UNIVERSITY OF SOUTHAMPTON

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING

Electronics and Computer Science

Extending the ERS Approach for Workflow Modelling in Event-B

by

Dana Dghaym

Thesis for the degree of Doctor of Philosophy

April 2017

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING
Electronics and Computer Science

Doctor of Philosophy

EXTENDING THE ERS APPROACH FOR WORKFLOW MODELLING IN EVENT-B

by Dana Dghaym

The Event Refinement Structures (ERS) approach augments the Event-B formal method with hierarchical diagrams, providing explicit support for control flow and refinement relationships. ERS was originally designed to decompose the atomicity of the events in Event-B and later enriched with control flow combinators.

Combining graphical workflow approaches with formal methods has been a subject of interest in both industry and academia, resulting in a diversity of approaches. In this thesis, we present an approach for workflow modelling that addresses both control flow and data handling. ERS is used for control flow, while Event-B mathematical notation supports the data handling. This separation simplifies the modelling by avoiding an extensive number of patterns, though separation does not mean the independence of control flow from data handling. The dependency is achieved by the ERS semantics, which are acquired by transforming the diagrams to Event-B. This combination not only benefits from the verification capabilities of Event-B and the graphical nature of ERS, but also supports incremental modelling through refinement and hierarchy.

Our studies resulted in extending the ERS approach to support more flexible behaviour like unbounded replication and exception handling. Unbounded replication is needed when the number of instances of a flow to be executed is unknown and additional instances can be initiated during execution. We also enhance some of the existing ERS combinators such as the loop. We validate our approach and extensions by applying them to two complex workflows, the fire dispatch system and the travel agency booking system. Finally, we extend the ERS formal language with new translation rules to support our new ERS extensions. We formally define the new translation rules of ERS to Event-B, using the Augmented Backus-Naur Form (ABNF), to be easily integrated in the ERS plug-in. The ERS plug-in is a tool providing automatic generation of part of the Event-B model representing types and sequencing. We also evaluate the ERS combinators in control flow modelling against already published criteria.

Contents

List of Figures

List of Tables

Declaration of Authorship

I, Dana Dghaym , declare that the thesis entitled *Extending the ERS Approach for Workflow Modelling in Event-B* and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as: (Dghaym et al., 2013), (Dghaym et al., 2016)

Signed:...

Date:..

Acknowledgements

I would like to express my sincere gratitude to my supervisor Professor Michael Butler, for all his guidance and support during my PhD. I cannot thank him enough for all what I have learned and will continue to learn from him, and for his great understanding of my family commitments. I would also like to thank my second supervisor Dr Asieh Salehi Fathabadi, who was not only a supervisor but also a great friend.

I would like to thank Dr Corina Cirstea for all her valuable feedback and comments. I would also like to thank Matheus Garay Trindade and Dr Colin Snook for all their help in development with the iUML-B framework.

I would also like to express my gratitude to my former colleagues at the Hampshire Fire and Rescue Service for their help with my case study requirements.

Special thanks to all my friends in the ESS group for all their encouragement, support and great company during my PhD.

I cannot thank enough my parents, sisters and brothers, who despite all the distance were always beside me and gave me the strength and encouragement to pursue all my dreams.

Lastly, I would like to thank my beautiful little family. My husband and soulmate for all his love, support and encouragement. My angel, Celine who is not only just my little daughter, but also my best friend and the most amazing little helper. My forever baby, Adam for all the laughter and smiles that enlightened my days.

Chapter 1

Introduction

Formal methods are mathematical techniques used for the systematic specification, verification and development of both hardware and software systems. However, these techniques have been mainly criticised for their difficulty (Wing, 1990; Hall, 1990; Bowen and Hinchey, 1995; Abrial, 2007). On the other hand, semi-formal notations are graphical representations of the system, that lack a precise or formal semantics. Despite all the criticisms about lacking precise semantics, semi-formal notations, such as the Unified Modelling Language (UML) (Rumbaugh et al., 1998), are very popular and widely used in industry. Probably, the main reason for the popularity of semi-formal representation is the use of both textual and graphical notions, where many studies have shown that such integration of graphical and textual representations can improve the comprehensibility of problems (Mayer et al., 1996; Razali et al., 2007).

The thesis builds on the Event-B formalism (Abrial, 2010). Event-B is a formal method based on set theory and first order logic. Refinement and decomposition are important concepts in Event-B, for solving the complexity of formal modelling (Abrial and Hallerstede, 2007). Refinement is the gradual building of a model, where details of the model are added gradually at different refinements levels. Decomposition is dividing the model into sub-models, where each can be refined independently. Writing Event-B models and verifying them is supported by a tool called, Rodin (Abrial et al., 2005, 2010).

The integration of formal methods with semi-formal representations can be the solution to the complexity of formal modelling, on one hand; and the lack of formality in semi-formal methods, on the other hand. Many researchers have studied such integration with a variety of formalisms. Of direct relevance to our work is UML-B, which is a UML-like representation based on Event-B (Snook and Butler, 2008; Said et al., 2009).

The Event Refinement Structures (ERS) approach, introduced by Butler (Butler, 2009a) is a hierarchical graphical representation that aims at structuring Event-B refinements and explicitly describe the ordering of events. The control flow of events is done implicitly in Event-B using guards, which are event enabling conditions. Semantics is given to an ERS diagram by transforming it into an Event-B model using some transformation rules (Salehi Fathabadi et al., 2012).

1.1 Aims and Contributions

During recent years workflow modelling with formal semantics has been a subject of interest in both industry and academia, resulting in a diversity of workflow modelling approaches. Nowadays, BPMN (Chinosi and Trombetta, 2012), for Business Process Model and Notation, is adopted as a de-facto language for business process modelling. UML (Rumbaugh et al., 1998) diagrams such as activity and state-machine diagrams have also been used widely for workflow modelling. However, both BPMN and UML have been criticised for their lack of formal semantics. This resulted in many attempts to give these modelling languages precise semantics by integrating them with different formal methods, such as the work in Börger and Sörensen (2011); Storrle (2004).

Petri Nets (Murata, 1989), is a formal method which has also been used for workflow modelling due to its graphical nature, its state-based approach, in addition to the abundance of its analysis techniques (van der Aalst, 1998). YAWL (van der Aalst and ter Hofstede, 2005), for Yet Another Workflow Language, inspired by Petri Nets, was introduced to address the Petri Nets limitations in workflow modelling, which are limitations in multiple instances modelling, complex synchronisations and cancellation patterns. However, YAWL is still criticised for many reasons such as its complex formal semantics (Börger, 2012).

All the previously mentioned workflow modelling approaches have limitations in some aspects. When modelling a complex workflow, we are looking for an approach that supports the following criteria:

1. A range of control structures covering sequencing, choice and parallelism with precise semantics.
2. Flexible control flow that supports dynamic changes in the degree of concurrency.
3. Data handling, due to the influence data can have on the course of execution of activities.
4. Exception handling mechanisms.
5. Incremental modelling and verification to reduce the modelling complexity and ensure the consistency of the workflow models.
6. Graphical representation to facilitate the understanding of the workflow models and for easier communication.

In this thesis, we show how ERS, an approach originally designed for decomposing the atomicity of events in further refinements, can also be used at a high level to model workflows and help overcome some of the challenges of workflow modelling. Our workflow modelling approach uses ERS to model control flow, while data handling is done using Event-B.

Why Event-B? Event-B (Abrial, 2010), is a formal method with a formal modelling language and tool support for formal verification. Using Event-B supports the incremental modelling of workflows through refinement, and also supports the verification of invariants and refinement correctness. In addition, various plug-ins extend the Event-B tool, Rodin (Abrial et al., 2006), with several validation and verification techniques. The language of Event-B, which is based on set theory and first order logic, provides comprehensive support for data handling.

Why ERS? An Event-B model consists of a collection of guarded atomic actions (i.e. events). In Event-B, event execution is interleaved and there is no explicit control flow. ERS (Fathabadi et al., 2014) augments Event-B with explicit control flow structures, in addition to graphical representation of control flow. The ERS control flow combinators support sequencing, iteration, choice, synchronisation and concurrency.

In order to address our workflow modelling criteria, we introduce two types of extensions to the original ERS approach:

- **First:** Extensions related to dynamic workflow modelling, where we provide support to data-dependent workflows in which case, data changes are possible. The extensions include introducing a new combinator called the *par-replicator* that support replication of behaviour with the possibility of providing new data values while executing. Extending the parameter definition to allow parameter ranges to be defined by expressions on the state, which replaces the original ERS parameter which only uses static sets. We also generalise the *loop*, which allows zero or more executions of an activity, to allow non-deterministic executions of events.
- **Second:** The exception handling extension, where we introduce two new combinators (*interrupt* and *retry*). These two new combinators provide the flexibility to allow one flow to interrupt and block the execution of another flow, while *retry* provides an additional functionality that permits handling the exception by executing the expected normal flow again.

In order to validate our workflow modelling approach and ERS extensions, we apply them to two complex workflows, the fire dispatch workflow and the travel agency booking system. The case studies require treatment of exceptions. To an extent ERS already deals with exceptional behaviour by using *xor* (a mutual exclusive constructor) or by using separate diagrams. However, we want something more general like the ability to allow one flow to interrupt another and represent changes to on-going workflows such as:

- Ability to change the action plan and respond to those changes after the action plan starts execution.
- Ability to model one workflow instance impacting another workflow instance, e.g., re-allocation of resources from a low priority incident to a high priority incident.
- Ability to interrupt the execution of the workflow due to an exception, and how to handle such exceptions.

In our case studies, the fire dispatch system and the travel agency booking system, we show how ERS is used to model the control flow and how we use Event-B to handle data. We also illustrate how we can combine ERS with other graphical approaches like state machines.

The ERS approach gets its semantics by translating the visual representation of the model to the Event-B language. For this reason, we present formally the translation rules of the new ERS extensions to Event-B. Some of these translation rules are already integrated to the ERS plug-in.

1.2 Research and Modelling Methodology

Our research focuses at facilitating the Event-B modelling and the methodology followed in developing our approach can be summarised by the following activities which are repeated in an iterative manner:

1. Model case study using Event-B and original ERS.
2. Identify limitations.
3. Propose extensions based on limitations and literature review.
4. Revisit case study by applying extensions.
5. Extend ERS language and include extensions to ERS tool.
6. Validate extensions by applying them to new case studies.

In modelling the case studies we adopt the following modelling methodology:

1. Represent the control-flow requirements using ERS diagrams and analyse the Event-B refinement levels resulting from the ERS diagrams to come up with a final version of an ERS representation of the case study. This stage is merely a diagrammatic representation of the model and does **not** involve the application of the ERS tool to generate Event-B models.

2. During the analysis stage of the ERS diagrams, make modelling decisions about where (i.e., at which refinement level) to introduce some data related requirements and what are the possible ERS combinators that can be applied.

3. Using the ERS tool, add the ERS diagram and generate the Event-B model at each refinement level of the ERS representation of the model.

4. At each refinement level, after generating the Event-B model from the ERS diagram, add manually using the Event-B editor any required data elements that are not part of the ERS control flow, before moving to the next refinement level.

5. After each Event-B refinement, verify any proof obligation that is not automatically discharged by the Rodin Provers. This can sometimes lead to some changes to the Event-B models in order to discharge the Event-B proof obligations. However, the Event-B parts that are generated by the ERS tool cannot be modified, except the events can allow the addition of new guards and actions.

6. Finally, we validate the Event-B models using *ProB* (Leuschel and Butler, 2003), which is an animator and model checker for the B-Method and it also supports Event-B. In modelling the case studies, we used *ProB* to check for any inconsistencies in the models, which may also result in some modelling changes.

1.3 Thesis Organisation

This thesis is divided into the following chapters:

Chapter 2 gives an overview of some of the popular specification techniques and graphical approaches for workflow modelling.

Chapter 3 provides the background information needed to understand the work done in this thesis, which includes formal methods, Event-B, and the ERS approach.

The following chapters represent the main contributions of this thesis:

Chapter 4 presents the dynamic extensions we suggest to enhance the ERS approach for workflow modelling. These extensions include adding a new pattern called the *par-replicator* to the ERS combinators, generalising the *loop* combinator to include non-determinism in execution and finally extending the parameter definition to allow the use of variables. We also provide a concise tabular summary of the enabling and disabling conditions related to the ERS combinators.

Chapter 5 introduces the fire dispatch case study and how we applied our workflow approach to formally model the case study. In this case study, we apply our dynamic extensions to ERS, where we highlight some limitations in ERS, which we address in Chapter 6.

Chapter 6 addresses the ERS limitations in exception handling and introduces two new combinators to address these limitations, which are also applied to the fire dispatch case study, introduced in Chapter 5.

In Chapter 7 we apply our workkflow modelling to the on-line booking system of a travel agency, where we also apply some of our ERS extensions.

In Chapter 8, we formalise the extensions introduced in chapters 4 and 6. We use the formal syntax of "Augmented Backus–Naur Form" (ABNF) in the form of translation rules mapping the ERS elements to Event-B elements. We also explain how these new translations are added to the improved ERS plug-in.

Finally, Chapter 9 evaluates our workflow approach against published workflow criteria and presents our conclusions and plans for future work. We also compare ERS with other workflow modelling approaches and other techniques integrating the formal methods with semi-formal representations.

Chapter 2

Workflow Modelling Approaches and Specification Techniques

2.1 Introduction

Workflow Modelling can be associated with different concepts like requirements engineering, system specification, formal modelling, business management, and graphical representations. All these concepts aim at better understanding and analysis of the requirements in order to produce a system that conforms with the customers expectations and requirements.

In this chapter we will present some of the existing graphical workflow modelling approaches and specification techniques that are related to our work.

2.2 Jackson Structure Diagrams (JSD)

Jackson Structure Diagrams (JSD) (Jackson, 1983), are tree-like diagrams that aim at showing the ordering of an entity's actions in a hierarchical way. The diagram can describe sequencing, iteration and selection. JSD diagrams also have a special notation for null actions.

Figure 2.1 shows an example of a structure diagram. The root of the tree "*Order*" represents the entity's name and the leaves of the tree must be actions of that entity. The sequencing is read from left to right. The asterisk in "*Delay*" indicates iteration. The circle in "*Allocate*" indicates selection, where exactly one child can be selected. The dash represents a **null** or **empty** action where nothing happens. The null action must be part of a selection and it must be a leaf node.

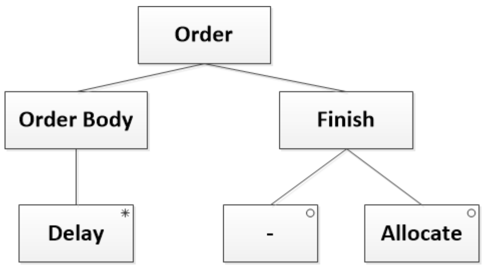


Figure 2.1: Example of Jackson Structure Diagram (Jackson, 1983)

In Jackson structure diagrams, concurrency cannot be represented in a single diagram. It can be done using more than one diagram with common actions. Similarly, a premature termination of an entity's life can not be easily represented in structure diagrams. To express this kind of behaviour, a special kind of selection is required. Figure 2.2 shows an example of premature termination as a result of natural death in a soldier's career promotion (Jackson, 1983). The normal structure of a complete career of a soldier is augmented by a choice at the beginning between a normal complete career and a prematurely terminated career. There is also an additional feature represented by an iteration over the normal events (Career Event) in order to avoid representing the several options of a prematurely terminated career.

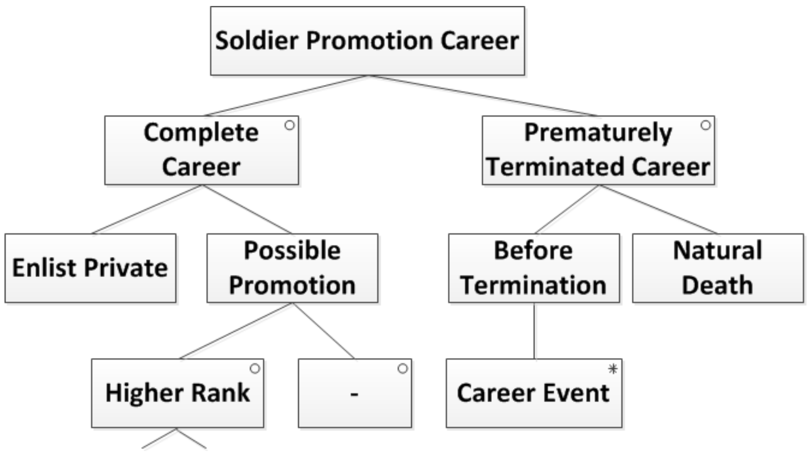


Figure 2.2: Example of Premature Termination Using Jackson Structure Diagram (Jackson, 1983)

2.3 Unified Modelling Language (UML)

Unified Modelling Language (UML) (Rumbaugh et al., 1998), is a graphical representation for modelling object oriented systems. UML consists of many graphical types representing objects, behaviours, and states. Version 2.0. of UML includes thirteen diagrams divided into two semantics groups (OMG, 2013):

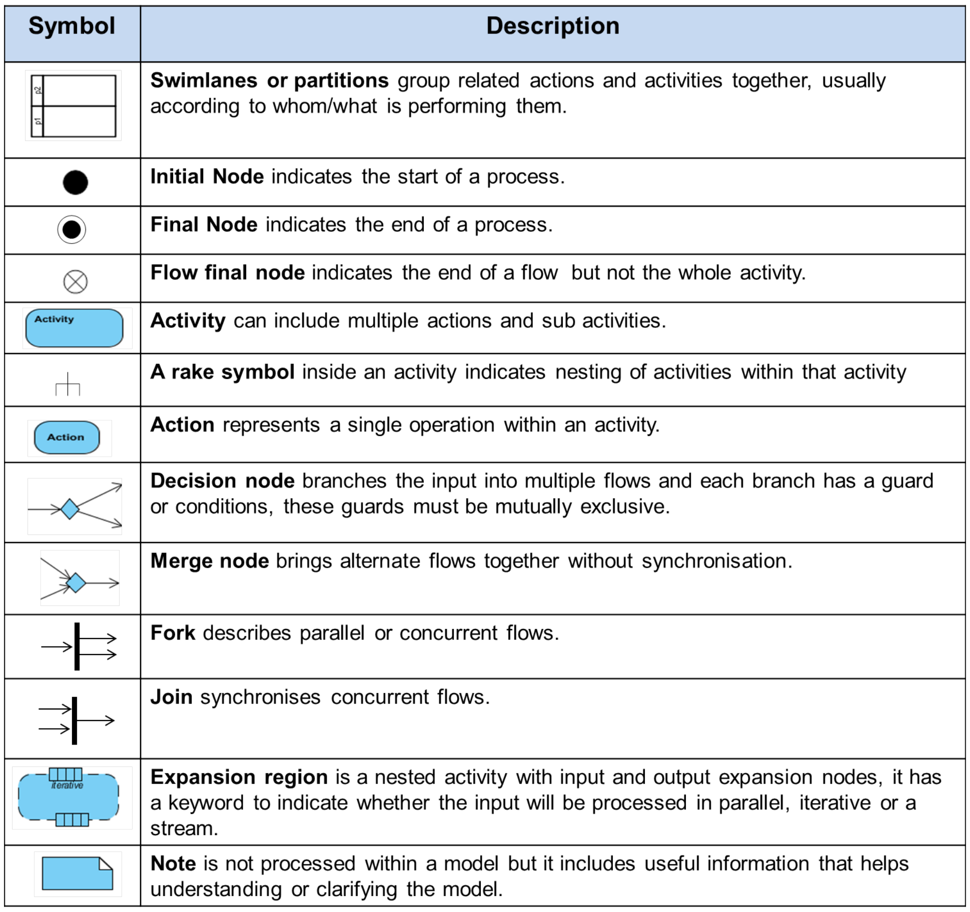
- Structure Semantics: Describes the structure of the model and this category includes six diagrams: Class Diagram, Object Diagram, Component Diagram, Composite Structure Diagram, Package Diagram, and Deployment Diagram.
- Behaviour Semantics: Describes the behaviour or the dynamics of the model and it can be divided into two sub-categories. The first is Behaviour Diagrams that include three types: Use Case Diagram, Activity Diagram, and State Machine Diagram. The second is Interaction Diagrams, which include four diagram types: Sequence Diagram, Communication Diagram, Timing Diagram, and Interaction Overview Diagram.

In the following sections, we will present two UML diagrams with behavioural semantics, activity diagrams and state machine diagrams.

2.3.1 Activity Diagrams

A UML activity diagram is one of the UML behaviour diagrams. An activity diagram is a directed graph in which the direction of the arrows suggests the flow of activities. In older versions of UML, activity diagrams were considered as a type of state machine diagram, but since the introduction of version 2.0, activity diagram semantics is widely enriched, based on token flow inspired by Petri Nets, described in more detail in Section 2.5. Despite this fact, UML activity diagrams still lack a complete formal definition and that is why many researchers have tried to define a formal semantics of UML activity diagrams in different ways (omg.org, 2012).

Figure 2.3 describes some of the notation of UML activity diagrams. Activities and actions are connected through a range of flow types including decision points, merges, forks and joins. Activities may be nested hierarchically. Figure 2.4 shows a simple example of an *order* activity using some of these notation. The start of the *order* activity is indicated through the use of the *initial node* with a customer making an order. The directed arrow indicates that the next action will be receiving the order in the *order department*. After receiving the order, a decision should be taken to whether accept it or not. If it is accepted, then it will continue to the *fork-node*, which means the execution of the actions *Send Invoice* followed by *Receive Payment*, in parallel with the execution of *Ship Order* Action. The *join-node* waits for both *Receive Payment* and *Ship Order* to finish, to proceed to *Close Order* action, and finally closing the *order* activity using the *activity final node*. In the case of rejection decision, the order will be closed immediately. The use of the *merge-node* before the *Close Order* action, means this action will execute if any of the previously described scenarios take place.



Symbol	Description
	Swimlanes or partitions group related actions and activities together, usually according to whom/what is performing them.
●	**Initial Node** indicates the start of a process.
◉	**Final Node** indicates the end of a process.
⊗	**Flow final node** indicates the end of a flow but not the whole activity.
Activity	**Activity** can include multiple actions and sub activities.
	A rake symbol inside an activity indicates nesting of activities within that activity
Action	**Action** represents a single operation within an activity.
	Decision node branches the input into multiple flows and each branch has a guard or conditions, these guards must be mutually exclusive.
	Merge node brings alternate flows together without synchronisation.
	Fork describes parallel or concurrent flows.
	Join synchronises concurrent flows.
iterative	**Expansion region** is a nested activity with input and output expansion nodes, it has a keyword to indicate whether the input will be processed in parallel, iterative or a stream.
	Note is not processed within a model but it includes useful information that helps understanding or clarifying the model.

Figure 2.3: Some Constructs of UML Activity Diagrams

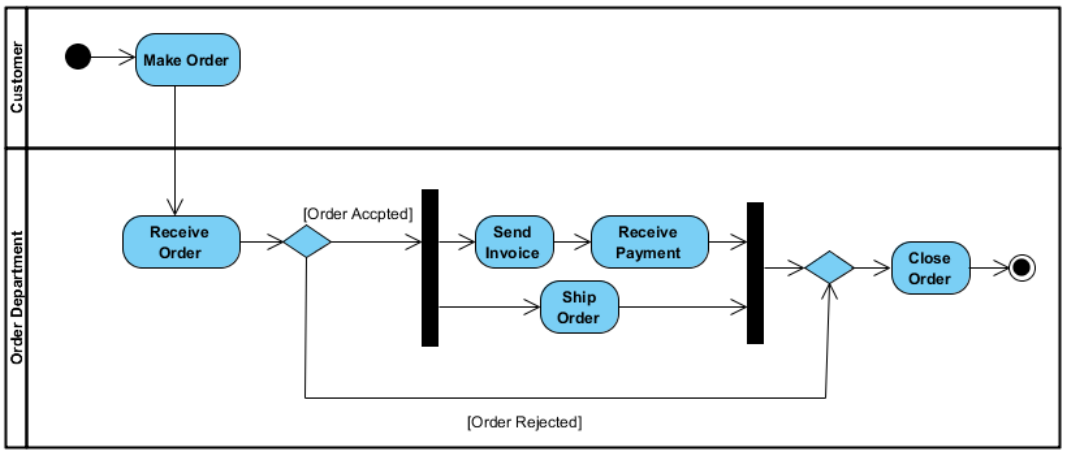


Figure 2.4: "Order" Activity: Simple Example of Activity Diagram

2.3.2 State Machines

A State machine (omg.org, 2012) is another UML behavioural diagram that describes the sequence of states an individual object goes through. States and transitions are important concepts in a state machine. A *state* models a situation in an object's life, for example a telephone in an idle state represents the situation when a telephone is not in use. A *transition* is a directed arc connecting two vertices, where a vertex can be a *state* or a *pseudostate* such as a choice or fork etc. A transition can indicate that an object in a source state may perform certain actions resulting in changing the state of the object to the target state as a result of an *event*. The source and target states of a transition might be the same, in this case it is called *self-transition*. A transition can be also associated with guards which are boolean conditions that must evaluate to true before a transition can execute.

States can be one of the following three types:

- **Simple State:** Has no internal sub-states or transitions.
- **Composite State:** can be composed of one or more regions. A *region* is delineated by a dashed line to describe the behaviour of a fragment that may execute concurrently with its orthogonal regions (The term *orthogonal* refers to regions owned by the same state or state machine). Any state within a region of a composite state is called a *sub-state*.
- **Submachine State:** is an entire state machine nested within a state.

Initial and final states are special states that respectively indicate the start and completion of a state machine. Figure 2.5 from omg.org (2012) shows an example of composite states, where "*CourseAttempt*" is a composite state with a single region and "*Studying*" is a composite state with three orthogonal regions that may execute in parallel.

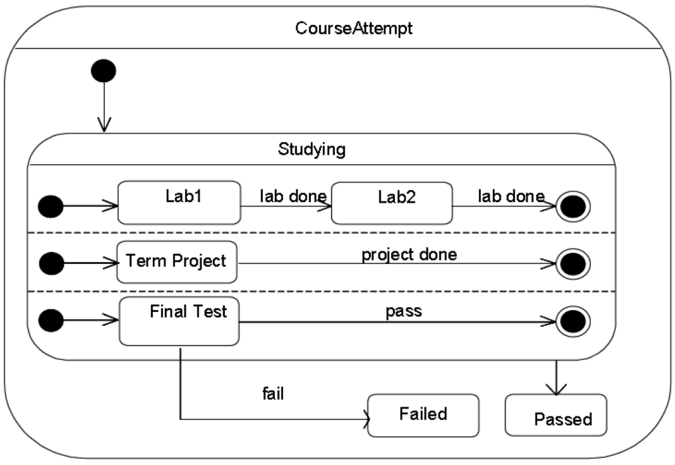


Figure 2.5: Composite State With Regions (omg.org, 2012)

2.4 Business Process Model and Notation (BPMN)

BPMN (Object Management Group, 2011), refers to "Business Process Model and Notation" or "Business Process Modeling Notation". BPMN is a graphical notation which is nowadays adopted as the de facto for business process modelling (Chinosi and Trombetta, 2012).

BPMN aims at modelling control flow rather than data flow, although it supports flow of messages and associations between activities and data. The main goal of BPMN is to provide a standard notation that can be easily understood by the different parties involved in the business process. Early versions of BPMN were inspired by UML activity diagrams, but lacked a well defined semantics. Then came BPMN 2.0 to extend the notation and provide a formal execution semantics (Object Management Group, 2011; Ouyang et al., 2009). However, BPMN is still criticised for its ambiguous and imprecise semantics (Börger, 2012).

BPMN has five categories of elements: Flow Objects, Data, Connecting Objects, Swimlanes and Artifacts. The Flow Objects category is the main category in any business process, and can be further categorised into three elements (Object Management Group, 2011):

1. **Events:** Can be a "Start", an "Intermediate" or an "End" event. An event represents something that **happens** during a process and it is represented as a circle.
2. **Activities:** Can be atomic or non-atomic. An atomic activity is called a **task** and it represents the work to be performed within the business process. An activity is represented by a rectangle with round corners. A task can be also marked by any one or two types of markers which are: Loop, Compensation and Multi-Instance. "Compensation" is an important aspect of BPMN which is concerned with undoing the steps that successfully happened and are later cancelled.
3. **Gateways:** Control the divergence or convergence of flows and are represented by diamond shapes.

Figure 2.6 from Object Management Group (2010) shows a BPMN representation of an "order" process. In this diagram the circle with the thin border represents a start event and the circle with a thick dark border represents an end event. The "Approver Order" activity, containing the activities "Approve Customer" and "Approve Product", shows an example of a non-atomic activity or a sub-process. The blank diamond shape represents an exclusive gateway or a decision gateway. The diamond with a plus sign in the middle represents a parallel gateway, the first parallel gateway is a parallel fork and the second one is a parallel join.

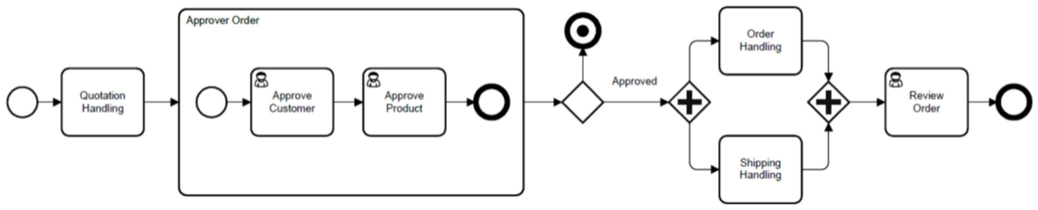


Figure 2.6: BPMN Diagram of an Order Process (Object Management Group, 2010)

2.5 Petri Nets

Petri Nets (Murata, 1989) are directed graphs used mainly for modelling concurrent and asynchronous systems. Petri Nets consist of two types of disjoint nodes called places and transitions, connected to each other by edges called arcs, which can only connect nodes of different types. Transitions denoted by solid bars or rectangles, can be interpreted as actions or events. Places are conditions for actions, and they are represented as circles. The dynamics of the system can be described using tokens, which are represented by solid dots in places. The presence of a token in a place represents the enabling condition of a transition. It is also possible to describe the behaviour of the system using mathematical models (Murata, 1989).

Petri Nets have been used for workflow modelling for three main reasons (van der Aalst, 1998; van der Aalst and ter Hofstede, 2005):

1. It has formal semantics in addition to the graphical nature.
2. It is a state-based approach, where the states of a system are explicitly modelled.
3. There are model checking techniques available for Petri Nets.

There are many extensions to Petri Nets, usually referred to as high level Petri Nets. For example, Coloured Petri Nets (Jensen, 1997), which extends the basic Petri Nets by allowing the tokens to carry data values of different types.

2.6 Yet Another Workflow Language (YAWL)

YAWL (van der Aalst and ter Hofstede, 2005) stands for "Yet Another Workflow Language", is a workflow language as its name suggests. YAWL was introduced to cover the twenty workflow patterns described in van der Aalst et al. (2003). YAWL is inspired by "Petri Nets", but extended to overcome the limitations of Petri Nets in workflow modelling. Although, YAWL is inspired by Petri Nets, it is considered as a different language with different semantics.

The three limitations in Petri Nets that YAWL tries to solve are limitations in complex synchronisation, limitations in multiple instances modelling and finally limitations in cancellation patterns.

The twenty workflow patterns that YAWL tries to cover can be categorised into six categories (van der Aalst and ter Hofstede, 2005) :

1. Basic control flow patterns
2. Advanced branching and synchronisation patterns
3. Structural patterns
4. Pattern involving multiple instances
5. State-based patterns
6. Cancellation patterns

Only one of the twenty patterns is not supported by YAWL which is "implicit termination", which falls under the structural patterns category.

