\widehat{=}} P_{A-B} \cup Q_{B-A} \cup \{P_c \parallel Q_c \mid c \in A \cap B\}$

Like alluded before,this view of action system is based on CSP. As a consequence, the parallel operator for action system enjoys the same properties as the CSP parallel operator w.r.t. refinement like commutativity, associativity and monotonicity [62], which is equivalent to our approach on Event-B. Because of the first two properties, we can write the parallel composition of a finite collection of action systems P_i as $(\parallel i \cdot P_i)$, where$(\parallel i \cdot P_i)$ can be calculated by successive application of the binary parallel operator. As consequence, we can deal with multi-interaction between actions that share the same action label [68]. The monotonicity property allows the further and independent refinement or decomposition of any parallel component of a distributed system [62].

Like Event-B, a similar definition of parallel composition with value passing is applied to action systems. This include parallel composition of actions with output/input behaviour as well as input/input.

3.6 Z notation

Z notation 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 variable values. Abstract Data Type (ADT) is a model that is characterised by its operations. It can be useful to describe object-oriented programs since the state variables and operations can be compared to instance variables and method, respectively [72]. Z served as basis for other notations (classical B) and several variants adapted for object-oriented programming. Z is a strictly specification notation, while B for instance has imperative programming constructs as part of the notation.

Z includes two notations [72]:

- Notation for ordinary discrete mathematics
- 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*. Using *schema calculus*, it is possible to build big schemas from small ones.

3.6.1 Schemas in Z

Schema is a naming construct that represents some specification. Defines the requirements through the use of mathematic entities such as sets, relations/functions or sequences. Their primary use is specify state spaces and operations for the mathematical modelling of systems [73].

One of the ways to represent a schema *StateSpace* is represented here (the shortest one) [73]:

- $StateSpace \mathrel{\widehat=} [x_1 : S_1; ...; x_n : S_n \mid Inv(x_1, ..., x_n)]$

$x_1...x_n$ are state variables, $S_1...S_n$ are expression that represent the variables types. $Inv(x_1, ..., x_n)$ is the state invariant. Schema are used to define the static and dynamic feature of a system. The static part includes the possible states and the rules that should be preserved during the system execution (invariant clauses). The dynamic part consist on the available operations and the change on the state after the execution of the operation, as well as the relationship between input and output [74].

3.6.1.1 Combining schemas

It is possible to combine schemas if they are considered type compatible. Signatures are considered type compatible if the set of variables that is common has the same type. If this property is respected, than a larger signature containing all the variables from all the components can be enabled.

The combination of schemas can be achieved using the schema calculus operator, though the use of logical connectives between schemas. A schema can be included in another schema by placing the name of the included schema in the signature of the including schema, resulting in the combination of signatures and predicates. [14] explain in more detail the combination of schemas. Two schemas S and T that are type compatible w.r.t. signatures can be combined to give a new schema $S \wedge T$. So $S \wedge T$ joins the signatures of S and T, and its property is in fact the conjunction of the properties of S and T. If:

- $S \mathrel{\widehat=} y : \mathbb{Z}; z : 1..10 \mid y = z * z$
- $T \mathrel{\widehat=} x, y : \mathbb{Z} \mid x < y$

- then $S \wedge T \mathrel{\widehat=} x, y : \mathbb{Z}; z : 1..10 \mid x < y \wedge y = z * z$

where all the expressions before (left hand side) '|' are the signatures and all the expression after (right hand side) are the predicates of the schema.

3.6.2 Composition 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 also permits the stuttering refinement of specifications, composition can be applied at same time the refinement is applied to the system, becoming closer to the implementation.

Using the operations of the Z schema calculus, it is possible to combine two descriptions into a stronger specification. [14] describes how this combination of schemas can be achieved, assuming that overloading - possibility that two distinct variables in the same scope might have identical names - is forbidden. This is similar to our approach, where it is not allowed to have variable sharing and thus no variables with same name - although since Z does not have a notion of machine, there are some differences.

The piping operator ($\gg$) is used to describe operations that have almost independent effect on two disjoint sets of state variables. If we consider again the schemas $Op1$ and $Op2$ and compose them using the piping operator: $Op1 \gg Op2$, the outputs parameters of $Op1$ are matched with the inputs of $Op2$ and hidden, while the other components are merged as they would be in $Op1 \wedge Op2$.

As an example, if we have schema $Op1$ that inputs a number $x?$, returning the square of that number $y!$:

- $Opt1 == x?, y! : \mathbb{N} \mid y! = x? * x?$

, a schema named $Counter$ that defines same variables and rules:

- $Counter == value, limit : \mathbb{N} \mid value \leq limit$

and $Op2$ that includes the schema $Counter$ and also inputs a number $y?$ and returns the sum between $value$ and the input number $new\_value!$:

- $Opt2 == Counter; Counter'; y? : \mathbb{N}; new\_value! : \mathbb{N} \mid value' = value + y? \wedge limit' = limit \wedge new\_value! = value'$

we can say that $AddSquare \mathrel{\widehat=} Op1 \gg Op2$ is equivalent to :

- $\Delta Counter; x? : \mathbb{N}; new\_value! : \mathbb{N} \mid value' = value + x? * x? \land limit' = limit \land new\_value! = value'$

The *pipping* operator describes an operation that is closest to our approach, where output parameters from the different sources are merged (as long as they have the same type) and the rest of the properties are combined. Although Z does not have a notion of machines, the combination of schemas through *piping* is similar to the parallel composition. In case of variables with the same name, there is a *Renaming* operation which allows the renaming of variables to different ones, and it is another way of seeing similarities between approaches.

Another approach for the composition is through the use of view [75; 76]. 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 that are tackled by other views should be omitted. But there will always exist some redundancies which are needed to represent a particular part of the entire system. An advantage is that views can be constructed and analysed independent of the other views. The interaction between views uses the schema calculus and standard logic operators. Views can be connected by an invariant relating their state (more like a state-based approach), or connected by synchronising their operations (like an event-based approach) or even a mix of both. [76] discusses with more details, reasons, advantages, disadvantages and some hints for a good view structuring using Z. [75] discusses a similar approach using views, but the composition is through coupling schemas. 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 view unification.

Z is not ideal for dealing with concurrency, although can be used if a system is modelled as a sequence of operations on an abstracted state. Despite that, some research has been undertaken to adjust Z to model concurrent systems [73]. Examples are TLZ : Temporal Logic of Actions (TLA) and Z by Lamport [77], Coombes and McDermid in [78] or Schuman et al in [79] between others.

3.7 VDM Notation

VDM (Vienna Development Method) is a model-oriented notation that was developed while a research group of IBM laboratory in Vienna was working on compiler development and language design. It consists of a formal modelling language VDM-SL, combination of data definitions, state variables, a set of operations that can describe the specification of a system and an invariant on the state variables, that must be verified before and after the execution of any operation [80]. VDM has 3 valued logic,

instead of only two (true or false), which allows treatment of undefinedness not explicitly treated in Z, B or Action System. The VDM syntax can be described using ASCII or mathematic notation. Nowadays, there is an extension of VDM, VDM^{++} which supports object-oriented design, concurrency and is capable to model real-time distributed systems [81].

A VDM development is made up of state descriptions at successive levels of abstraction and of implementation steps which link the state description. The implementation of an abstract state description S_a by means of a more concrete one S_c describes [80]:

- 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 implements the ones of S_a.

While modelling a specification using VDM, in particular for the operations,predicates pre and post condition are written explicitly. So 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 [81]. VDM objects must be validated by the verification of proof obligations [80] and for an operation to be valid, the satisfiability must be met [82].

Formal development by VDM uses data reitification from abstract to concrete model but also suitable operation decomposition. In general operation decomposition it is applied after the data reitification [82].

3.7.1 Parallel Composition (Interference) - Rely/Guarantee Conditions

There are some approaches for the development of composition using VDM. One of the famous approaches is based on rely/guarantee conditions where two states predicates are added to the pre and post conditions on a specification, allowing *interference* between systems, with variable sharing. This extension of VDM, developed by Jones [53], permits the specification and development of concurrent, shared-variable systems [83].

So, the interference problem makes the development process of the *ex post facto* proofs difficult. Trying to solve this problem, an specification can then be described as:

- (P, R, G, Q)

where P correspond to the pre-condition and is a condition describing a set of states, while R, G, Q are rely-condition, guarantee-condition and post-condition respectively and are conditions on state-transitions (predicates of two states: before and after state).

A definition for rely/guarantee condition is described in [84]:

- Rely-Condition: defines assumptions that can be made in program developments. Although the global state may alter, the changes will be constrained. Any state change made by other processes can be assumed to satisfy the rely condition. Must be reflexive and transitive. The default rely condition is that the state does not change: $rely - OP(\sigma, \sigma') \mathrel{\widehat{=}} \sigma' = \sigma$.
- Guarantee-Condition: any process must make its state changes in such a way that any other process observing the global variables will be only see (time-ordered) pair of states that satisfy the guarantee-condition. Must be reflexive and transitive. The default guarantee-condition is that there is no restrictions: $guar - OP(\sigma, \sigma') \mathrel{\widehat{=}}$ TRUE.

The guarantee of condition of parallel processes should implies the guarantee condition of the overall operation. Furthermore, each guarantee-condition should be at least as strong as the rely-condition of the other [84]:

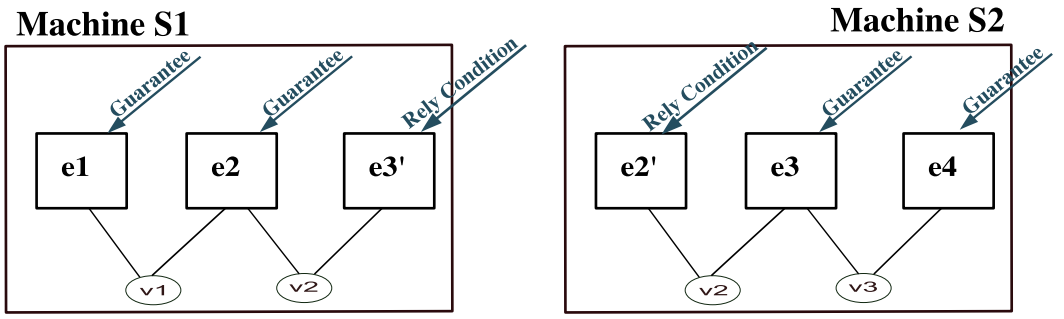
- for $i \neq j$, $guard - T_i(\sigma, \sigma') \Rightarrow rely - T_j(\sigma, \sigma')$

The disadvantage of this approach is that the specification of interference must be checked against every state update, even if it is "obvious" that the update cannot interfere with anything else [85]. So recent work which combines ideas in concurrent separation logic with the rely/guarantee formalism has been undertaken, as can be seen on [85]. Lu in [82], introduces a different kind of (de)composition, called data decomposition. This approach does not necessarily have to be applied after data reification like operation decomposition. It allows the splitting of the model, which is not possible with the classical approach, in sub-specifications without knowledge of their internal implementation and each sub-specification can be developed independently. This formal development method is called DD-VDM and unifies flexibly data reification, data decomposition and operation decomposition in a uniform framework.

While studying the several approaches for the composition/decomposition of systems, we realised that there is a strong similarity between the rely/guarantee approach proposed by Jones [83] and the decomposition using shared variable proposed by Abrial [5]. The shared variable approach splits the system in separated components: usually the events are allocated to each of the components, so they can be refined independently later. A problem may occur when a variable is used in two different events that belong to different components after decomposition (that variable has a special status defined as *external*). If the intention is to apply a further independent refinement to the components, *variable sharing* can be used to deal with this problem: by introducing *external events* that simulate the way the external variables are handled in the machine before the

decomposition. Those events cannot be refined in their components, since they just simulate the existence on that (internal) event on the other component. If the event $e2$ and event $e3$ share a variable $v2$, then one of the components after decomposition ($S1$) will have an external variable $v2$, an internal event $e2$ and an external event $e3'$. On the other component $S2$, it will exist as well the external variable $v2$, but an internal event $e3$ and an external event $e2'$. Depending on the point of view, let's say from $S1$, $e3'$ will correspond to the rely condition while $e2$ will be the guarantee condition. If we rely on $e3'$, then we guarantee $e2$. So we think it is possible to make a correlation between this two approaches, and develop a further study on using the developed worked on rely/guarantee for VDM and apply it on shared variable decomposition for Event-B.