Figure 2.7 from van der Aalst and ter Hofstede (2005), shows the modelling notations used by YAWL. The modelling elements of YAWL include conditions; AND, XOR, OR splits and joins, atomic and composite tasks in both single instance and multiple instances cases; in addition to "remove tokens" which can be used to model cancellation.

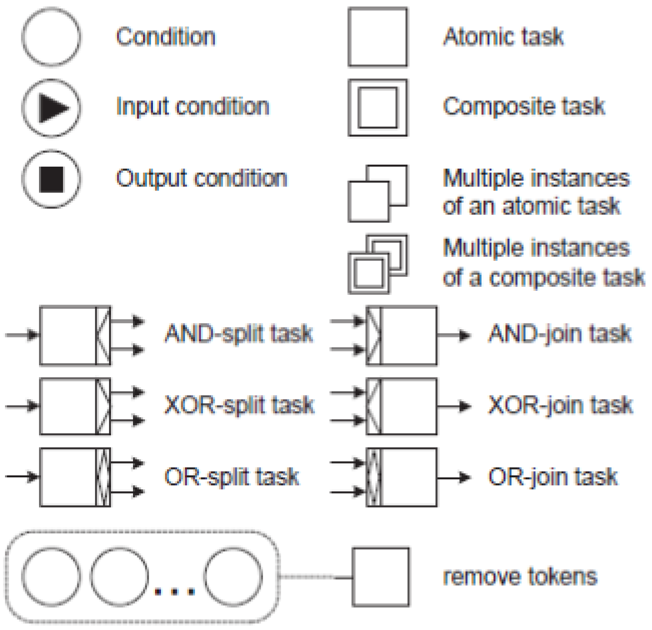


Figure 2.7: YAWL Symbols (van der Aalst and ter Hofstede, 2005)

YAWL defines four additional parameters to multiple instances tasks: lower bound, upper bound, threshold and static/dynamic parameter. Figure 2.8 from van der Aalst and ter Hofstede (2005) represents different scenarios of using multiple instances. The first case, "a", requires between 1 and 10 instances to process witness statements and does not allow the creation of new instances while being processed, since it is described as "static". Enabling the "archive" task in the first case requires the execution of all instances. In the second case, "b", new instances can be created while processing witness statements due to the dynamic parameter. Lastly, Case "c" enables the "archive" task if all instances or at least three instances, which is the threshold, have executed. All these cases can execute the multiple instances in parallel.

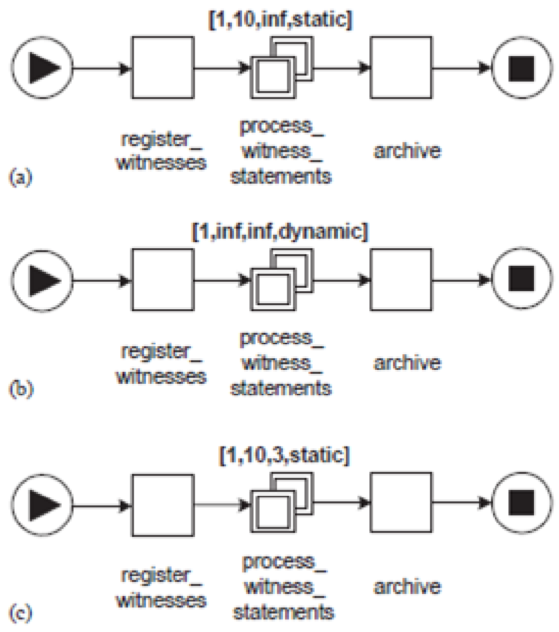


Figure 2.8: Examples of Multiple Instances Tasks Using YAWL (van der Aalst and ter Hofstede, 2005)

2.7 Formal Modelling of Workflows

The need to have precise semantics for workflows have resulted in several works formalising different workflow approaches. In this section we present some of the works that gave workflows formal semantics:

2.7.1 Formalising Control Flow Patterns Using Abstract State Machine (ASM)

ASM or Abstract State Machine (Börger, 2005), is a rule based modelling approach. In Börger (2007a), Börger uses ASM to represent the forty three workflow patterns defined in Russell et al. (2006a) by the business process modelling centre. However, Börger finds out using ASM, the forty three workflow patterns can be reduced into eight generic patterns out of which more complex patterns can be derived using parameter instantiation, composition and refinement.

The generic eight workflow patterns are described using ASM in Börger (2007b). The eight basic patterns are divided into two categories, sequential and parallel, each containing four patterns. The sequential category includes the following four patterns:

1. **Sequence Patterns:** Deals with sequencing of multiple flows, where an activity is enabled after the completion of another activity .
2. **Iteration Patterns:** Where an activity is repeatedly executed.
3. **Begin/Termination Patterns:** Where a flow is terminated depending on a specified stopping criteria. Börger also relates the termination pattern with the cancellation patterns.
4. **Selection Patterns:** Includes a choice among multiple flows.

The parallel category includes:

1. **Parallel Split Patterns:** Where one flow splits into multiple independent flows.
2. **Merge Patterns:** Where multiple flows converges into a single thread of control.
3. **Interleaving Patterns:** Where activities are executed one after another in an arbitrary order.
4. **Trigger Patterns:** The ability of an activity to be triggered by another part of the process or by an external environment.

In the eight different workflow patterns, Börger does not differentiate between single and multiple instances behaviour or between static and dynamic behaviour, i.e. whether the number of branches or instances is known or unknown until runtime, are left as a matter of how the parameter is declared.

2.7.2 Other Work on Formalising Workflows

Wang et al. (2010) describe a guarded workflow language based on the CSP (Communicating Sequential Processes) (Hoare et al., 1985) Notation. The actions in this formal language are guarded transactions which in turn will help in determining the pre-conditions for a successful completion of a workflow or a combination of workflows.

Butler et al. (2005a) use the StAC (Structured Activity Compensation) language (Butler and Ferreira, 2000), which is also based on process algebras such as CSP (Hoare et al., 1985). Similar to CSP, StAC supports operators for sequencing, concurrency, choice and hiding, but also provides support for exception handling using compensation. In Butler et al. (2005a), they use StAC for activity orchestration and the B formal method (Abrial, 1996) for data handling. They also map BPEL (OASIS, 2007), which is the Business Process Execution Language for Web Services to StAC, thus giving BPEL formal semantics.

Other works have also given BPMN formal semantics such as Bryans and Wei (2010) and Börger and Sörensen (2011). The former translated BPMN to the Event-B formal language, and the latter translated BPMN to ASM. Others have also given activity diagrams formal semantics. As a natural sequence of the activity diagrams semantics which is based on Petri Nets, Syriani and Ergin (2012) translated activity diagrams to Petri Nets, and in Börger et al. (2000), activity diagrams are given formal semantics using ASM.

In addition to formalising workflow modelling approaches, others have used a combination of state-based and event-based formal methods, such as CSP and B in Butler (2000) and Circus (Woodcock and Cavalcanti, 2002) which is a combination of Z (Spivey, 1992) and CSP. In these approaches, the event-based CSP is used to formally represent control flow.

2.8 Conclusion

In this chapter, we have presented some of the main specification and workflow modelling approaches. All approaches cover the basic control flow constructs, but their strengths vary when it comes to complex cases such as multiple instances and cancellations.

Jackson Structure Diagrams (JSD) can simplify complex problems by applying a hierarchical structure. UML diagrams and BPMN are popular modelling techniques, but are still criticised for their imprecise semantics. Advocates of Petri Nets praise its graphical nature as well as its formal semantics. Then, YAWL comes to address the limitations in Petri Nets when it comes to modelling multiple instances, complex synchronisation and cancellation. However, YAWL is still criticised for its complex formal semantics. That is why, in this thesis we focus on a structured workflow modelling of multiple instances using formal semantics.

Chapter 3

Formal Modelling Approaches

3.1 Introduction

In this chapter we introduce some key concepts in formal methods needed to understand our work. First we start with introducing formal methods and an overview of some formal modelling languages. Then we introduce Event-B, a formal method for system modelling. We describe the structure, refinement, proof obligation rules and finally the tool support of the Event-B formal method and one of its important plug-ins UML-B. We also introduce the Event Refinement Structures approach (ERS), a diagrammatic representation that we use to model workflows in Event-B. We present the different patterns of ERS, from their diagrammatic notation to their Event-B semantics.

3.2 Formal Methods

Formal methods are mathematical techniques used for the systematic specification, verification and development of both hardware and software systems (Hall, 1990; Wing, 1990; Bowen and Hinchey, 1995). Similar to blueprints used in other engineering disciplines, formal models can be used to formally reason about the system during its construction (Abrial, 2007). This will mainly help in finding some errors during the construction of the system, instead of delaying it to the testing phase.

Nevertheless, many people are still questioning the practicality of formal methods. Opponents of formal methods usually claim that they are difficult and only mathematicians can use them. However, according to Abrial (2007), some of the real difficulties in formal methods lie in the lack of good requirement documents, and the common practice of rushing towards coding without spending enough time to reason about the system. On the other hand, semi-formal notations which are graphical representations of the system are criticised for their lack of formal semantics. Therefore, to tackle the issues of mathematical difficulty and lack of formal semantics, in this thesis we use a combination of a graphical representation and a formal method.

3.2.1 Overview of Some Formal Methods

In this section, we will discuss briefly some formal methods that have been used in industry, in particular VDM, Z, the B-Method and ASM. Later in Section 3.3, we will describe in more detail Event-B, which is the formal method, we use in modelling our case studies.

- **VDM** or Vienna Development Method (Jones, 1990), was developed in an industrial environment at the IBM Laboratory, Vienna. A VDM specification describes the behaviour of the system using operations. These operations are defined using pre- and post-conditions, which describe the state variables of the system before and after the operation (Lindsay et al., 1991).

  One of the key features of *VDM* is *data reification*, this involves the transition from abstract data types to concrete data types, or in other words data types that are more complex and includes more detail. This transformation requires several steps until the data types can be expressed in the implementation language. In order to relate the abstract data types with the corresponding concrete data types, a function called a "*retrieve function*" is required, resulting in some proofs to justify the correctness of data reification (Jones, 1990).

  Another key feature is *operation decomposition*, which is required after the data types are refined into the level of an implementation language. This results in the decomposition of operations into combinators that combine the different types of operations, e.g., while loop. Operation decomposition also entails more proofs to justify its correctness (Jones, 1990).

- **Z** pronounced as "*Zed*", similar to *VDM*, it is based on set theory and first order predicate logic. However, *Z* includes *schema notation* (Figure 3.1), which improves the structuring and the readability of the specifications, but unlike *VDM*, *Z* does not explicitly separate pre- and post-conditions of the operations (Bowen, 2003).

  BirthdayBook

  $known : \mathbb{P}\, NAME$

  $birthday : NAME \pfun DATE$

  $known = \operatorname{dom} birthday$

  Figure 3.1: Example of Z Schema Notation (Bowen, 2003)

  Figure 3.1 shows an example of a schema notation, where the top part of the box describes the variables and their associated constraints defining their types, while the bottom half describes additional constraints between these variables.

Z is more considered as a formal specification notation rather than a formal method, and in industry *Z* is mainly used for specification rather than proof (Bowen, 2003). When we say formal specification, we mean a mathematical representation of the customer's requirements in order to remove ambiguities, whereas a formal method includes both a formal specification and a framework to develop a specification towards an implementation (Wing, 1990; Jones, 1990).

- **B-Method**, or *Classical B* was first introduced by Abrial (Abrial, 1996) for software development. Similar to *Z* and *VDM*, the B-Method is based on set theory and first order predicate logic. From a practical point of view, the main difference is that B has abstract assignments rather than postconditions. But unlike VDM, which uses three-valued logic to describe undefinedness, the B-Method uses the classical two-valued logic (True and False) (Cansell and Mery, 2003).

  The structure of the B-Method is based on *machines* (Figure 3.2) and *refinement* (Cansell and Mery, 2003; Butler et al., 2005c), refinement will be explained in more detail in Section 3.3.2.

```
machine
  m
sets
  s
constants
  c
properties
  P(s, c)
variables
  x
invariant
  I(x)
assertions
  A(x)
initialisation
  <substitution>
operations
  <list of operations>
end
```

Figure 3.2: B-Method Abstract Machine (Cansell and Mery, 2003)

Figure 3.2, describes an abstract machine "*m*". The "*sets*" and "*constants*" clauses define the sets and the constants of the system respectively. The "*properties*" clause contains definitions of the constants. The "*variables*" describe the state of the system and the "*invariants*" define the properties of these variables, where "*assertions*" describe the safety properties of the variables. The "*initialision*" gives the variables an initial value to describe the initial state of the system. Finally, "operations" can change the state of the variables but need to always preserve the invariants.

- **ASM** or Abstract State Machine (Börger, 2005), is a formal method that extends the Finite State Machine (FSM) with an enriched notion of state, where a state is an arbitrary data structure. The ASM method starts with an abstract model called *ground model*, and then more detail is added through stepwise refinement.

  ASM has a simple syntax in the form of pseudo code, and is defined as a set of rules called ASM rules with the general form (Börger, 2003, 2005):

  $$\textbf{if } \textit{Condition} \textbf{ then } \textit{Updates}$$

  The *condition*, also called guard, evaluates to true or false, and the *Updates* is a finite set of assignment statements. Simultaneous and non-deterministic execution of ASM rules can also be represented using the following notation, where *rule(x)* should satisfy the condition φ:

Simultaneous Execution:	**forall** x **with** φ $rule(x)$
Non-Deterministic Execution:	**choose** x **with** φ $rule(x)$

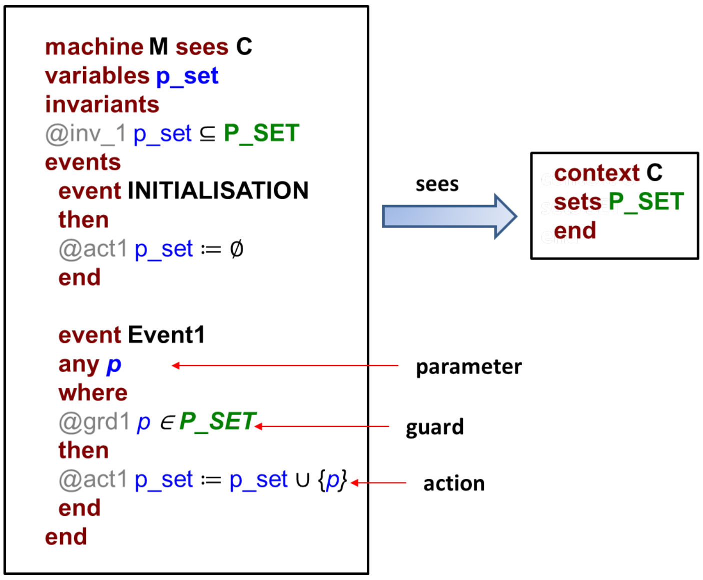
3.3 Event-B

Event-B is a formal method for modelling systems that may include hardware and mechanical components as well as software (Abrial, 2010). Event-B is mainly influenced by the B-Method (Abrial, 1996) and Action Systems (Back, 1990). The Event-B language is based on set theory and first order logic. An Event-B model starts with an abstract machine which is linked to a series of concrete machines through refinement. Correctness and consistency of models, are proved by means of mathematical proofs (Abrial et al., 2010).

3.3.1 Structure

The Event-B model, unlike the B-Method, is divided into a static part, the "Context" and a dynamic part, the "Machine". The context consists of the set types, constants and their properties as axioms. A machine consists of variables, invariants describing the constraints applied on the variables, and events that show how the variables' states change provided some guard conditions hold. In addition, theorems can be defined in both context and machine (Metayer et al., 2005; Abrial, 2010).

Figure 3.3 presents a simple Event-B model showing both the static and the dynamic parts of the model. Machine *M*, on the left, consists of two events: the initialisation event and "*Event1*". The variable *p_set* is defined by the invariant (*inv_1*) as a subset of the set *P_SET*. The set *P_SET* is defined in the context *C*, where the "*sees*" clause gives machine *M*, access to the context *C*. The event *Event1* updates the variable *p_set*, by adding new values in the action (*act1*), but first checks the type of the newly added values by the guard (*grd1*).



```
machine M sees C
variables p_set
invariants
@inv_1 p_set ⊆ P_SET
events
  event INITIALISATION
  then
  @act1 p_set ≔ ∅
  end

  event Event1
  any p
  where
  @grd1 p ∈ P_SET
  then
  @act1 p_set ≔ p_set ∪ {p}
  end
end
```

```
context C
sets P_SET
end
```

Figure 3.3: Simple Event-B Model

3.3.2 Refinement

Refinement is a key concept in the Event-B modelling, where instead of building a single model, we start with an abstract one, focusing on the main functionalities of the system. Then during refinement levels, details of the model and/or new functionalities are added gradually. The refinement process aids understanding the model and proving its correctness and consistency (Abrial and Hallerstede, 2007).

Refinement can be categorised into two types, horizontal refinement and vertical refinement (Damchoom and Butler, 2009). Horizontal refinement includes the introduction of new features or properties to the model without changing the previously existing ones, resulting in the addition of new events and variables to the model (Damchoom et al., 2008; Damchoom and Butler, 2009). On the other hand, vertical refinement aims at changing the structure of the abstract model using data refinement (Damchoom and Butler, 2009). Data refinement is used to change the state of the variable in a way that makes its implementation using a programming language possible (Abrial and Hallerstede, 2007).

In Event-B, refinement is described using the "*refines*" clause in a machine. In case of refinement, a special type of invariant, called "*gluing invariant*", can be used. Gluing invariants describe the relation between the abstract and the concrete, or refining, variables. An abstract event can be refined by one or more concrete events and new events can be added. In Event-B the variables of the abstract machine can be inherited by its refining machine which can also add new variables or even drop some of its abstract machine variables. Dropping variables in refinement requires gluing invariants. Similarly, where parameters are dropped in the concrete machine, "*Witnesses*" for them should be added to the refining events, relating the refining parameters to their corresponding abstract parameters in the same way gluing invariants relate abstract and concrete variables, thus making the refinement proof obligations easier to discharge (Abrial, 2010; Abrial and Hallerstede, 2007).

The new events added during refinement, refine *skip* which is an implicit event that does nothing at the abstract level. This allows a refining event to modify variables introduced in the refinement as if they make no change at the abstract level. An event's status can also be described using the "*status*" clause as ordinary, convergent or anticipated. A convergent event is an event which cannot take control forever, while an anticipated event is typically refined into a convergent event. In case of convergent or anticipated events, a variant is needed. A variant must be a natural number or a finite set, which a convergent event must decrease and an anticipated event must not increase. An ordinary event is the most commonly used, it is an event that is neither convergent nor anticipated (Abrial, 2010).

When applying machine refinement, a machine can only refine one other machine and so on, where one can construct a chain of refinements. Machine refinement results in additional proof obligations, including guard strengthening, invariant preservation, action simulation and witness feasibility (Hallerstede et al., 2010).

Unlike a machine, a context can extend more than one other context. This extension can include adding new sets, constants defined by axioms, but does not result in associated proof obligations (Abrial and Hallerstede, 2007). Figure 3.4 adopted from Butler (2013) describes the different relationships between machines and contexts. The dashed line indicates an implicit "*sees*" between Machine *M2* and Context *C1*, since Machine *M2* sees Context *C2* which extends Context *C1*.

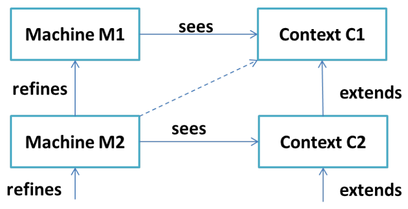


Figure 3.4: Machine and Context Relationships

3.3.3 Proof Obligation Rules

Proof Obligations are used in proving the consistency properties of the Event-B model and the consistency between refinement levels. These proof obligations are automatically generated by the Rodin tool, and can be proved both automatically and interactively (Abrial et al., 2006, 2010). Proof Obligations are concerned with model consistency, refinement, theorems, convergence, and well-definedness of the model. In this thesis we make a lot of use of Event-B and proof, so a brief description of the main proof obligation rules is described in Table 3.1. Regarding the variant proof obligations, numeric variant (NAT) and finite set variant (FIN), only one of these proof obligations can be generated for a given variant, since a variant can be either numeric or a finite set.

Proof Obligation Rule	**Description**
Invariant Preservation **(INV)**	Ensures that invariants are maintained by all events
Feasibility **(FIS)**	Ensures that a non-deterministic action is feasible when its guards are True
Guard Strengthening **(GRD)**	Ensures that the guard of the concrete event is stronger than the guard of its corresponding abstract event
Simulation **(SIM)**	Ensures that all actions in abstract events are properly simulated in the corresponding concrete events
Numeric Variant **(NAT)**	Ensures that the numeric Variant of a convergent or anticipated event is a natural number under their guards
Finite Set Variant **(FIN)**	Ensures that the Variant of a convergent or anticipated event is a finite set under their guards
Variant **(Var)**	Ensures that the convergent event decreased the numeric or finite set variant
Non-Deterministic Witness **(WFIS)**	Ensures the existence of all witnesses proposed in a concrete event
Theorem **(THM)**	Ensures the provability of theorems described in contexts and machines
Well-Definedness **(WD)**	Ensures that all elements of the model are well defined e.g. we cannot divide by zero
Preserved Variable **(EQL)**	Ensures that the value of a variable at the abstract event does not change, if assigned a value at a concrete new event

Table 3.1: Event-B Proof Obligation Rules (Abrial, 2010)

3.3.4 Rodin: An Event-B Modelling Tool

Rodin is a tool for Event-B that generates proof obligations and automatically discharge the trivial ones (Butler and Hallerstede, 2007; Abrial et al., 2010).

Rodin is an open tool, implemented on top of the Eclipse Framework (Butler and Hallerstede, 2007). The openness of this tool allows other parties to contribute to this tool and extend it, through the use of plug-ins (Abrial et al., 2006; Butler and Hallerstede, 2007). UML-B is one of these important plug-ins that integrates semi-formal modelling with Event-B. This plug-in is explained in more detail in Section 3.3.5.

The graphical user interface of Rodin, allows the user to easily switch the perspective between modelling and proving (Abrial et al., 2010; Butler and Hallerstede, 2007).

The Rodin tool consists of three major components(Abrial et al., 2006, 2010):

- **Static Checker:** Performs syntax and type checking of the contexts and machines.
- **Proof Obligation Generator:** Generates proof obligations automatically. These proof obligations are described in more details in Section 3.3.3.
- **Proof Obligation Manager:** Keeps track of proof obligations and associated proofs. It also discharges proof obligations automatically, and allows the user to interactively prove un-discharged proofs.

3.3.5 UML-B

UML-B combines both graphical representation and formal modelling using Event-B. UML-B is a UML-like formal modelling language based on Event-B (Snook and Butler, 2008; Said et al., 2009). UML-B is first introduced by Snook, as a UML profile that is translated to classical B, or B-Method explained in Section 3.2.1, with the help of U2B tool (Snook and Butler, 2006; Hallerstede and Snook, 2011). UML-B is supported by the UML-B tool, which is an extension of the Rodin tool, generating Event-B models from the corresponding UML-B diagrams. Event-B translation of the state machines in UML-B can be done as disjoint sets or state functions representations (Said et al., 2009).

UML-B includes the following four diagram types (Snook and Butler, 2008):

- **Package Diagrams:** Based on UML package diagrams, these describe the relationships between contexts and machines.
- **Context Diagrams:** Based on UML class diagrams, these describe the static part or the context of the model.
- **Class Diagrams:** Classes are described as variables subset of the types or class types introduced in the context.

- **State Machine Diagrams:** Are attached to the classes of the class diagram and introduce a variable that partitions the behaviour of the class. The transitions in the state machine are translated into events in the Event-B machine that describe the change of states.

Refinement in UML-B is based on Event-B refinement, it can be either a class refinement or a state machine refinement. Both of these two types of refinement can be described as node refinement (Hallerstede and Snook, 2011). In a class refinement, classes and attributes can be inherited, added, or dropped variables as in Event-B refinement. Events are refined in the same way as Event-B refinement and new events can be introduced. State machine refinement can refine states or state machines. This can be done through state elaboration or transition elaboration. In a state elaboration a super state can be refined into sub-states resulting in the addition of new invariants. In transition elaboration, a transition is replaced by several transitions similar to event refinement in Event-B, where an event can be refined by one or more events and new events can be added refining *skip* (Said et al., 2009).

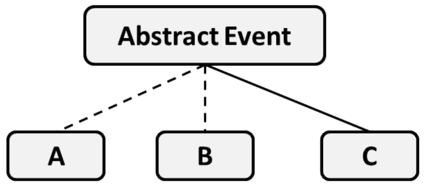
A new refinement method for UML-B is introduced in Said et al. (2013), called "*Context Diagram Extension*". In this method, the class types are extended with new features such as associations, axioms etc. without changing and repeating the old features.

3.4 Event Refinement Structures Approach (ERS)

The Event Refinement Structures (ERS) approach was first introduced by Butler (Butler, 2009a). ERS provides a graphical representation based on Jackson Structure Diagrams (JSD) (Jackson, 1983). JSD provides a tree-like graphical representation of structured programs with sequential behaviour indicated by left to right placement of nodes. ERS diagrams help in understanding the relations between the abstract and concrete levels of the Event-B models, and also help to explicitly describe the control flow of events, which is implicitly described through guarded events in an Event-B model (Butler, 2009a).

Figures 3.5 and 3.6 show examples of ERS diagrams, where each node represents an event. The ERS diagram indicates that the atomicity of the abstract event, appearing in the root node, is decomposed into some concrete sub-events, appearing in the leaf nodes. Similar to Jackson's structure diagrams, the leaf events are read from left to right, so A is executed first, followed by B and finally C.

The dashed line represents an event that refines *skip*, or in other words a newly added event. The solid line represents a refining event, where exactly one event can refine an abstract event. A combinator can also be applied to the events and it can be one of the following: *and, or, xor, loop, all, some, one.* A combinator can be connected to one or more events depending on the combinator type. We can also add parameters to the

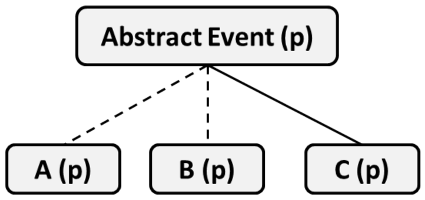


```
Invariants
  @inv_A_type A ∈ BOOL
  @inv_B_seq B = TRUE⇒ A = TRUE
```

```
event B
   where
     @grd_seq A = TRUE
     @grd_self B = FALSE
   then
     @act B  ≔ TRUE
 end
```

Figure 3.5: A Basic Structure for an ERS Diagram Using Single Instance (SI) Modelling and its Event-B Semantics

events to indicate multiple instances modelling (Figure 3.6). Multiple instances modelling can be used to model concurrency, where leaf events of different instances of a flow may interleave. These parameters need to be inherited by subsequent children (Butler, 2009a; Salehi Fathabadi et al., 2012).



```
invariants
  @inv_A_type A ⊆ TYPE(p)
 @inv_B_seq B ⊆ A
```

```
event B
   any p
   where
     @grd_seq p ∈ A
     @grd_self p ∉ B
   then
     @act B  ≔ B  ∪ {p}
 end
```

Figure 3.6: A Basic Structure for an ERS Diagram Using Multiple Instances (MI) Modelling and its Event-B Semantics

Semantics is given to the ERS diagram by transforming it into an Event-B model. Salehi Fathabadi et al. (2012) describe some of the rules of the ERS diagram transformation into Event-B modelling language. They apply ERS, or what is previously referred to as the atomicity decomposition approach, in two case studies (Salehi Fathabadi and Butler, 2010; Salehi Fathabadi et al., 2011) to evaluate its role in clarifying the refinement and the explicit ordering of events in the Event-B modelling language.

Figures 3.5 and 3.6 also show the corresponding Event-B translation of leaf B, in the single instance and the multiple instances case respectively. In both cases, leaf B is transformed to an event in Event-B with a corresponding control variable (B), that is used to explicitly describe the ordering of the leaf event (B). In the single instance case (Figure 3.5) the control variables are represented as boolean flags in Event-B, with the interpretation that the flag is *false* before the occurrence of the event and is set to *true* after the event occurs, as shown by the disabling guard (grd_self) and the action (act) of the leaf event B. The sequencing guard ($A = TRUE$) checks that event A has executed before B is allowed to execute, which conforms with the *left-to-right* sequencing of the ERS diagram. The ordering of B is also specified by the sequencing invariant (inv_B_seq).

In the multiple instances case (Figure 3.6), the control variables are defined as subsets of the instance parameter set ($TYPE(p)$). In this case, the first leaf of the ERS diagram (A) is defined as a subset set of $TYPE(p)$ (inv_A_type), and B as a subset set of A (inv_B_seq), since the execution of B must follow the execution of A. In the corresponding event of leaf B, the sequencing guard checks that A has occurred for the parameter value p, while the second guard (grd_self) prevents B from executing again for the same instance value.

3.4.1 Event Refinement Structures (ERS) Combinators

In this section, we describe the existing combinators of the ERS approach. For more details, refer to Salehi Fathabadi (2012). The ERS approach defines four groups of combinators:

- Sequencing Pattern: Defined by the left to right arrangement of events.
- Logical Combinators Pattern: and, or, xor
- Loop Pattern
- Replicator Patterns: all, some, one

In ERS the combinators are represented within an oval applied to one or more events, where the events are represented within a rectangle. It is also worth noting that, when the lines, whether solid or dashed, originate from a rectangle, this indicates a sequential behaviour; and when they originate from an oval shape, this indicates a non-sequential behaviour, as shown by Figure 3.7.

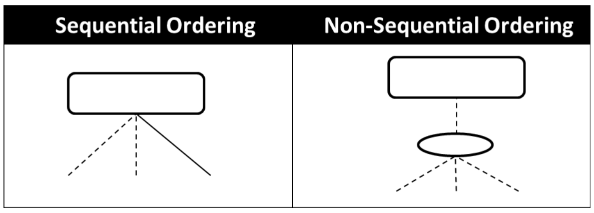


Figure 3.7: ERS Representation of Sequential and Non-Sequential Behaviour

The sequencing pattern is described earlier in Section 3.4 (Figures 3.5 and 3.6). ERS also allows the choice between strong sequencing and weak sequencing within different refinement levels. Strong sequencing means sequencing rules are applied to each sub-event, whereas weak sequencing only applies the sequencing rules to the solid line sub-event with sub-events from the previous refinement level. Weak sequencing cannot be applied within the same refinement level.

To make things clearer, consider the example in Figure 3.8, where *Event_2* is refined into the sequence of events *Event_a*, *Event_b* and *Event_c*. If weak sequencing interpretation is applied to the refinement of *Event_2*, then there is no ordering constraints between *Event_1* and *Event_a* and also no ordering constraints between *Event_c* and *Event_3*. However, within the same refinement level ordering must be applied, so that *Event_a* must execute first followed by *Event_b* then *Event_c*. Moreover, the solid line of *Event_b* enforces ordering with sub-events from previous refinements, so *Event_b* must follow *Event_1* and precede *Event_3*.

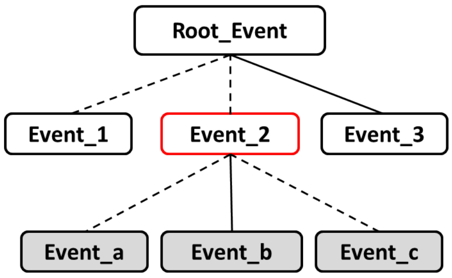


Figure 3.8: ERS Example Diagram

Under the weak sequencing interpretation, both of these events traces are possible behaviours of Figure 3.8:

$< \textbf{Event\_1}, Event\_a, \textbf{Event\_b}, Event\_c, \textbf{Event\_3} >$
$< Event\_a, \textbf{Event\_1}, \textbf{Event\_b}, \textbf{Event\_3}, Event\_c >$

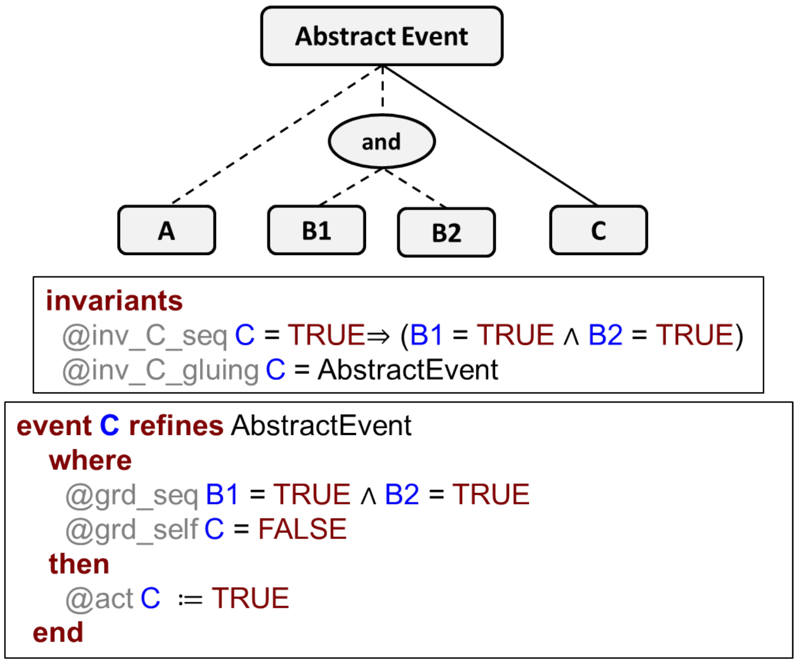
These are not possible under the strong sequencing interpretation, where the only possible execution trace is:

$< Event\_1, Event\_a, Event\_b, Event\_c, Event\_3 >$

The rest of the ERS combinators are described in the following sub-sections:

3.4.1.1 And Combinator

The *and-combinator* is used to describe that the execution of all its sub-events should finish, before the next event can execute. The order in which its sub-events execute is non-deterministic. It should also include at least two sub-events. In Figures 3.9 and 3.10, C can only execute after both *B1* and *B2* finish executing.



```
invariants
  @inv_C_seq C = TRUE⇒ (B1 = TRUE ∧ B2 = TRUE)
  @inv_C_gluing C = AbstractEvent
```

```
event C refines AbstractEvent
   where
     @grd_seq B1 = TRUE ∧ B2 = TRUE
     @grd_self C = FALSE
   then
     @act C  ≔ TRUE
  end
```

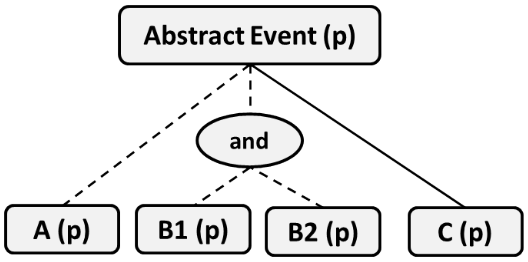
Figure 3.9: ERS *and-combinator* (Single Instance Case)

Applying the *and-combinator* will affect the sequencing constraint of its following event (C). In the single instance case (Figure 3.9), the sequencing invariant ensures that the control variable C cannot be *true*, unless both *B1* and *B2* are already set to *true*. The effect of the *and-combinator* also appears in the sequencing guard, which requires the occurrence of both *B1* and *B2* ($B1 = TRUE \wedge B2 = TRUE$) to enable the execution of C.

Similarly in the multiple instances case (Figure 3.10), enabling the execution of C requires the execution of both *B1* and *B2*, which in this case is done by ensuring that the set C is subset of the intersection of the sets *B1* and *B2*. The gluing invariant in both the single and multiple instances cases ($C = AbstrcatEvent$) is the result of applying the solid line to C, which is also reflected in the event C by using the keyword *refines*.

3.4.1.2 Or Combinator

The *or-combinator* is used to describe multiple choice, in other words at least one of its sub-events is required to execute, but possibly more than one can also execute. In the



```
invariants
  @inv_C_seq C ⊆ B1 ∩ B2
  @inv_C_gluing C = AbstractEvent
```

```
event C refines AbstractEvent
   any p
   where
    @grd_seq p ∈ (B1∩ B2)
    @grd_self p ∉ C
   then
    @act C  ≔ C  ∪ { p }
 end
```

Figure 3.10: ERS *and-combinator* (Multiple Instances Case)

case that more than one sub-event is executed, the order is non-deterministic. Similar to the *and-combinator*, it should be applied to at least two sub-events.

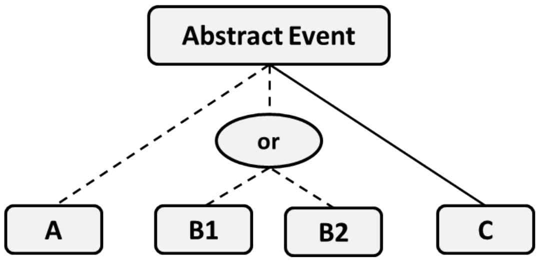
In Figure 3.11, we show the effect of applying the *or-combinator* in the single instance case. The *or-combinator* affects the sequencing constraint of C, which requires the occurrence of either *B1* or *B2* to enable C, this is indicated by the sequencing guard of C $(B1 = TRUE \vee B2 = TRUE)$ and is also enforced by the sequencing invariant which states that the control variable C cannot be *true*, unless either *B1* or *B2* has occurred.

In the multiple instances case Figure 3.12, the use of *union* in the sequencing invariant of C, means that C should be either a subset of *B1* or *B2* or both.

3.4.1.3 Xor Combinator

The *xor-combinator* is used to describe mutual exclusiveness, in other words exactly one of its sub-events can be executed at any time. Similar to the *or-combinator* and the *and-combinator*, it should be applied to two or more sub-events.

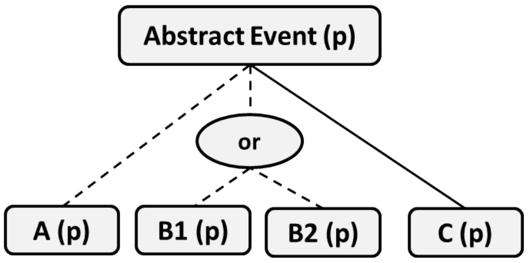
Figures 3.13 and 3.14 show the effect of applying the *xor-combinator* in the single instance case and the multiple instances case respectively. In both the single instance and multiple instances case the sequencing constraint of C is similar to applying the *or-combinator*. However, the *xor-combinator* adds additional constraints to its sub-events to ensure that exactly one sub-event can execute at any time. Applying the *partition*



```
invariants
  @inv_C_seq C = TRUE⇒ (B1 = TRUE ∨ B2 = TRUE)
  @inv_C_gluing C = AbstractEvent
```

```
event C refines AbstractEvent
  where
    @grd_seq B1 = TRUE ∨ B2 = TRUE
    @grd_self C = FALSE
  then
    @act C  ≔ TRUE
 end
```

Figure 3.11: ERS *or-combinator* (Single Instance Case)

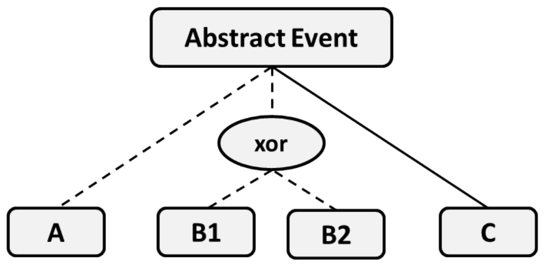


```
invariants
  @inv_C_seq C ⊆ B1 ∪ B2
  @inv_C_gluing C = AbstractEvent
```

```
event C refines AbstractEvent
  any p
  where
    @grd_seq p ∈ (B1∪ B2)
    @grd_self p ∉ C
  then
    @act C  ≔ C  ∪ {p}
 end
```

Figure 3.12: ERS *or-combinator* (Multiple Instances Case)

operator in Figure 3.13 ensures that at any time only one control variable of the *xor-combinator* sub-events can be *true*. An additional guard (*grd_xor*) will be added to both *B1* and *B2* to check the execution of the other. In Figure 3.13, we also show *B1* which checks that *B2* has not occurred ($B2 = FALSE$), the same also applies to *B1*.



```
invariants
  @inv_C_seq C = TRUE⇒ (B1 = TRUE ∨ B2 = TRUE)
  @inv_C_gluing C = AbstractEvent
 @inv_xor partition({B1, B2} ∩ {TRUE}, {B1} ∩ {TRUE}, {B2} ∩ {TRUE})
```

```
event B1
   where
    @grd_seq A = TRUE
    @grd_self B1= FALSE
    @grd_xor B2= FALSE
   then
    @act B1  ≔ TRUE
 end
```

```
event C refines AbstractEvent
   where
    @grd_seq B1 = TRUE ∨ B2 = TRUE
    @grd_self C = FALSE
   then
    @act C  ≔ TRUE
 end
```

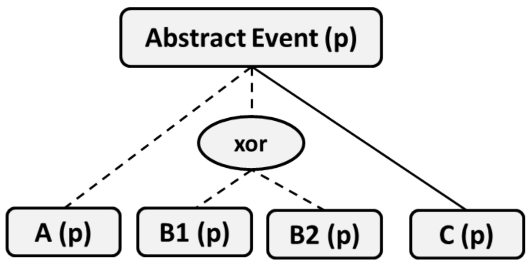
Figure 3.13: ERS *xor-combinator* (Single Instance Case)

In the multiple instances case (Figure 3.14), the *partition* operator used in *inv_xor* ensures that the sets (*B1* and *B2*) are disjoint. The *grd_xor* is added to *B1* to ensure the mutual exclusiveness property by checking that *B2* has not occurred for the same instance value, the same also applies to *B2*.

3.4.1.4 Loop Combinator

The *loop-combinator* when applied to an event, means zero or more executions of that event. A *loop-combinator* should only be applied to one event. In case we need to apply it to more than one event, an extra level of refinement is added and a resetting event will be generated. The resetting event is required because a control variable will be generated for the *loop* sub-events to control their ordering, hence the need to reset the *loop* sub-events control variables to enable more than one execution. A *loop* is represented using the star symbol as shown in figures 3.15 and 3.16.

Unlike other events in ERS, the loop event does not have a corresponding control variable in its Event-B representation. In Figure 3.15, we show the effect of applying this combinator on the loop event, *B*, and on the event following this combinator, *C*. B can execute zero or more times, that is why, there is no need for a control variable to record its execution. The execution of *B* is terminated by the execution of its follow-on event (C), this is ensured by the additional guard ($C = FALSE$), which is added to *B*.

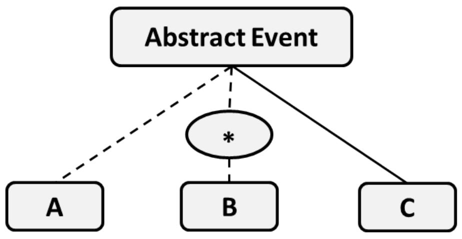


```
invariants
  @inv_C_seq C ⊆ B1 ∪ B2
  @inv_C_gluing C = AbstractEvent
 @inv_xor partition((B1∪ B2), B1, B2)
```

```
event B1
   any p
   where
     @grd_seq p ∈ A
     @grd_self p ∉ C
     @grd_xor p ∉ B2
   then
     @act B1  ≔ B1  ∪ {p}
 end
```

```
event C refines AbstractEvent
   any p
   where
     @grd_seq p ∈ (B1∪ B2)
     @grd_self p ∉ C
   then
     @act C  ≔ C  ∪ {p}
 end
```

Figure 3.14: ERS *xor-combinator* (Multiple Instances Case)



```
invariants
  @inv_C_seq C = TRUE⇒ A = TRUE
  @inv_C_gluing C = AbstractEvent
```

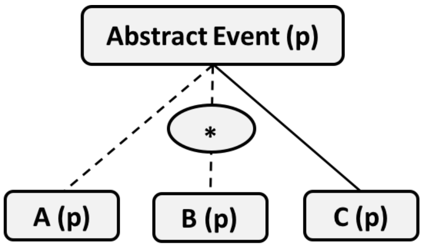
```
event B
   where
     @grd_seq A = TRUE
     @grd_loop C = FALSE
end
```

```
event C refines AbstractEvent
   where
     @grd_seq A = TRUE
     @grd_self C = FALSE
   then
     @act C  ≔ TRUE
 end
```

Figure 3.15: ERS *loop-combinator* (Single Instance Case)

Because the execution of B is optional, the sequencing of C depends on A and **not** B (*inv_C_seq*).

Similarly in the multiple instances case (Figure 3.16), the execution of B for an instance value is disabled if C executes for the same instance value ($p \notin C$). We can also see that



```
invariants
  @inv_C_seq C ⊆ A
  @inv_C_gluing C = AbstractEvent
```

```
event B
   any p
   where
    @grd_seq p ∈ A
    @grd_loop p ∉ C
end
```

```
event C refines AbstractEvent
   any p
   where
    @grd_seq p ∈ A
    @grd_self p ∉ C
   then
    @act C ≔ C  ∪ {p}
  end
```

Figure 3.16: ERS *loop-combinator* (Multiple Instances Case)

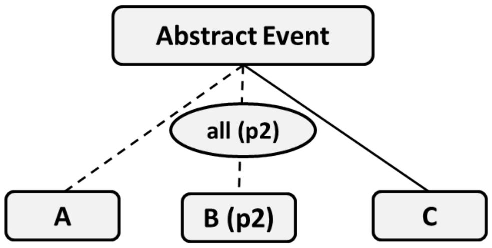
the set C is subset of A because of the possible zero execution case of B.

3.4.1.5 All Combinator

The *all-combinator* is used to represent execution of an event for all values of a parameter from some set. This combinator is a generalisation of the *and-combinator*, so the event following this combinator cannot be executed, until the *all-combinator* event completes execution for all the instance values of the parameter set. Figures 3.17 and 3.18 show an example of applying this combinator with *p2* as the *all-combinator* parameter, in the single and multiple instances case respectively. If we need to apply this combinator to a sequence of events, an extra level of refinement is required.

In the single instance case (Figure 3.17), the type of the *all-combinator* event (B) is the same as that of the *all-combinator* parameter ($B \subseteq Type(p2)$). The sequencing invariant of B means that if B has executed for at least one instance value ($B \neq \phi$), then A must have occurred. The sequencing invariant of C indicates that C can only occur after B completes execution for all instance values of the set of *p2*. These sequencing invariants are maintained by the sequencing guards added to the events (B and C).

In the multiple instances case (Figure 3.18), the type of B is extended with the type of *p2*, which is the *all-combinator* parameter ($B \subseteq Type(p) \times Type(p2)$). The sequencing invariant of C ensures that for any instance value of parameter p in the set *Type(p)*, B has executed for all instances of *p2* in the set *Type(p2)*.

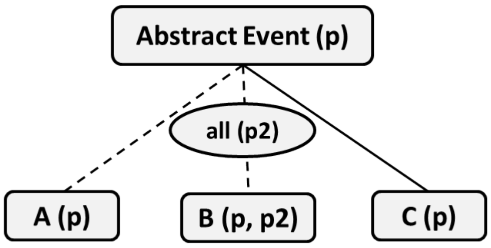


```
invariants
  @inv_B_type B ⊆  TYPE(p2)
  @inv_B_seq B ≠ ϕ ⇒ A = TRUE
  @inv_C_seq C = TRUE ⇒ B = TYPE(p2)
  @inv_C_gluing C = AbstractEvent
```

```
event B
    any p2
    where
      @grd_seq  A= TRUE
      @grd_self p2 ∉ B
 then
      @act B ≔ B ∪ { p2 }
end
```

```
event C refines AbstractEvent
    where
      @grd_seq B = TYPE(p2)
      @grd_self C = FALSE
    then
      @act C  ≔ TRUE
  end
```

Figure 3.17: ERS *all-combinator* (Single Instance Case)



```
invariants
  @inv_B_type B ⊆  TYPE(p)×TYPE(p2)
  @inv_B_seq dom(B) ⊆ A
  @inv_C_seq p ∈ C ⇒ B[{p}] = TYPE(p2)
  @inv_C_gluing C = AbstractEvent
```

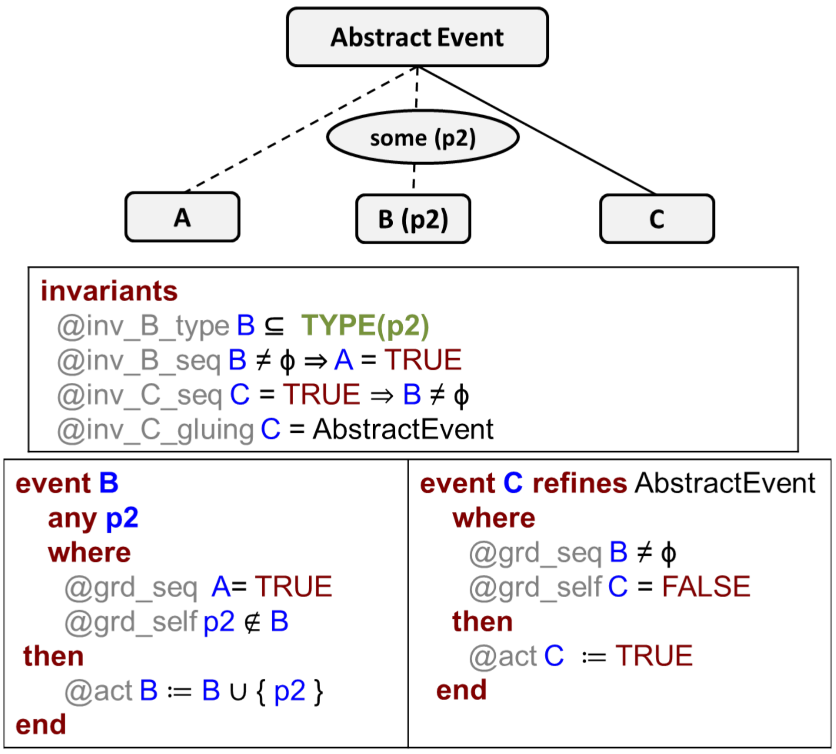
```
event B
    any p p2
    where
      @grd_seq  p ∈ A
      @grd_self p2 ∉ B
    then
      @act B ≔ B ∪ {p ↦ p2}
end
```

```
event C refines AbstractEvent
    any p
    where
      @grd_seq B[{p}] = TYPE(p2)
      @grd_self p ∉ C
    then
      @act C  ≔ C  ∪ { p }
  end
```

Figure 3.18: ERS *all-combinator* (Multiple Instances Case)

3.4.1.6 Some Combinator

The *some-combinator* is a generalisation of the *or-combinator*. Similar to the *all-combinator*, it includes the addition of a parameter (Figures 3.19 and 3.20). As an *or-combinator* generalisation, it means that the *some-combinator* sub-event must be executed for at least one instance of the *some-combinator* parameter before the next events are enabled. However, the execution of the events following the *some-combinator* does not mean the *some-combinator* sub-events have to stop execution.



```
invariants
  @inv_B_type B ⊆ TYPE(p2)
  @inv_B_seq B ≠ ϕ ⇒ A = TRUE
  @inv_C_seq C = TRUE ⇒ B ≠ ϕ
  @inv_C_gluing C = AbstractEvent
```

```
event B
   any p2
   where
     @grd_seq  A= TRUE
     @grd_self p2 ∉ B
 then
     @act B ≔ B ∪ { p2 }
end
```

```
event C refines AbstractEvent
   where
     @grd_seq B ≠ ϕ
     @grd_self C = FALSE
   then
     @act C  ≔ TRUE
 end
```

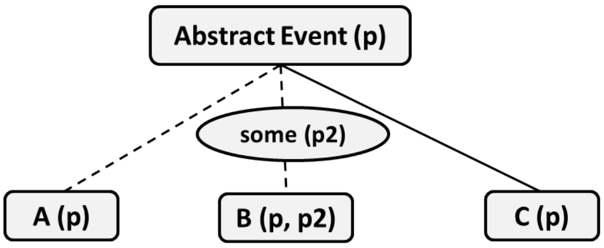
Figure 3.19: ERS *some-combinator* (Single Instance Case)

Figure 3.19 shows the effect of applying the *some-combinator* to the Event-B translation of both B and C in the single instance case. Compared to the *all-combinator*, the difference is in the sequencing guard and invariant of C, which requires only the execution of B for one instance of the *some-combinator* parameter $p2$ to be enabled.

Similarly in the multiple instances case (Figure 3.20), the sequencing invariant of C ($C \subseteq dom(B)$) ensures that B has executed for at least one instance value of $p2$.

3.4.1.7 One Combinator

The *one-combinator* is a generalisation of the *xor-combinator*, so it requires the execution of its sub-event for exactly one instance of its parameter. Similar, to the other replicators *all* and *some*, it requires the addition of a parameter (Figures 3.21 and 3.22).

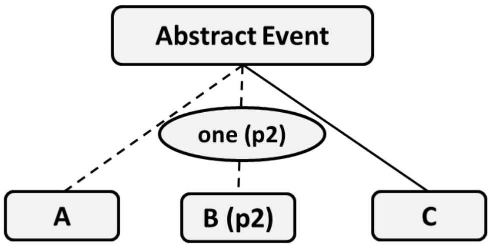


```
invariants
  @inv_B_type B ⊆ TYPE(p)×TYPE(p2)
  @inv_B_seq dom(B) ⊆ A
  @inv_C_seq C ⊆ dom(B)
  @inv_C_gluing C = AbstractEvent
```

```
event B
   any p p2
   where
     @grd_seq  p ∈ A
     @grd_self p2 ∉ B
   then
     @act B ≔ B ∪ {p ↦ p2}
end
```

```
event C refines AbstractEvent
   any p
   where
     @grd_seq p ∈  dom(B)
     @grd_self p ∉ C
   then
     @act C  ≔ C  ∪ { p }
 end
```

Figure 3.20: ERS *some-combinator* (Multiple Instances Case)



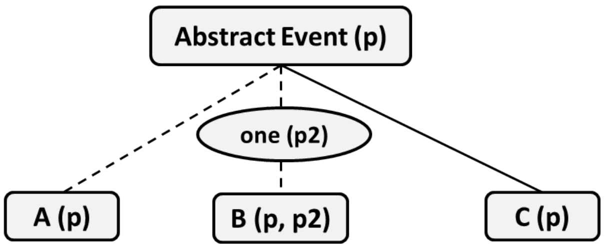
```
invariants
  @inv_B_type B ⊆  TYPE(p2)
  @inv_B_seq B ≠ ϕ ⇒ A = TRUE
  @inv_C_seq C = TRUE ⇒ B ≠ ϕ
  @inv_B_one card(B) ≤ 1
  @inv_C_gluing C = AbstractEvent
```

```
event B
   any p2
   where
     @grd_seq  A= TRUE
     @grd_self p2 ∉ B
    @grd_one B = ϕ
 then
     @act B ≔ B ∪ { p2 }
end
```

```
event C refines AbstractEvent
   where
     @grd_seq B ≠ ϕ
     @grd_self C = FALSE
   then
     @act C  ≔ TRUE
 end
```

Figure 3.21: ERS *one-combinator* (Single Instance Case)

The sequencing of B and C is similar to the *some-combinator*, but the main effect of the *one-combinator* appears in its sub-event (B). Figure 3.21 shows the result of applying the *one-combinator* in a single instance case, which is illustrated by the one invariant and guard. The *one* invariant ($card(B) \leq 1$) ensures that B can at most occur once. The guard $B = \phi$ checks that the B has not executed before.



```
invariants
  @inv_B_type B ⊆  TYPE(p)×TYPE(p2)
  @inv_B_seq dom(B) ⊆ A
  @inv_C_seq C ⊆ dom(B)
  @inv_B_one ∀p. card(B[{p}] ) ≤ 1
  @inv_C_gluing C = AbstractEvent
```

```
event B
   any p p2
   where
    @grd_seq  p ∈ A
    @grd_self p ↦  p2 ∉ B
    @grd_one p ∉ dom(B)
   then
    @act B ≔ B ∪ {p ↦ p2}
end
```

```
event C refines AbstractEvent
   any p
   where
    @grd_seq p ∈  dom(B)
    @grd_self p ∉ C
   then
    @act C  ≔ C  ∪ { p }
 end
```

Figure 3.22: ERS *one-combinator* (Multiple Instances Case)

Similarly in the multiple instance case (Figure 3.22), the *one* invariant ensures that B can occur at most once for any instance value ($p2$). This is maintained by the guard ($p \notin dom(B)$) which is added to B.

3.4.2 Event Refinement Structures (ERS) Constraints

ERS has some restrictions that are required for refinement to work smoothly and to simplify the modelling process. One of the significant restrictions is the *single solid line rule*, where exactly one leaf can be connected to its parent event by a solid line. The *single solid line rule* can be justified by the event execution trace of the model, where hiding newly added events from the event execution trace of the concrete model, should result in the same event execution trace of the abstract model (Butler, 2009b,a).

Consider the ERS diagram in Figure 3.23, which represents a generic pattern for introducing a replicator in a refinement step. The event execution trace at the abstract level is $< A >$, while the event execution trace at the concrete level allows multiple executions of B for different instances of the parameter p. One possible execution is $< B(p1), B(p2), B(p3), C >$, where *p1*, *p2* and *p3* represent different instance values of the replicator parameter p. Hiding the newly added events will result in the event execution trace $< C >$, which is the same as the abstract level, since C refines A. However, if we violate the *single solid line rule* and allow *B(p)* to refine A, we will have multiple executions of events that refine A, represented by $< B(p1), B(p2), B(p3) >$, which does not correspond to the abstract trace which only allows a single execution of event A.

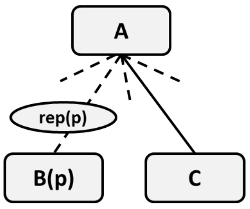


Figure 3.23: General Pattern for Introducing Replicator

The rest of the restrictions can be summarised as follows (Salehi Fathabadi et al., 2012):

- As a consequence of the *single solid line rule*, where exactly a single event can refine an abstract event, the combinators *and*, *or*, *loop*, *all*, *some* and their directly connected children are always connected to the previous level using dashed lines.
- In ERS the only case where more than one event can be connected to its parent using solid line is when applying the *xor-combinator*, since the *xor-combinator* ensures the execution of exactly one of its sub-flows.
- The only replicator which can be refining (i.e., connected to its parent by a solid line) is the *one-combinator*, because it allows the execution of exactly one instance value.
- The events which are directly connected to a combinator, will inherit their refining property from their combinator.
- A combinator cannot be applied to another combinator at the same refinement level, i.e., a combinator can only be connected to an event. That event can be elaborated in later refinements.
- Replicators (Figure 3.23) can have only one child event. If more than one event needs to be replicated, this has to be done through refinement.
- Similar to replicators, a *loop* can only be connected to one child event.
- A *loop* cannot be the last child of a flow.
- Multiple instance modelling can be introduced by either adding a parameter to the root of an ERS diagram and/or by using a replicator.

- The set of values of a parameter added to a replicator can only be a static set, i.e., a constant or a carrier set.
- When a parameter is applied for multiple instance modelling, it is inherited by all the children of a tree or sub-tree.
- The abstract level is distinguished from the other refinements by having events with only dashed lines, and its root is not an event, it is the name of the ERS flow.
- The same event name cannot be used in different branches of the tree. The only case the same event name can be used is the case of a parent and its solid child leaf.

3.5 Conclusion

In this chapter, we have introduced the main concepts required to understand our work. Our work focuses on the formal method Event-B and the diagrammatic notation ERS which extends Event-B with a visual representation. We choose Event-B for its mathematical simplicity and important concepts in facilitating the formal modelling using refinement. ERS, on the other hand provides a visual side of Event-B supporting control-flow as well as refinement. The combination of Event-B with the ERS approach can result in a formal workflow modelling approach facilitating formal modelling on one hand, and formalising workflow modelling on the other hand.

Chapter 4

Extending ERS by Dynamic Workflow Modelling

4.1 Introduction

The ERS approach provides Event-B with explicit control flow using different control flow combinators. The ERS combinators not only have simple logical combinators (*and*, *or*, *xor*), but also supports behaviour replication through quantified combinators (*all*, *some*, *one*).

Despite the variety of ERS combinators, ERS still lacks the flexibility to model dynamic control flow that supports dynamic changes in the degree of concurrency. Specifically in the cases where the degree of parallelism is data dependent and data values can change during execution. In this chapter we start by identifying some of the limitations of the ERS approach, and we also extend the original ERS approach to address the identified limitations.

The work presented in this chapter and Chapter 5 is submitted to "Software and Systems Modeling (SoSyM)" journal.

4.2 Limitations of the Original ERS Approach in Control Flow Modelling

4.2.1 Data Dependent Workflow Limitations

One of the main objectives of ERS is the explicit representation of control flow in Event-B. While ERS has a variety of combinators to support control flow, ERS still lacks flexibility to support dynamic changes in the degree of concurrency.

To better illustrate this limitation, consider the example ERS diagram in Figure 4.1. The ERS diagram describes part of the flow of an emergency dispatch system, which is responsible for sending appropriate resources to emergency incidents. The root of the tree represents the name of the workflow model. The parameter i, introduced at the root of the tree and inherited by all the nodes and leaves of the tree, indicates that there are multiple instances of the *Emergency_Dispatch* workflow with each instance being distinguished by parameter i. The leaves of the tree represents events in Event-B, where a leaf is an atomic or non-decomposed task. The leaf events of different instances of a workflow are interleaved.

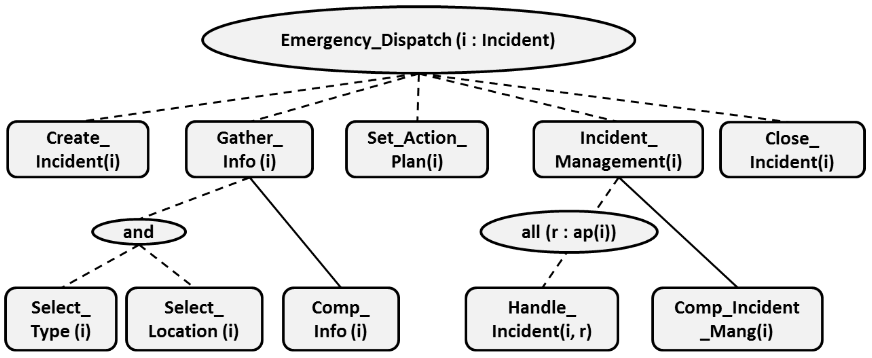


Figure 4.1: ERS Diagram of a Fire Dispatch Workflow

As explained earlier in Chapter 3 the order of execution of the ERS leaf events is from left to right. In Figure 4.1, the event *Set_Action_Plan* sets an action plan *ap* for the incident, identifying the resources needed to handle an incident in the *Incident_Management* activity. The *Incident_Management* activity is in turn decomposed into *Handle_Incident* where we apply the *all* combinator, the generalisation of *and*. *Handle_Incident* can interleave for different instances r in the set $ap(i)$, i.e., the action plan of incident i. The expression $ap(i)$ represents the set of resources required for incident i and parameter r of ranges over this set.