Since the shared variable approach requires further study, it would be interesting combine this two theories and unfold the result of such combination.



(a) Machine S1 that result from decomposition of machine S

(b) Machine S2 that result from decomposition of machine S

FIGURE 3.3: Shared Variable Decomposition Result

3.8 A Comparison

All the formal methods described include interaction between systems and a notion of creating a larger system. Comparing Event-B and classical B, we can say that on Event-B there is no equivalence to classical B **Includes**, but we extend the syntax and add that same keyword, although with some differences:

- The composition it is not over another machine. In theory it is generated another machine with the properties of all the included machines.
- It is possible to access to the variables of the included machine, since the invariant it is a conjunction of all the invariants. Events available depend on the events to be composed, and may include singular events.
- There is no notion of transitivity.

- On classical B, *includes* means extending a machine, while on Event-B, there is a semantic behind the composition, involving interaction and independent refinement of each of the sub-systems to be composed.

On Event-B, there is a **Sees** as well, but only concerns to the static part of the model like carrier sets, constants, axioms and theorems. Also allow the extension to other contexts. This is different from the **Sees** on classical B, which does not separate the static and the dynamic part of the model, and **Sees** can be applied to complement machines.

Comparing the other formal notations, we can say that Z notation does not have the notion of machines but the schemas are used to make the composition. CSP, being a process algebra formal method, describes the evolution of a system through a sequence of processes - event-based view , while classical B, Action System, Z, VDM and even Event-B are more state-based view and the evolution of the system is seen based on the change of state (change on the system's variables). Our approach has an event-based behaviour, because the composition is done through the composition of events and similar approaches happen in classical B, Action System, CSP and even Z. The composition in VDM uses variable sharing and because of that it is necessary to restrict the behaviour of the environment and the operation itself in order to consider the composition valid. One the other hand, VDM has already incorporated some processes for decomposition. So several features of our composition is based on the described formal methods and adjusted to Event-B, with some new ideas like compose events that not necessarily have the same name (on the others notations, this is essential to define which operations to merge). Event-B does not have a distinction between input/output parameters but Z,classical B, Action Systems and CSP use a similar representation: '!' for input and '?' for output. VDM also distinguish between variables types, using "rd" and "wr" for reading and writing behaviour respectively. We intend to further our study on that matter since this distinction seems related to the composition of events and the necessary proof obligations to generate. Event-B takes advantage of the tool support for modelling and reasoning about a system, something that the other notations do not share at the same level at least. It is also suitable for development of system that are constantly changing state (reactive systems), becoming easier to model parallel and concurrent systems.

Chapter 4

Work Description

This chapter describes the work that has been developed since the beginning of the PhD programme. We start describing our aim in a conceptual way and the justification for the chosen approach. The next section discusses how the work was implemented in detail.

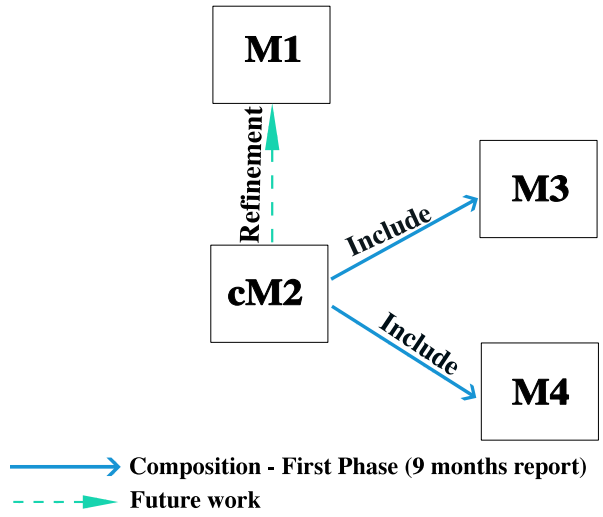
4.1 Methodology

The goal of this work is to understand and implement the `decomposition` technique while modelling a system. The modelling is done using formal methods, in particular, using Event-B and it will be used a platform, RODIN, as a base to test the decomposition implementation. While modelling a system, in our best understanding, the goal is to improve the understanding of a system, and implement verification by detecting eventual flaws and bad design implementations. Proofs generated by the model help on this understanding, but should not be the prime objective while modelling - prime goal is the better understanding of the system.

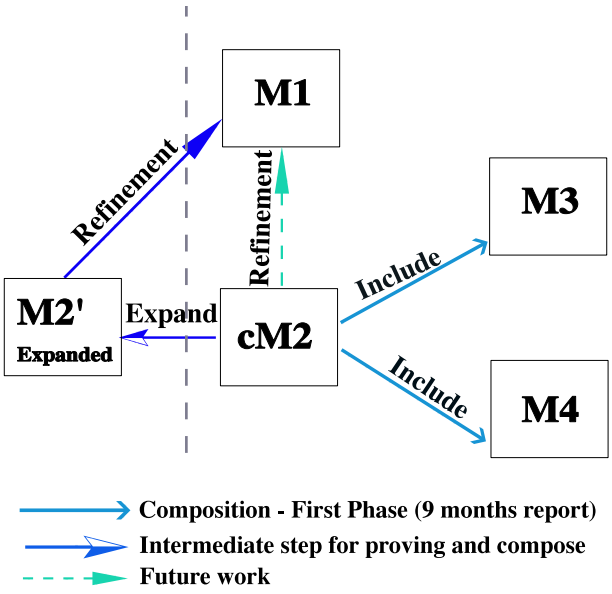
The approach chosen starts with the study and development of another technique, `composition`, which can be seen as the reverse process of `decomposition`. Based on this inverse relation, figure 4.1(a) shows what we intend to achieve now (composition) and on figure 4.1(b) what we achieved so far. Machines $M3$ and $M4$ are included in the composition file $cM2$. The current approach creates a temporary machine $M2'$ that is an expansion of all the properties of $cM2$. If composition is applied while refining, it is necessary to prove that $M2'$ is a refinement of $M1$, using the refinement technique that already exist on Rodin. In the future, this intermediate step will eventually disappear and the refinement shall be proved straight from $cM2$ and $M1$.

After the composition is achieved, we intend to develop the *Decomposition*. Seeing again the figure 4.2, the idea is to have $M1$ and to be able to decompose into $M3$ and $M4$.

This operation is valid, if we can prove that the result of the decomposition process ($M3$ and $M4$), if composed again, have a refinement relationship w.r.t. $M1$.



(a) Composition Structure



(b) Composition Structure with intermediary step (Expand)

FIGURE 4.1: Composition Structure

There have been some studies on the composition/decomposition using other formal methods, like described on chapter 3 . Based on that work, it was decided to develop this techniques on the Rodin platform. Conceptuality speaking `composition` is the first step on the process: one must add more detail and more complexity in a system in order to be able to decompose it.

So, the aim of this report is to show the understanding of what formal methods are, how to integrate them on the system's development (modelling) and the benefits of using it. On a deeper look, it will be shown the development of the *Composition* technique and

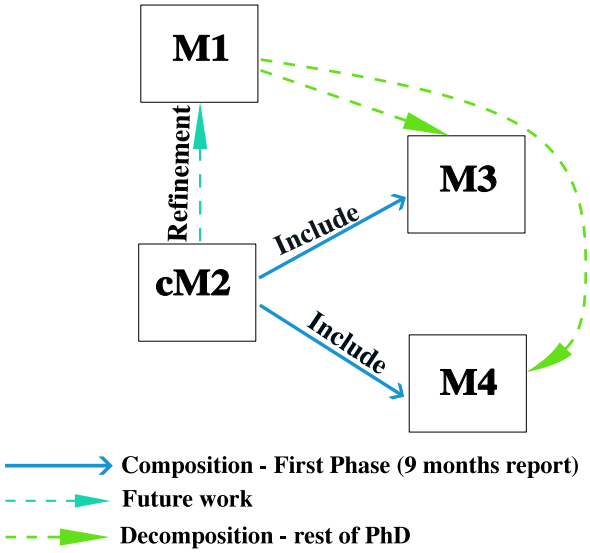


FIGURE 4.2: Composition and Decomposition Structure

using the refinement technique on the RODIN platform, generate an expanded model (containing the combination of two or more sub-systems). The expanded model will be used to validate the refinement process of an abstract system. A case study backs up this study and understand better which proofs/validations must be done to show that a composed machine is a refinement of other machine. After composition proofs are analysed and disposed, they will be used on the inverse process, `decomposition`, which besides this proves, will probably reveal more necessary ones. From the refinement process [5; 86] it is possible to infer the complexity of proofs to be generated on the decomposition.

4.2 Overview of the Railway System case study

The Railway System that is used in this case study describes a formal approach for the development of embedded controllers for a railway. [7] makes a description of such system, but using classical B formal method. We base our case study in that work, converting the B notation into Event-B and making the necessary adjustments.

The model starts with an abstract view of the railway system. It is modelled the connection between sections (constant *net*), the connection section for each switch (variable *next*), all the trains (variable *trns*), the sections that are occupied by trains (variable *conn*), the initial and the final section occupied by each train (variables *occpA* and *occpZ* respectively). The speed and when braking, for each train is modelled through the variables *speed* and *braking*. The invariant introduces properties of the system such as all the section occupied by the trains have to be connected to each other, if the train occupies for than one section then *occpA* and *occpZ* must be different sections, *next* is

a subset of sections (*net*), between others. At the events level, the operations specified are entry and leaving section by the trains (*enterCDV* and *LeaveCDV*), the change of speed by a train (*ChangeSpeed* - if braking the new speed must be less than the current speed), the braking of a specified train (*Brake*), the change of switch positions (*SwitchChangeDiv* and *SwitchChangeDiv* - no train can occupy the switch while is been changed). The events *SendTrainMsg* and *RecvTrainMsg* are introduced in this model although not implemented (implement `skip`). Those two events are implemented on the first refinement step, through the inclusion of a communication layer. The messages are very simple at this level, represented by a function that maps trains to a set of boolean variable (`tmsgs` $\in$ `trns` $\rightarrow$ $\mathbb{P}$`(BOOL)`) and other one that confirms if there is any message to be read (`permit` $\in$ `trns` $\rightarrow$ `BOOL`). The communication layer affects the event *Brake*, since it makes an emergency break if receives a message saying that the next section is already occupied (`permit(t1) = FALSE`). In the second refinement, we compose sub-systems using our plug-in. The composition introduces the concept that some events, from different machines, happen in parallel and because of that, they are "merged". Because Rodin platform does not support this kind of parallel composition, we introduce this plug-in which allows the composition of events in a parallel fashion.

So the railway system can be decomposed in several components such as Tracks, Trains and Communication module. All this components, interacting between each other, as we can see in the figure 4.3.

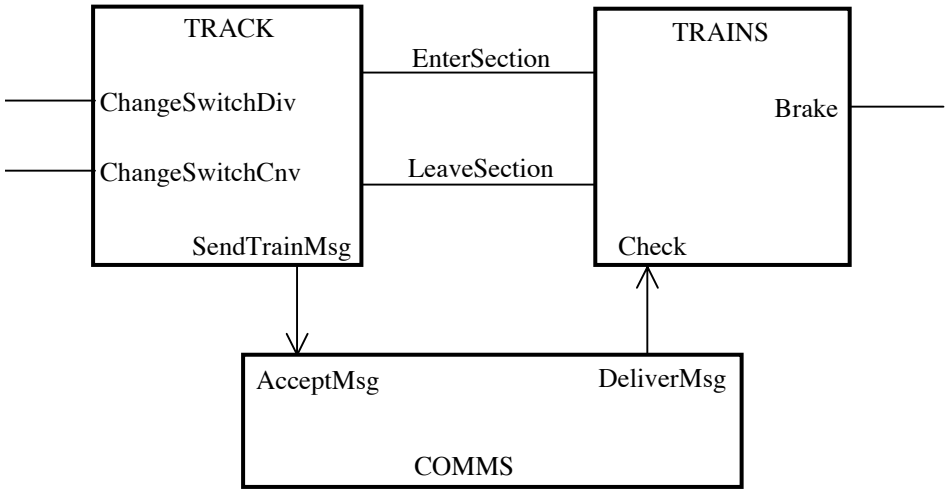


FIGURE 4.3: Components of Railway System

The *Track* component is responsible for defining which sections are occupied, which are free and when to change the switch positions (switchs are special sections that can change the position of the tracks; can be divergent, with one incoming section and two outgoing sections or convergent, having two incoming sections and one ongoing section). For more details, see Appendix A.3.