The $all(r : ap(i))$ structure is an example of a data dependent workflow. The set $ap(i)$ represents the action plan of the incident i, and the value of $ap(i)$ is determined by the event *Set_Action_Plan*. Now the challenge is that in a real system the action plan of an incident can change during the execution of this data dependent workflow. This is not allowed in the original ERS. For this reason we extend the ERS approach with a new flexible combinator that allows dynamic modelling of data dependent workflows. We also improve the parameter definition to allow the parameter set to be any expression and not only constants as the original ERS did.

4.2.2 Loop Limitations

The compositional structure of ERS means that the *loop* is a structured loop, where there is only one entry point and one exit point. Originally ERS required the *loop* to have only one child node, to be executed zero or more times. To give the *loop* more flexibility without breaking its structured nature, we generalise the loop to have one or more child nodes. Such generalisation supports dynamic behaviour through non-deterministic execution of event at each iteration. For example, the itinerary selection of a travel agency booking system allows the user to book different flights, hotels and/or cars at the same time, this can be represented using a *loop*, where at each iteration the user can perform a different selection (flight, hotel or car), but all selections still have the same exit point, which is when the user decides to check out.

Using the original form of the *loop* will require refinement and additional combinators like the *and-combinator*. For example to represent the non-deterministic iterations of the events A and B, we cannot simply do as shown in Figure 4.2. The first diagram is invalid because it implies some sequencing between the loops A and B. The second case is also invalid because it applies two consecutive combinators at the same level.

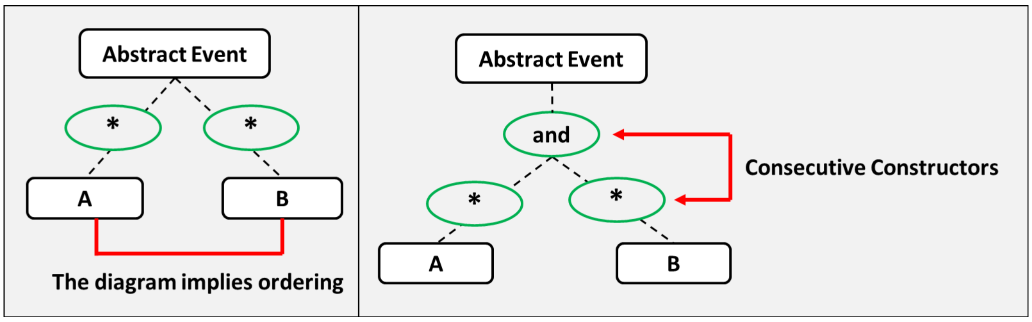


Figure 4.2: Invalid Representation of Non-Deterministic Loop Events

The correct representation is in Figure 4.3, which can be only done by introducing refinement. There are two possible ways either using the *or-combinator* or using the *and-combinator*. The first disadvantage is the inconsistent representation of such a simple behaviour. The second disadvantage is that, the refinements will result in the addition of unnecessary events, event C in the *or-refinement* case, and events $A2$ and $B2$ in the *and-refinement* case. The simple solution is to generalise the *loop* definition to allow its application to more than one event, as shown in Figure 4.4.

4.3 Advantages of ERS in Control Flow Modelling

The restrictions described in Section 4.2 should not undermine the importance of ERS in modelling workflows. In ERS the main characteristics of control flow modelling are supported through different combinators described in Chapter 3. These combinators support the typical aspects of control flow modelling which are: sequencing (left to right placement of nodes), choice (*xor*, *or*), concurrency (*and*, *all*, *some*), repetition (*loop*, *all*, *some*) and synchronisation (*and*, *all*). We have seen in Figure 4.1 an example of using ERS in modelling sequencing, concurrency, repetition and synchronisation. Not

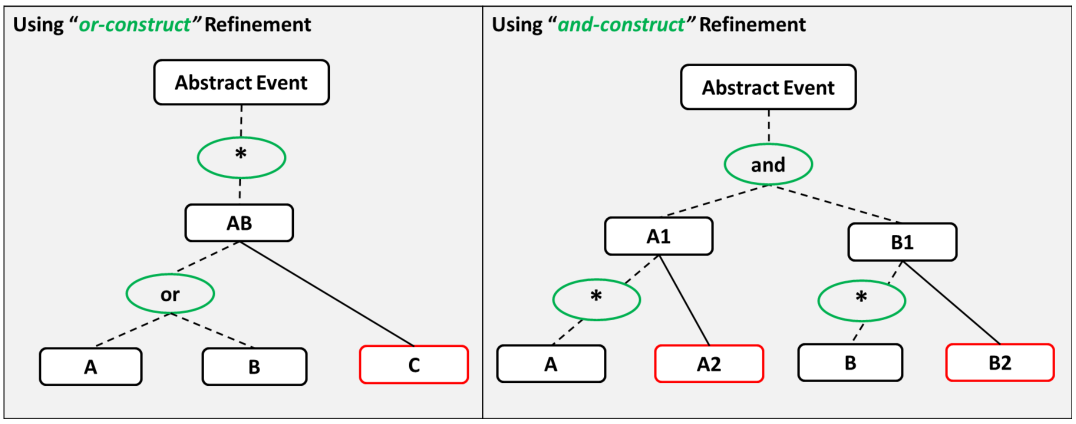

Using "or-construct" Refinement
Abstract Event
*
AB
or
A
B
C
Using "and-construct" Refinement
Abstract Event
and
A1
B1
*
*
A
A2
B
B2

Figure 4.3: Correct Representation of Non-Deterministic Loop Events in the Original ERS

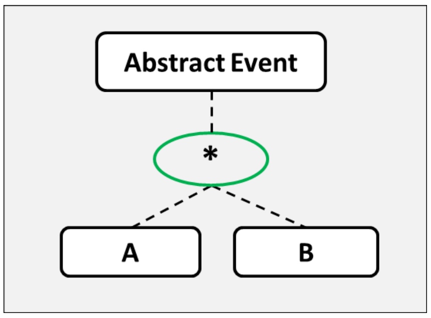


Figure 4.4: Generalisation of the Loop

only does ERS support control flow, but its translation to Event-B allows the handling of data, e.g., updating variables through event actions. Supporting control flow and handling data using Event-B, in addition to the graphical nature of ERS and its Event-B formal semantics, which allows validation and verification, makes ERS well suited for workflow modelling.

In order to better understand the different ERS combinators and how they can be used to control the flow of events, we will summarise the disabling conditions of the events under a combinator, i.e., the conditions that stop the flow of events from executing again, and the enabling conditions of their successor events according to the combinator applied.

Consider Figure 4.5, where P and Q are themselves ERS diagrams. P and Q are represented using the ERS diagram in a triangle to indicate the possibility of being a single event or a sub-flow. P occurs to the left of Q, and the combinator C applied to P defines the disabling conditions for P and the enabling conditions for Q.

Table 4.1 describes the disabling conditions of P and the enabling conditions for Q according to the different forms of combinator C. The enabling condition for P is determined by treating P as Q and whatever precedes P as P. If nothing precedes P then no additional enabling condition is required. Note that the conditions for disabling P and

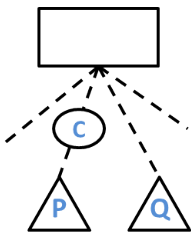
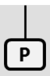
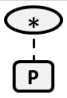
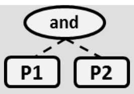
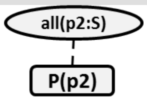
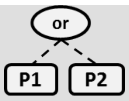
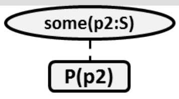
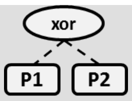
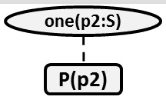


Figure 4.5: Generalised ERS Diagram

enabling Q are **not** necessarily the same. Moreover, the events within P may have other explicit data guards which might cause P to be disabled. The conditions in the middle and last columns define conditions (represented as guards) that are conjoined with any explicit event guards.

Table 4.1: Disabling and Enabling Conditions for ERS Patterns

Form of C		**Disable P**	**Enable Q**
once P		P has executed	P has executed
loop P		Q has made a step	True
P1 **and** P2		Both P1 and P2 have executed	Both P1 and P2 have executed
all $p2$:S.P($p2$)		P($p2$) has executed for each $p2$ in S	P($p2$) has executed for each $p2$ in S
P1 **or** P2		Both P1 and P2 have executed	P1 or P2 has executed
some $p2$:S.P($p2$)		P($p2$) has executed for each $p2$ in S	P($p2$) has executed for some $p2$ in S
P1 **xor** P2		P1 or P2 has executed	P1 or P2 has executed
one $p2$:S.P($p2$)		P($p2$) has executed for one $p2$ in S	P($p2$) has executed for some $p2$ in S

The first form describes the case where P is a single execution flow (indicated diagrammatically by a line with no oval). Then one execution of P will disable further executions of P and enable Q. The *loop* pattern will disable P once Q starts execution, but there are no conditions on enabling Q, which means it is possible there will be zero executions of P.

The *and-combinator* requires both sub-flows *P1* and *P2* to execute before P is disabled and Q is enabled. In the generalised form of the *and-combinator*, the *all-replicator* requires the sub-flow to be executed once for each $p2 \in S$ before P is disabled and Q is enabled.

The *or-combinator* requires at least one sub-flow of P to execute before enabling Q, and even if Q has executed, P can continue its execution until all its sub-flows have executed, in this case both *P1* and *P2* should execute to disable P. The *some-replicator* requires the sub-flow P to be executed for at least one $p2 \in S$ to enable Q. The sub-flow P will be disabled when all $p2 \in S$ have executed.

Similar to the *or-combinator*, the *xor-combinator* requires at least one sub-flow, *P1* or *P2*, to execute before Q is enabled. However unlike the *or-combinator*, the *xor-combinator* sub-flow will be disabled once one of its sub-flows (*P1* or *P2*) executes. The *one-replicator*, which is the generalisation of the *xor-combinator* will enable Q, when the sub-flow P executes for at least one $p2 \in S$, while P will be disabled when exactly one $p2 \in S$ executes.

As we can see in Table 4.1, ERS not only supports simple combinators like (*or*, *and*, *xor*), but also explicitly supports behaviour replication through different combinators (*all*, *some*, *one*). The multiple instances support is one of the ERS key strengths, that is why in the following sections, we extend ERS to make better use of it in workflow modelling.

4.4 Parameter Extension

In the original translation of ERS diagrams to Event-B, the parameter set of a replicator is always described as a constant defined in the context of the model (Wiki.event-b.org, 2012a; Fathabadi et al., 2014). However, it is not always possible to define the parameter set in the context, as we have seen in Figure 4.1, the parameter set of r is determined by an event (*Set_Action_Plan*) in the machine.

This requires extending the definition of the parameter and adding some more guards and invariants when applying a replicator to enable the use of variables and not only constants. We start by extending the parameter definition by adding a *range_expression* which can be a variable or a constant. The definition of a parameter will be extended as follows:

par = "par"(name, Type, **range_expression**)

We will explain this through a simple example represented by the ERS diagram of Figure 4.6, where each doctor is allocated a set of patients to visit at home. The patients that the doctor needs to visit will differ from one day to the other. The patients will be allocated to each doctor (d) in the event *Allocate_Patients*.

In this case the parameter par of the *all-combinator* is: par = “par”(p, *PATIENTS*, *patients*[{d}]), where the parameter name is p, the type is *PATIENTS* and the range_expression is the variable *patients*[{d}], which is the set of patients allocated to doctor d.

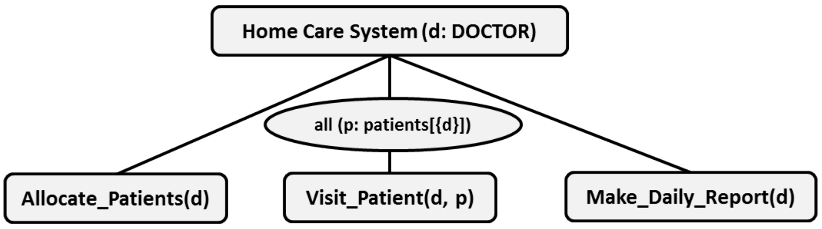


Figure 4.6: Home Care System Example

Using a variable instead of a constant requires the event *Visit_Patient* to explicitly check that it is executing for an instance in the variable set. Therefore, we need to explicitly add the guard $p \in patients[\{d\}]$ to the event *Visit_Patient*. This guard can be enforced by an invariant to ensure that *Visit_Patient* can only execute for values in the range_expression set ($patients[\{d\}]$) as follows:

$$\forall d.d \in DOCTOR \Rightarrow Visit\_Patient[\{d\}] \subseteq patients[\{d\}]$$

If the range_expression guard ($p \in patients[\{d\}]$) is removed from the event (*Visit_Patient*), then in this case the doctor can visit any patient, whether allocated to them or not. Moreover, the invariant enforcing the guard, will also ensure that patients cannot be removed from the allocated patients of the doctor after visiting them.

4.5 A Dynamic Replicator: Par-Replicator

Par-Replicator Behaviour: The purpose of the *par-replicator* is to execute a sub-flow for several instances of a certain parameter, but the difference to other ERS replicators (*all, some, one*) is that the number of instances of that parameter is not predetermined and can change during the execution of its flow, depending on the environment.

The behaviour of the *par-replicator* can be considered as a combination of the *loop* and the *some-replicator*. It is similar to a *loop* in the sense that it can execute zero or more times and it stops execution when the follow-on event executes. It is similar to the *some-replicator*, since it does not have to be applied to all the instances of the parameter set, and its sub-flow may be interleaving for different instances. However, the main difference with the *some-replicator* is that the *some-replicator* sub-flow can continue execution even after the execution of its follow-on event and the parameter instances cannot change during the execution of the *some-replicator* sub-flow.

Diagrammatic Representation: Similar to the ERS replicators, the *par-replicator* is represented within an oval with the *par* keyword followed by the name of the parameter, as shown in Figure 4.7. The parameter *p2* of the *par-replicator* has *p_Set* as the *range_expression*. The non-connected lines in the ERS diagram, means we are only showing part of the flow. The non-refining restriction which is the result of the *single solid line rule*, explained in Chapter 3, applies to the *par-replicator*, i.e. the *par-replicator* sub-event cannot be a direct refinement to an event at the previous abstract level and will always be connected to the previous level using a dashed line. The non-refining restriction is due to the fact that the *par-replicator* sub-event can execute zero or more times, which violates the *single solid line rule* of ERS, where exactly one event can be a direct refinement of the parent event.

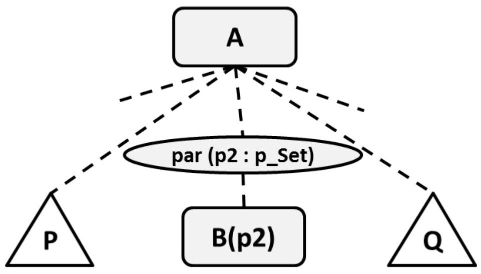


Figure 4.7: Diagrammatic Representation of the *par-replicator*

Parallel Producer-Consumer Pattern: The *par-replicator* supports dynamic behaviour, where the number of replications cannot be predetermined. Take the example of a travel agency, where the travel agency attempts to book flights, hotels etc., while the user is still selecting the itinerary. In this case the number of instances to be booked changes dynamically as the user extends their itinerary. This example can be generalised to any parallel *Producer-Consumer* pattern, as shown in Figure 4.8. The parallel *Producer-Consumer* pattern shows how elements are consumed, while other elements are still being produced. In this case the *and-combinator* indicates the concurrent production and consumption of elements, where the left hand side sub-flow of the *and-combinator*, shows how new elements are produced and added to the set *a* in the *Produce* event, while the right hand sub-flow of the *and-combinator* represents the elements consumption in the *Consume* event.

In both producing and consuming elements we use the *par-replicator*. When producing elements, we do not know how many elements are needed, and disabling element production is determined by the *End Produce* event, that is why the *par-replicator* was a good candidate to represent this behaviour. The *Consume* event can start consuming elements from the set *a*, while other elements are still being produced, hence the need for the *par-replicator*. This pattern cannot be supported by the previous ERS replicator combinators (Salehi Fathabadi et al., 2012), because all existing replictors do not allow adding new elements while executing.

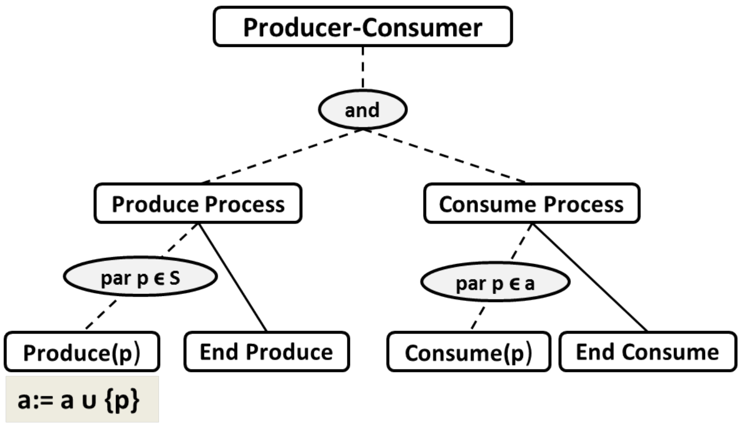
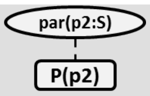


Figure 4.8: Parallel Producer-Consumer Pattern

4.5.1 Par-Replicator Semantics

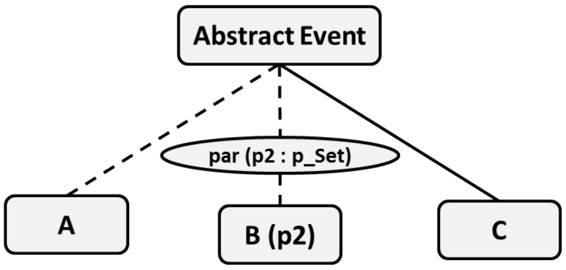
The *par-replicator* has the same disabling and enabling conditions as the *loop* in Table 4.1. We can add the *par-replicator* to Table 4.1, as shown in Table 4.2. The *par-replicator* sub-flow (P) will be disabled when Q starts execution. The enabling of Q is independent of the *par-replicator*, that is why it is *True*.

Table 4.2: Disabling and Enabling Conditions for the *par-replicator*

Form of C		**Disable P**	**Enable Q**
par $p2{:}S.\mathrm{P}(p2)$		Q has made a step	True

Figures 4.9 and 4.10, show the Event-B encoding of the *par-replicator* in the single instance case and the multiple instances case respectively. In both cases the atomicity of the *Abstract Event* is decomposed into the events A, followed by B where we apply the *par-replicator*, and finally event C which is the refining event. In addition to the normal sequencing, typing and gluing invariants and guards, the *par-replicator* results in an additional guard *grd_par* added to the *par-replicator* event (B). This guard disables the execution of the *par-replicator* event (B), if the follow-on C event has executed. In the single instance case the *par-replicator* disabling guard is $C = FALSE$, while in the multiple instances case, it is $p \notin C$ checking that C has not executed for a particular instance of the parameter set.

The *par-replicator* is similar to any ERS *replicator*, where its child type is extended by the parameter type (*Type(p2)*) as shown by *inv_B_type* in both the single instance and multiple instances cases. The invariant *inv_B_rep* and the guard *grd_exp* are the result of



```
Invariants
  @inv_A_type A ∈ BOOL
  @inv_B_type B ⊆ Type(p2)
  @inv_B_seq B ≠ ∅ ⇒ A = TRUE
  @inv_B_rep B ⊆ p_Set
  @inv_C_seq C = TRUE ⇒ A= TRUE
  @inv_C_gluing C = Abstract_Event
```

```
event A
   where
     @grd_self A = FALSE
   then
     @act A ≔ TRUE
  end
```

```
event C refines AbstractEvent
   where
     @grd_self  C = FALSE
     @grd_seq  A = TRUE
   then
     @act  C = TRUE
end
```

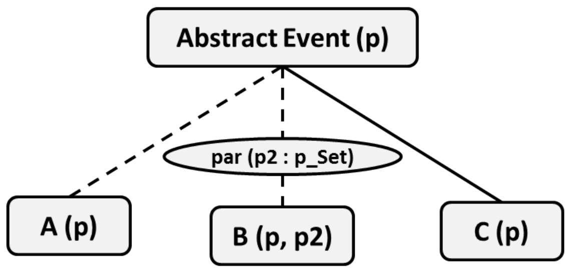
```
event B
   any p2
   where
     @grd_exp p2 ∈ p_Set
     @grd_self p2 ∉  B
     @grd_seq  A = TRUE
     @grd_par  C = FALSE
    then
     @act  B ≔ B ∪ {p2}
End
```

Figure 4.9: Single Instance Semantics of the *par-replicator*

extending the parameter with *range_expression*, as explained in Section 4.4, which apply to any replicator (*all*, *some*, *one*, *par*). The *range_expression* invariant and guard will ensure that the values of *p2* which are added to *B* are only from the *range_expression*.

There is also a possibility that the *par-replicator* event does not execute, which is why we can see the sequencing guard (*grd_seq*) and invariant (*inv_C_seq*) of the follow-on event *C* is similar to *B*.

We explained earlier that the set of the *par-replicator* parameter can change during execution. However, this is restricted to "non-decreasing" change. Otherwise this will violate the *range_expression* invariant (*inv_B_rep*) in figures 4.9 and 4.10, which requires that the *par-replicator* event (*B*) to execute only for values in the *range_expression* (p_Set).



```
Invariants
  @inv_A_type A ∈ Type(p)
  @inv_B_type B ⊆ Type(p) ×Type(p2)
  @inv_B_seq dom(B) ⊆ A
  @inv_B_rep ∀p. p ∈ Type(p) ⇒ B[{p}] ⊆ p_Set
  @inv_C_seq C ⊆ A
  @inv_C_gluing C = Abstract_Event
```

```
event A
   any p
   where
     @grd_self  p ∉  A
    then
     @act A ≔ A  ∪ {p}
  end
```

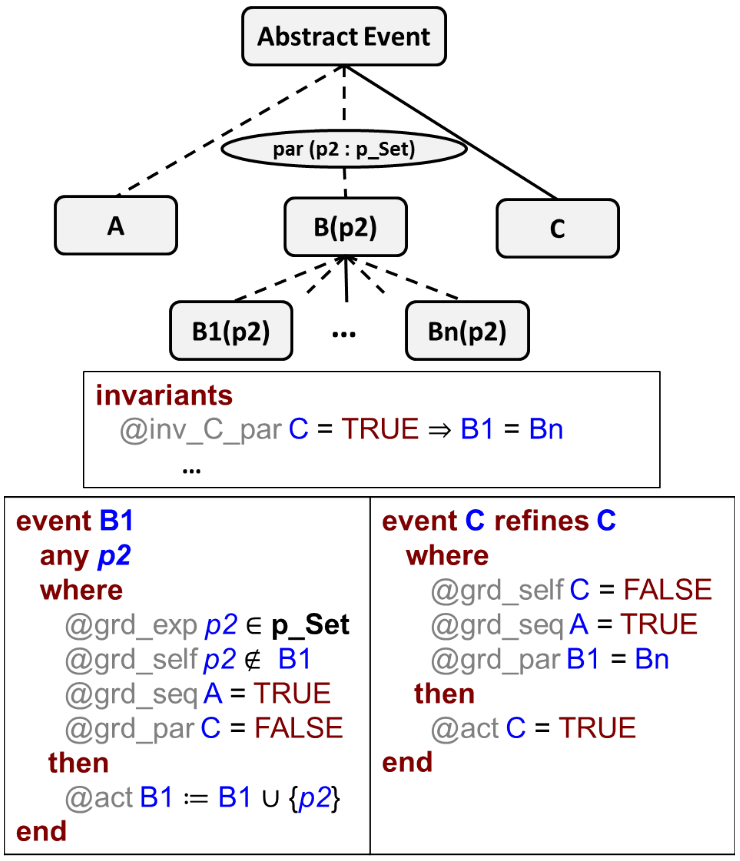
```
event C refines AbstractEvent
   any p
   where
     @grd_self p ∉  C
     @grd_seq p ∈ A
   then
     @act C ≔ C ∪ {p}
  end
```

```
event B
   any p p2
   where
     @grd_exp p2 ∈ p_Set
     @grd_self p ↦ p2  ∉  B
     @grd_seq  p ∈ A
     @grd_par  p ∉  C
    then
     @act  B ≔ B ∪ {p ↦  p2}
End
```

Figure 4.10: Multiple Instances Semantics of the *par-replicator*

Refinement: When a *par-replicator* child is refined into some sequence of events, we need to ensure that the follow-on event, C, which can disable their execution, does not interrupt the execution of their sub-events in the middle of their execution. This is because when decomposing the *par-replicator* event into some sequence of sub-events, we consider the activity of the *par-replicator* event is represented by the execution of all the sub-events and not only the refining event. This can be done by adding a guard to the follow-on event (C), that prevents it from executing before the sub-flows of the *par-replicator* complete their execution. This constraint on the follow-on event C is also represented by an invariant.

Consider the scheme in Figure 4.11 where the *par_replicator* child, B, is refined into the sequence of events starting with *B1* and ending with *Bn*. The disabling guard of the *par-replicator* event is added to the first child of the refined *par-replicator* (*B1*), disabling the execution of the *par-replicator* sub-flow when the follow-on event, *C*, executes. The invariant (*inv_C_par*) and guard (*grd_par*) of the *par-replicator* follow-on event (*C*) ensure that *C* does not interrupt the *par-replicator* sub-flow in the middle of its execution. This is done by checking that if the first child of the *par-replicator* flow has executed, then the last child of the *par-replicator* flow must have executed, before the follow-on event can disable the *par-replicator* sub-flow. In Figure 4.11, *inv_C_par* and *grd_par* of *C* check the equality of *B1* and *Bn* ensuring that *B1* and *Bn* executed for the same instance values, before *C* can be executed.



```
invariants
  @inv_C_par C = TRUE ⇒ B1 = Bn
  ...
```

```
event B1                          event C refines C
  any p2                            where
  where                               @grd_self C = FALSE
    @grd_exp p2 ∈ p_Set               @grd_seq A = TRUE
    @grd_self p2 ∉ B1                 @grd_par B1 = Bn
    @grd_seq A = TRUE               then
    @grd_par C = FALSE                @act C = TRUE
  then                            end
    @act B1 ≔ B1 ∪ {p2}
end
```

Figure 4.11: Refinement of the *par-replicator* Event

In the case of multiple instances, the same guard $B1 = Bn$ will be added to C if both events have the same parameter sets. In both single instance and multiple instances cases, if the parameter sets are not the same for the first and last events (*B1* and *Bn*) as a result of applying a replicator, we need to project out the non-common parameters by applying the *dom* operator. For example if *B1* has an additional parameter *p3* as a result of applying a replicator, the *par-replicator* guard added to *C*, will be $dom(B1) = Bn$. In the case a logical combinator is applied to the first and/or the last child of the refined *par-replicator*, we apply union and intersection of sets as needed.

Note that if the *par-replicator* is refined into a single event (i.e., $n \leqslant 1$) or a refining *xor* or *one* flows only, the guard will not be added to C, as the execution cannot be interrupted.

There is also another possibility that we only block the execution of the follow-on event C, until the refining event has occured. In this case the guard that will be added to C in Figure 4.11 will be $B1 = Br$, where Br as shown in Figure 4.12 is the refining event of B and $1 \leq r \leq n$, i.e., if *B1* has occurred C will be blocked until Br has occurred. However, we have decided to adopt the approach that blocks the *par-replicator* follow-on event until the completion of the *par-replicator* flow, this is because we do not have enough cases to demonstrate the necessity of the partial block of the follow-on event.

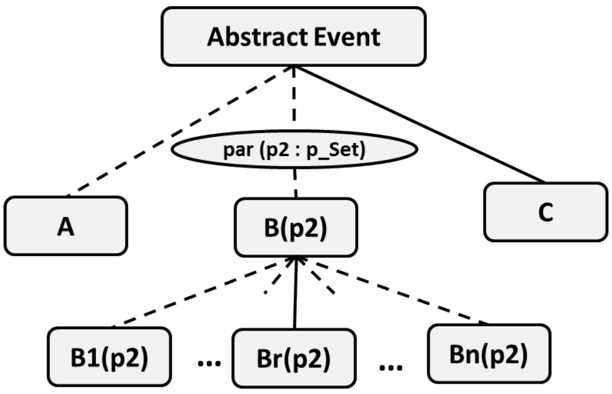


Figure 4.12: The *par-replicator* Refining Event

4.6 Non-Deterministic Loop

The compositional structure of ERS means that the *loop* is a structured loop, where there is only one entry point and one exit point. Originally ERS requires the *loop* to have only one child flow, to be executed zero or more times (Salehi Fathabadi et al., 2012). To give the loop more flexibility, we generalise the *loop* to have one or more child flows, with one sub-flow being chosen non-deterministically for execution in each iteration. The disabling and enabling conditions of P and Q of Figure 4.5 of the generalised form of the *loop* are still the same as the original *loop* shown in Table 4.1, as all the *loop* events will be disabled by the execution of the follow-on flow.

Diagrammatic Representation: A *loop* is represented by an asterisk within an oval, as shown in Figure 4.13. Similar to Figure 4.7, the non-connected lines indicate that we are only showing part of the flow. A *loop* can have one or more child events. Similar to the *par-replicator*, the *loop* events cannot be the last events of a flow, and they are always connected to their parent by a dashed line. The previous form of the *loop* is a special case of the general *loop* where the *loop* has only one child.