The *Train* component represents each train, controlling the speed, when to brake, the entering and leaving track sections and based on the received messages from the communication component, react and produce an action. For more details, see Appendix A.4.

The *Comms* component represents the communication layer which interconnects *Track* and *Trains*. Whenever a train enters/leaves a section, *Track* sends a message to *Trains* through *Comms*. So the events of entering and leaving sections from Track and Train are interconnected and it is possible to represent that as a single event using the composition. For more details, see Appendix A.5.

The detailed specification and all the refinements, as well as the sub-systems can be seen on Appendix A.

4.2.1 Specification of the composition using the Railway System

We describe here the specification of the case study, using Event-B notation plus the extension described on the section 3.2 . The composed machine is the second refinement of the Railway system which refines *Railway_M1* as seen on figure 4.4 . Includes the machines *Track_M0,Trains_M0* and *Comms_M0*. The Invariant section contains some properties originated by the composition of different machines. All the composed events refine an event that already exist on the first refinement (*Railway_M1*). Note that the *INITIALISATION* event is the parallel composition of each included machine.

COMPOSITION MACHINE RailWayComposition.bcp

REFINES RailWay_M1

INCLUDES

$Trains\_M0$

$Track\_M0$

$Comms\_M0$

INVARIANTS

$inv1 : occpZTrain = occpZtmsgs$

$inv2 : occpATrain = occpA$

$inv4 : trns = trnsTrain$

$inv5 : mm\_1 \in TRAIN \rightarrow BOOL$

COMPOSES EVENTS

Initialisation

Combines Events

Trains_M0.INITIALISATION ‖ Track_M0.INITIALISATION ‖ Comms_M0.INITIALISATION

Combined Event SendTrainMsg $\widehat{=}$

Refines SendTrainMsg

Combines Events

Track_M0.SendTrainMsg ‖ Comms_M0.Send

Combined Event RecvTrainMsg $\widehat{=}$

Refines RecvTrainMsg

Combines Events

Train_M0.RecvTrainMsg ‖ Comms_M0.Recv

Combined Event ChangeSpeed $\widehat{=}$

Refines ChangeSpeed

Combines Events

Train_M0. ChangeSpeed

Combined Event Brake $\widehat{=}$

Refines Brake

Combines Events

Train_M0. Brake

Combined Event EnterCDV $\widehat{=}$

Refines enterCDV

Combines Events

Track_M0. EnterCDV ‖ Trains_M0.EnterCDV

Combined Event LeaveCDV $\widehat{=}$

Refines LeaveCDV

Combines Events

Track_M0. LeaveCDV ‖ Trains_M0. LeaveCDV

Combined Event SwitchChangeDiv1 $\widehat{=}$

Refines SwitchChangeDiv1

Combines Events

Track_M0. SwitchChangeDiv1

Combined Event SwitchChangeCnv1 $\widehat{=}$

Refines SwitchChangeCnv1

Combines Events

Track_M0. SwitchChangeCnv1

END

FIGURE 4.4: Composition file for the Railway System - Second Refinement

Some events are the result of parallel composition (SendTrainMsg,RecvTrainMsg,EnterCDV and LeaveCDV) and others are just simple events (ChangeSpeed,Brake,SwitchChangeDiv1,SwitchChang Although the names of the events match between abstract and composition machine, that is not a restriction: the events can have any name as long it does not clash with other that already exist.

4.3 Composition Plug-in: Requirements, Purpose and Developed Work

In this section is described the semantics, reasons and proposed outcome for the composition plug-in. The overall purpose is:

- Enable the composition of models, allowing the interaction between systems, highlighting the power of reusability and development of large systems based on subsystems, using Event-B.
- Composition is achieved through the "fusion" of events whose source are different models. In other words, it is used an event-based view where models can be combined through synchronised events. It is possible to pass values from one system to other while synchronized events are composed.
- When composing, the properties of the system are merged. Meaning that variables, invariants and contexts properties are combined (for that reason, it is not possible to exist variables nor context properties with the same name). The result is the creation of a new system that is a combination of different models, which can be an abstract model itself or a refinement of an existing abstract model.

- The proof obligation to be generated should be as effortless as possible for the user. This includes minimization of proofs generated (try to reuse proofs from the original models) as well as maximization of proofs automatically discharged.
- Possibility of adding "gluing invariant" clauses when composing in order to define properties between included systems.
- The events that are part of the composition model are chosen by the user and do not depend on the name of event itself (different from Action systems or CSP. Our approach is more flexible since allows composition without name restrictions.
- The properties of the *Composition* are hosted in a composition file, with *bcp* extension.
- The included machines can be refined independently after the conclusion of the composition (monotonicity).

So to achieve this previous goals, the plug-in has the following sections:

- **Refines** section
- **Includes** section
- **See** section
- **Invariant** section
- **Variant** section
- **Composes Events** section
  - **Kind of Event** section
  - **Refines** section
  - **Combines Events** section

4.3.1 Refines section

A machine can be chosen to be refined in the composed model. It can be used in two ways:

- As a new abstract machine, which includes (at least) two machines.
- As a refinement, which includes (at least) two machines. In this case, the proofs generated must include refinement proofs.

4.3.1.1 Constrains/Properties

- It is only possible to refine one abstract machine each time while composing - Refinement rule

Using the case study, the figure 4.5 represent that the composed machine is a refinement of the machine RailWay_M1.

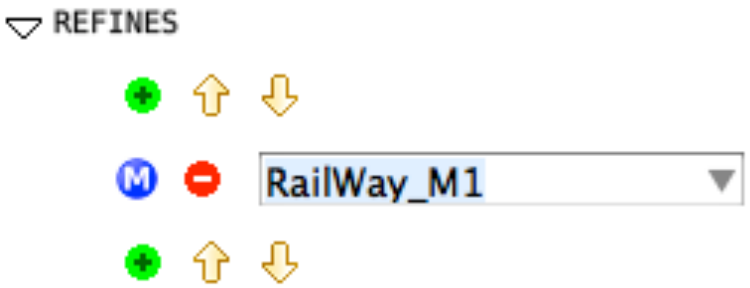


FIGURE 4.5: Refines section on the Composed Machine Railway on the second refinement

4.3.1.2 Developed work

The list of machines available must belong to the same project. Changing this clause, has effects on the events to be refined (on the section 4.3.6.2). Optional element for our composition plug-in.

4.3.2 Includes section

To compose a model, it is necessary to define which sub-system interact. This section allows the selection of the models to be composed.

4.3.2.1 Constrains/Properties

- Composition is done with at least two sub-systems.
- It is not possible to compose machines that have a relation between them, i.e, that have any kind of abstract/refinement relation.
- It is only possible to include machines that are abstract (without any refinement): simplification (may changed in the future).
- Possibility to include the sub-systems invariant clauses to the composed system.

The inclusion of the machines *Trains_M0,Track_M0, Comms_M0* can be seen on the figure 4.6. Note that it was chosen not to include the invariant of the last two machines (since they exist already on the abstract machine).

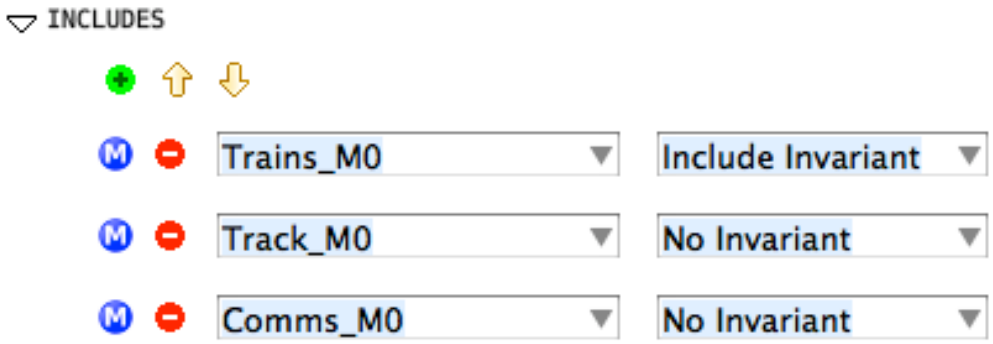


FIGURE 4.6: Includes section on the Composed Machine Railway on the second refinement

4.3.2.2 Developed work

The list of machines available belongs to the same project. It is not possible to add twice the same machine. There is no limit on number of included machines per composition file, but at least two machines must be included. Changes on the list of included machines are reflected on the events to be composed (`Composes Events`), on the contexts seen (`See`), on the `Invariant` and possibly on the `Variant` (if used). Compulsory element for our composition plug-in.

4.3.3 See section

This section is used to enable the inclusion of a context file into the composed model. Contexts referred by any of the included machines are also referred on the composed file. So all the context properties are visible to all the elements of the composed file.

4.3.3.1 Constrains/Properties

- There is no limitation to the number of contexts a composed file can see. To avoid redundancy, a context referred by an included machine is only shown once.

4.3.3.2 Developed Work

The list of contexts available are the ones existing on the same project. Only validated contexts can be seen. Optional element for our composition plug-in.

4.3.4 Invariant section

This section allows the inclusion of invariant clauses to the composed file. The inclusion of more invariants (from the included machines) depends on the user's choice on `Includes`. The defined invariants on the composed machine are "joint" properties between the included machines (gluing invariants), so variables and contexts from all the included machines become part of the composed machine scope. Note that the inclusion of this clauses does not change the monotonicity of the system nor the sub-systems, since parallel composition is monotonic [33].

4.3.4.1 Constrains/Properties

- Composed machine invariant is a conjunction of all the included invariants.

The invariant section on the composed machine can be seen on figure 4.7. Note that the first three clauses result from the renaming of some variables on the Train machine, in order to be composed.

FIGURE 4.7: Invariant section on the Composed Machine Railway on the second refinement

4.3.4.2 Developed Work

The invariant on the composed file is the conjunction of all the invariants that are included by the machines plus the clauses added to the composed file itself. The invariant of the composed file must be preserved by all the events like in an ordinary machine. Optional element for our composition plug-in.

4.3.5 Variant section

Since the composed machine can be a refinement of an abstract model, there is the possibility of introducing new events. In order to avoid divergence, there variants are necessary for the new events.

4.3.5.1 Constrains/Properties

- Just like an ordinary variant in a machine.

4.3.5.2 Developed Work

Optional element for our composition plug-in.

4.3.6 Composes Events section

The interaction between systems only happens when the composed events are synchronised and ready to be executed. The systems can interact through shared parameters.

4.3.6.1 Constrains/Properties

- The event have a name that is different from any other event on the composed machine and must have a defined type from the list: ordinary, convergent, anticipated.
- *INITIALISATION* event initialises the composed machine with the respective initialisation state of the included machines. The actions of this event must preserve the invariant of the composed model.
- If refining, the events of the abstract model must be refined.
- It is not possible to interact between events of the same system.But when composing events, the original events can have the same name.
- A composed event has at least one singular event. By composing an event, the guard of that event is the conjunction of the guards from the original events. Because the abstract events are not changed, there are never witnesses. The actions of the composed events is the parallel execution of all the actions from the original events.
- The parameters of a combined event is a list defined by all the parameters of each individual event that constitute the combined event. If on the list of parameters,

there is some that share the same name, those parameters will be merged into only one. Another necessary validation is to check if the type of the parameters with the same name match or have same sub-type. Meaning that a parameter p? with type T1 and the p! with type T2 can be merged if:

1. The types must match or at least be from the same sub-type. In other words, $T1 \cap T2 \neq \varnothing$.
2. The guard of the input event will accept the output value coming from the output event.

The composition of two events on the composed machine can be seen on the figure 4.8.

4.3.6.2 Developed Work

It must be possible to prove that each event of the composition file preserves the invariant. To compose a new event, it is necessary to specify a name for the event and define if the event is a refinement or not. If refining, a list of events from the abstract machine are available to be selected. To conclude the composition, it is necessary to select which events to compose: only one event per machine. Must exist at least one event to be composed.

Although not in a direct way, there are some other properties of the machines that are affected by the composition (`Variables` and `Theorems`). This properties are discussed on the next sections.

4.3.6.3 Variables section

The list of variables of the composed machine includes all the variables existing on the included machines. In case of name clash, it is necessary to rename one of the variables.

4.3.6.4 Theorems section

The theorems of the composition file is the conjunction of all the theorems belonging to the included machines.