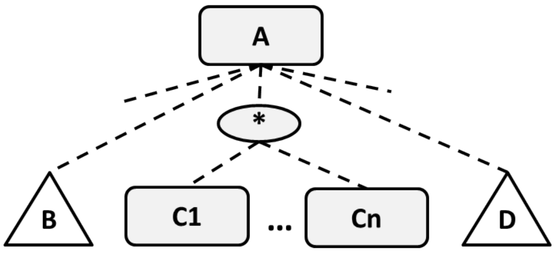
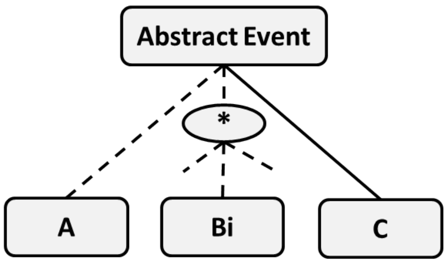


Figure 4.13: Diagrammatic Representation of the Generalised Loop

4.6.1 Non-Deterministic Loop Semantics

Figures 4.14 and 4.15 represent the Event-B encoding of the generalised form of the *loop* in the single instance case and the multiple instances cases respectively. The *loop* sub-events (Bi) do not require control variables and no action is required to record their execution. This is because of the zero execution case of the *loop*, which is also reflected by the sequencing invariant (inv_C_seq) and guard of the follow-on event C, which is similar to the *loop* sub-events depends on the execution of event A.



```
Invariants
  @inv_A_type A ∈ BOOL
  @inv_C_seq C = TRUE ⇒ A = TRUE
  @inv_C_gluing C = Abstract_Event
```

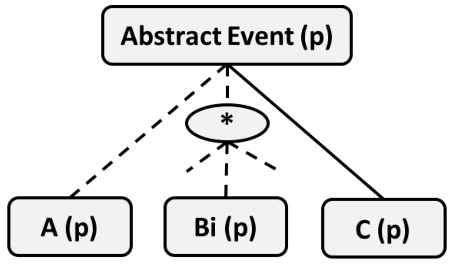
```
event A
   where
     @grd_self A = FALSE
   then
     @act A ≔ TRUE
  end
```

```
event C refines AbstractEvent
  where
    @grd_self C = FALSE
    @grd_seq A = TRUE
  then
    @act C = TRUE
end
```

```
event Bi  // 1 ≤ i ≤ n
  where
    @grd_seq  A = TRUE
    @grd_loop  C = FALSE
end
```

Figure 4.14: Single Instance Semantics of the Loop

The effect of applying a *loop* is the addition of the *grd_loop* to all the *loop* sub-events (Bi), which checks the execution of the follow-on event. In the single instance case the loop disabling guard is $(C = FALSE)$, and in the multiple instances case it is $(p \notin C)$



```
Invariants
  @inv_A_type A ⊆ Type(p)
  @inv_C_seq C ⊆ A
  @inv_C_gluing C = Abstract_Event
```

```
event A
   any p
   where
     @grd_self  p ∉  A
    then
     @act A ≔ A  ∪ {p}
  end
```

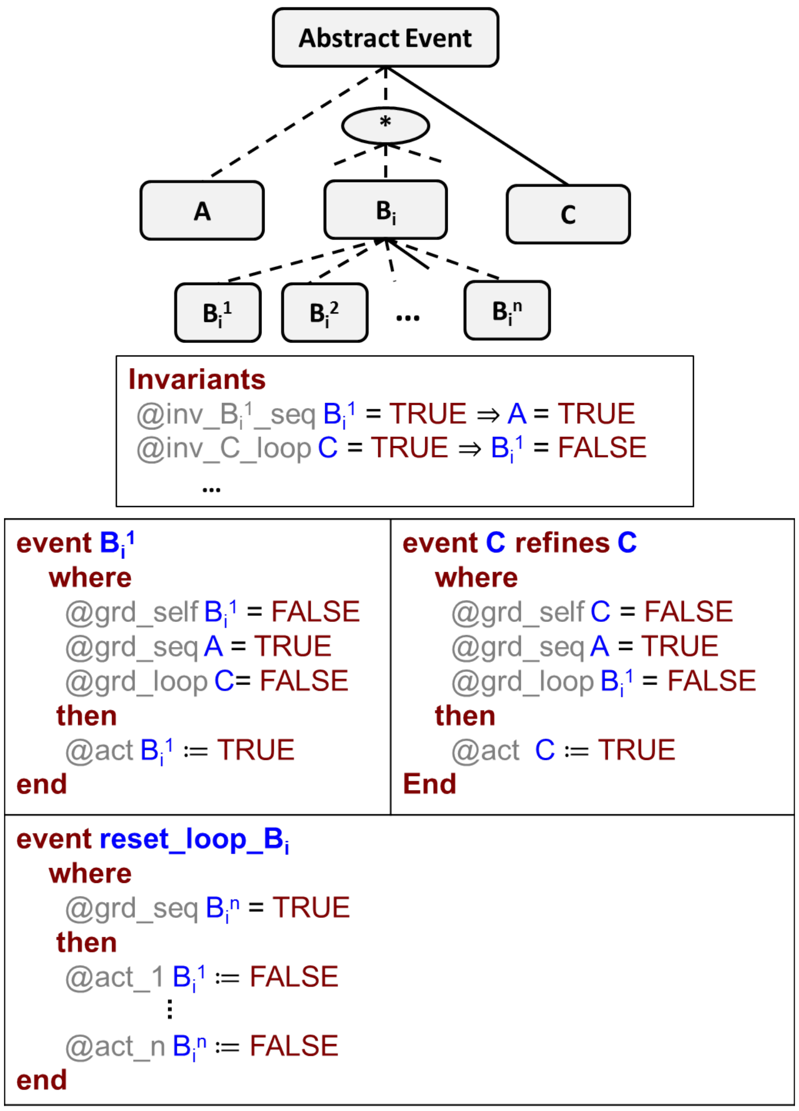
```
event C refines AbstractEvent
   any p
    where
     @grd_self p ∉  C
     @grd_seq p  ∈ A
    then
     @act C ≔ C ∪ {p}
  end
```

```
event Bi   // 1 ≤ i ≤ n
   where
     @grd_seq  p ∈ A
     @grd_loop  p ∉  C
end
```

Figure 4.15: Multiple Instances Semantics of the Loop

Refinement: If any of the *loop* sub-events is refined, the children of the refined sub-event will require control variables to ensure their proper execution order (Figure 4.16). These control variables need to be reset after their execution in order to enable more than one execution of the *loop*. Resetting the control variables is done in a separate event $(reset\_loop\_B_i)$ for each loop refined sub-event, after the execution of its last leaf event (B_i^n), as shown by the sequencing guard $(B_i^n = TRUE)$. The control variables of the refined *loop* will be reset by setting them to *FALSE* in the single instance case (Figure 4.16), while in the multiple instances case, the control variables are reset by removing the parameter instance from the control variable set $(B_i^i = B_i^i \setminus \{p\})$. As explained in Wiki.event-b.org (2012b), the resetting of control variables is not done in the last event of the loop child (B_i^n), in order to avoid complicated cases like the case when the last child is an *and-combinator* child.

Similar to the *par-replicator*, in order to avoid interrupting the *loop* refined child events in the middle of their execution, for each refined *loop* sub-event, we add a guard (grd_loop) and an invariant (inv_C_loop) to the *loop* follow-on event (C) checking that the first leaf



```
Invariants
  @inv_B_i^1_seq B_i^1 = TRUE ⇒ A = TRUE
  @inv_C_loop C = TRUE ⇒ B_i^1 = FALSE
     ...
```

```
event B_i^1
   where
     @grd_self B_i^1 = FALSE
     @grd_seq A = TRUE
     @grd_loop C= FALSE
   then
     @act B_i^1 ≔ TRUE
end
```

```
event C refines C
   where
     @grd_self C = FALSE
     @grd_seq A = TRUE
     @grd_loop B_i^1 = FALSE
   then
     @act  C ≔ TRUE
End
```

```
event reset_loop_B_i
   where
     @grd_seq B_i^n = TRUE
   then
     @act_1 B_i^1 ≔ FALSE
              ⋮
     @act_n B_i^n ≔ FALSE
end
```

Figure 4.16: Refinement of a Loop Child

event of the refined *loop* child has not executed. The *loop* guard in the follow-on event is ensured by the resetting of the control variables.

4.7 Conclusion

In this chapter we have extended the ERS approach to address the limitations presented in Section 4.2. Our extended ERS approach has given ERS the flexibility to model the dynamic behaviour of the *Producer-Consumer* pattern. The newly introduced replicator, the *par-replicator* can be used to model data dependent workflows where data is not predetermined and can change during the workflow execution. Such replication behaviour without a priori knowledge is not supported by many important workflow modelling approaches like BPMN and activity diagrams (Russell et al., 2006a).

Our parameter extension with the range expression is a natural extension of the parameter sets, as data cannot always be known statically in contexts, but in lots of cases other dynamic factors can determine the parameter sets, as we have seen in different examples where the parameter set are data dependent and can be determined by other executing events.

The extended ERS approach still does not support arbitrary loops, where the loops can have different entry and/or exit points like BPMN and activity diagrams can do with their flexible notations. Such arbitrary loops cannot be supported by ERS due to its compositional structure, however this does not mean we cannot make the loops more flexible than their original definition. Our generalised form of the *loop* supports non-deterministic execution of the *loop* events that have the same entry and exit points.

In this chapter we have also summarised the enabling and disabling conditions of the different control flow combinators of ERS, making it easier for the modeller to understand and select the required combinator. The ERS approach with its new dynamic extensions are applied to a case study, which will be presented in Chapter 5.

Chapter 5

Fire Dispatch Case Study

5.1 Introduction

In this chapter, we present the fire dispatch case study, which is an improved version of the case study presented in Dghaym et al. (2013) in order to support dynamic changes in the fire dispatch workflow. We model the fire dispatch workflow using the extended ERS approach, presented in Chapter 4, and Event-B. The complete version of the model is presented in Appendix A.

The aim of this case study is first to validate our ERS extensions by applying them to a complex workflow like the fire disptach workflow, and second to present our workflow modelling approach, which uses ERS to model control flow and Event-B to model data flow.

The combination of ERS and Event-B has reduced the complexity of this case study, with ERS supporting hierarchical decomposition and multiplicity in flows and sub-flows using the replication combinators, in addition to the simple mathematical language of Event-B, which made it easier to represent the different data structures, their relations and formulate and prove the consistency properties of the fire dispatch workflow.

5.2 Fire Dispatch System Overview

The main goal of the fire and rescue service is the public safety as well as the safety of the fire fighters. To achieve this goal, the fire service works on responding to fire or emergency incidents as fast as possible and in an expedient manner. This requires a 24/7 reliable dispatch system, as well as a robust communication system.

As described by Hantsfire.gov.uk (2000), response to emergency calls starts from the control room. The control operator workstation consists of 3 systems as shown in Figure 5.1.

- Integrated Communications Control System (ICCS): Which allows all telephone and radio communications.
- The Command and Control System (C and C) : Which is the main focus of our case study, this system is responsible for handling all the incident information, finding the nearest available resources for the incident, alerting the stations and fire fighters and monitoring the status of the appliances, so basically this system is the centre of the whole dispatch operation.
- The Geographical Information System (GIS): This has 6 different mapping views of the covered areas showing the stations, incidents and resources locations.

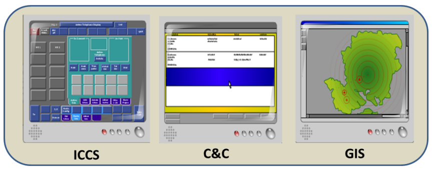


Figure 5.1: The Control Operator Workstation

These 3 systems can communicate and link together to facilitate the role of a control operator, but the way these systems communicate together is not studied in this case study. We focus on the dispatch system and how resources are allocated to incidents and the change of status of both resources and incidents from the start of an incident call until its closure.

5.3 Fire Dispatch System Requirements

In this section, we present the main functional requirements of the dispatch system that we are going to model. We group these requirements according to the problem they are going to solve into the following categories:

- Incident Information: The control operator tries to get information about the incident, which will allow the system to determine if it is a duplicate incident or not. These requirements are summarised in Table 5.1.

- Incident Handling: From the incident's information, the system must identify a possible action plan to handle the incident. The action plan identifies the resources required to handle the incident's type. Table 5.2 presents the incident handling requirements.

REQ_ID	**Description**
INF_1	The control operator must get information about the location and type of the incident
INF_2	Subsequent calls to an existing incident are referred to as duplicate calls
INF_3	A control operator can decide that an incident is duplicate at any point and there might be cases where an incident can be described as duplicate upon a resource arrival
INF_4	Each incident type is associated to a default priority level
INF_5	An operator can change the priority level of an incident

Table 5.1: Incident Information Requirements

REQ_ID	**Description**
HND_1	The system should identify the *predetermined action plan* (PDA) according to the location type and type of the incident
HND_2	The action plan must identify the resource types that need to attend the incident and their number
HND_3	The action plan can be changed by requesting more resources to attend the incident
HND_4	A stop message can be sent to the operator to stop sending resources to the incident when there is sufficient resources to deal with it, and this message should be sent to all resources that are not in attendance
HND_5	When proposing available resources to be allocated to the incident, this can include resources mobilised to lower priority incidents

Table 5.2: Incident Handling Requirements

- Resource Manipulation: The resource types to be allocated to an incident are identified by the action plan associated to that incident. The allocation of the resources to an incident is done by the operator, who chooses a resource from the system's suggestions. The system suggests resources according to their type, availability and how quick they can attend an incident. Table 5.3 summarises the resource manipulation requirements.

REQ_ID	**Description**
RES_1	If a resource is reallocated to a higher priority incident or becomes unavailable due to a break down for example, the system must alert the operator for a replacement resource
RES_2	Before a resource is in attendance, the system can keep looking for quicker resources that become available and suggest reallocating them to the operator
RES_3	A resource status can be available at station, mobilised to an incident, in attendance when it is at an incident location, or unavailable

Table 5.3: Resource Manipulation Requirements

- Incident Closure: In this workflow we restrict closing an incident to two cases, the case where an incident is handled by the fire service and the case where it is considered as a duplicate incident, which means it is handled by another incident call. Table 5.4 describes when an incident can be closed.

REQ_ID	**Description**
CLS_1	An incident must not be closed if any of the items in the action plan are not completed
CLS_2	An incident must not be closed if it still have allocated resources
CLS_3	An incident must not be closed if a stop message has not been sent
CLS_4	An incident should be closed if identified as a duplicate incident

Table 5.4: Incident Closure Requirements

5.4 Incremental Development of the Fire Dispatch Model

Using Refinement (Section 3.3.2) in modelling helps in reducing the complexity of the model and enhances our understanding of the problem by gradually adding more details to the model. Refinement is also supported by the hierarchical elaboration of the ERS diagrams. That is why when modelling the fire dispatch system we use refinement supported by the ERS diagrams.

The model of the fire dispatch system consists of five contexts to define the static part of the model, while the dynamic part of the model consists of an abstract and seven refinement levels as follows:

- **Abstract Level:** Models creation, information gathering and identifying the incident as duplicate or not.
- **First Refinement:** Introduces the closure of duplicate incidents and management of non-duplicate ones.
- **Second Refinement:** Introduces the ability to update the action plan during incident management.
- **Third Refinement:** Models the ability to update required resources during incident handling.
- **Fourth Refinement:** Introduces further incident details such as incident location and type.
- **Fifth Refinement:** Refines incident handling, showing how a resource can be allocated and deallocated from an incident.
- **Sixth Refinement:** Introduces physical resources, which are assigned to handle incidents and their relation with the required resources of an incident action plan.
- **Seventh Refinement:** Adds further details about the location and the duration of a physical resource to the incident.

5.4.1 Abstract and First Refinement: Duplicate and Non-Duplicate

When modelling the fire dispatch system using ERS diagrams, we apply multiple instances modelling to enable possibly simultaneous dispatch to different incidents. Multiple instances modelling is indicated by the addition of the parameter i, representing incidents, to the root of the diagram (Figure 6.11). The set *INCIDENT* is defined in the context, representing all potential incidents.

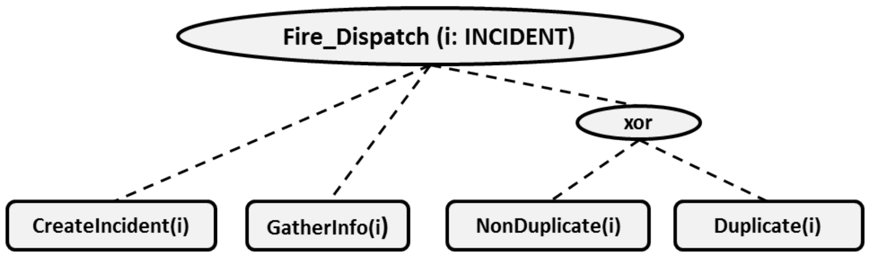
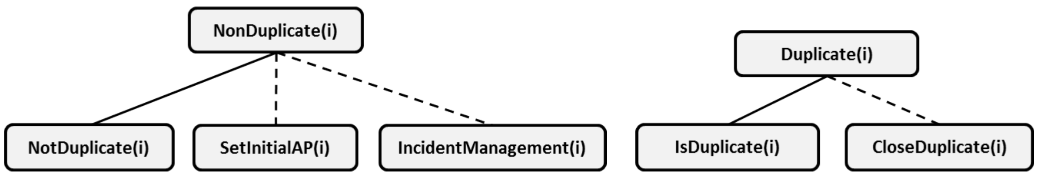


Figure 5.2: ERS Diagram for the Fire Dispatch Abstract Level

At the abstract level (Figure 6.11), we distinguish between two cases of incident handling which are duplicate and non-duplicate incidents as indicated by the application of the *xor-combinator*. The decision about the possibility of the incident being duplicate is determined after getting information (*GatherInfo*) about the incident during the incident call, which is initiated at the beginning by the *CreateIncident* event.

Having two different types of incidents (non-duplicate and duplicate) will result in two different paths when dealing with the incident, the normal path when the incident is not duplicate and the exceptional path when the incident is duplicate. This is shown in the refinement of the events *Non-Duplicate* and *Duplicate* in Figure 5.3.



(a) Non-Duplicate Refinement (b) Duplicate Refinement

Figure 5.3: Refinement of Non-Duplicate and Duplicate Events

In the case of non-duplicate incident (Figure 5.3(a)), an initial plan must be determined (*SetInitialAP*) to manage the incident (*IncidentManagement*), while in the case of a duplicate incident (Figure 5.3(b)), the incident will be simply closed (*CloseDuplicate*). Up to this stage we have not introduced any data varaibles or any additional constraints manually, and both machines of the abstract and first refinement are generated automatically from the ERS diagrams.

5.4.2 Second and Third Refinement: Incident Management

Once it is established that an incident is not duplicate, an initial action plan is established. The action plan is followed through during the incident management. However, in some cases the initial action plan can be updated during the incident management to request additional resources. Such behaviour is a parallel *producer-consumer* pattern, which was introduced in Chapter 4. In Figure 5.4 we show the refinement of the *IncidentManagement* event of the Non-Duplicate case (Figure 5.3(a)) using the parallel *producer-consumer* pattern. In this case the event *RequestAdditionalRes* will produce more resources to be added to the action plan, while *HandleIncident* will consume the resources in the action plan.

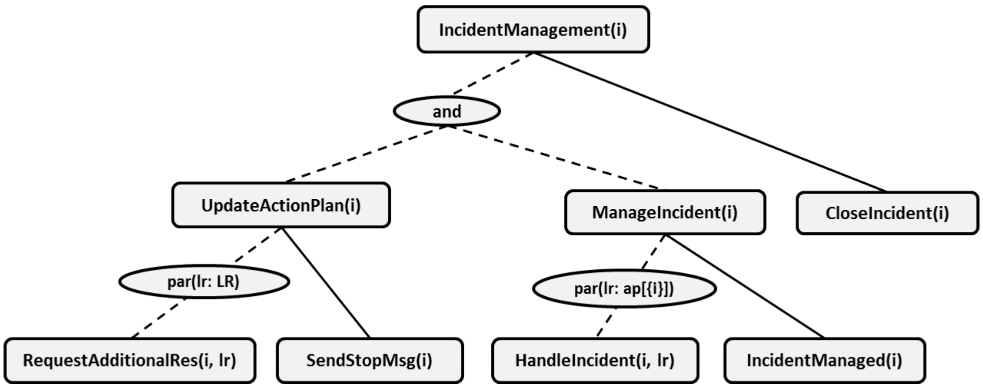


Figure 5.4: Application of the Parallel *Producer-Consumer* Pattern for Incident Management

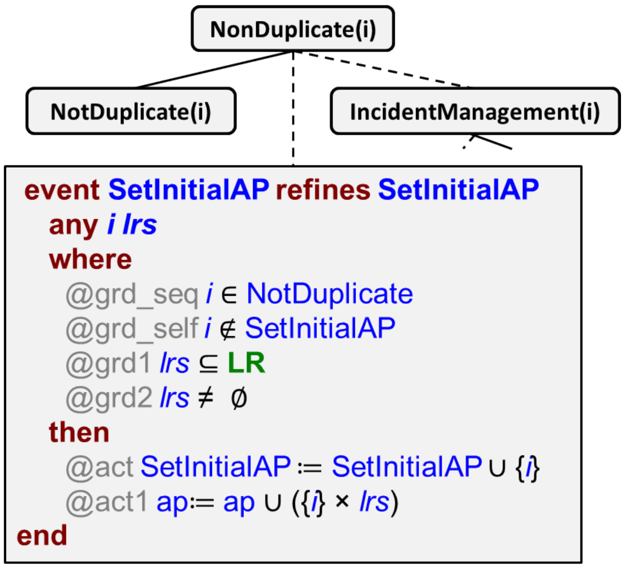
In the third refinement, we extend the context with the carrier set LR, which represents the logical resources. In the fire dispatch case study, we make a distinction between logical resources (LR) and physical resources (PR). Logical resources represent the resources required by the action plan to handle the incident, e.g., an incident may have several different logical resources with the property *class 3 fire engine*. Physical resources will be introduced in Section 5.4.4, they are the actual resources that are going to be assigned to the incident, e.g., a specific fire engine. A physical resource, unlike a logical resource, can be only allocated to one incident. In our model we use logical resources for the action plan and physical resources for the resource allocation to the incident.

The action plan of the incident is represented by the data variable ap, which is added manually in the third refinement of the model and is defined using the following invariants:

$$\textbf{inv\_ap\_type: } ap \in INCIDENT \leftrightarrow LR$$
$$\textbf{inv\_ap\_seq: } dom(ap) = SetInitialAP$$

The first invariant (*inv_ap_type*) defines the type of the data variable *ap*, which is a relation between incidents and logical resources (LR). The variable *ap* is a relation because an incident can be associated to different logical resources and a logical resource can be associated to different incidents. In the second invariant, we are relating the data variable *ap* to the control variable *SetInitialAP*, this is to ensure that setting the action plan (*ap*) cannot start before the *SetInitialAP* event, and when this event occurs then the incident must have an action plan (*ap*). This invariant (*inv_ap_seq*) shows how we can combine data variables (*ap*) with control variables (*SetInitialAP*) to represent and enforce the requirements using invariants.

In Figure 5.5, we present the Event-B specification of *SetInitialAP* event after introducing the data variable *ap* in the third refinement.

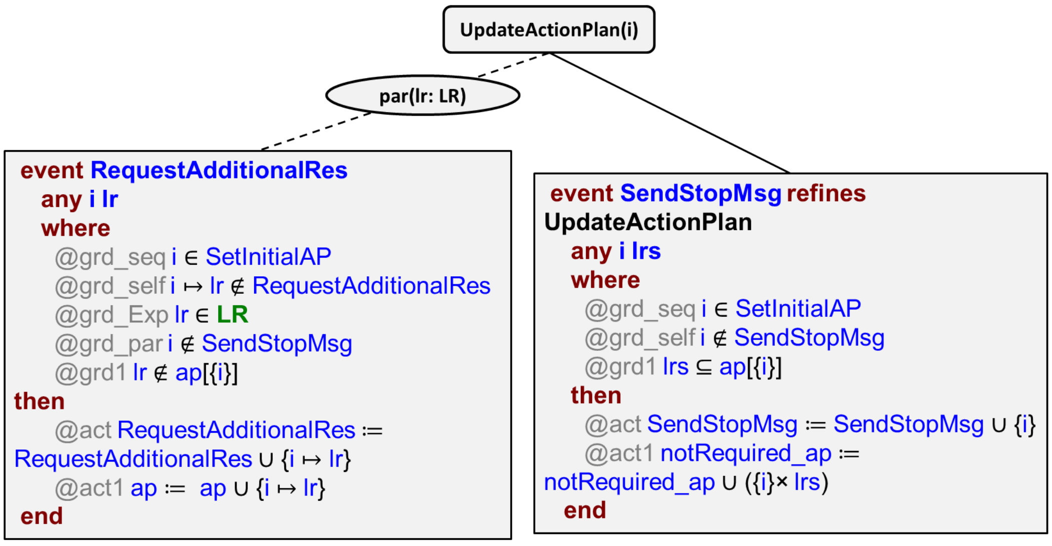


```
event SetInitialAP refines SetInitialAP
  any i lrs
  where
    @grd_seq i ∈ NotDuplicate
    @grd_self i ∉ SetInitialAP
    @grd1 lrs ⊆ LR
    @grd2 lrs ≠ ∅
  then
    @act SetInitialAP ≔ SetInitialAP ∪ {i}
    @act1 ap≔ ap ∪ ({i} × lrs)
end
```

Figure 5.5: Setting the Action Plan (*ap*) in *SetInitialAP* Event Using Event-B

In Figure 5.5, the numbered guards (*grd1*, *grd2*) and action (act1) are added manually, while the rest are generated from the ERS diagram. The data variable *ap* is initialised to the empty set, and then the *SetInitialAP* sets the action plan of the incident i for the first time in *act1* to the set of logical resources *lrs*. At this point *lrs* is defined as any set of logical resources which is not empty (*grd1* and *grd2*). The set *lrs* will be defined more precisely in Section 5.4.3.

Regarding the producer part of Figure 5.4, the Event-B specifications of its events are shown in Figure 5.6. After setting the initial action plan, it is possible to update the action plan (*ap*), by requesting additional resources. This is done manually by updating *ap* in *act1* of *RequestAdditionalRes*. The last guard (*grd1*) of *RequestAdditionalRes* is added manually. The guard $lr \notin ap[\{i\}]$ is to ensure that the newly added resource is a fresh resource, where $ap[\{i\}]$ is the relational image of ap with respect to $\{i\}$ i.e., $ap[\{i\}]$ is the set of logical resources required for incident i.



```
event RequestAdditionalRes
  any i lr
  where
    @grd_seq i ∈ SetInitialAP
    @grd_self i ↦ lr ∉ RequestAdditionalRes
    @grd_Exp lr ∈ LR
    @grd_par i ∉ SendStopMsg
    @grd1 lr ∉ ap[{i}]
then
    @act RequestAdditionalRes ≔
RequestAdditionalRes ∪ {i ↦ lr}
    @act1 ap ≔  ap ∪ {i ↦ lr}
end
```

```
event SendStopMsg refines
UpdateActionPlan
  any i lrs
  where
    @grd_seq i ∈ SetInitialAP
    @grd_self i ∉ SendStopMsg
    @grd1 lrs ⊆ ap[{i}]
  then
    @act SendStopMsg ≔ SendStopMsg ∪ {i}
    @act1 notRequired_ap ≔
notRequired_ap ∪ ({i}× lrs)
  end
```

Figure 5.6: Event-B Specifications of the Producer Events

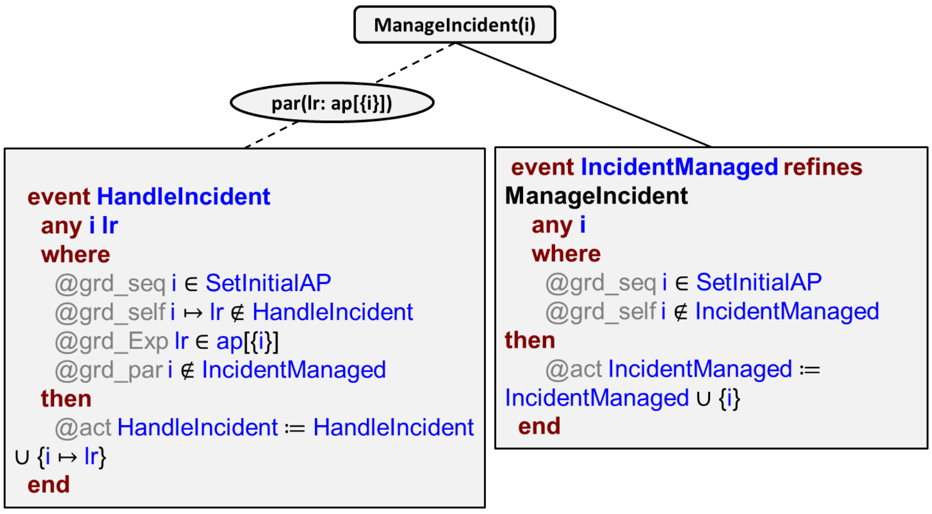
The reason for not combining the guard $lr \notin ap[\{i\}]$ as part of the range expression of the *par-replicator*, i.e., having the *grd_Exp* of the parameter set as $lr \in (LR \setminus ap[\{i\}])$, is the “non-decreasing” restriction of the *par-replicator* range expression.

Requesting additional resources is disabled by sending a stop message, which is sent by a system operator to indicate that there are sufficient resources to deal with the incident, in this case it is the follow-on event of the *par-replicator* (*SendStopMsg*). When a stop message is sent, then it means the incident is satisfied by whatever resources it already had and no more resources are required to attend, even if there are still more logical resources in the action plan that are not satisfied yet. That is why we also introduce the data variable *notRequired_ap* which is defined by the invariant: $notRequired\_ap \subseteq ap$. This data variable will be updated with the logical resources of the action plan (ap) that are not needed any more to manage the incident, as shown by the manually added action (*act1*) of *SendStopMsg*, which could be also empty. The logical resources that can be added to *notRequired_ap*, which are defined by *grd1* as any set of logical resources in the action plan of the incident, will be defined more precisely in Section 5.4.4.

Handling the incident will consume logical resources in the action plan of the incident ($ap[\{i\}]$). In Figure 5.7, we show the Event-B encodings of the consumer events (*HandleIncident* and *IncidentManaged*). All the guards and actions associated with the consumer events are generated automatically from the ERS diagram.

5.4.3 Fourth Refinement: Incident Details

In the fourth refinement level, we introduce more details about the incident by refining *GatherInfo* to introduce the type and the location of the incident, Figure 5.8. We



```
event HandleIncident
  any i lr
  where
    @grd_seq i ∈ SetInitialAP
    @grd_self i ↦ lr ∉ HandleIncident
    @grd_Exp lr ∈ ap[{i}]
    @grd_par i ∉ IncidentManaged
  then
    @act HandleIncident ≔ HandleIncident
∪ {i ↦ lr}
  end
```

```
 event IncidentManaged refines
ManageIncident
  any i
  where
    @grd_seq i ∈ SetInitialAP
    @grd_self i ∉ IncidentManaged
then
    @act IncidentManaged ≔
IncidentManaged ∪ {i}
 end
```

Figure 5.7: Event-B Specifications of the Consumer Events

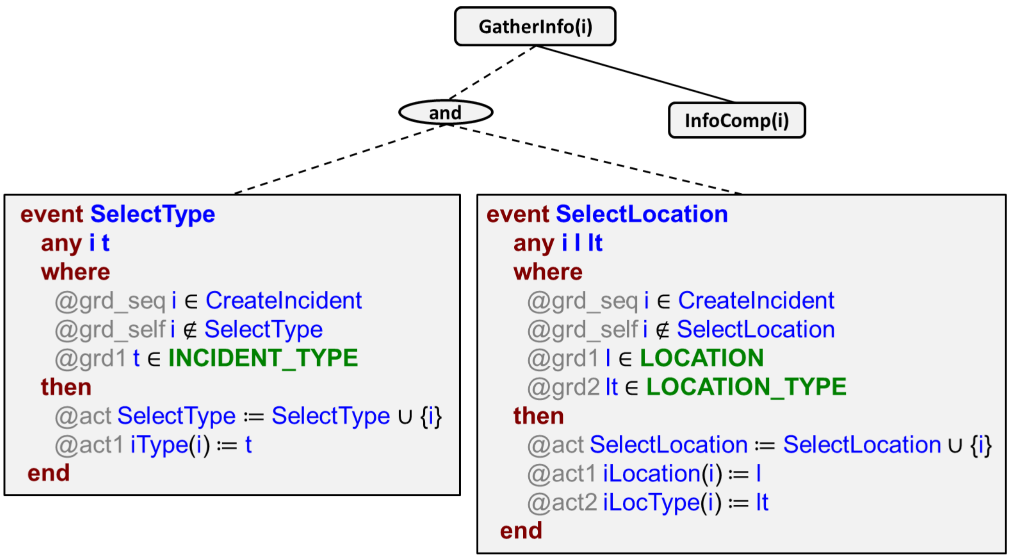
extend the context to include the carrier sets *INCIDENT_TYPE*, *LOCATION* and *LOCATION_TYPE* representing the possible types, locations and location types of the incident. We also introduce three data variables *iType*, *iLocation* and *iLocType* to associate an incident with a specific type, location and location type. These data variables are defined as functions in the invariants *inv_iType*, *inv_iLocation* and *inv_ilocType* to ensure that an incident can have one type, one location and one location type. Their domains are equal to the control variable *SelectType* and *SelectLocation* to ensure that these events will set the type and the location of the incident respectively. This is done in the manually added actions *act1* and *act2* of both events *SelectType* and *SelectLocation*, Figure 5.8.

$$\textbf{inv\_iType: } iType \in SelectType \rightarrow INCIDENT\_TYPE$$

$$\textbf{inv\_iLocation: } iLocation \in SelectLocation \rightarrow LOCATION$$

$$\textbf{inv\_ilocType: } iLocType \in SelectLocation \rightarrow LOCATION\_TYPE$$

The requirements state that the type and the location type of the incident will help determine the action plan of the incident. This is done by introducing the predetermined action plan *PDA* constant. *PDA* is defined in the context by axioms *axm_PDA_1* and *axm_PDA_2*. The first axiom defines *PDA* as a total function between the incident type and location type on one hand and the power set of logical resources i.e., $PDA(t, l)$ is the set of logical resources required for an incident of type t at location type l. The second axiom is to ensure the different types and location types are not assigned to the



```
event SelectType
  any i t
  where
    @grd_seq i ∈ CreateIncident
    @grd_self i ∉ SelectType
    @grd1 t ∈ INCIDENT_TYPE
  then
    @act SelectType ≔ SelectType ∪ {i}
    @act1 iType(i) ≔ t
 end
```

```
event SelectLocation
  any i l lt
  where
    @grd_seq i ∈ CreateIncident
    @grd_self i ∉ SelectLocation
    @grd1 l ∈ LOCATION
    @grd2 lt ∈ LOCATION_TYPE
  then
    @act SelectLocation ≔ SelectLocation ∪ {i}
    @act1 iLocation(i) ≔ l
    @act2 iLocType(i) ≔ lt
 end
```

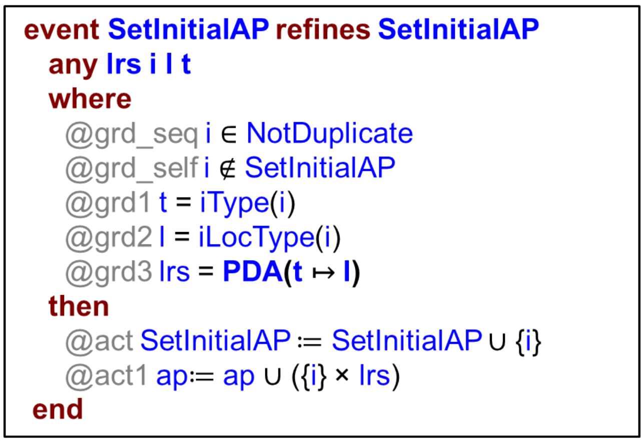
Figure 5.8: Refinement of *GatherInfo* Introducing Type and Location

empty set.

axm_PDA_1: $PDA \in (INCIDENT\_TYPE \times LOCATION\_TYPE) \rightarrow \mathbb{P}(LR)$

axm_PDA_2: $\forall it, loc \cdot it \in INCIDENT\_TYPE \wedge loc \in LOCATION\_TYPE \implies PDA(it \mapsto loc) \neq \phi$

After introducing *PDA*, the *SetInitialAP* event will be refined to set the action plan (ap), according to the type and location type of the incident as can be seen from *grd1*, *grd2*, *grd3* and *act1* of Figure 5.9.



```
event SetInitialAP refines SetInitialAP
  any lrs i l t
  where
    @grd_seq i ∈ NotDuplicate
    @grd_self i ∉ SetInitialAP
    @grd1 t = iType(i)
    @grd2 l = iLocType(i)
    @grd3 lrs = PDA(t ↦ l)
  then
    @act SetInitialAP ≔ SetInitialAP ∪ {i}
    @act1 ap≔ ap ∪ ({i} × lrs)
 end
```

Figure 5.9: Refinement of the Initial Action Plan According to Type and Location

Another extension refinement related to the location of the incident is done to the *IsDuplicate* event, where we manually add some guards to satisfy the newly added invariant:

$$\forall i \cdot i \in IsDuplicate \rightarrow (\exists i0 \cdot i0 \in NotDuplicate \wedge iLocation(i) = iLocation(i0))$$

This invariant means for an incident to be considered as duplicate, there must be at least one incident with the same location.

5.4.4 Fifth, Sixth and Seventh Refinement: Physical Resources

In the previous sections, we only dealt with logical resources. In the following refinements we start by refining the *HandleIncident* event showing the different steps in handling an incident, then introducing physical resources which are related to the different stages in handling an incident, finally we add more details related to physical resources.

The *HandleIncident* flow is shown in Figure 5.10, illustrating how a logical resource *lr* is managed for incident *i*. Once a physical resource is allocated to a logical resource by the *AllocateRes* event, it is possible for two changes to occur to the resource any number of times prior to the resource attending the incident or receiving a stop a message: either the physical resource is replaced by a quicker one that becomes available or the physical resource location is updated. Finally after a resource attends the incident and completes its task or if the resource receives a stop a message, the resource will be deallocated from the incident in *DeallocateRes* event.

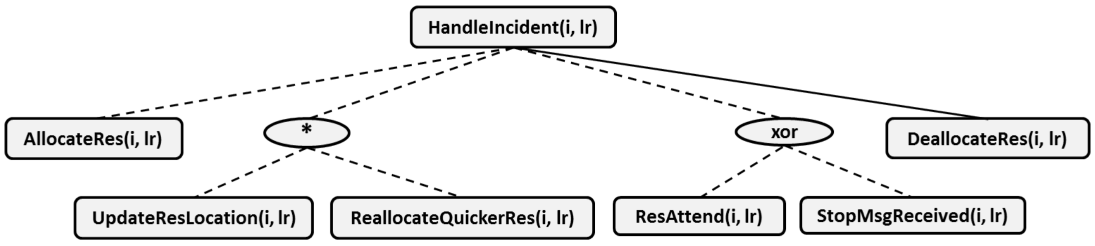


Figure 5.10: Different Steps of Incident Handling

After introducing *ResAttend*, it is now possible to specify the action plan resources that are not required to be completed (*notRequired_ap*) in *SendStopMsg* more precisely, by updating its manually added guard (*grd1*) of Figure 5.6 to become:

$$lrs = ap[\{i\}] \setminus ResAttend[\{i\}]$$

This update is to ensure a stop message is sent to all resources that are part of the action plan, but not in attendance yet, letting them know that the incident is satisfied with the resources currently attending the incident location and no more resources are required to handle the incident.

In the sixth refinement, physical resources are introduced by the data variables *alloc* and *alloc_LR*. The following invariants describe *alloc* and *alloc_LR* and the relationship between them.

inv1: $alloc \in PR \rightarrowtail\!\!\!\!\!\!+ \; INCIDENT$

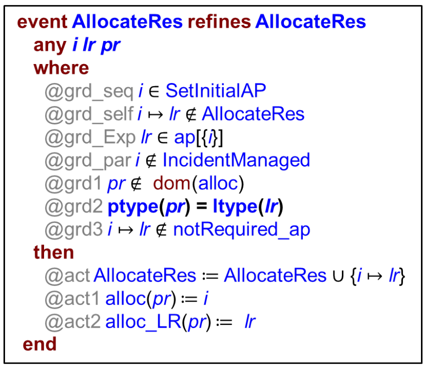
inv2: $ran(alloc) \subseteq SetInitialAP$

inv3: $alloc\_LR \in PR \rightarrowtail\!\!\!\!\!\!+ \; LR$

inv4: $\forall pr, lr . pr \in dom(alloc\_LR) \wedge alloc\_LR(pr) = lr \Rightarrow ptype(pr) = ltype(lr)$

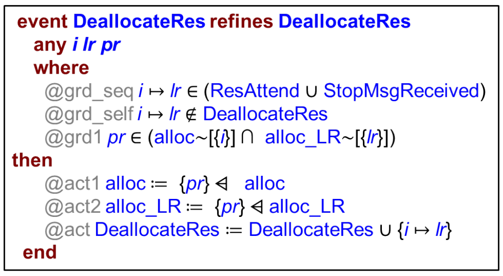
inv5: $dom(alloc) = dom(alloc\_LR)$

The data variable *alloc* maps physical resources to the incident they are allocated to, *inv1*. The variable *alloc* is defined as a partial function so that each physical resource can be assigned to at most one incident. The incident must have an action plan which is why we introduce *inv2* to ensure that incidents are in *SetInitialAP*. Invariant *inv3* defines *alloc_LR*, which assigns a logical resource to a physical resource. This variable will help us in keeping track of which logical resource in the action plan, the physical resource is assigned to. Finally, *ptype* and *ltype* are constants defined in the context to describe the type of the physical and logical resources respectively. The last two invariants ensure that the type of a physical resource must be the same as the type of the logical resource assigned to it, and the equality of the domains ensures that all physical resources allocated to an incident are associated with a logical resource.



```
event AllocateRes refines AllocateRes
  any i lr pr
  where
   @grd_seq i ∈ SetInitialAP
   @grd_self i ↦ lr ∉ AllocateRes
   @grd_Exp lr ∈ ap[{i}]
   @grd_par i ∉ IncidentManaged
   @grd1 pr ∉  dom(alloc)
   @grd2 ptype(pr) = ltype(lr)
   @grd3 i ↦ lr ∉ notRequired_ap
  then
   @act AllocateRes ≔ AllocateRes ∪ {i ↦ lr}
   @act1 alloc(pr) ≔ i
   @act2 alloc_LR(pr) ≔  lr
 end
```

(a) Allocation of a Physical Resource *pr*



```
event DeallocateRes refines DeallocateRes
  any i lr pr
  where
   @grd_seq i ↦ lr ∈ (ResAttend ∪ StopMsgReceived)
   @grd_self i ↦ lr ∉ DeallocateRes
   @grd1 pr ∈ (alloc~[{i}] ∩  alloc_LR~[{lr}])
then
   @act1 alloc ≔  {pr} ⩤  alloc
   @act2 alloc_LR ≔  {pr} ⩤ alloc_LR
   @act DeallocateRes ≔ DeallocateRes ∪ {i ↦ lr}
 end
```

(b) Deallocation of a Physical Resource *pr*

Figure 5.11: Allocation and Deallocation of a Physical Resource *pr* in Event-B

In Figure 5.11, the event *AllocateRes* will add new values to *alloc* and *alloc_LR*, whereas *DeallocateRes*, Figure 5.11, will remove physical resources from *alloc* and *alloc_LR*. Therefore, *DeallocateRes* will ensure that an incident cannot be closed if it still has allocated physical resources. This is enforced by invariant *inv6* which states that an incident whether duplicate or not can be only closed if it has no physical resources allocated to it. The following invariant, *inv7*, ensures that all required resources have completed their tasks and have been deallocated before the incident can be closed. This is done by checking that all logical resources in the action plan of the incident i, that are not marked as not required as a result of a send stop message, are part of the relational image of *DeallocateRes* with respect to incident i.

inv6: $\forall i.i \in (CloseIncident \cup CloseDuplicate) \Rightarrow i \notin ran(alloc)$

inv7: $\forall i.i \in CloseIncident \Rightarrow ap[\{i\}] \setminus notRequired\_ap[\{i\}] \subseteq DeallocateRes[\{i\}]$

These invariants show another example of combining control variables with data variables.

In the last refinement level, we extend the model with the location of the physical resource and the duration it needs to get to the incident by introducing the data variables rLocation and duration as follows:

inv_rLocation: $rLocation \in PR \rightarrow LOCATION$

inv_duration: $duration \in (LOCATION \times LOCATION) \rightarrow \mathbb{N}$

Initially the physical resources will be associated with some stations, and their location will be set to the station location. The event *UpdateResLocation* will be extended to update the location of the resource, i.e., update the data variable *rLocation* of *pr*. We also extend the event *ResAttend* with an extra guard to ensure the location of the resource is the same as that of the incident, i.e., $rLocation(pr) = iLoction(i)$ where pr is a resource allocated to the incident i. The *duration*, which represents the time needed to get from one location to another, is mainly needed when reallocating a new quicker resource to the incident as shown in Figure 5.12, where *grd5* checks that the duration of the new physical resource pr to be allocated to incident i, is less than the initially allocated resource *pr0*.

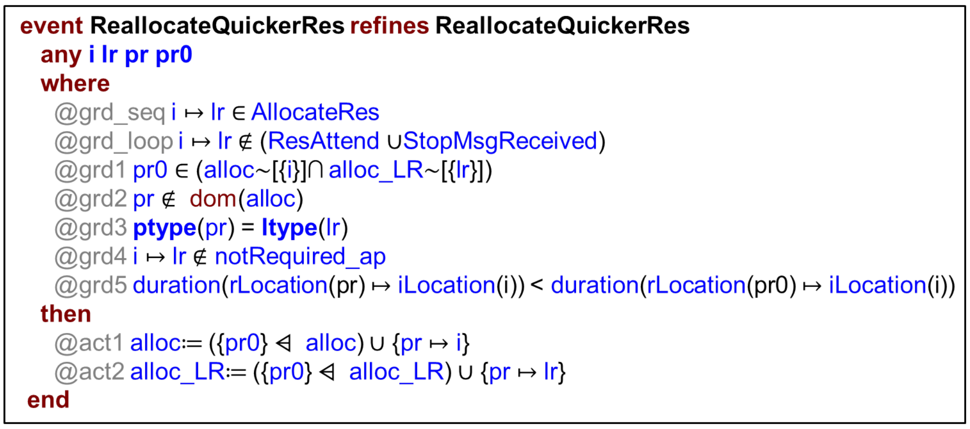

```
event ReallocateQuickerRes refines ReallocateQuickerRes
  any i lr pr pr0
  where
   @grd_seq i ↦ lr ∈ AllocateRes
   @grd_loop i ↦ lr ∉ (ResAttend ∪StopMsgReceived)
   @grd1 pr0 ∈ (alloc~[{i}]∩ alloc_LR~[{lr}])
   @grd2 pr ∉  dom(alloc)
   @grd3 ptype(pr) = ltype(lr)
   @grd4 i ↦ lr ∉ notRequired_ap
   @grd5 duration(rLocation(pr) ↦ iLocation(i)) < duration(rLocation(pr0) ↦ iLocation(i))
  then
   @act1 alloc:= ({pr0} ⩤  alloc) ∪ {pr ↦ i}
   @act2 alloc_LR:= ({pr0} ⩤  alloc_LR) ∪ {pr ↦ lr}
 end
```

Figure 5.12: Reallocation of a Quicker Resource to the Incident

In Figure 5.12 the only Event-B parts that are generated from the ERS diagram in Figure 5.10 are the first two guards *grd_seq* and *grd_loop*, while the rest of the guards and actions are manually added. *grd1* checks that *pr0* is a physical resource allocated to incident i and corresponds to the logical resource *lr*. Guards, *grd2* and *grd3*, ensure that the new physical resource *pr* to be allocated to incident *i* is not allocated to another incident and has the same type as the logical resource *lr*. *grd4* checks that the logical resource *lr* is still required for incident *i*, otherwise there is no need for reallocation. The actions will swap the allocation of the physical resource *pr0* to *i* with the new physical resource *pr*. This is another example of how we used ERS to manage the control flow of events, while the data flow is done manually using Event-B.

5.5 Evaluation of the Fire Dispatch Case Study

We evaluate the model presented in Section 5.4, by tracing the requirements in Table 5.5 to validate that the model (Section 5.4) incorporates all the requirements presented in Section 5.3, and to assess the suitability of the ERS features for the case study. In this requirements trace we show at which refinement levels the requirements were mainly addressed and the directly related data variables and events.

Table 5.5: Requirements Tracing of the Fire Dispatch Case Study

Req_ID	**Machine**	**Data Variables**	**Events and Control Variables**
INF_1	M0, M4	iType, iLocation, iLocType	GatherInfo, SelectLocation, SelectType
INF_2	M0		NonDuplicate
HND_1	M2, M4	PDA	SetInitialAP
HND_2	M3	ap	SetInitialAP
HND_3	M2, M3	ap	UpdateActionPlan, RequestAdditionalRes
HND_4	M3, M4	notRequired_ap	SendStopMsg
RES_2	M5, M7	duration, rLocation	ReallocateQuickerRes, UpdateResLocation, ResAttend
CLS_1	M3, M5, M6	alloc, alloc_LR, ap	HandleIncident, CloseIncident, AllocateRes, DeallocateRes
CLS_2	M3, M5, M6	alloc, alloc_LR	CloseIncident, DeallocateRes, CloseDuplicate
CLS_3	M2		SendStopMsg
CLS_4	M1		IsDuplicate, CloseDuplicate

Comparing Table 5.5 with the requirements tables in Section 5.3, we have the following requirements missing from the model: *INF_3*, *INF_4*, *INF_5*, *HND_5*, *RES_1* and *RES_3*.

Regarding requirement *INF_3*, we introduced the possibility of having duplicate incidents, but we restricted the decision making of a duplicate incident to be only done after gathering information about the incident, and this decision cannot be made if an incident is marked as *NonDuplicate*, as indicated by the application of the *xor-combinator* in Figure 5.2. Using the *xor-combinator* we have *not* modelled the possibility of interrupting an incident handling if the duplication is discovered late as requirement *INF_3* gives the flexibility of making such decision at any stage, even after the arrival of a resource to the incident. The reason for this restriction is that in ERS we do not have combinators that can interrupt a flow especially in this case the interruption can possibly happen at different places during the incident handling which consist of many events as shown in Figure 5.10. Using the existing ERS combinators this can be done by using multiple *xor-combinators* to show the different possibilities, which can lead to a complex model, that is why we did **not** fully model requirement *INF_3*.

Requirements *INF_4* and *INF_5* can be easily modelled with the existing ERS combinators, but at this stage we decided to postpone the introduction of the incident priority due to its effect on resource allocation as required by *HND_5* and *RES_1*. Requirement *HND_5* states that we can allocate a resource to an incident even if it is allocated to another incident on the condition that reallocation takes place from a lower priority incident to a higher priority incident. Furthermore, *RES_1* requires replacing the resource in the lower priority incident or in any incident if its resource becomes unavailable due to a breakdown for example.

Requirement *HND_5* can be easily modelled by changing the guard on resource allocation using Event-B to allow allocating from a lower priority incident to a higher priority incident, but the problem is in requirement *RES_1*. In order to replace a lost resource due to reallocation or breakdown, we need to repeat some events again, such as *AllocateRes*. The problem in ERS is once an event executes for an instance *i*, it cannot execute again for the same instance due to the disabling guard *grd_self*, unless we use a *loop*. The problem in using a *loop* in this case, is the control variables will reset every time, whereas here resetting is only needed in the case a replacement is needed. Another option is to do the resetting of control variables manually, but the manual modification of the control variables is *not* recommended in ERS and could lead to lots of problems due to the sequencing dependencies between events, hence, we decided to postpone modelling this behaviour at this stage. We also decided to postpone modelling requirement *RES_3* because we have not introduced the unavailability of a resource which is also related to requirement *RES_1*.

This shows the limitation in the ERS approach when dealing with interruptions and cancellation of behaviour. These limitations will be addressed in Chapter 6.

By overcoming the previously mentioned limitations in ERS, we managed to model most of the requirements of the fire dispatch workflow as shown in Table 5.5. We have also shown how in many cases we combined control variables with data variables to model the requirements, for instance invariants *inv6* and *inv7* use both control variables (*CloseIncident*, *DeallocateRes* ...), and data variables (*alloc*, *ap* ...) to enforce requirements *CLS_1* and *CLS_2*.

The approach we followed in modelling workflows, where we do the control flow using ERS diagrams and leave the data handling to Event-B, seems successful as this leads to simpler diagrams and the generated model can be manually extended with data variables without causing problems. The *ReallocateQuickerRes* event in Figure 5.12 is an example of how we combined both ERS generated Event-B elements with manually added elements.

In addition to that, we managed to relate the data variables to the control variables using invariants, which helps in knowing when or at which stage the data variable will be available. For example the invariant ($dom(ap) = SetInitialAP$), indicates that when the *SetInitialAP* event occurs then the data variable *ap* will be set, or in other words the incident will have an action plan. This would not have been possible, if we did not have explicit control variables associated to the events.

In this case study we have also applied the ERS extensions introduced in Chapter 4. We used the parallel *producer-consumer* pattern in Figure 5.4 to show how additional resources can be requested while managing an incident, this pattern provides flexibility when dealing with dynamic variables such as the action plan of an incident ($ap[\{i\}]$).

Requirement *CLS_1* which requires that all the tasks in the action plan be satisfied before closing an incident suggests that the *all-replicator* is a possible candidate to represent this behaviour. However, we applied the *par-replicator* due to the dynamic nature of the variable set. We managed to model *CLS_1*, using a combination of control and data variables as shown by *inv7*. This invariant ensures that for an incident to be closed, all the resources that are still required by the action plan are part of the deallocated resources.

If we replaced the *par-replicator* with an *all-replicator*, we will end up with an invariant stronger than *inv7* which cannot be satisfied. This is because the *all-replicator* will require the equality of all the items in the action plan with the deallocated resources before considering an incident as managed, which is not always possible. The problem is in the case when some unallocated resources in the action plan are no longer required due to sending a stop message, hence they cannot be allocated and consequently cannot be deallocated. However, we managed to express the requirement by the manually added invariant (*inv7*), but in a way that can be satisfied by the model.

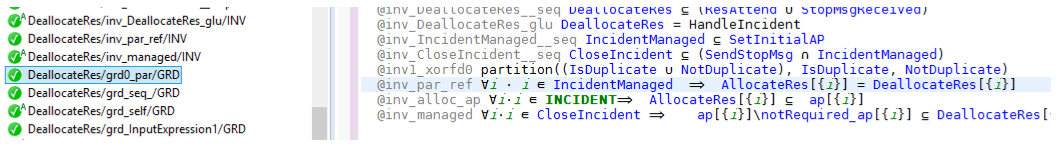
We have also used the parameter extension, as we have seen in the case study, where the action plan cannot be determined statically, it depends on the data collected (location and type) by the events, hence the extension of the parameter was needed in this case. The *loop* generalisation was also used in the case of updating the resource location and allocating a quicker resource of the incident (Figure 5.10).

In Event-B, a variable introduced at a refinement level i can only be modified in subsequent refinement levels by events that refine events of level i. For example, the data variable *ap* in the fire dispatch workflow has been updated and changed by several events, so we postponed the introduction of this application-specific variable until all the events that change its value were introduced. The overall ERS diagram of the dispatch system has helped us to make such decision.

In addition to that, the explicit invariants generated by ERS help in discharging proof obligations. Take for example *M5*:

$$\textbf{inv\_par\_ref: } \forall i \cdot i \in IncidentManaged \implies AllocateRes[\{i\}] = DeallocateRes[\{i\}]$$

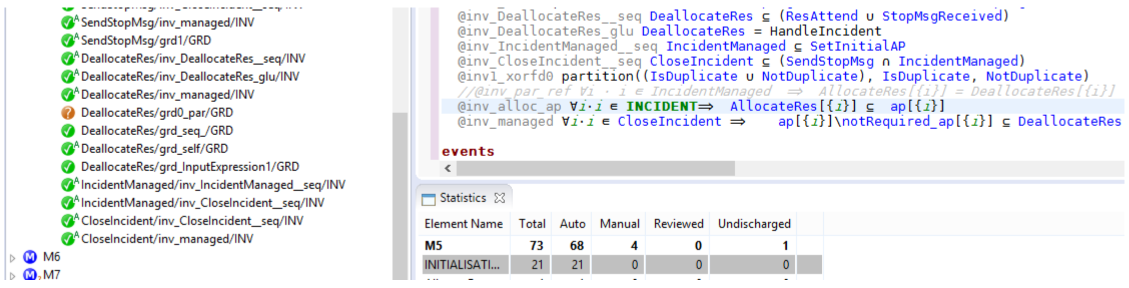
If we do not have an explicit *par-replicator* invariant related to refinement, it will not be possible (without adding an explicit guard) to discharge the proof obligation related to the guard strengthening (GRD) of the *par-replicator* guard ($i \notin IncidentManaged$) in *DeallocateRes* event, as shown in figures 5.13 and 5.14.



```
✓ DeallocateRes/inv_DeallocateRes_glu/INV
✓ DeallocateRes/inv_par_ref/INV
✓ DeallocateRes/inv_managed/INV
✓ DeallocateRes/grd0_par/GRD
✓ DeallocateRes/grd_seq_/GRD
✓ DeallocateRes/grd_self/GRD
✓ DeallocateRes/grd_InputExpression1/GRD
```

```
@inv_DeallocateRes__seq DeallocateRes ⊆ (ResAttend ∪ StopMsgReceived)
@inv_DeallocateRes_glu DeallocateRes = HandleIncident
@inv_IncidentManaged__seq IncidentManaged ⊆ SetInitialAP
@inv_CloseIncident__seq CloseIncident ⊆ (SendStopMsg ∩ IncidentManaged)
@inv1_xorfd0 partition((IsDuplicate ∪ NotDuplicate), IsDuplicate, NotDuplicate)
@inv_par_ref ∀i · i ∈ IncidentManaged  ⇒  AllocateRes[{i}] = DeallocateRes[{i}]
@inv_alloc_ap ∀i·i ∈ INCIDENT⇒  AllocateRes[{i}] ⊆  ap[{i}]
@inv_managed ∀i·i ∈ CloseIncident ⇒    ap[{i}]\notRequired_ap[{i}] ⊆ DeallocateRes[
```

(a) GRD Discharged with ERS Invariant



```
✓ SendStopMsg/inv_managed/INV
✓ SendStopMsg/grd1/GRD
✓ DeallocateRes/inv_DeallocateRes__seq/INV
✓ DeallocateRes/inv_DeallocateRes_glu/INV
✓ DeallocateRes/inv_managed/INV
? DeallocateRes/grd0_par/GRD
✓ DeallocateRes/grd_seq_/GRD
✓ DeallocateRes/grd_self/GRD
✓ DeallocateRes/grd_InputExpression1/GRD
✓ IncidentManaged/inv_IncidentManaged__seq/INV
✓ IncidentManaged/inv_CloseIncident__seq/INV
✓ CloseIncident/inv_CloseIncident__seq/INV
✓ CloseIncident/inv_managed/INV
M6
M7
```

```
@inv_DeallocateRes__seq DeallocateRes ⊆ (ResAttend ∪ StopMsgReceived)
@inv_DeallocateRes_glu DeallocateRes = HandleIncident
@inv_IncidentManaged__seq IncidentManaged ⊆ SetInitialAP
@inv_CloseIncident__seq CloseIncident ⊆ (SendStopMsg ∩ IncidentManaged)
@inv1_xorfd0 partition((IsDuplicate ∪ NotDuplicate), IsDuplicate, NotDuplicate)
//@inv_par_ref ∀i · i ∈ IncidentManaged  ⇒  AllocateRes[{i}] = DeallocateRes[{i}]
@inv_alloc_ap ∀i·i ∈ INCIDENT⇒  AllocateRes[{i}] ⊆  ap[{i}]
@inv_managed ∀i·i ∈ CloseIncident ⇒    ap[{i}]\notRequired_ap[{i}] ⊆ DeallocateRes

events
```

Statistics

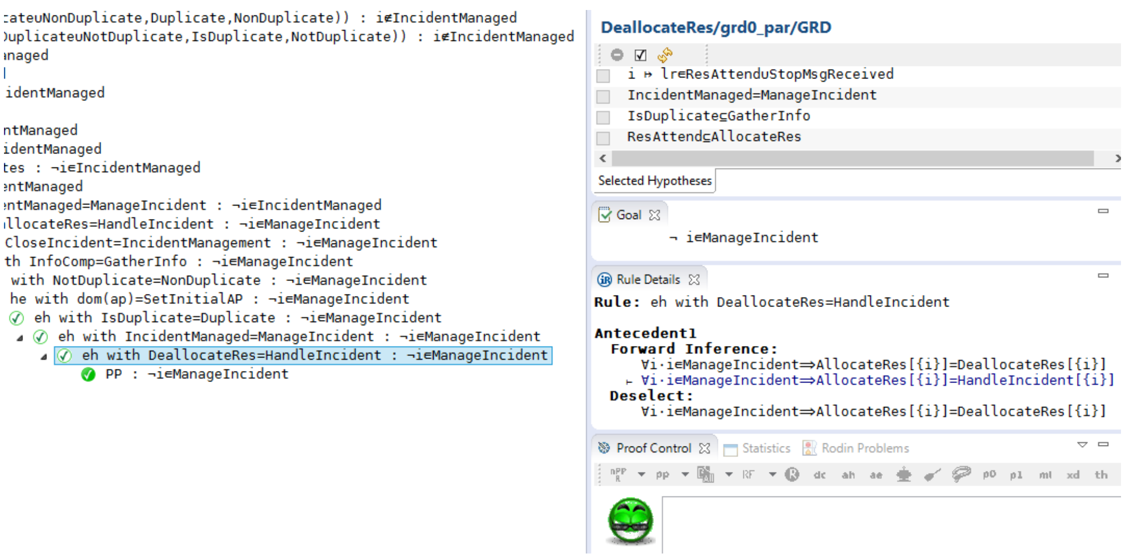
Element Name	Total	Auto	Manual	Reviewed	Undischarged
M5	73	68	4	0	1
INITIALISATI...	21	21	0	0	0

(b) GRD Undischarged without the ERS Invariant

Figure 5.13: Effect of ERS Invariant in Discharging Proof Obligations

5.6 Conclusion

In this chapter we have modelled the fire dispatch system workflow, by applying the ERS approach to model the control flow and representing the data handling using Event-B. In modelling the case study we tried to make small changes in each refinement, making the verification and the validation of the model easier than making big changes in each refinement as explained by some studies (Butler and Yadav, 2008; Butler, 2009b). The ERS approach helped in achieving this goal by its refinement restrictions. Most of the proof obligations in this case study where either proved automatically using Rodin provers or automatically with the help of additional external prover plug-ins such as SMT solvers (Figure 5.15). In Figure 5.15, most of the proofs marked as manual are



```
:ateuNonDuplicate,Duplicate,NonDuplicate)) : i∉IncidentManaged
)uplicateuNotDuplicate,IsDuplicate,NotDuplicate)) : i∉IncidentManaged
ınaged
|
identManaged

ntManaged
identManaged
tes : ¬i∈IncidentManaged
entManaged
entManaged=ManageIncident : ¬i∈IncidentManaged
llocateRes=HandleIncident : ¬i∈ManageIncident
CloseIncident=IncidentManagement : ¬i∈ManageIncident
th InfoComp=GatherInfo : ¬i∈ManageIncident
 with NotDuplicate=NonDuplicate : ¬i∈ManageIncident
 he with dom(ap)=SetInitialAP : ¬i∈ManageIncident
  ✓ eh with IsDuplicate=Duplicate : ¬i∈ManageIncident
    ✓ eh with IncidentManaged=ManageIncident : ¬i∈ManageIncident
      ✓ eh with DeallocateRes=HandleIncident : ¬i∈ManageIncident
          ✓ PP : ¬i∈ManageIncident
```

DeallocateRes/grd0_par/GRD

```
i ↦ lr∈ResAttenduStopMsgReceived
IncidentManaged=ManageIncident
IsDuplicate⊆GatherInfo
ResAttend⊆AllocateRes
```

Selected Hypotheses

Goal

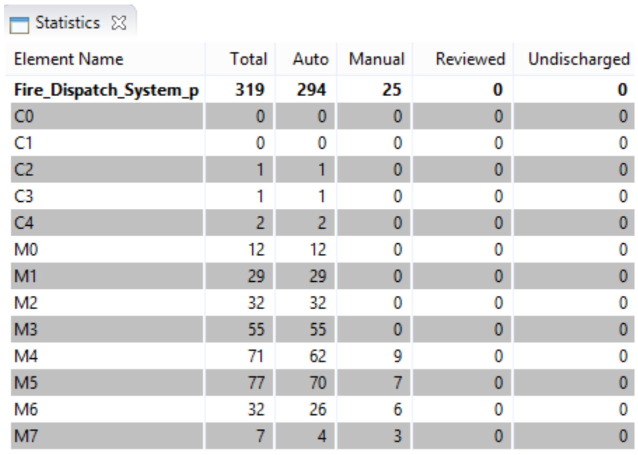
```
¬ i∈ManageIncident
```

Rule Details

```
Rule: eh with DeallocateRes=HandleIncident

Antecedent1
  Forward Inference:
      ∀i·i∈ManageIncident⇒AllocateRes[{i}]=DeallocateRes[{i}]
    ⊢ ∀i·i∈ManageIncident⇒AllocateRes[{i}]=HandleIncident[{i}]
  Deselect:
      ∀i·i∈ManageIncident⇒AllocateRes[{i}]=DeallocateRes[{i}]
```

Figure 5.14: Proof Details of the DeallocateRes/GRD

actually done automatically using external provers or interactively with the help of different external provers.