4.4 Constraints/Properties of the plug-in

This section describes the constrains and properties of the plug-in. Some results as a for simplification step, other will have future developments, since this is a work in progress.

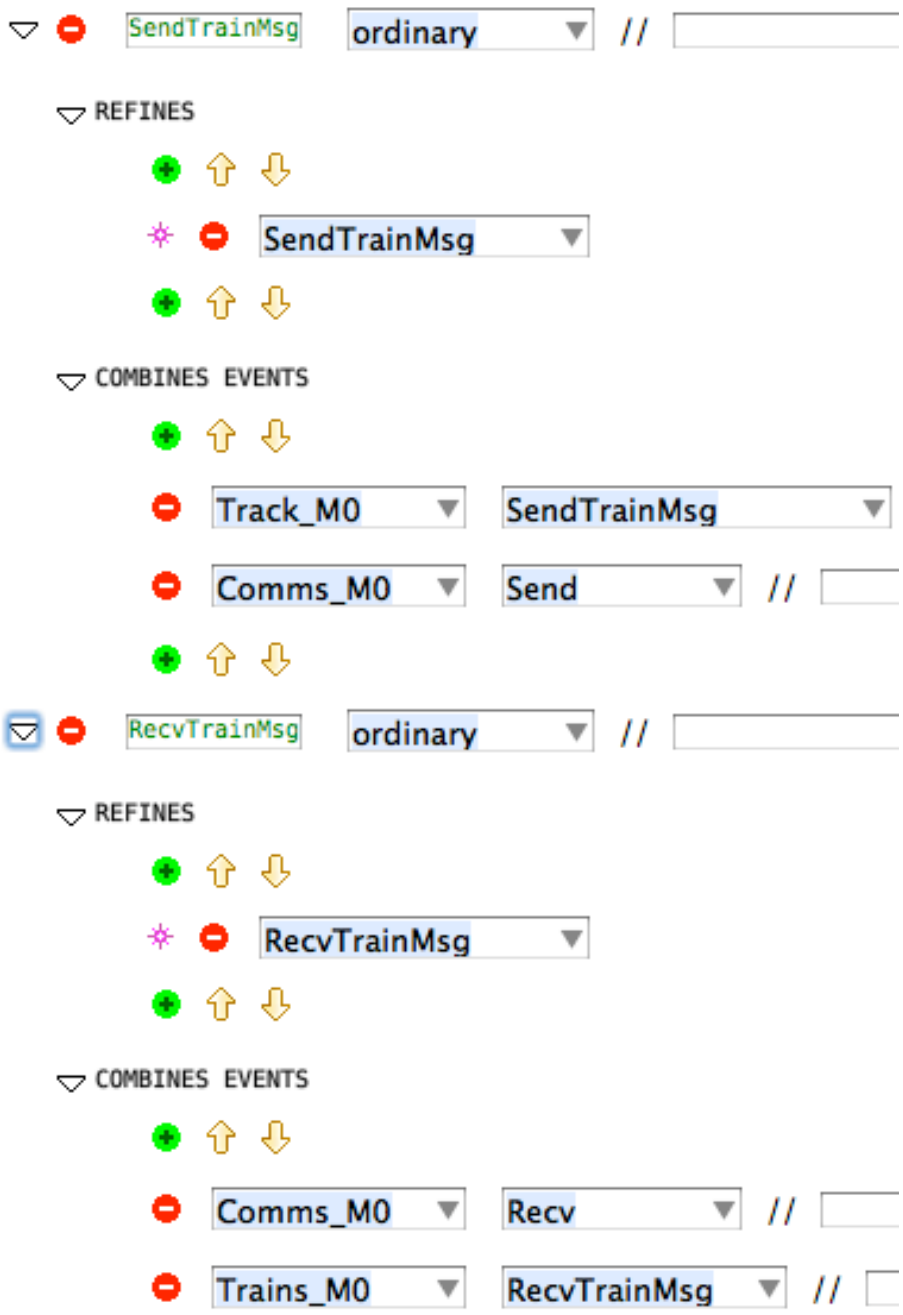


FIGURE 4.8: Composes Event section on the Composed Machine Railway on the second refinement

- Rodin Platform does not distinguish explicitly between kind of parameters nor internal or external events. In order to assure the preservation of liveness, this distinction is necessary. For the time being, and since this is a work in progress, the parameters with the same name are merged (still must fulfill the two conditions referred in 4.3.6.1).

- Since we adopt an event-based view of the composition, there is no variable sharing.

In case of name clash, it is necessary to change manually in the original models the name of the element that is clashing. In the future, there should exist a "refactoring" option, allowing variable renaming.

- Parallel composition preserves monotonicity when referring to the model's behaviour [33]. The events on the composition model, just like the original ones, are atomic.

- For now, there is no validation on the composed model that is generated. There is already a requirement document with the validations to be applied. The intention is to create a model (using Event-B) to make the validation. The implementation would extend Rodin's Static Checker and Proof Generator.

- For the purpose of study and reuse of the functionalities that already exist on machines on Rodin, after the conclusion of the composed file, it is generated a new machine. So the validation is applied to the new machine (although there are some composition validations not implemented). It is intended to use this machine file as an auxiliary one, helping the understanding of which validations and conditions to be applied for a composed machine. The goal in the future is not to generate this temporary machine file but instead make the validation on the bcp file instead.

- For simplification purposes, it was not added a "Theorems" section on the composition file. If, in the future, based on more case studies, the composition model reveals a necessity to include theorems, this will be done. For now, the theorems existing on the original machines, are conjoined in the new composed machine file.

4.5 Discussion

In theory, the parallel composition is very well known and exist in different other notations. Based on those, it was possible to create this plug-in. But when implementing the theory, usually there are limitations or performance issues that are not seen and have to be solved. In our case, while applying the railway case study, we discovered that a excessive high number of proofs were generated. Those proofs had different sources like well-definedness (WD), invariant strengthening (INV), guard checks (GRD) and action simulation (SIM - only when the composition involved a refinement): more details about the meaning of each of this proofs can be found in Rodin's User Manual[36]. Some of the proofs will have to be generated anyway (part of the validation of a machine), but others do not, since they are already discharged on the original machines. For instance, the INV proofs that already exist on the abstract machine do not need to be re-generated since they were already discharged (that is the reason why the user can decide to add or not the invariant for each included machine). But other proofs are also re-generated. Some of the events when composed, do not change nothing from the previous refinement.

In that case, instead of re-generating proofs, it should be possible to reuse proofs from the other models and decrease the number of generated proofs. This is an issue that we intend to tackle in the future. For a small example, the number of proofs to be generated is not that important, but if we intend to apply on a large scale project, involving big sub-systems, the number of proofs to be generated and discharged can interfere on the performance or even when to decide if it worth to use parallel composition or not. So we can use the number of generated proofs as a way to measure how accurate the composition plug-in is.

On the decomposition level, we intend to split the system into sub-systems. This splitting might not be straightforward since the machine may not be ready for the decomposition. In other words, because we intend to make an event-based decomposition by splitting an event without variable sharing, it may be necessary to rearrange the model in a way that facilitate the decomposition. In an abstract model of a system, the idea is not to have a very detailed view of the system, but instead a simple model of the system's properties. Because of that, it may not be possible to apply the decomposition directly because the model is not *mature* enough. It may be necessary an intermediary step, which could involve a refinement of the model in order to prepare it for the decomposition. The conditions to apply the decomposition straight or the need of a refinement step in the middle needs further study.

Tackling the decomposition itself: this will involve discover the conditions that permit a valid separation of systems, the possible consequences of this operation and the properties of each of the resulting components.

4.6 Conclusion

Although a work in progress, the composition plug-in already achieved some outcomes. The main achievement is the ability to choose events from different machines and combine them, resulting in a new machine where the properties are merged. This allows combine independent sub-systems. Also assures that those sub-systems when combined are a refinement of an abstract machine. Because the composition technique maintains the monotonicity of the included sub-systems, those sub-systems can be refined independently. Another major achievement is the possibility of applying the parallel composition with message value passing, where parameters can be passed from one event to another. Although still some study have to be done, it is already possible to achieve that goal. With this developed work (besides the rest that needs to be concluded for the composition technique), we expect to have the necessary conditions to develop the decomposition. There are some issues to be solve (as mention above), but this should be a good starting point to our next aim.

Chapter 5

Future Work - Work Plan

After concluding the 9 months report, it was possible to learn some important points and achieve the following:

- Development of a prototype plug-in for composition using the Event-B notation and the Rodin platform.
- Application of this plug-in in a case study which helps to understand more about the advantages, constraints, how to use and improve the prototype for the future.
- Understanding of the Rodin platform structure and how to add new features on the tool.
- Understanding of the Event-B notation, the construction of models, how to use them, advantages, disadvantages.
- Based on the study of other notations (B, CSP, Z), it was possible to implement this plug-in and make some comparisons.
- Although Composition/Decomposition are techniques well known in different areas, the implementation it is always complicated, it is difficult to define the boundaries between how automatic the tool can be and the user decisions on how to decompose a system. Also implementation constraints that are not foreseen in theory rise.

So the next goals are to use the composition plug-in in some other cases studies until it becomes more stable. Also it is intended to improve/add some features as part of improving the usability. After concluding this part, starts the development of the decomposition. There are two kind of decomposition that can be developed:

- Event based view, where the system is decomposed through events. Since we use a similar approach for the composition, this approach is considered so far the suitable for the decomposition development.
- State based view, where the system is decomposed through variables (variable sharing). This approach for the composition/decomposition development using a state-based view, can be seen on [29] and [5].

For now, it is decided to be applied the spitting of events (Butler style), but depending on the complexity of the study and development, it might be possible to to develop the other style.

The next stages (in terms of time) of the PhD can be defined as:

- Rest of PhD: roughly 2 years (26 months):
  - Mini-Thesis: 1 year (12 months)
  - PhD: roughly 1 year (14 months)

So the plan for the composition is:

- Refactoring system: in case of clash of elements, to be possible to rename one of the elements that clashes. Also allow the possibility to change the name only for the composition operation, while the original machine remains unchanged **[1 month]**.
- Creation of a validation model using Event-B for our composition plug-in. The validation should be done straight on the bcp file (composed file) **[2 months]**.
- Static Checker extension and reusing proofs: minimize number of generated proofs and thus minimizing the user's effort to discharge the proofs. Minimize the complexity related to the proofs and composition. Try to reuse as many proof as possible that are already discharged on the original machines and also on the abstract machine (in case of refinement) **[3 months]**.
- To have enough material and information that supports the submission of a paper about composition techniques and tools developed, as well as achievements **[1 month]**.
- Input/Output Parameters and Internal/External Parameters. Internal parameters cannot be seen by other machines, so if there is a name clash for those, it should be generated an error. Internal parameters don not have the notion of input/output parameters, unlike external ones. Issue that need to be studied since Rodin does not support this distinction and for Composition it is important to be aware of of kind of parameters while composing events. **[3 months]**.

- Safety and Liveness properties should be preserved while composing. The proof obligations generated for an input event can be different from a similar event but with an output behaviour. In particular, one of the liveness sub-property is important when discussing about input/output parameters: *enableness*. Depending on type of event, the proofs obligations generated to assure that the concrete models keep the abstract model behaviour, will change. [**2 months**].

- Deal with instance of machines (similar concept as oriented objects) and representation within the Event-B notation. Important for distributed systems that are physical separated but are similar systems [**2 months**].

For the decomposition, the plan is:

- Study of related work on decomposition and possible application on tools. Also begin to analyse in more detail what it is involved while applying decomposition: model preparation (possible intermediate step and conditions on when to apply this step), consequences at the components level (properties inherited), proof obligations that need to be generated to permit correctness of the sub-systems [**3 month**].

- Study more the decomposition process and features in order to decide how to implement it on the Rodin platform [**1 month**].

- Write Mini-Thesis based on the worked developed between the 9 months report and the next 12 months [**3 months**]. Before than eventually the publication of a paper.

- Build a model for the decomposition (using Event-B and the Rodin Platform), similar to the one done for the composition. It will involve the creation of a document with the requirements, the possible operations, the rules and the initial proves to be generated to allow the validation of this process [**2 months**].

- After the construction of the model (creation and specification validation), use it to implement the decomposition plug-in. Based on the development of the composition plug-in, this phase may take a long period, involving architecture decisions, programming, user interface decisions and the implementation/generation/learning of the proves related to this technique [**4 months**].

- While developing the decomposition plug-in prototype, it should be tested with some case studies (to be decided which case studies later, but for the time being, it should be used the railway system) [**1 month**].

- To have enough material and information to support a submission of a paper on decomposition techniques, tools and achievements [**1.5 months**].

- With the application of a case study, it is expected to get some conclusions as well as some improvements to be done so this technique can be used in a more abroad fashion [**1 month**].
- After the implementation of this possible conclusions/changes, the final test should be done using more different and complex case studies to make sure that the plug-in is robust, reliable and handy for the development, validation and above all,better understanding of systems' models/specification [**2.5 months**].
- Write PhD-Thesis, which would be a wrap-up of the all the developed work during the previous 30 months [**5 months**]. Before than eventually the publication of a paper.

Appendix A

Case Study : Railway System

A.1 Railway Abstract Specification

A.1.1 Railway Context - RailWay_C0

CONTEXT RailWay_C0

SETS

$TRAIN$

CDV Track Sections

CONSTANTS

aig_cdv Switches

net Total connectivity of sections */

div_aig_cdv divergent switches 1 – 2

cnv_aig_cdv convergent switches 2 – 1

new_cc1

new_cc2

AXIOMS

$axm2 : net \in CDV \leftrightarrow CDV$

net represents the connectivity between track sections

$axm3 : net \cap id(CDV) = \varnothing$

no cdv is connected to itself

$axm1 : aig\_cdv \subseteq CDV$

aig_cdv is a subset of CDV representing those cdv which are switches

$axm4 : div\_aig\_cdv \subseteq aig\_cdv$

$axm5:\ cnv\_aig\_cdv \subseteq aig\_cdv$

$axm6:\ div\_aig\_cdv \cap cnv\_aig\_cdv = \varnothing$

$axm10:\ finite(net)$

explicite declaration to simplify the proving

$axm11:\ finite(net^{-1})$

explicite declaration to simplify the proving

$axm9:\ (aig\_cdv \times aig\_cdv) \cap net = \varnothing$

switches are not directly connected

$axm7:\ \forall cc \cdot (cc \in (CDV \setminus aig\_cdv) \Rightarrow card(net[\{cc\}]) \leq 1 \wedge card(net^{-1}[\{cc\}]) \leq 1)$

non switch cdv has at most one successor and at most one predecessor

$axm8:\ \forall cc \cdot (cc \in aig\_cdv \Rightarrow ((card(net[\{cc\}]) \leq 2 \wedge card(net^{-1}[\{cc\}]) \leq 1) \vee (card(net[\{cc\}]) \leq 1 \wedge card(net^{-1}[\{cc\}]) \leq 2)))$

switch cdv has at most two predecessors and one successor or one predecessor and two successors

$axm12:\ new\_cc1 \in CDV \rightarrow CDV$

$axm13:\ new\_cc2 \in CDV \rightarrow CDV$

END

A.1.2 Railway Machine - RailWay_M0

MACHINE RailWay_M0

SEES RailWay_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

$inv1:\ next \subseteq net$

net represents the total possible connectivity,next represents the current connectivity based on the positions of switches

$inv2:\ next \in CDV \rightarrowtail CDV$

$inv3:\ trns \subseteq TRAIN$

trns is the set of trains on the network. Each train occupies several cdv. The set of cdv occupied by a train should be contiguous under the next function, i.e., there are no gaps in the train and all the switches occupied by the train are in the correct position. occpA is the starting cdv and occpZ is the end cdv of a train.

$inv4:\ occp \in CDV \leftrightarrow trns$

$inv5:\ occpA \in trns \rightarrow CDV$

$inv6:\ \forall tt \cdot (tt \in trns \Rightarrow occpA(tt) \in occp^{-1}[\{tt\}])$

$inv7:\ occpZ \in trns \rightarrow CDV$

$inv8:\ \forall tt \cdot (tt \in trns \Rightarrow occpZ(tt) \in occp^{-1}[\{tt\}])$

Note next does not indicate the direction that a train is moving in,the direction can be occpA to occpZ or occpZ to occpA. Also, since both occpA and occpZ are in the set of cdv occupied by a train, a train occupies at least one cdv

$inv9:\ braking \subseteq trns$

$inv10:\ speed \in trns \rightarrow \mathbb{N}$

$inv12:\ finite(occp^{-1})$

$inv11:\ \forall tt \cdot tt \in trns \wedge card(occp^{-1}[\{tt\}]) > 1 \Rightarrow occpA(tt) \neq occpZ(tt)$

occpA and occpZ must be different if tt occupies more than one cdv:

EVENTS

Initialisation

begin

$act1:\ next := \varnothing$

$act2:\ trns := \varnothing$

$act3:\ occp := \varnothing$

$act4:\ occpA := \varnothing$

$act5:\ occpZ := \varnothing$

$act6:\ braking := \varnothing$

$act7:\ speed := \varnothing$

end

Event enterCDV $\widehat{=}$

any

$t1$ Start occupying the successor of occpZ, i.e., change from
... →0 →t1 →... →t1 →0 →0 →...
to
... →0 →t1 →... →t1 →t1 →0 →...

$c1$

$c2$

where

$grd1 : t1 \in trns$

$grd2 : c1 \in CDV$

$grd3 : c2 \in CDV$

$grd4 : speed(t1) > 0$

$grd5 : c1 = occpZ(t1)$

$grd6 : c1 \in dom(next)$

$grd7 : c2 = next(occpZ(t1))$

$grd8 : \forall tt \cdot tt \in trns \wedge card((occp \cup \{c2 \mapsto t1\})^{-1}[\{tt\}]) > 1 \Rightarrow (occpZ \lhd\!\!\!\!- \{t1 \mapsto c2\})(tt) \neq occpA(tt)$
If tt size is > 1, the beginning section(occpA(tt)) has to be fifferent from the new next end section (occpZ(tt))

then

$act1 : occpZ(t1) := c2$

$act2 : occp := occp \cup \{c2 \mapsto t1\}$

end

Event LeaveCDV $\widehat{=}$

any

$t1$ Stop occupying occpA, i.e., change from

... →0 →t1 →t1 →... →t1 →0 →...
to
... →0 →0 →t1 →... →t1 →0 →...
This is only possible if t1 currently occupies more than one section.

$c1$

$c2$

where

$grd1 : t1 \in trns$

$grd2 : c1 \in CDV$

$grd3 : c2 \in CDV$

$grd4 : speed(t1) > 0$

$grd8:\ c1 \in dom(next)$

$grd5:\ c1 = occpA(t1)$

$grd6:\ c2 = next(c1)$

$grd7:\ occpA(t1) \neq occpZ(t1)$

$grd13:\ \forall tt \cdot tt \in trns \wedge card(((occp \setminus \{c1 \mapsto t1\}) \cup \{c2 \mapsto t1\})^{-1}[\{tt\}]) > 1 \Rightarrow (occpA \lhd\!\!\!+ \{t1 \mapsto c2\})(tt) \neq occpZ(tt)$

then

$act1:\ occpA(t1) := c2$

$act2:\ occp := (occp \setminus \{c1 \mapsto t1\}) \cup \{c2 \mapsto t1\}$

end

Event ChangeSpeed $\widehat{=}$

any

$t1$

$s1$

where

$grd1:\ t1 \in trns$

$grd2:\ s1 \in \mathbb{N}$

$grd3:\ t1 \in braking \Rightarrow s1 < speed(t1)$

then

$act1:\ speed(t1) := s1$

end

Event Brake $\widehat{=}$

any

$t1$

where

$grd1:\ t1 \in TRAIN$

$grd2:\ t1 \in trns \setminus braking$

then

$act1:\ braking := braking \cup \{t1\}$

end

Event SwitchChangeDiv $\widehat{=}$

Here ac is a switch, and it's successor is changed from c1 to c2;ac must not be occupied.

any

ac

$c1$

$c2$

where

$grd1 : ac \in div\_aig\_cdv$

$grd2 : c1 \in CDV$

$grd3 : c2 \in CDV$

$grd8 : c2 \notin ran(next)$

$grd4 : (ac \mapsto c1) \in next$

$grd5 : (ac \mapsto c2) \in net$

$grd6 : c1 \neq c2$

$grd7 : ac \notin dom(occp)$

then

$act1 : next := next \lhd\!\!+ \{ac \mapsto c2\}$

end

Event SwitchChangeCnv $\widehat{=}$

Here ac is a switch, and it's predecessor is changed from c1 to c2;

ac must not be occupied.

any

ac

$c1$

$c2$

where

$grd1 : ac \in cnv\_aig\_cdv$

$grd2 : c1 \in CDV$

$grd3 : c2 \in CDV$

$grd8 : c2 \notin dom(next)$

$grd4 : (c1 \mapsto ac) \in next$

$grd5 : (c2 \mapsto ac) \in net$

$grd6 : c1 \neq c2$

$grd7 : ac \notin dom(occp)$

then

$act1 : next := (\{c1\} \lhd next) \cup \{c2 \mapsto ac\}$

end

Event SendTrainMsg $\widehat{=}$

any

$t1$

where

$grd1 : t1 \in trns$

end

Event RecvTrainMsg $\widehat{=}$

any

$t1$

where

$grd1 : t1 \in trns$

end

END

A.2 First Refinement Railway Machine - RailWay_M1: Introduction of the Communication Layer

MACHINE RailWay_M1

REFINES RailWay_M0

SEES RailWay_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

tmsgs

permit

INVARIANTS

$inv1 : tmsgs \in trns \rightarrow \mathbb{P}(BOOL)$

$inv2:\ permit \in trns \rightarrow BOOL$

EVENTS

Initialisation

begin

$act1:\ next := \varnothing$

$act2:\ trns := \varnothing$

$act3:\ occp := \varnothing$

$act4:\ occpA := \varnothing$

$act5:\ occpZ := \varnothing$

$act6:\ braking := \varnothing$

$act7:\ speed := \varnothing$

$act8:\ tmsgs := \varnothing$

$act9:\ permit := \varnothing$

end

Event enterCDV $\widehat{=}$

Refines enterCDV

any

$t1$

$c1$

$c2$

where

$grd1:\ t1 \in trns$

$grd2:\ c1 \in CDV$

$grd3:\ c2 \in CDV$

$grd4:\ speed(t1) > 0$

$grd5:\ c1 = occpZ(t1)$

$grd6:\ c1 \in dom(next)$

$grd7:\ c2 = next(occpZ(t1))$

$grd8:\ \forall tt \cdot tt \in trns \wedge card((occp \cup \{c2 \mapsto t1\})^{-1}[\{tt\}]) > 1 \Rightarrow (occpZ \lhd\!\!\!\!- \{t1 \mapsto c2\})(tt) \neq occpA(tt)$

then

$act1:\ occpZ(t1) := c2$

$act2:\ occp := occp \cup \{c2 \mapsto t1\}$

end

Event LeaveCDV $\widehat{=}$

Refines LeaveCDV

any

$t1$

$c1$

$c2$

where

$grd1: t1 \in trns$

$grd2: c1 \in CDV$

$grd3: c2 \in CDV$

$grd4: speed(t1) > 0$

$grd8: c1 \in dom(next)$

$grd5: c1 = occpA(t1)$

$grd6: c2 = next(c1)$

$grd7: occpA(t1) \neq occpZ(t1)$

$grd13: \forall tt \cdot tt \in trns \wedge card(((occp \setminus \{c1 \mapsto t1\}) \cup \{c2 \mapsto t1\})^{-1}[\{tt\}]) > 1 \Rightarrow (occpA \lhd\!\!\!\!- \{t1 \mapsto c2\})(tt) \neq occpZ(tt)$

then

$act1: occpA(t1) := c2$

$act2: occp := (occp \setminus \{c1 \mapsto t1\}) \cup \{c2 \mapsto t1\}$

end

Event ChangeSpeed $\widehat{=}$

Refines ChangeSpeed

any

$t1$

$s1$

where

$grd1: t1 \in trns$

$grd2: s1 \in \mathbb{N}$

$grd3: t1 \in braking \Rightarrow s1 < speed(t1)$

then

$act1: speed(t1) := s1$

end

Event Brake $\widehat{=}$

Refines Brake

any

$t1$

where

$grd1 : t1 \in TRAIN$

$grd2 : t1 \in trns \setminus braking$

$grd3 : permit(t1) = FALSE$

then

$act1 : braking := braking \cup \{t1\}$

end

Event SendTrainMsg $\widehat{=}$

Refines SendTrainMsg

any

$t1$

where

$grd1 : t1 \in trns$

$grd2 : tmsgs(t1) = \varnothing$

then

$act1 : tmsgs(t1) := \{bool(occpZ(t1) \in dom(next) \wedge next(occpZ(t1)) \notin dom(occp))\}$