Statistics

Element Name	Total	Auto	Manual	Reviewed	Undischarged
Fire_Dispatch_System_p	**319**	**294**	**25**	**0**	**0**
C0	0	0	0	0	0
C1	0	0	0	0	0
C2	1	1	0	0	0
C3	1	1	0	0	0
C4	2	2	0	0	0
M0	12	12	0	0	0
M1	29	29	0	0	0
M2	32	32	0	0	0
M3	55	55	0	0	0
M4	71	62	9	0	0
M5	77	70	7	0	0
M6	32	26	6	0	0
M7	7	4	3	0	0

Figure 5.15: Fire Dispatch Model Statistics

In this case study we have also validated the extensions presented in Chapter 4 showing that they are needed to give ERS the flexibility to model dynamic behaviour. While modelling the case study we identified some limitations in the case study related to interruptions and cancellation behaviour, these limitations will be addressed in Chapter 6.

Chapter 6

Extending ERS by Exception Handling

6.1 Introduction

When developing workflows, we usually concentrate on how we are expecting the system to behave in normal circumstances. However, errors and unexpected behaviours are unavoidable, therefore a good workflow modelling approach should be capable of handling exceptions. According to Casati et al. (1999), exceptions are those occasional asynchronous deviations from the normal flow, which may cause backtracking of previous steps or even sudden termination. Modelling exceptions can be done in Event-B, by defining events that can deal with exceptions, but how can we include exceptions and error recovery in ERS?

First when dealing with exceptions, it is better to distinguish between the normal flow and the exception handling flow. This is to avoid complicating the model with various alternatives related to exceptions (Russell et al., 2006b,c). Therefore, in order to avoid complicating the ERS diagrams with various *xor-combinator* branches capturing unexpected behaviour and to focus on the expected behaviour of the system, we introduce two explicit combinators that deal with exceptions.

Second, what are the recovery actions or how exceptions can be handled in ERS? In Eder and Liebhart (1996), which is one of the early works about workflow recovery, when recovering from failures or exceptions in workflows, we can do forward execution, backward recovery or forward recovery. Forward execution entails progressing forward by executing remaining activities, backward recovery requires controlled rollback where successfully completed activities are undone, while forward recovery is in most cases a combination of backward recovery and forward execution. Also in Russell et al. (2006b,c), they identify three possible actions for exception handling; no action, rollback, or compensate. In all of the above recovery actions, we need to provide means to allow the undoing of actions in ERS. This could be also part of the cancellation control patterns which ERS does not support explicitly, but still can be achieved using Event-B.

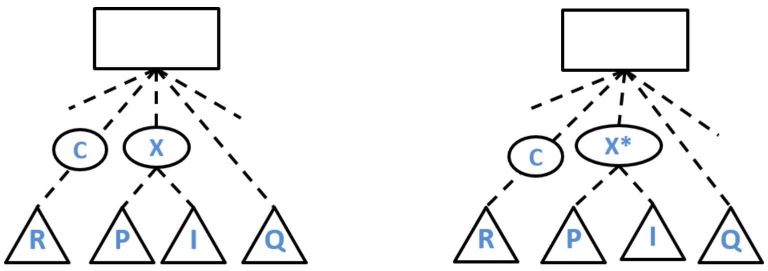
In this chapter we start with introducing two exception handling combinators for ERS, *interrupt* and *retry*. We also show the application of these exception handling combinators to the case study introduced in Chapter 5. Finally we compare our approach in exception handling with other workflow modelling approaches.

6.2 Exception Handling Combinators

In this section we extend the ERS approach with two combinators to support exception handling. The two exception handling combinators are *interrupt* and *retry*. The *interrupt* will stop the execution of a flow, once the interrupt event occurs. The *retry* will also stop the flow execution once the interrupt event takes place, but *retry* will reset the control variables of the flow, in order to enable execution of the normal flow again.

The first child, the left hand side, of the exception handling combinator represents the normal control flow, while the second child, the right hand side, represents the exception or the interrupting flow which can stop the normal flow and transfer control to the exception handling flow. Unlike the other ERS combinators (*xor*, *and*...), the children (*normal-child* and the *interrupting-child*) of the exception handling combinators are not commutative, since their effect is **not** symmetric.

In Figure 6.1, we present a generalised representation of the *interrupt* and the *retry* combinators respectively. P represents the normal sub-flow, while I represents the interrupting sub-flow. R with combinator C and Q, represent respectively the preceding and follow-on flows of the exception handling combinator.



(a) Interrupt (b) Retry

Figure 6.1: Generalised Form of the *interrupt* and the *retry* Combinators

Table 6.1 defines the enabling and disabling conditions of the normal sub-flow P and the interrupting sub-flow I, and the enabling conditions for the follow-on flow Q, for both the *interrupt* (Figure 6.1(a)) and the *retry* (Figure 6.1(b)) combinators. In both cases, the enabling condition for P is determined by the combinator C of the preceding flow R. However in the case of *retry*, P can also be enabled by the completion of I.

Table 6.1: Disabling and Enabling Conditions for *interrupt* and *retry*

	Interrupt Combinator	**Retry Combinator**
Enable P	Determined by combinator C of R	Determined by combinator C of R or I has completed
Disable P	P has executed or I has made a step	P has executed or I has made a step
Enable I	Same enabling condition as P	Same enabling condition as P
Disable I	P has executed	P has executed
Enable Q	P or I has executed	P has executed

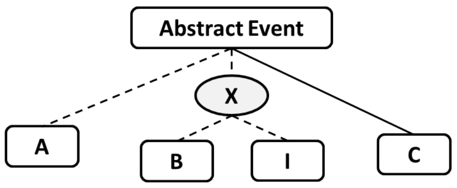
The other difference between the two combinators is the enabling condition of the follow-on flow Q. Enabling Q following the *interrupt-combinator* requires the execution of either P or I, while for the *retry-combinator*, Q can be only enabled after the execution of P. The rest of the conditions are the same for both combinators, so P will be disabled if it completes execution or if I starts execution. The interrupting sub-flow I has the same enabling conditions as the normal sub-flow P of its exception handling combinator, which depends on the previous flow in both cases while *retry* has an additional possible enabling condition which is the completion of I. In both cases I will be disabled if P completes execution.

6.2.1 The Interrupt Combinator Semantics

The aim of the *interrupt* combinator is to stop the execution of a sub-flow once a certain event, which we refer to as the interrupting event, takes place.

The similarity between the *interrupt-combinator* and the *xor-combinator* is that the sequencing of the follow-on sub-flow, if it exists, requires the execution of either child of the *interrupt-combinator*. However the main difference between the *interrupt* and the *xor* is that in the *xor-combinator*, once one child starts execution, it disables the execution of all other children of the *xor-combinator*, whereas the *interrupt-combinator* allows the interrupting sub-flow of the *interrupt-combinator* to stop the execution of the normal sub-flow at any point before it completes execution.

For example, a user shopping on-line can decide at any time before completion to abort or cancel the order. In this case the cancel order event is the interrupting event of the *interrupt-combinator*, and the on-line shopping process is the normal sub-flow of the *interrupt-combinator*.



```
invariants
  @inv_A_type A ∈ BOOL
  @inv_B_seq B = TRUE ⇒ A = TRUE
  @inv_I_seq I = TRUE ⇒ A = TRUE
  @inv_I_interrupt partition ({B, I} ∩ {TRUE}, {B} ∩ {TRUE}, {I} ∩ {TRUE})
  @inv_C_seq C = TRUE ⇒ B = TRUE ∨ I = TRUE
```

```
event A
  where
    @grd_self A = FALSE
  then
    @act A ≔ TRUE
 end
```

```
event I
  where
    @grd_seq A = TRUE
    @grd_self I = FALSE
    @grd_interrupt B = FALSE
  then
    @act I ≔ TRUE
 end
```

```
event B
  where
    @grd_seq A = TRUE
    @grd_self B = FALSE
    @grd_interrupt I = FALSE
  then
    @act B ≔ TRUE
 end
```

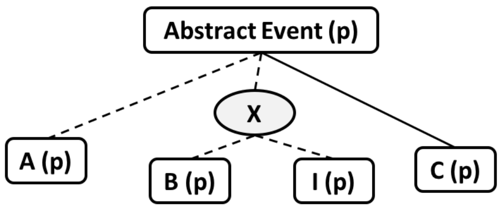
```
event C refines AbstractEvent
  where
    @grd_seq B = TRUE ∨ I = TRUE
    @grd_self C = FALSE
  then
    @act C ≔ TRUE
 end
```

Figure 6.2: Single Instance Semantics of the *Interrupt* Combinator

Diagrammatic Representation: Similar to the other ERS combinators, the *interrupt-combinator* is represented within an oval shape with the "X" symbol, as shown in Figure 6.1(a). The *interrupt-combinator* is always connected to its parent using dashed lines to satisfy the *single solid line* rule of ERS, since the normal sub-flow can execute partially if the interrupt sub-flow executes. The *interrupt-combinator* is always connected to exactly two dashed nodes, the first node represents the normal flow and the second node represents the interrupting flow.

Event-B Semantics: The Event-B semantics of the *interrupt-combinator* is shown in Figures 6.2 and 6.3 for the single instance case and the multiple instances case respectively. In both figures the first child, in this case B, represents the normal child of the *interrupt-combinator* and the second child, in this case I, represents the interrupting child. At the abstract level (Figures 6.2 and 6.3), the *interrupt-combinator* is similar to the *xor-combinator*, the difference between the two combinators is only relevant once refinement is introduced (Figure 6.5).

As shown in the Event-B models of both the single instance case and the multiple instances case, the sequencing constraint of I is the same as B, which represents the normal sub-flow. In this case, I can follow the execution of A as shown by the sequencing



```
invariants
  @inv_A_type A ⊆ TYPE(p)
  @inv_B_seq B ⊆ A
  @inv_I_seq I ⊆ A
  @inv_I_interrupt partition((B ∪ I), B, I)
  @inv_C_seq C ⊆ B ∪ I
```

```
event A
   any p
   where
     @grd_self  p ∉  A
    then
     @act A ≔ A  ∪ {p}
  end
```

```
event I
   any p
   where
     @grd_seq p ∈ A
     @grd_self p ∉  I
     @grd_interrupt p ∉ B
   then
     @act I ≔ I ∪ {p}
  end
```

```
event B
   any p
   where
     @grd_seq p ∈ A
     @grd_self p ∉ B
     @grd_interrupt p ∉ I
   then
     @act B ≔ B ∪ {p}
End
```

```
event C refines AbstractEvent
   any p
   where
     @grd_seq p ∈ (B ∪ I)
     @grd_self p ∉  C
   then
     @act C ≔ C ∪ {p}
  end
```

Figure 6.3: Multiple Instances Semantics of the *Interrupt* Combinator

guards (*grd_seq*) and invariants (*inv_I_seq*) of the interrupting event I. In the single instance case I requires the execution of A ($A = TRUE$) to be enabled, while in the multiple instances case, I can be enabled for a certain instance of parameter p, only if A has executed for that instance ($p \in A$).

The interrupting event I will be disabled once the normal sub-flow (B) completes execution, as shown by the guard *grd_interrupt* of I. In the single Instance case, the disabling guard is ($B = FALSE$) and in the multiple instances case ($p \notin B$).

For the normal sub-flow, we add a disabling guard, *grd_interrupt*, to stop the execution of the normal sub-flow if I executes. In the single instance case the disabling guard of B is ($I = FALSE$) and in the multiple instances cases ($p \notin I$).

Both B and I will be disabled if the other executes, so we enforce this behaviour by the *inv_I_interrupt*, where the $partition(S, s1, .., sn)$ means that, the union of all the elements ($s1..sn$) constitute S, and all the elements ($s1..sn$) are disjoint.

Refinement: Both the normal leaf and the interrupting leaf of the *interrupt-combinator* can be refined. However, when refining the normal child of the *interrupting-combinator*, *B* in Figures 6.2 and 6.3, the last child in the refinement must be the refining event or the solid child, which can be explained by Figure 6.4. Allowing *I* to interrupt between *B1* and *B2* is inconsistent with refinement: at the abstract level, *I* is disabled once *B* occurs. Therefore in the refinement, *I* must be disabled once *B1* occurs. Hence, the need for the restriction that the solid child of the normal flow of the *interrupt-combinator* must be always the last child of the flow.

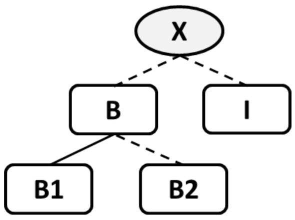


Figure 6.4: Invalid Refinement of the *Interrupt* Combinator

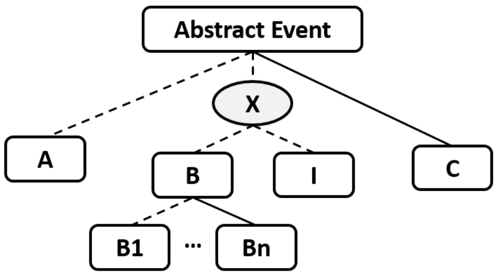
The Event-B semantics of introducing refinement to the normal sub-flow (*B*) of Figure 6.2 is shown in Figure 6.5. As we can see the interrupt disabling guard (*grd_interrupt*) will be added to each leaf (*Bi*) of the normal sub-flow, stopping the execution of the normal sub-flow at any point *I* executes. The interrupting event *I* can be only disabled if the last child of the normal sub-flow (*Bn*) executes, that is why the mutual exclusiveness property is only between *I* and *Bn* as shown by the invariant *inv_I_interrupt*. The same thing applies to the multiple instances case of Figure 6.3.

Refining the interrupting event *I* does not have any restrictions, in this case the execution of the first child of *I* will always stop the normal flow, whether the first child is solid or dashed and in both strong and weak sequencing.

6.2.2 The Retry Combinator Semantics

The *retry-combinator* is similar to the *interrupt-combinator*, in the sense that an interrupting event can stop the execution of the normal flow at any time before completion. However, the main difference between the two combinators is that, once the interrupting event stops the normal flow and completes execution, the normal flow is enabled again. This combinator is useful when some exception happens and the user needs to roll back to some point in the workflow.

Diagrammatic Representation: Similar to the other ERS combinators, the *retry-combinator*, Figure 6.1(b), is represented within an oval shape with the "X*" symbol. The "X", the symbol of the *interrupt-combinator*, to indicate the interruption action and the "*", the symbol of the *loop*, to indicate the possible repetition of the flow.



```
invariants
  @inv_A_type A ∈ BOOL
  @inv_B1_seq B1 = TRUE ⇒ A = TRUE
  @inv_Bi_seq Bi = TRUE ⇒ Bi-1 = TRUE        // 1 ≤ i ≤ n
  @inv_Bn_seq Bn = TRUE ⇒ Bn-1 = TRUE
  @inv_I_seq I = TRUE ⇒ A= TRUE
  @inv_I_interrupt partition ({Bn, I} ∩ {TRUE}, {Bn} ∩ {TRUE}, {I} ∩ {TRUE})
  @inv_C_seq C = TRUE ⇒ Bn = TRUE ∨ I = TRUE
```

```
event A
  where
    @grd_self A = FALSE
  then
    @act A ≔ TRUE
end
```

```
event I
  where
    @grd_seq A = TRUE
    @grd_self I = FALSE
    @grd_interrupt Bn = FALSE
  then
    @act I ≔ TRUE
end
```

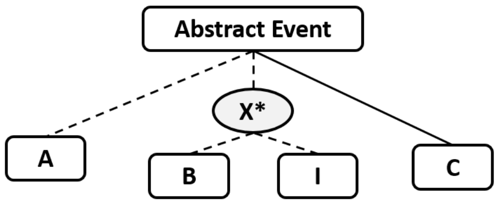
```
event Bi  // 1 ≤ i ≤ n
  where
    @grd_seqBi-1 = TRUE
    @grd_self Bi = FALSE
    @grd_interrupt I = FALSE
  then
    @act Bi ≔ TRUE
end
```

```
event C refines AbstractEvent
  where
    @grd_seq Bn = TRUE ∨ I = TRUE
    @grd_self C = FALSE
  then
    @act C ≔ TRUE
end
```

Figure 6.5: Refinement Semantics of the normal sub-flow B of Figure 6.2

Similar to the *interrupt-combinator*, the *retry-combinator* is always connected to its parent node using dashed line and is always connected to exactly two dashed nodes, the first representing the normal sub-flow and the second representing the interrupting sub-flow.

Event-B Semantics: The Event-B semantics of the *retry-combinator* is shown in Figures 6.6 and 6.7, for the single instance and multiple instances cases respectively. The encoded Event-B related to sequencing and disabling conditions of an event in the normal sub-flow B and the interrupting event I, are similar to those of the *interrupt-combinator*, Figures 6.2 and 6.3. However, the interrupting event I of the *retry-combinator* has additional actions to reset the control variables of the normal sub-flow. In the single instance case the normal sub-flow B is reset to $FALSE$, and in the multiple instances case the parameter p is removed from the set B.



```
invariants
  @inv_A_type A ∈ BOOL
  @inv_B_seq B = TRUE ⇒ A = TRUE
  @inv_I_seq I = TRUE ⇒ A = TRUE
  @inv_I_interrupt partition ({B, I} ∩ {TRUE}, {B} ∩ {TRUE}, {I} ∩ {TRUE})
  @inv_C_seq C = TRUE ⇒ B = TRUE
```

```
event A
  where
    @grd_self A = FALSE
  then
    @act A ≔ TRUE
 end
```

```
event I
  where
    @grd_seq A = TRUE
    @grd_self I = FALSE
    @grd_interrupt B = FALSE
  then
    @act I ≔ TRUE
    @act_B B ≔ FALSE
end
```

```
event B
  where
    @grd_seq A = TRUE
    @grd_self B = FALSE
    @grd_interrupt I = FALSE
  then
    @act B ≔ TRUE
 end
```

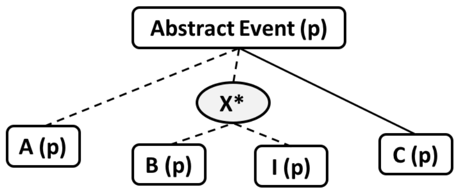
```
event C refines AbstractEvent
  where
    @grd_seq B = TRUE
    @grd_self C = FALSE
  then
    @act C ≔ TRUE
 end
```

```
event reset_I
  where
    @grd_seq I = TRUE
then
    @act I ≔ FALSE
 end
```

Figure 6.6: Single Instance Semantics of the *Retry* Combinator

At the abstract level (Figures 6.6 and 6.7), the behaviour of the *retry-combinator* can be represented by a *loop* as shown in Figure 6.8. However, when the *normal-child* (B) is refined, this equivalence with the *loop* is not applicable.

Resetting the control variable, B, of the normal sub-flow is not enough to enable the normal sub-flow due to the presence of the interrupting guard *grd_interrupt*. Therefore, we add a resetting event, event *reset_I*, to reset the value of the interrupting event after its execution. The resetting event can execute after the completion of the interrupting flow, which is represented by the sequencing guard of the reset event (*reset_I*). The sequencing guard of the resetting event is $I = TRUE$ in the single instance case and $p \in I$ in the multiple instances case. The action(s) of the reset event will reset the control variables of the interrupting sub-flow, in this case there is only one event I, so



```
invariants
  @inv_A_type A ⊆ TYPE(p)
  @inv_B_seq B ⊆ A
  @inv_I_seq I ⊆ A
  @inv_I_interrupt partition((B ∪ I), B, I)
  @inv_C_seq C ⊆ B
```

```
event A
   any p
   where
     @grd_self p ∉ A
   then
     @act A ≔ A ∪ {p}
End
```

```
event I
   any p
   where
     @grd_seq p ∈ A
     @grd_self p ∉ I
     @grd_interrupt p ∉ B
   then
     @act I ≔ I ∪ {p}
     @act_B B ≔ B ∖ {p}
end
```

```
event B
   any p
   where
     @grd_seq p ∈ B
     @grd_self p ∉ B
     @grd_interrupt p ∉ I
   then
     @act B ≔ B ∪ {p}
 end
```

```
event C refines AbstractEvent
   any p
   where
     @grd_seq p ∈ B
     @grd_self p ∉ C
   then
     @act C ≔ C ∪ {p}
end
```

```
event reset_I
  any p
   where
     @grd_seq p ∈ I
then
     @act I ≔ I ∖ {p}
end
```

Figure 6.7: Multiple Instances Semantics of the *Retry* Combinator

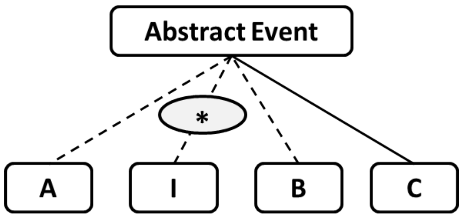


Figure 6.8: Behaviour of *retry* (Figure 6.6) at Abstract Level

the resetting action will be $I = FALSE$ in the single instance case, and $I = I \setminus \{p\}$ in the multiple instances case.

Another difference to the *interrupt-combinator* is the sequencing of the follow-on event, C, which depends only on the normal sub-flow, so the normal sub-flow must complete execution without interruption to enable C. This is represented by the sequencing guard $B = TRUE$ in the single instance case and $p \in B$ in the multiple instances case and their associated invariants *inv_C_seq*. If it is possible for the *retry* exception to persist, we can add an *interrupt-combinator* at a higher level to avoid retrying forever. For example in Figure 6.9, *I0*, the interrupting event of the *interrupt-combinator*, can interrupt both B and I of the *retry-combinator*.

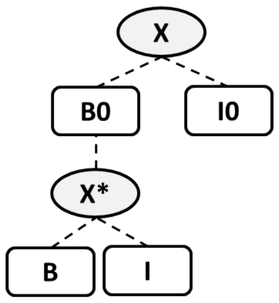


Figure 6.9: Introducing Exception Handling at Two Levels

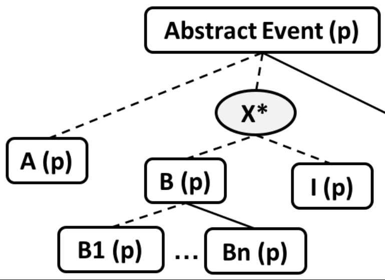
Refinement: The same refinement restrictions of the *interrupt-combinator* applies to the *retry-combinator*, i.e., when refining the normal child of the *retry-combinator* the last child must be the solid child.

One can argue that a solution to avoid the restriction of the solid last child of the normal flow, is to have the follow-on event of the *retry-combinator* (C) to disable the interrupting sub-flow. This does not work, first because we do not always have an event after the *retry-combinator*. Second and most importantly in the *retry-combinator* case, this solution would make it possible for the normal sub-flow to execute successfully more than once. This contradicts the aim of the *retry* combinator which is only to retry the normal flow if an exception occurs, represented by the occurrence of the interrupting event, i.e., the normal sub-flow can complete execution only once, while the interrupting sub-flow can complete execution more than once.

Figure 6.10 shows the encoded Event-B of the refinement of B in the multiple instances case. Similar to the *interrupt-combinator* the interrupt disabling guard (*grd_interrupt*) will be added to each child leaf *Bi*, and the interrupting event I will be disabled by the execution of the last child of the normal sub-flow (*Bn*) and enforced by the invariant *inv_I_interrupt*. The event I will also reset each child leaf *Bi* of the normal sub-flow. The same applies to the single instance case.

6.3 Application to the Fire Dispatch Workflow

In this section we model the fire dispatch case study introduced in Chapter 5 using the exception handling combinators. The aim of this section is to model the requirements not



```
invariants
  @inv_A_type A ⊆ TYPE(p)
  @inv_B1_seq B1 ⊆ A
  @inv_Bi_seq Bi ⊆ Bi-1        // 1 ≤ i ≤ n
  @inv_Bn_seq Bn ⊆ Bn-1
  @inv_I_seq I ⊆ A
  @inv_I_interrupt partition((Bn ∪ I), Bn, I)
  @inv_C_seq C ⊆ Bn
```

```
event A
   any p
   where
    @grd_self p ∉ A
   then
    @act A ≔ A ∪ {p}
end
```

```
event I
   any p
   where
    @grd_seq p ∈ A
    @grd_self p ∉ I
    @grd_interrupt p ∉ Bn
   then
    @act I ≔ I ∪ {p}
    @act_Bi Bi ≔ Bi ∖ {p}
end
```

```
event Bi   // 1 ≤ i ≤ n
   any p
   where
    @grd_seq p ∈ Bi-1
    @grd_self p ∉ Bi
    @grd_interrupt p ∉ I
   then
    @act Bi ≔ Bi ∪ {p}
 end
```

```
event C refines
AbstractEvent
   any p
   where
    @grd_seq p ∈ Bn
    @grd_self p ∉ C
   then
    @act C ≔ C ∪ {p}
end
```

```
event reset_I
  any p
   where
    @grd_seq p ∈ I
then
    @act I ≔ I ∖ {p}
end
```

Figure 6.10: Refinement Semantics of the normal sub-flow B of Figure 6.7

covered in Chapter 5 due to limitations in the ERS approach as explained in Section 5.5 of Chapter 5. The complete version of this model is presented in Appendix B.

In this version of the fire dispatch model, the dynamic part consists of an abstract and ten refinement levels, while the static part consists of seven contexts. The control flow of the model is done using the ERS approach, but in the 9th refinement ($M9$), we use a state machine to model the different states of a physical resource. Data handling of the case study is also done manually using Event-B, where additional data variables were added alongside the ERS control variables. Most of the requirements which were already done in Section 5.4 of Chapter 5 are the same and we will only show the differences in this section.

The machines of the case study are modelled as follows:

- **M0:** Models the creation of an incident and identifying it as duplicate or not.
- **M1:** Refines the non-duplicate incident case, introducing the information gathering and setting the action plan of the incident and managing the incident.
- **M2:** Models the ability to update the action plan of the incident by requesting additional resources.
- **M3:** Introduces further incident details such as incident location and type.
- **M4:** Introduces the priority of the incident.
- **M5:** Refines the incident management according to the action plan of the incident
- **M6:** Introduces the possibility of losing an allocated resource and how to handle this exception.
- **M7:** Shows in further details, how an incident is handled and how allocated resources can be lost.
- **M8:** Introduces physical resources which are actually assigned to an incident and how a duplicate incident is handled in case duplication is discovered after resource allocation
- **M9:** Introduces the different states of a physical resource using a state machine.
- **M10:** Adds further details about the location and the duration of a physical resource to the incident.

6.3.1 Distinction Between Duplicate and Non-Duplicate Incidents Using the Interrupt Combinator

The main difference starts right at the abstract level (Figure 6.11), where in order to model requirement *INF_3* in Table 5.1, we represent the *IsDuplicate* event as an interrupting event of the *interrupt-combinator* introduced in Chapter 6. Replacing the *xor-combinator* in Figure 5.2 with the *interrupt-combinator* in Figure 6.11 gives the controller the flexibility to determine that an incident is duplicate at any point, hence there is no need to continue with the normal activities associated with the non-duplicate incidents, which are shown by refining the *Not-Duplicate* event.

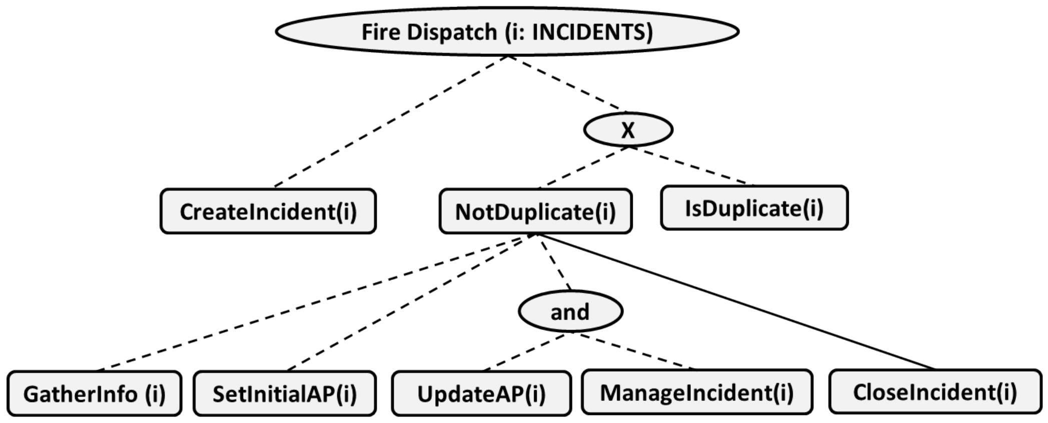
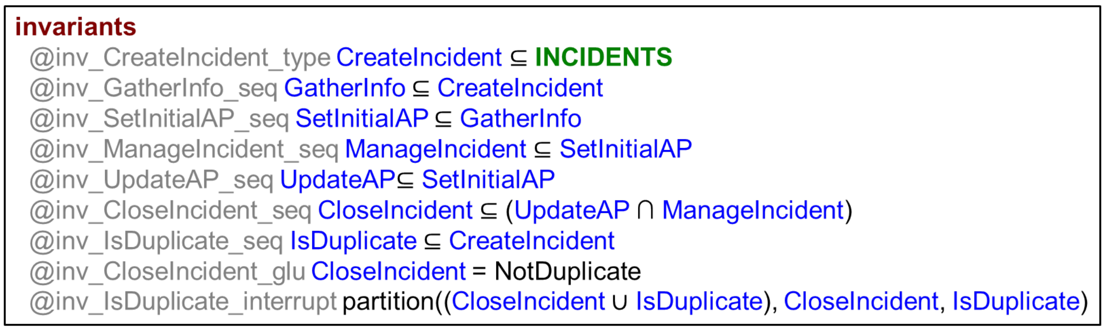


Figure 6.11: Distinction Between Duplicate and Non-Duplicate Incidents

In the first refinement (*M1*), the atomicity of *NotDuplicate* is broken into the sequence of events starting with gathering information about the incident (*GatherInfo*), followed by setting an initial action plan to manage the incident (*SetInitialAP*), then the concurrent execution of managing the incident (*ManageIncident*) and updating the action plan (*UpdatAP*), finally after the incident is managed the incident call can be closed (*CloseIncident*), after which the incident cannot be interrupted by the *IsDuplicate* event.