end

Event RecvTrainMsg $\widehat{=}$

Refines RecvTrainMsg

any

$t1$

bb

where

$grd1 : t1 \in trns$

$grd2 : bb \in tmsgs(t1)$

then

$act1 : permit(t1) := bb$

$act2 : tmsgs(t1) := \varnothing$

end

Event SwitchChangeDiv $\widehat{=}$

Refines SwitchChangeDiv

any

ac

$c1$

$c2$

where

$grd1: ac \in div\_aig\_cdv$

$grd2: c1 \in CDV$

$grd3: c2 \in CDV$

$grd4: (ac \mapsto c1) \in next$

$grd5: (ac \mapsto c2) \in net$

$grd9: (ac \mapsto c2) \notin next$

$grd8: c2 \notin ran(next)$

$grd6: c1 \neq c2$

$grd7: ac \notin dom(occp)$

then

$act1: next := next \lhd\!\!\!- \{ac \mapsto c2\}$

end

Event SwitchChangeCnv $\widehat{=}$

Refines SwitchChangeCnv

any

ac

$c1$

$c2$

where

$grd1: ac \in cnv\_aig\_cdv$

$grd2: c1 \in CDV$

$grd3: c2 \in CDV$

$grd8: c2 \notin dom(next)$

$grd4: (c1 \mapsto ac) \in next$

$grd5: (c2 \mapsto ac) \in net$

$grd6: c1 \neq c2$

$grd7: ac \notin dom(occp)$

then

$act1: next := (\{c1\} \lhd\!\!\!- next) \cup \{c2 \mapsto ac\}$

end

END

A.3 Track Specification

A.3.1 Track Context - Tracks_C0

CONTEXT Track_C0

EXTENDS RailWay_C0

SETS

POS Switch Position

CONSTANTS

dev_pos

dir_pos

ind_pos

dir

dev

AXIOMS

$axm1: POS = \{dev\_pos, dir\_pos, ind\_pos\}$

$axm2: dev\_pos \neq dir\_pos$

$axm3: dev\_pos \neq ind\_pos$

$axm4: dir\_pos \neq ind\_pos$

$axm5: dev \subseteq net$

$axm6: dir \subseteq net$

END

A.3.2 Track Machine - Tracks_M0

MACHINE Track_M0

SEES Track_C0

VARIABLES

next Current connectivity based on switch positions

trns Set of trains on network

occp Occupancy function for sections

occpA Initial cdv occupied by train

occpZ Finl cdv occupied by train

INVARIANTS

$inv1:\ next \subseteq net$

$inv2:\ next \in CDV \rightarrowtail CDV$

$inv3:\ trns \subseteq TRAIN$

$inv4:\ occp \in CDV \leftrightarrow trns$

$inv5:\ occpA \in trns \rightarrow CDV$

$inv6:\ occpZ \in trns \rightarrow CDV$

$inv7:\ finite(occp^{-1})$

EVENTS

Initialisation

begin

$act1:\ next := \varnothing$

$act2:\ trns := \varnothing$

$act3:\ occp := \varnothing$

$act4:\ occpA := \varnothing$

$act5:\ occpZ := \varnothing$

end

Event SendTrainMsg $\widehat{=}$

any

$t1$

bb

where

$grd1:\ t1 \in TRAIN$

$grd4:\ t1 \in trns$

$grd2:\ bb \in BOOL$

$grd3:\ bb = bool(occpZ(t1) \in dom(next) \land next(occpZ(t1)) \notin dom(occp))$

end

Event EnterCDV $\widehat{=}$

any

$t1$

$c1$

$c2$

where

$grd1:\ t1 \in TRAIN$

$grd2:\ c1 \in CDV$

$grd3:\ c2 \in CDV$

$grd4:\ t1 \in trns$

$grd5:\ c1 \in dom(next)$

$grd6:\ c2 = next(c1)$

$grd7:\ \forall tt \cdot tt \in trns \land card((occp \cup \{c2 \mapsto t1\})^{-1}[\{tt\}]) > 1 \Rightarrow (occpZ \lhd\!\!\!\!- \{t1 \mapsto c2\})(tt) \neq occpA(tt)$

then

$act1:\ occpZ(t1) := c2$

occpZ(t1) := c1 -¿ mistake??

$act2:\ occp := occp \cup \{c2 \mapsto t1\}$

end

Event LeaveCDV $\widehat{=}$

any

$t1$

$c1$

$c2$

where

$grd2:\ c1 \in CDV$

$grd3:\ c2 \in CDV$

$grd4:\ t1 \in trns$

$grd5:\ c1 \in dom(next)$

$grd6:\ c2 = next(c1)$

$grd7:\ \forall tt \cdot tt \in trns \land card(((occp \setminus \{c1 \mapsto t1\}) \cup \{c2 \mapsto t1\})^{-1}[\{tt\}]) > 1 \Rightarrow (occpA \lhd\!\!\!\!- \{t1 \mapsto c2\})(tt) \neq occpZ(tt)$

then

$act1:\ occpA(t1) := c2$

$act2:\ occp := (occp \setminus \{c1 \mapsto t1\}) \cup \{c2 \mapsto t1\}$

end

Event SwitchChangeDiv1 $\widehat{=}$

any

ac

$c1$

$c2$

where

$grd1 : ac \in CDV$

$grd2 : c1 \in CDV$

$grd3 : c2 \in CDV$

$grd4 : ac \in div\_aig\_cdv$

$grd5 : (ac \mapsto c1) \in next$

$grd6 : (ac \mapsto c2) \in net$

$grd7 : c1 \neq c2$

$grd8 : ac \notin dom(occp)$

$grd9 : c2 \notin ran(next)$

Added for helping the proving. Confirms that section(CDV) c2 is not the end connected of any other section

then

$act1 : next := next \lhd\!\!\!- \{ac \mapsto c2\}$

end

Event SwitchChangeDiv2 $\widehat{=}$

any

ac

$c1$

$c2$

where

$grd1 : ac \in CDV$

$grd2 : c1 \in CDV$

$grd3 : c2 \in CDV$

$grd4 : ac \in div\_aig\_cdv$

$grd5 : c1 \neq c2$

$grd6 : ac \notin dom(occp)$

$grd7 : (ac \mapsto c1) \in next$

$grd8 : (ac \mapsto c2) \in net$

$grd9 : c2 \notin ran(next)$

Added for helping the proving. Confirms that section(CDV) c2 is not the end connected of any other section

then

$act1 : next := next \lhd\!\!\!- \{ac \mapsto c2\}$

end

Event SwitchChangeCnv1 $\widehat{=}$

any

ac

$c1$

$c2$

where

$grd1 : ac \in cnv\_aig\_cdv$

$grd2 : c1 \in CDV$

$grd3 : c2 \in CDV$

$grd4 : c1 \mapsto ac \in next$

$grd5 : c2 \mapsto ac \in net$

$grd6 : c1 \neq c2$

$grd7 : ac \notin dom(occp)$

$grd8 : c2 \notin dom(next)$

then

$act1 : next := ((\{c1\} \lhd next) \cup \{c2 \mapsto ac\})$

end

Event SwitchChangeCnv2 $\widehat{=}$

any

ac

$c1$

$c2$

where

$grd1 : ac \in cnv\_aig\_cdv$

$grd2 : c1 \in CDV$

$grd3 : c2 \in CDV$

$grd4 : c1 \mapsto ac \in next$

$grd5 : c2 \mapsto ac \in net$

$grd6 : c1 \neq c2$

$grd7 : ac \notin dom(occp)$

$grd8 : c2 \notin dom(next)$

then

$act1 : next := ((\{c1\} \lhd next) \cup \{c2 \mapsto ac\})$

end

Event InitSwitchChangeDiv1 $\widehat{=}$

Which is anticipated

any

ac

$c1$

$c2$

where

$grd1 : ac \in CDV$

$grd2 : c1 \in CDV$

$grd3 : c2 \in CDV$

end

Event InitSwitchChangeDiv2 $\widehat{=}$

Which is anticipated

any

ac

$c1$

$c2$

where

$grd1 : ac \in CDV$

$grd2 : c1 \in CDV$

$grd3 : c2 \in CDV$

end

Event InitSwitchChangeCnv1 $\widehat{=}$

Which is anticipated

any

ac

$c1$

$c2$

where

$grd1 : ac \in CDV$

$grd2 : c1 \in CDV$

$grd3 : c2 \in CDV$

end

Event InitSwitchChangeCnv2 $\widehat{=}$

Which is anticipated

any

ac

$c1$

$c2$

where

$grd1 : ac \in CDV$

$grd2 : c1 \in CDV$

$grd3 : c2 \in CDV$

end

Event SendSectorMsg $\widehat{=}$

Which is anticipated

begin

end

Event RecvSectorMsg $\widehat{=}$

Which is anticipated

begin

end

END

A.4 Train Specification

A.4.1 Train Machine - Trains_M0

MACHINE Trains_M0

SEES RailWay_C0

VARIABLES

$nextTrain$

$trnsTrain$

$occpTrain$

$occpATrain$

$occpZTrain$

$speed$

$permit$

$braking$

INVARIANTS

$inv1: nextTrain \in CDV \rightarrowtail CDV$

$inv2: trnsTrain \subseteq TRAIN$

$inv3: nextTrain \subseteq net$

$inv4: occpTrain \in CDV \leftrightarrow trnsTrain$

$inv5: occpATrain \in trnsTrain \rightarrow CDV$

$inv6: occpZTrain \in trnsTrain \rightarrow CDV$

$inv7: speed \in trnsTrain \rightarrow \mathbb{N}$

$inv8: permit \in trnsTrain \rightarrow BOOL$

$inv9: braking \subseteq trnsTrain$

EVENTS

Initialisation

begin

$act1: nextTrain := \varnothing$

$act2: trnsTrain := \varnothing$

$act3: occpTrain := \varnothing$

$act4: occpATrain := \varnothing$

$act5: occpZTrain := \varnothing$

$act6: speed := \varnothing$

$act7: permit := \varnothing$

$act8: braking := \varnothing$

end

Event RecvTrainMsg $\widehat{=}$

any

$t1$

bb

where

$grd1: t1 \in trnsTrain$

$grd2: bb \in BOOL$

then

$act1: permit(t1) := bb$

end

Event ChangeSpeed $\widehat{=}$

any

$t1$

$s1$

where

$grd1: t1 \in TRAIN$

$grd2: s1 \in \mathbb{N}$

$grd3: t1 \in trnsTrain$

$grd4: t1 \in braking \Rightarrow s1 < speed(t1)$

then

$act1: speed(t1) := s1$

end

Event Brake $\widehat{=}$

any

$t1$

where

$grd1: t1 \in TRAIN$

$grd2: t1 \in trnsTrain \setminus braking$

$grd3: permit(t1) = FALSE$

then

$act1: braking := braking \cup \{t1\}$

end

Event EnterCDV $\widehat{=}$

any

$t1$

$c1$

$c2$

where

$grd2: c1 \in CDV$

$grd3: c2 \in CDV$

$grd4: t1 \in trnsTrain$

$grd5: c1 = occpZTrain(t1)$

$grd6: speed(t1) > 0$

then

$act1:\ occpZTrain(t1) := c2$

$act2:\ occpTrain := occpTrain \cup \{c2 \mapsto t1\}$

end

Event LeaveCDV $\widehat{=}$

any

$t1$

$c1$

$c2$

where

$grd2:\ c1 \in CDV$

$grd3:\ c2 \in CDV$

$grd4:\ t1 \in trnsTrain$

$grd5:\ c1 = occpATrain(t1)$

$grd6:\ speed(t1) > 0$

$grd1:\ occpATrain(t1) \neq occpZTrain(t1)$

then

$act1:\ occpATrain(t1) := c2$

$act2:\ occpTrain := (occpTrain \setminus \{c1 \mapsto t1\})$

end

END

A.5 Communication Specification

A.5.1 Communication Machine - Comms_M0

CONTEXT Comms_C0

EXTENDS RailWay_C0

SETS

$CommsCSPState$

CONSTANTS

CM

CM_1

new_mm_1

AXIOMS

$axm1:\ CommsCSPState = \{CM, CM\_1\}$

$axm2:\ CM \neq CM\_1$

$axm3:\ new\_mm\_1 \in TRAIN \rightarrow BOOL$

END

A.5.2 Communication Machine - Comms_M0

MACHINE Comms_M0

SEES Comms_C0

VARIABLES

$tmsgs$ CommsCSP

mm_1

INVARIANTS

$inv2:\ tmsgs \in TRAIN \rightarrow \mathbb{P}(BOOL)$

$inv3:\ mm\_1 \in TRAIN \rightarrow BOOL$

EVENTS

Initialisation

begin

$act2:\ tmsgs := \varnothing$

$act3:\ mm\_1 := new\_mm\_1$

end

Event Send $\widehat{=}$

any

$t1$

bb

where

$grd1:\ t1 \in TRAIN$

$grd4:\ t1 \in dom(tmsgs)$

$grd2:\ bb \in BOOL$

CommsCSP(t1) = CM

$grd3: tmsgs(t1) = \varnothing$

then

$act1: tmsgs(t1) := \{bb\}$
CommsCSP(t1) := CM_1

$act3: mm\_1(t1) := bb$

end

Event Recv $\widehat{=}$

any

$t1$

bb

where

$grd1: t1 \in TRAIN$

$grd3: t1 \in dom(tmsgs)$

$grd4: bb = mm\_1(t1)$
CommsCSP(t1) = CM_1

$grd2: mm\_1(t1) \in tmsgs(t1)$

then

$act1: tmsgs(t1) := \varnothing$
CommsCSP(t1):= CM

end

END

A.6 Second Refinement Railway Machine - RailWay_M1: Parallel Composition of machines Trains_M0,Tracks_M0,Comms

MACHINE RailWayComposition4

REFINES RailWay_M1

SEES Track_C0, Comms_C0

VARIABLES

$nextTrain$

$trnsTrain$

$occpTrain$

$occpATrain$

$occpZTrain$

$speed$

$permit$

$braking$

$next$ Current connectivity based on switch positions

$trns$ Set of trains on network

$occp$ Occupancy function for sections

$occpA$ Initial cdv occupied by train

$occpZ$ Finl cdv occupied by train

$tmsgs$ CommsCSP

mm_1

INVARIANTS

$Trains\_M0/inv1 : nextTrain \in CDV \rightarrowtail CDV$

$Trains\_M0/inv2 : trnsTrain \subseteq TRAIN$

$Trains\_M0/inv3 : nextTrain \subseteq net$

$Trains\_M0/inv4 : occpTrain \in CDV \leftrightarrow trnsTrain$

$Trains\_M0/inv5 : occpATrain \in trnsTrain \rightarrow CDV$

$Trains\_M0/inv6 : occpZTrain \in trnsTrain \rightarrow CDV$

$Trains\_M0/inv7 : speed \in trnsTrain \rightarrow \mathbb{N}$

$Trains\_M0/inv8 : permit \in trnsTrain \rightarrow BOOL$

$Trains\_M0/inv9 : braking \subseteq trnsTrain$

$RailWayComposition/inv1 : occpZTrain = occpZ$

$RailWayComposition/inv2 : occpATrain = occpA$

$RailWayComposition/inv4 : trns = trnsTrain$

$RailWayComposition/inv5 : mm\_1 \in TRAIN \rightarrow BOOL$

EVENTS

Initialisation

begin

$Trains\_M0/act1 : nextTrain := \varnothing$

$Trains\_M0/act2 : trnsTrain := \varnothing$

$Trains\_M0/act3 : occpTrain := \varnothing$

$Trains\_M0/act4 : occpATrain := \varnothing$

$Trains\_M0/act5 : occpZTrain := \varnothing$

$Trains\_M0/act6 :\ speed := \varnothing$

$Trains\_M0/act7 :\ permit := \varnothing$

$Trains\_M0/act8 :\ braking := \varnothing$

$Track\_M0/act1 :\ next := \varnothing$

$Track\_M0/act2 :\ trns := \varnothing$

$Track\_M0/act3 :\ occp := \varnothing$

$Track\_M0/act4 :\ occpA := \varnothing$

$Track\_M0/act5 :\ occpZ := \varnothing$

$Comms\_M0/act2 :\ tmsgs := \varnothing$

$Comms\_M0/act3 :\ mm\_1 := new\_mm\_1$

end

Event SendTrainMsg $\widehat{=}$

Refines SendTrainMsg

any

$t1$

bb

where

$Track\_M0/grd1 :\ t1 \in TRAIN$

$Track\_M0/grd4 :\ t1 \in trns$

$Track\_M0/grd2 :\ bb \in BOOL$

$Track\_M0/grd3 :\ bb = bool(occpZ(t1) \in dom(next) \land next(occpZ(t1)) \notin dom(occp))$

$Comms\_M0/grd1 :\ t1 \in TRAIN$

$Comms\_M0/grd4 :\ t1 \in dom(tmsgs)$

$Comms\_M0/grd2 :\ bb \in BOOL$

$Comms\_M0/grd3 :\ tmsgs(t1) = \varnothing$

then

$Comms\_M0/act1 :\ tmsgs(t1) := \{bb\}$

$Comms\_M0/act3 :\ mm\_1(t1) := bb$

end

Event RecvTrainMsg $\widehat{=}$

Refines RecvTrainMsg

any

$t1$

bb

where

$Comms\_M0/grd1 :\ t1 \in TRAIN$

$Comms\_M0/grd3 :\ t1 \in dom(tmsgs)$

$Comms\_M0/grd4 :\ bb = mm\_1(t1)$

$Comms\_M0/grd2 :\ mm\_1(t1) \in tmsgs(t1)$

$Trains\_M0/grd1 :\ t1 \in trnsTrain$

$Trains\_M0/grd2 :\ bb \in BOOL$

then

$Comms\_M0/act1 :\ tmsgs(t1) := \varnothing$

$Trains\_M0/act1 :\ permit(t1) := bb$

end

Event ChangeSpeed $\widehat{=}$

Refines ChangeSpeed

any

$t1$

$s1$

where

$Trains\_M0/grd1 :\ t1 \in TRAIN$

$Trains\_M0/grd2 :\ s1 \in \mathbb{N}$

$Trains\_M0/grd3 :\ t1 \in trnsTrain$

$Trains\_M0/grd4 :\ t1 \in braking \Rightarrow s1 < speed(t1)$

then

$Trains\_M0/act1 :\ speed(t1) := s1$

end

Event Brake $\widehat{=}$

Refines Brake

any

$t1$

where

$Trains\_M0/grd1 :\ t1 \in TRAIN$

$Trains\_M0/grd2 :\ t1 \in trnsTrain \setminus braking$

$Trains\_M0/grd3 :\ permit(t1) = FALSE$

then

$Trains\_M0/act1:\ braking := braking \cup \{t1\}$

end

Event EnterCDV $\widehat{=}$

Refines enterCDV

any

$t1$

$c1$

$c2$

where

$Trains\_M0/grd2:\ c1 \in CDV$

$Trains\_M0/grd3:\ c2 \in CDV$

$Trains\_M0/grd4:\ t1 \in trnsTrain$

$Trains\_M0/grd5:\ c1 = occpZTrain(t1)$

$Trains\_M0/grd6:\ speed(t1) > 0$

$Track\_M0/grd1:\ t1 \in TRAIN$

$Track\_M0/grd2:\ c1 \in CDV$

$Track\_M0/grd3:\ c2 \in CDV$

$Track\_M0/grd4:\ t1 \in trns$

$Track\_M0/grd5:\ c1 \in dom(next)$

$Track\_M0/grd6:\ c2 = next(c1)$

$Track\_M0/grd7:\ \forall tt \cdot tt \in trns \wedge card((occp \cup \{c2 \mapsto t1\})^{-1}[\{tt\}]) > 1 \Rightarrow (occpZ \lhd\!\!\!\!- \{t1 \mapsto c2\})(tt) \neq occpA(tt)$

then

$Trains\_M0/act1:\ occpZTrain(t1) := c2$

$Trains\_M0/act2:\ occpTrain := occpTrain \cup \{c2 \mapsto t1\}$

$Track\_M0/act1:\ occpZ(t1) := c2$

$Track\_M0/act2:\ occp := occp \cup \{c2 \mapsto t1\}$

end

Event LeaveCDV $\widehat{=}$

Refines LeaveCDV

any

$t1$

$c1$

$c2$

where

$Trains\_M0/grd2:\ c1 \in CDV$

$Trains\_M0/grd3:\ c2 \in CDV$

$Trains\_M0/grd4:\ t1 \in trnsTrain$

$Trains\_M0/grd5:\ c1 = occpATrain(t1)$

$Trains\_M0/grd6:\ speed(t1) > 0$

$Trains\_M0/grd1:\ occpATrain(t1) \neq occpZTrain(t1)$

$Track\_M0/grd2:\ c1 \in CDV$

$Track\_M0/grd3:\ c2 \in CDV$

$Track\_M0/grd4:\ t1 \in trns$

$Track\_M0/grd5:\ c1 \in dom(next)$

$Track\_M0/grd6:\ c2 = next(c1)$

$Track\_M0/grd7:\ \forall tt \cdot tt \in trns \wedge card(((occp \backslash \{c1 \mapsto t1\}) \cup \{c2 \mapsto t1\})^{-1}[\{tt\}]) > 1 \Rightarrow (occpA \lhd\!\!\!\!- \{t1 \mapsto c2\})(tt) \neq occpZ(tt)$

then

$Trains\_M0/act1:\ occpATrain(t1) := c2$

$Trains\_M0/act2:\ occpTrain := (occpTrain \setminus \{c1 \mapsto t1\})$

$Track\_M0/act1:\ occpA(t1) := c2$

$Track\_M0/act2:\ occp := (occp \setminus \{c1 \mapsto t1\}) \cup \{c2 \mapsto t1\}$

end

Event SwitchChangeDiv1 $\widehat{=}$

Refines SwitchChangeDiv

any

ac

$c1$

$c2$

where

$Track\_M0/grd1:\ ac \in CDV$

$Track\_M0/grd2:\ c1 \in CDV$

$Track\_M0/grd3:\ c2 \in CDV$

$Track\_M0/grd4:\ ac \in div\_aig\_cdv$

$Track\_M0/grd5:\ (ac \mapsto c1) \in next$

$Track\_M0/grd6:\ (ac \mapsto c2) \in net$

$Track\_M0/grd7:\ c1 \neq c2$

$Track\_M0/grd8:\ ac \notin dom(occp)$

$Track\_M0/grd9 :\ c2 \notin ran(next)$

Added for helping the proving. Confirms that section(CDV) c2 is not the end connected of any other section

then

$Track\_M0/act1 :\ next := next \Leftarrow \{ac \mapsto c2\}$

end

Event SwitchChangeCnv1 $\widehat{=}$

Refines SwitchChangeCnv

any

ac

$c1$

$c2$

where

$Track\_M0/grd1 :\ ac \in cnv\_aig\_cdv$

$Track\_M0/grd2 :\ c1 \in CDV$

$Track\_M0/grd3 :\ c2 \in CDV$

$Track\_M0/grd4 :\ c1 \mapsto ac \in next$

$Track\_M0/grd5 :\ c2 \mapsto ac \in net$

$Track\_M0/grd6 :\ c1 \neq c2$

$Track\_M0/grd7 :\ ac \notin dom(occp)$

$Track\_M0/grd8 :\ c2 \notin dom(next)$

then

$Track\_M0/act1 :\ next := ((\{c1\} \lhd next) \cup \{c2 \mapsto ac\})$

end

END

Bibliography

[1] "Free on-line dictionary of computing." http://foldoc.org, September 2008.

[2] J.-R. Abrial, "Formal methods: Theory becoming practice," *J. UCS*, vol. 13, pp. 619–628, may 2007.

[3] S. Schneider, *The B method: an introduction*. Palgrave, 2001.

[4] "*RODIN* project homepage." http://rodin.cs.ncl.ac.uk, September 2008.

[5] J.-R. Abrial and S. Hallerstede, "Refinement, decomposition, and instantiation of discrete models: Application to event-b," *Fundam. Inf.*, vol. 77, no. 1-2, pp. 1–28, 2007.

[6] "Eclipse homepage." http://www.eclipse.org, September 2008.

[7] M. Butler, "A system-based approach to the formal development of embedded controllers for a railway," *Design Automation for Embedded Systems*, vol. 6, pp. 355–366, 2002.

[8] M. J. Butler, "csp2b: A practical approach to combining csp and b," *Formal Aspects of Computing*, vol. 12, pp. 182–196, 2000.

[9] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science, 1985.

[10] J.-R. Abrial, "Formal methods in industry: achievements, problems, future," in *ICSE '06: Proceedings of the 28th international conference on Software engineering*, (New York, NY, USA), pp. 761–768, ACM, 2006.

[11] "Waterfall model - wikipedia." http://en.wikipedia.org/wiki/Waterfall_model, September 2008.

[12] X. Liu, Z. Chen, H. Yang, H. Zedan, and W. 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*, (Washington, DC, USA), p. 342, IEEE Computer Society, 1997.

[13] X. Liu, H. Yang, and H. 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*, (Washington, DC, USA), p. 409, IEEE Computer Society, 1997.

[14] J. M. Spivey, *The Z Notation: a Reference Manual.* Prentice-Hall, Inc., 1989.

[15] J.-R. Abrial, *The B-book: assigning programs to meanings.* New York, NY, USA: Cambridge University Press, 1996.

[16] C. B. Jones, *Sytematic Software Development Using VDM.* Prentice Hall International, second edition, 1990.

[17] E. W. Dijkstra and C. S. Scholten, *Predicate Calculus and ProgramSemantics.* Springer-Verlag, 1990. No pdf file.

[18] E. W. Dijkstra, *A Discipline of Programming.* EnglewoodCliffs, New Jersey: Prentice-Hall, 1976.

[19] M. A. Orgun and W. Ma, "An overview of temporal and modal logic programming," in *ICTL '94: Proceedings of the First International Conference on Temporal Logic* (D. M. Gabbay and H. J. Ohlbach, eds.), (Berlin Heidelberg), pp. 445–479, Springer-Verlag, 1994.

[20] Z. Manna and A. Pnueli, *The temporal logic of reactive and concurrent systems.* New York, NY, USA: Springer-Verlag New York, Inc., 1992. No pdf file.

[21] J. Goguen and J. Tardo, "An introduction to obj: a language for writing and testing software specifications," in *Specification of Reliable Software* (M. K. Zelkowitz, ed.), pp. 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.

[22] J. V. Guttag and J. J. Horning, *Larch: languages and tools for formal specification.* New York, NY, USA: Springer-Verlag New York, Inc., 1993.

[23] R. Milner, *Communication and concurrency.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.

[24] J. C. M. Baeten and J. A. Bergstra, "Real time process algebra," *Formal Asp. Comput.*, vol. 3, no. 2, pp. 142–188, 1991. No pdf file.

[25] ISO, "Information systems processing - open systems interconnection-lotos," tech. rep., ISO, 1987.

[26] W. Reisig, *Petri nets: an introduction.* New York, NY, USA: Springer-Verlag New York, Inc., 1985. No pdf file.

[27] J. Hooman, S. Ramesh, and W. d. Roever, "A compositional semantics for statecharts," in *Liber Amicorum: J.W. de Bakker, 25 jaar semantiek* (J.-J. M. J.W. Klop and J. Rutten, eds.), (Amsterdam: CWI), pp. 275–287, 1989.

[28] C. Snook and M. Butler, "Uml-b and event-b: an integration of languages and tools," in *The IASTED International Conference on Software Engineering - SE2008*, February 2008.

[29] M. Abadi and L. Lamport, "Composing Specifications," in *Stepwise Refinement of Distributed Systems - Models, Formalisms, Correctness* (J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, eds.), vol. 430, (Berlin, Germany), pp. 1–41, Springer-Verlag, 1989.

[30] T. 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)*, (Washington, DC, USA), pp. 107–116, IEEE Computer Society, 2004.

[31] L. Fiege, G. Mühl, and F. C. Gärtner, "Modular event-based systems," *Knowl. Eng. Rev.*, vol. 17, no. 4, pp. 359–388, 2002.

[32] H. Baumeister, "Using algebraic specification languages for model-oriented specifications," Technical Report MPI-I-96-2-003, Max-Planck-Institut für Informatik, Saarbrücken, Feb. 1996.

[33] M. 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*, pp. 221–241, 1997.

[34] M. Butler and J. M. Silva, "Lectures notes of comp1001 introduction to formal methods university of southampton - ecs." https://secure.ecs.soton.ac.uk/notes/comp1001/SE.pdf, 2008.

[35] A. Rezazadeh, *Formal Patterns for Web-based Systems Design.* PhD thesis, Southampton University, 2007.

[36] "Event-b.org." http://www.event-b.org, 2008.

[37] S. Hallerstede, *Justifications for the Event-B Modelling Notation*, pp. 49–63. Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2006.

[38] M. Butler, "Synchronisation-based decomposition for event b," in *RODIN Deliverable D19 Intermediate report on methodology*, 2006.

[39] J.-R. Abrial, M. J. Butler, S. Hallerstede, and L. Voisin, "An open extensible tool environment for event-b," in *ICFEM*, pp. 588–605, 2006.

[40] A. Rezazadeh, N. Evans, and M. Butler, "Redevelopment of an industrial case study using event-b and rodin," in *BCS-FACS Christmas 2007 Meeting - Formal Methods In Industry*, December 2007.

[41] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Trans. Softw. Eng.*, vol. 3, no. 2, pp. 125–143, 1977.

[42] E. Ball, *An Incremental Process for the Development of Multi-Agent Systems in Event-B*. PhD thesis, Southampon University, 2008.

[43] M. Butler and S. Hallerstede, "The rodin formal modelling tool," *BCS-FACS Christmas 2007 Meeting - Formal Methods In Industry, London.*, December 2007.

[44] A. Bolour, "Eclipse plug-in architecture." http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html, September 2008.

[45] "B4free." http://www.b4free.com, September 2008.

[46] "Prob." http://www.stups.uni-duesseldorf.de/ProB/overview.php, September 2008.

[47] "Brama: Graphical tool of modeling applied to the b formal method." http://www.brama.fr/, September 2008.

[48] "Formal modelling with uml." http://users.ecs.soton.ac.uk/cfs/umlb.html, September 2008.

[49] S. Berezin, S. V. A. Campos, and E. M. Clarke, "Compositional reasoning in model checking," in *COMPOS'97: Revised Lectures from the International Symposium on Compositionality: The Significant Difference*, (London, UK), pp. 81–102, Springer-Verlag, 1998.

[50] M. Charpentier, "Reasoning about composition: A predicate transformer approach," in *In Speci and Veri of Component-Based Systems (SAVCBS'2001)*, pp. 42–49, 2001.

[51] D. Giannakopoulou and C. S. Păsăreanu, "Special issue on learning techniques for compositional reasoning," *Form. Methods Syst. Des.*, vol. 32, no. 3, pp. 173–174, 2008.

[52] F. de Boer, U. Hannemann, and W. de Roever, *A compositional proof system for shared variable concurrency*, pp. 515–532. Springer Berlin / Heidelberg, 1997.

[53] C. 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.

[54] F. S. de Boer and W. P. de Roever, "Compositional proof methods for concurrency: A semantic approach," in *COMPOS'97: Revised Lectures from the International Symposium on Compositionality: The Significant Difference*, (London, UK), pp. 632–646, Springer-Verlag, 1998.

[55] A. W. Roscoe, *The theory and practice of concurrency*. Prentice Hall, 1998.

[56] N. Moffat and M. Goldsmith, "Assumption-commitment support for csp model checking," *Electron. Notes Theor. Comput. Sci.*, vol. 185, pp. 121–137, 2007.

[57] K. M. Chandy and B. A. Sanders, "Reasoning about program composition," tech. rep., University of Florida, Department of Computer and Information Science and Engineering, 2000.

[58] Y. Deng, C. Palamidessi, and J. Pang, "Compositional reasoning for probabilistic finite-state behaviors," in *In Processes, Terms and Cycles: Steps on the Road to Infinity, Essays Dedicated to Jan Willem Klop, on the Occasion of His 60th Birthday, LNCS 3838*, pp. 309–337, Springer, 2005.

[59] S. Berezin and D. Gurov, "A compositional proof system for the modal μ - calculus and ccs," tech. rep., School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, January 1997.

[60] "Functional decomposition." http://en.wikipedia.org/wiki/Functional_decomposition, September 2008.

[61] C. Métayer, J.-R. Abrial, and L. Voisin, "Event-b language," tech. rep., Deliverable 3.2, EU Project IST-511599 - RODIN, May 2005.

[62] M. J. Butler, "Stepwise refinement of communicating systems," *Science of Computer Programming*, vol. 27, pp. 139–173, September 1996.

[63] "B-core, the b-technology company: B-toolkit.." http://www.b-core.com/btoolkit.html, September 2008.

[64] "Atelier b web page.." http://http://www.atelierb.eu/, September 2008.

[65] M.-L. Potet and Y. 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*, (London, UK), pp. 46–65, Springer-Verlag, 1998.

[66] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[67] S. Brookes, *Retracing the Semantics of CSP*, pp. 1–14. Springer Berlin / Heidelberg, 2005.

[68] M. Butler, *A CSP Approach to Action Systems*. PhD thesis, Oxford University, 1992.

[69] R. Back and J. von Wright, *Trace refinement of action systems*, pp. 367–384. Springer Berlin / Heidelberg, 1994.

[70] M. Butler and M. Waldén, "Distributed system development in B," Tech. Rep. TUCS-TR-53, Turku Centre for Computer Science, 14, 1996.

[71] R. J. Back, "Refinement of parallel and reactive programs," tech. rep., California Institute of Technology, Pasadena, CA, USA, 1992.

[72] J. Jacky, *The way of Z: practical programming with formal methods*. New York, NY, USA: Cambridge University Press, 1996. No pdf file.

[73] J. P. Bowen, *Formal Specification and Documentation using Z: A Case Study Approach*. International Thomson Computer Press, 1996.

[74] H. Reza, "The z notation presentation." http://www.cs.und.edu/~reza/Csci562Intr-Z.ppt, September 2008.

[75] E. Boiten, J. Derrick, H. Bowman, and M. Steen, "Coupling schemas: data refinement and view(point) composition," in *2nd BCS-FACS Northern Formal Methods Workshop* (D. Duke and A. Evans, eds.), Workshops in Computing, p. 18, Springer-Verlag, July 1997.

[76] D. Jackson, "Structuring z specifications with views," *ACM Trans. Softw. Eng. Methodol.*, vol. 4, no. 4, pp. 365–389, 1995.

[77] L. Lamport, "Tlz," *Z Users Conference*, pp. 267–268, 1994.

[78] A. Coombes and J. McDermid, "Specifying temporal requirements for distributed real-time systems in z," *Software Engineering Journal*, vol. 8, pp. 273–283, Sep 1993.

[79] S. A. Schuman, D. H. Pitt, and P. H. Byers, "Object-oriented process specification," in *Proceedings of the BCS-FACS Workshop on Specification and Verification of Concurrent Systems*, (London, UK), pp. 21–70, Springer-Verlag, 1990.

[80] Y. Ledru and P.-Y. Schobbens, "Applying vdm to large developments," in *Conference proceedings on Formal methods in software development*, (New York, NY, USA), pp. 55–58, ACM, 1990.

[81] J. S. Fitzgerald, "Triumphs and challenges for model-oriented formal methods: The vdm++ experience (abstract)," Nov. 2006.

[82] J. Lu, "Introducing data decomposition into vdm for tractable development of programs," *SIGPLAN Not.*, vol. 30, no. 9, pp. 41–50, 1995.

[83] J. Woodcock and B. Dickinson, "Using vdm with rely and guarantee-conditions," in *Proceedings of the 2nd VDM-Europe Symposium on VDM—The Way Ahead*, (New York, NY, USA), pp. 434–458, Springer-Verlag New York, Inc., 1988.

[84] C. B. Jones, "Tentative steps toward a development method for interfering programs," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 4, pp. 596–619, 1983.

[85] L. V. Vafeiadis, *A Marriage of Rely/Guarantee and Separation.* Springer Berlin / Heidelberg, 2007.

[86] J.-R. Abrial, D. Cansell, and D. Méry, "Refinement and reachability in event_b," *ZB 2005: Formal Specification and Development in Z and B*, pp. 222–241, 2005.