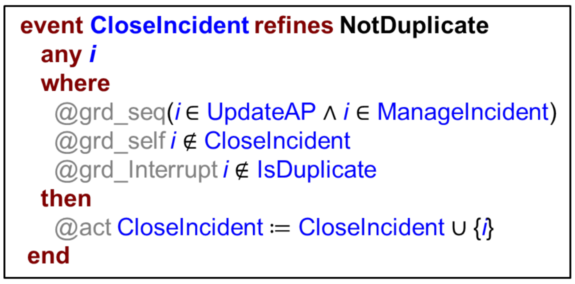
In the case study, we are using multiple instances modelling, therefore sequencing between the events is determined by the subset sets approach, for instance the invariant to describe the order of execution of *GatherInfo* event will be $GatherInfo \subseteq CreateIncident$, similarly the ordering of *IsDuplicate* depends only on *CreateIncident*, as a result of using the *interrupt-combinator*. Figure 6.12 shows the invariants of the first refinement level of the model (*M1*), where all these invariants are the result of transforming the ERS diagram of Figure 6.11 to Event-B.



```
invariants
  @inv_CreateIncident_type CreateIncident ⊆ INCIDENTS
  @inv_GatherInfo_seq GatherInfo ⊆ CreateIncident
  @inv_SetInitialAP_seq SetInitialAP ⊆ GatherInfo
  @inv_ManageIncident_seq ManageIncident ⊆ SetInitialAP
  @inv_UpdateAP_seq UpdateAP⊆ SetInitialAP
  @inv_CloseIncident_seq CloseIncident ⊆ (UpdateAP ∩ ManageIncident)
  @inv_IsDuplicate_seq IsDuplicate ⊆ CreateIncident
  @inv_CloseIncident_glu CloseIncident = NotDuplicate
  @inv_IsDuplicate_interrupt partition((CloseIncident ∪ IsDuplicate), CloseIncident, IsDuplicate)
```

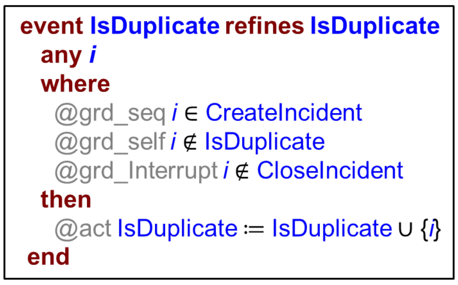
Figure 6.12: Invariants of the First Refinement Level (M1)

The gluing invariant (*inv_CloseIncident_glu*) showing the equality of *CloseIncident* with *NotDuplicate* is the result of the solid line connecting *CloseIncident* to *NotDuplicate*, which conforms to the restriction of the *interrupt-combinator*, where the last event must be the refining event. The last invariant is the result of applying the *interrupt-combinator*, where if the normal sub-flow (*Not-Duplicate*) completed execution successfully, it cannot be interrupted by *IsDuplicate*, and if *IsDuplicate* executes before completing execution, the normal sub-flow cannot complete execution. This invariant is different from the *xor-combinator* invariant where in case of the *xor-combinator* the partition will be between the first event (*GatherInfo*) and *IsDuplicate*. At the abstract level (*M0*) there is no difference between applying the *xor-combinator* or the *interrupt-combinator* to the events *NotDuplicate* and *IsDuplicate*, the difference appears in the refinement of the normal sub-flow (*M1*), where each event will have the guard $i \notin IsDuplicate$ and not only the first event like applying the *xor-combinator*.



```
event CloseIncident refines NotDuplicate
  any i
  where
    @grd_seq(i ∈ UpdateAP ∧ i ∈ ManageIncident)
    @grd_self i ∉ CloseIncident
    @grd_Interrupt i ∉ IsDuplicate
  then
    @act CloseIncident ≔ CloseIncident ∪ {i}
 end
```

(a) *CloseIncident* Event (*M1*)



```
event IsDuplicate refines IsDuplicate
  any i
  where
    @grd_seq i ∈ CreateIncident
    @grd_self i ∉ IsDuplicate
    @grd_Interrupt i ∉ CloseIncident
  then
    @act IsDuplicate ≔ IsDuplicate ∪ {i}
 end
```

(b) *IsDuplicate* Event (*M1*)

Figure 6.13: Event-B Specification of some Events of the Refinement Level (*M1*)

In Figure 6.13, we show the events *Closeincident* and *IsDuplicate* of the first refinement level of the model. All the guards and the actions of these events are the result of the ERS diagram in Figure 6.11. The guard $i \notin IsDuplicate$ (*grd_Interrupt*) is the result of the *interrupt-combinator*, and will be also inherited by the events (*GatherInfo*, *SetInitialAP*, *ManageIncident*, *UpdateAP*) to show the interruption effect of executing the *IsDuplicate* event.

6.3.2 Introducing the Priority of the Incident

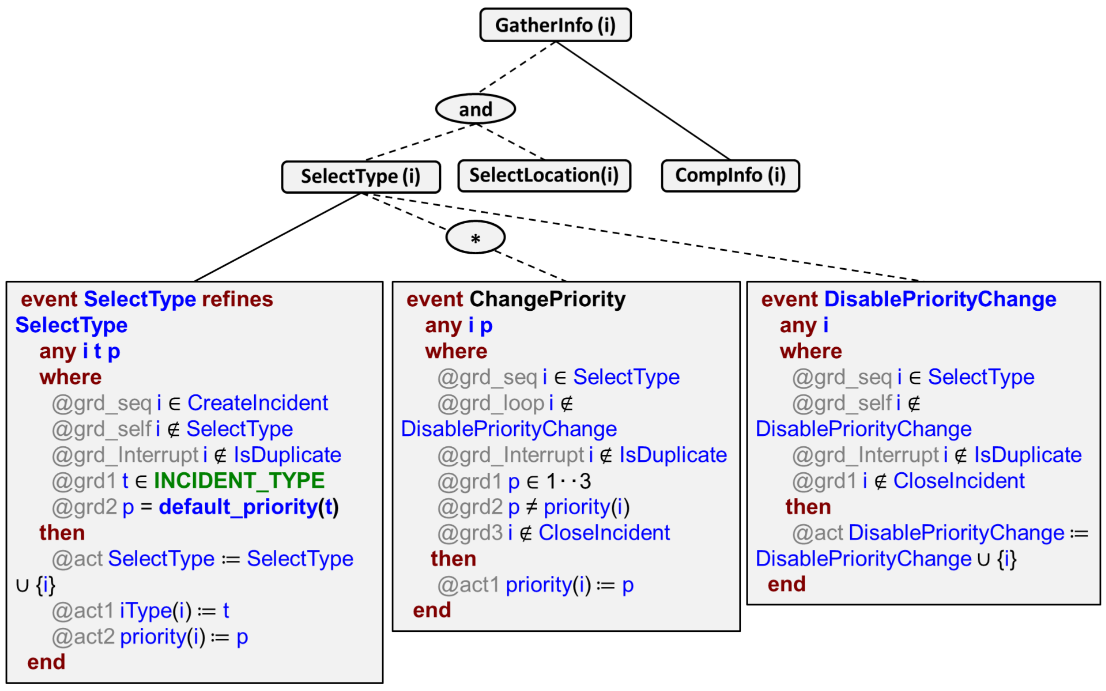
Introducing the priority of the incident, requirements *INF_4* and *INF_5* in Table 5.1, was possible in the model presented in Chapter 5, but we decided to leave it due to its relation with requirements *HND_5* and *RES_1* in tables 5.2 and 5.3 respectively.

In order to relate incident types with a priority level, we add the axiom ($default\_priority \in INCIDENT\_TYPE \rightarrow 1 \cdot\cdot 3$) to the context associating each incident type with a default priority level that can range between one and three, where priority level one is the highest priority.

In the dynamic part of the model, we introduce the data variable *priority* which is defined by *inv_priority*, the invariant uses a total function to ensure that each incident with a selected type will have a priority level.

$$\textbf{inv\_priority: } priority \in SelectType \rightarrow 1 \cdot\cdot 3$$

We refine *SelectType* as shown in Figure 6.14 to introduce the priority of the incident. Requirement *INF_4* does not specify how often the priority of the incident is allowed to change and when we can change it, therefore when introducing the priority we left it open, by allowing changing the priority zero or more times using the *loop*, moreover since it is not specified when the priority can be updated, we introduced the priority change in the refinement using weak sequencing. Applying the weak sequencing interpretation means that sequencing constraints only applies between the solid event (*SelectType*) and the left and right hand side events of the previous refinement level. As a result, *CompInfo* ordering will remain the same as before where its sequencing invariant is $CompInfo \subseteq (SelectLocation \cap SelectType)$. However, sequencing still holds at the same level, i.e., between *SelectType*, *ChangePriority* and *DisablePriorityChange* events. Therefore, it is possible to change the priority during the incident management, which makes sense because the incident officer can assess upon arrival the right priority of the incident, and having the possibility to be changed more than once is also acceptable as things might happen during the incident requiring the change of the priority.



```
event SelectType refines
SelectType
  any i t p
  where
    @grd_seq i ∈ CreateIncident
    @grd_self i ∉ SelectType
    @grd_Interrupt i ∉ IsDuplicate
    @grd1 t ∈ INCIDENT_TYPE
    @grd2 p = default_priority(t)
  then
    @act SelectType ≔ SelectType
∪ {i}
    @act1 iType(i) ≔ t
    @act2 priority(i) ≔ p
 end
```

```
event ChangePriority
  any i p
  where
    @grd_seq i ∈ SelectType
    @grd_loop i ∉
DisablePriorityChange
    @grd_Interrupt i ∉ IsDuplicate
    @grd1 p ∈ 1‥3
    @grd2 p ≠ priority(i)
    @grd3 i ∉ CloseIncident
  then
    @act1 priority(i) ≔ p
 end
```

```
event DisablePriorityChange
  any i
  where
    @grd_seq i ∈ SelectType
    @grd_self i ∉
DisablePriorityChange
    @grd_Interrupt i ∉ IsDuplicate
    @grd1 i ∉ CloseIncident
  then
    @act DisablePriorityChange ≔
DisablePriorityChange ∪ {i}
 end
```

Figure 6.14: Introducing Priority of the Incident Using Weak Sequencing

In Figure 6.14, we manually set the priority of the incident i to the default priority when selecting the type of the incident (*SelectType*). We can also see how all the events related to the non-duplicate case, inherited the interrupting guard ($i \notin IsDuplicate$). The *ChangePriority* event will update the current priority to a new one as shown by the action of the event ($priority(i) :\ = p$). In both events, *ChangePriority* and *DisablePriority*, we add the guard $i \notin CloseIncident$, this means that we will disable the priority change when the incident is closed.

6.3.3 Handling the Lost Allocated Resources with the Retry Combinator

The next main update in the model is handling resources which are lost due to reallocation to a higher priority incidents or due to breakdown, in order to cover requirements *HND_5* and *RES_2* in Table 5.2 and Table 5.3 respectively. For this, we use the *retry-combinator* introduced in Chapter 6 to interrupt the flow of the resource and allow a replacement resource to be allocated to the incident.

The incident management of a non-duplicate incident, which also constitutes the consumer part of the action plan in Figures 5.4 and 5.10 of Chapter 5 will be updated as shown in Figure 6.15 to introduce the *retry-combinator* which handles the resource loss.

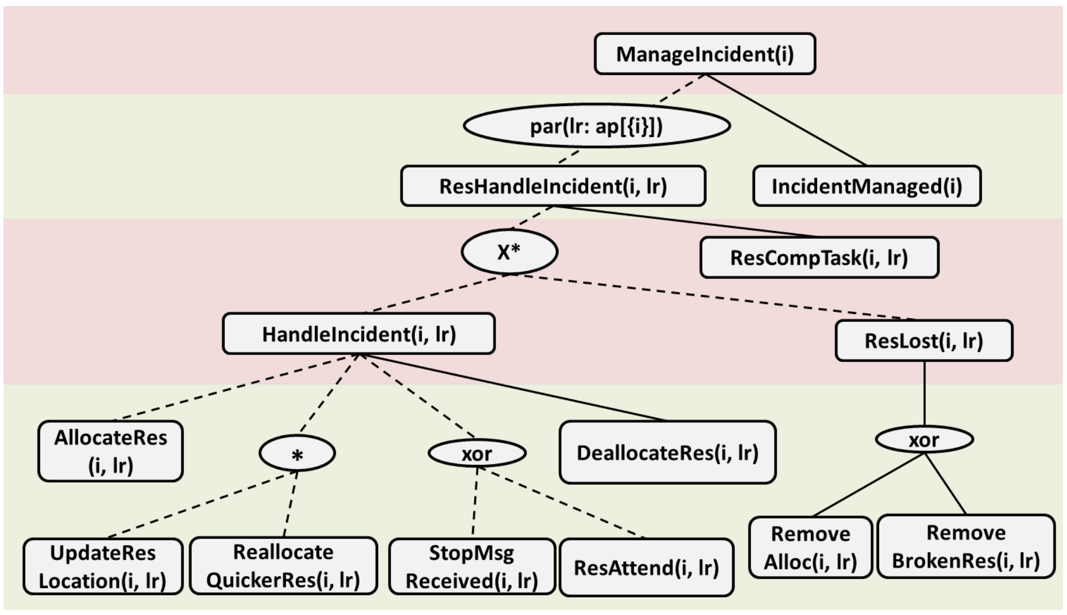


Figure 6.15: Incident Management Using the *Retry* Combinator

In Figure 6.15 we ended up with an additional refinement level due to the ERS restriction that do not allow more than one combinator to be applied to the same event at the same level, which actually makes the modelling process simpler by making smaller changes at each level. For example, applying more than one combinator to several events at the same time can result in more complex sequencing invariants that are harder to read and prove, whereas dividing these changes in separate stages and making use of the ERS refinement gluing invariants will result in simpler sequencing invariants. In this case we had to introduce the resource handling of a resource in two steps before introducing a detailed handling process, which are first introducing the *par-replicator* then the *retry-combinator*, because the loss and replacement of a resource applies to a single resource at a time.

As explained earlier in Chapter 6, the last event in the refinement of the normal part of an exception handler must be the solid line, as the interrupting event, in this case *RemoveAlloc* and *RemoveBrokenRes*, cannot interrupt the handling of the incident and the incident does **not** require a replacement after being deallocated.

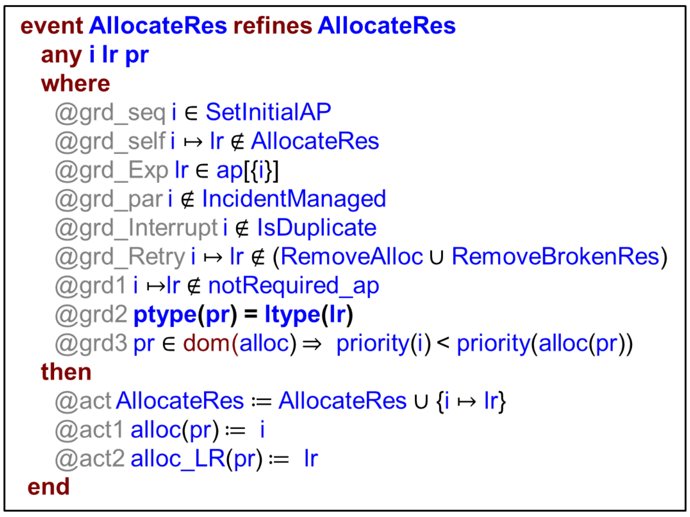
After introducing the physical resources, allocation of resources is updated to allow reallocation from lower priority incidents to higher priority incidents so the condition that ($pr \notin dom(alloc)$) in Figure 5.11(a) is not applicable. In the updated version of the model, *AllocateRes* will allow allocation of physical resources that are already in the domain of *alloc* as shown in Figure 6.16(a), where *grd3* ensures that if *pr* which is the physical resource to be allocated, was in the domain of *alloc*, then the incident it was allocated to must be of lower priority than the current incident *i*. *RemoveAlloc* (Figure 6.16) is one of the interrupting events that is related to *AllocateRes* event, where *RemoveAlloc* is enabled when a resource gets reallocated from a lower priority incident to a higher priority incident. This relation is specified by the manually added guards *grd1* and *grd2*, where *grd1* specifies that allocation of a resource occurred for the logical resource *lr*, but *grd2* indicates that there is no actual physical resource allocated to the logical resource *lr* by checking that the mapping from *i* to *lr* is not in the forward composition of the inverse of *alloc* and *alloc_LR*, in other words the allocated resource for the logical resource is lost.

In this case *RemoveAlloc* will remove the maplet ($i \mapsto lr$) from *AllocateRes*, *ResAttend*, *StopMsgReceived*, *DeallocateRes* to enable allocating a new physical resource for the logical resource *lr*. The replacement will be done after resetting *RemoveAlloc* (Figure 6.17(b)) because *RemoveAlloc* is an interrupting event. Another possible interruption is a broken down resource as indicated by the interrupting event *RemoveBrokenRes* in Figure 6.15, the only difference with *RemoveAlloc* is that removing the physical resource allocation from the incident (*alloc*, *alloc_LR*) will be done in this event and the broken resource will not be allocated to another incident (Figure 6.17(a)).

In both interrupting events *RemoveAlloc* and *RemoveBrokenRes*, we can see the interrupting guard $i \notin IsDuplicate$, since the duplicate interrupt is at higher level, therefore it can still interrupt the lower level interrupts (*RemoveAlloc*, *RemoveBrokenRes*).

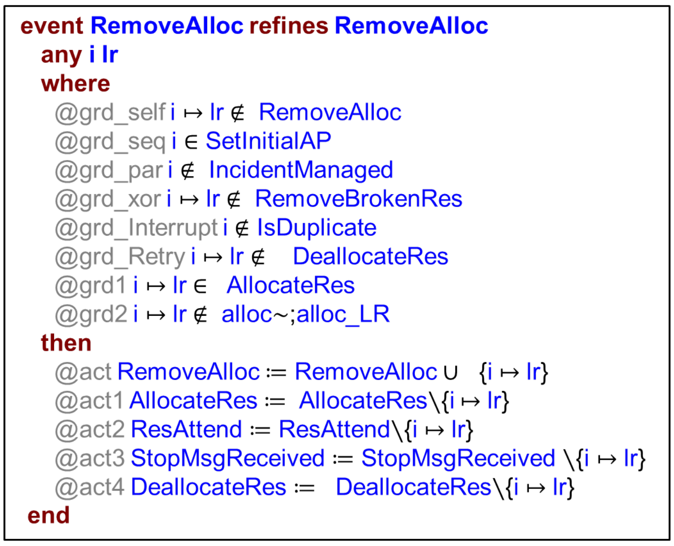
6.3.4 Handling Allocated Resources to Duplicate Incidents

In the previous version of the case study in Chapter 5, deciding that the incidents are duplicate was only done before handling the incidents and allocating physical resources



```
event AllocateRes refines AllocateRes
  any i lr pr
  where
   @grd_seq i ∈ SetInitialAP
   @grd_self i ↦ lr ∉ AllocateRes
   @grd_Exp lr ∈ ap[{i}]
   @grd_par i ∉ IncidentManaged
   @grd_Interrupt i ∉ IsDuplicate
   @grd_Retry i ↦ lr ∉ (RemoveAlloc ∪ RemoveBrokenRes)
   @grd1 i ↦lr ∉ notRequired_ap
   @grd2 ptype(pr) = ltype(lr)
   @grd3 pr ∈ dom(alloc) ⇒  priority(i) < priority(alloc(pr))
  then
   @act AllocateRes ≔ AllocateRes ∪ {i ↦ lr}
   @act1 alloc(pr) ≔  i
   @act2 alloc_LR(pr) ≔  lr
 end
```

(a) *AllocateRes* Event (*M8*)



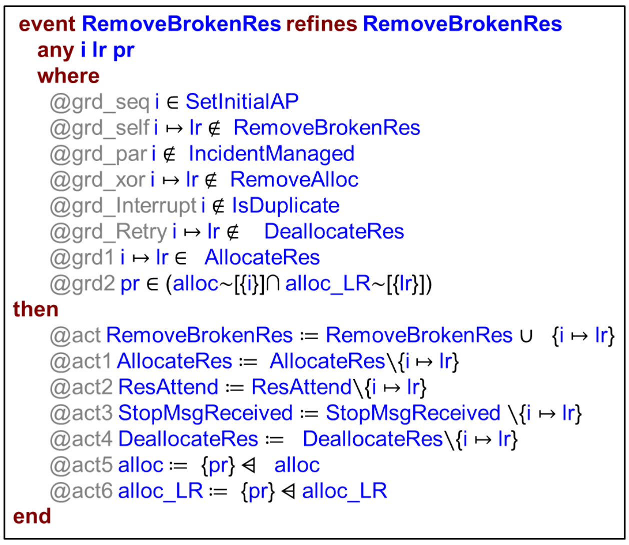
```
event RemoveAlloc refines RemoveAlloc
  any i lr
  where
   @grd_self i ↦ lr ∉  RemoveAlloc
   @grd_seq i ∈ SetInitialAP
   @grd_par i ∉  IncidentManaged
   @grd_xor i ↦ lr ∉  RemoveBrokenRes
   @grd_Interrupt i ∉ IsDuplicate
   @grd_Retry i ↦ lr ∉    DeallocateRes
   @grd1 i ↦ lr ∈   AllocateRes
   @grd2 i ↦ lr ∉  alloc~;alloc_LR
  then
   @act RemoveAlloc ≔ RemoveAlloc ∪   {i ↦ lr}
   @act1 AllocateRes ≔  AllocateRes\{i ↦ lr}
   @act2 ResAttend ≔ ResAttend\{i ↦ lr}
   @act3 StopMsgReceived ≔ StopMsgReceived \{i ↦ lr}
   @act4 DeallocateRes ≔   DeallocateRes\{i ↦ lr}
 end
```

(b) *RemoveAlloc* Event (*M8*)

Figure 6.16: Event-B Specification of the Allocation of Resources and its Cancellation at the Refinement Level (*M8*)

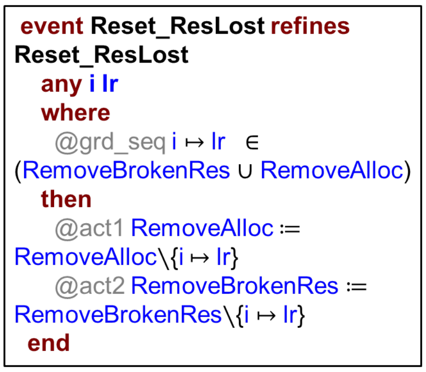
to it, thus requirement *CLS_2* of Table 5.4 is always satisfied for closing duplicate incidents since allocation of resources cannot be done for duplicate incidents. However, in Section 6.3.1, we have updated the model to allow the *IsDuplicate* event to interrupt the handling of resources, which means that in some cases it is possible to have allocated physical resources to a duplicate incident. That is why we refine the *IsDuplicate* event as shown in Figure 6.18 to handle the allocated resources of a duplicate incident that is discovered late.

Using the *all-replicator* in *HandleDupAlloc* ensures that requirement *CLS_2* is satisfied by removing all the physical allocations before closing the duplicate incident in the manually added actions *act1* and *act2*. This is an example of how we compensated the



```
event RemoveBrokenRes refines RemoveBrokenRes
  any i lr pr
  where
    @grd_seq i ∈ SetInitialAP
    @grd_self i ↦ lr ∉ RemoveBrokenRes
    @grd_par i ∉ IncidentManaged
    @grd_xor i ↦ lr ∉ RemoveAlloc
    @grd_Interrupt i ∉ IsDuplicate
    @grd_Retry i ↦ lr ∉ DeallocateRes
    @grd1 i ↦ lr ∈ AllocateRes
    @grd2 pr ∈ (alloc~[{i}]∩ alloc_LR~[{lr}])
then
    @act RemoveBrokenRes ≔ RemoveBrokenRes ∪ {i ↦ lr}
    @act1 AllocateRes ≔ AllocateRes\{i ↦ lr}
    @act2 ResAttend ≔ ResAttend\{i ↦ lr}
    @act3 StopMsgReceived ≔ StopMsgReceived \{i ↦ lr}
    @act4 DeallocateRes ≔ DeallocateRes\{i ↦ lr}
    @act5 alloc ≔ {pr} ⩤ alloc
    @act6 alloc_LR ≔ {pr} ⩤ alloc_LR
end
```

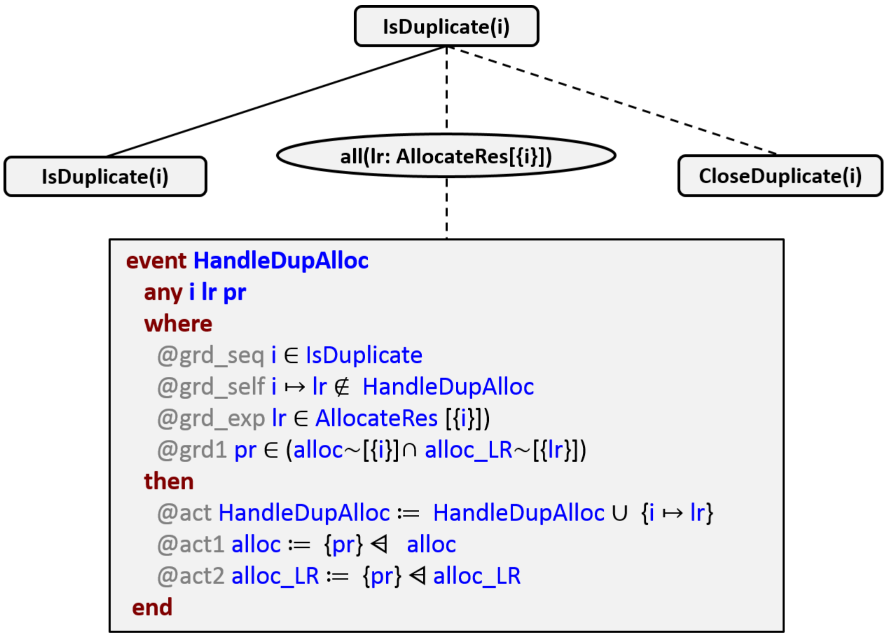
(a) *RemoveBrokenRes* Event (*M8*)



```
event Reset_ResLost refines
Reset_ResLost
  any i lr
  where
    @grd_seq i ↦ lr ∈
(RemoveBrokenRes ∪ RemoveAlloc)
  then
    @act1 RemoveAlloc ≔
RemoveAlloc\{i ↦ lr}
    @act2 RemoveBrokenRes ≔
RemoveBrokenRes\{i ↦ lr}
 end
```

(b) *Reset_ResLost* Event (*M8*)

Figure 6.17: Event-B Specification of *RemoveBrokenRes* and the Resetting Event at the Refinement Level (*M8*)



```
event HandleDupAlloc
  any i lr pr
  where
    @grd_seq i ∈ IsDuplicate
    @grd_self i ↦ lr ∉ HandleDupAlloc
    @grd_exp lr ∈ AllocateRes [{i}])
    @grd1 pr ∈ (alloc~[{i}]∩ alloc_LR~[{lr}])
  then
    @act HandleDupAlloc ≔ HandleDupAlloc ∪ {i ↦ lr}
    @act1 alloc ≔ {pr} ⩤ alloc
    @act2 alloc_LR ≔ {pr} ⩤ alloc_LR
 end
```

Figure 6.18: Refining the *IsDuplicate* Interrupting Event to Handle Allocated Resources

allocated resources in the case of duplicate exception. In Figure 6.18, it was possible to use the *all-replicator* and benefit from its sequencing constraint that affect *CloseDuplicate* because the control variable *AllocateRes*, which in this case represents the parameter set of the *all-replicator*, cannot be changed once *IsDuplicate* event executes.

6.3.5 Extending the Model with State Machines

As explained in Section 5.5 of Chapter 5, we did not model requirement *RES_3* of Table 5.3, because we have not modelled the unavailability of a resource due to breakdown for example. Now we have introduced the possible unavailability of a resource as shown in Section 6.3.3, hence it is possible to model the different states of physical resources.

In modelling the different states of a resource we found using a state machine is more suitable than an ERS diagram, because state machines are best known for describing the state changes of objects. Hence in this case the ERS diagram will descibe the overall behaviour of the fire dispatch system, while the state machine will be used to focus on the state changes of the physical resources.

We extended the model with the state machine in Figure 6.19, which shows the different states of a physical resource, which are in this case *Available* and *Unavailable*. The available state of a physical resource is in turn divided into two cases when the resource is free, i.e. not allocated, the physical resource can be either *StationAvailable* or *Returning*, while in the case of an allocated physical resource, it can be either *Mobilised* or *InAttendance*.

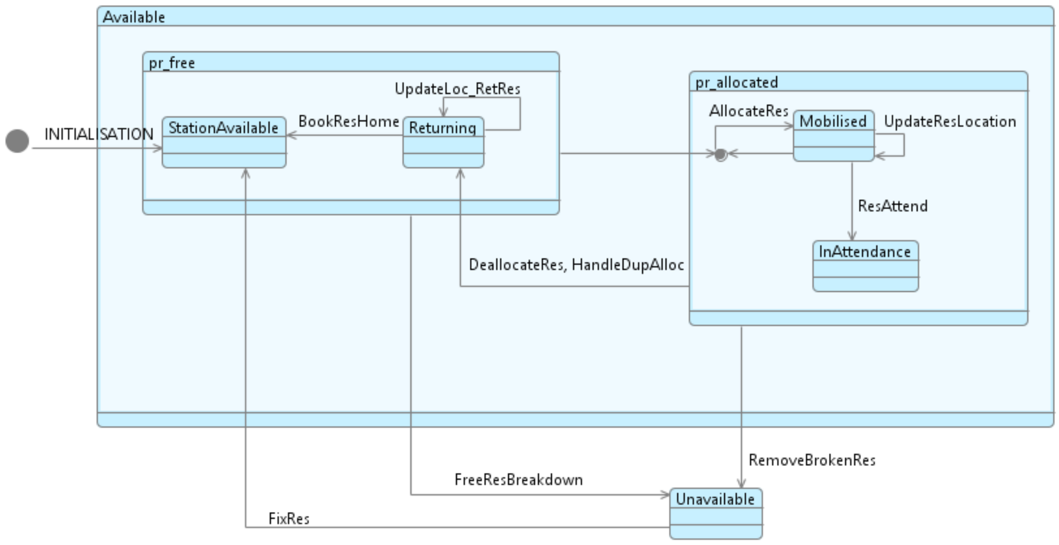
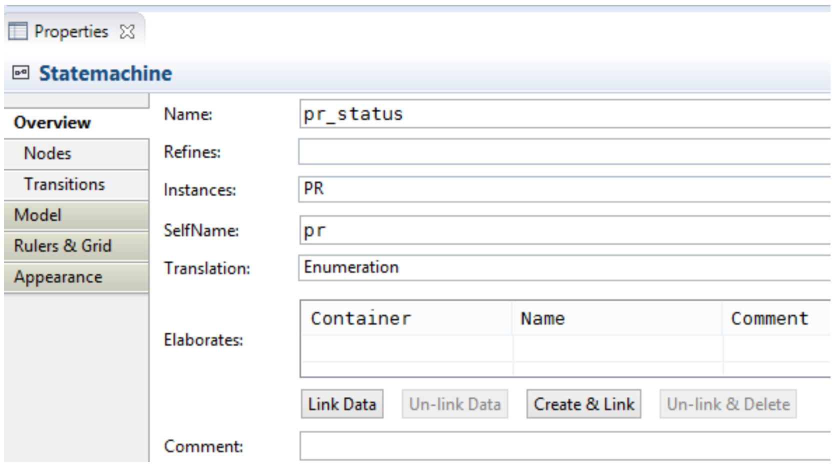


Figure 6.19: State Machine of the Status Change of a Physical Resource

In modelling the state machine in Figure 6.19, we used the iUML-B statemachine (Snook, 2014; Wiki.event-b.org, 2014), which is another form of UML-B (presented in Section 3.3.5 of Chapter 3), but with a stronger integration with Event-B. The dark circle in the middle of the state *pr_allocated* is a feature of iUML-B called *junction* which generates a disjunctive guard.

In the iUML-B state machine, we used the *Enumeration* translation to Event-B, where the other translation option is *Variable* translation, we also applied lifting as shown in Figure 6.20, where we specified the instance set and the self-name to *PR* and *pr* respectively. Using lifting in iUML-B is another feature which allows the statemachine to be lifted to a set of instances so that the state machine represents the behaviour of all the instances in the set (Wiki.event-b.org, 2014), which is *PR* in our case.



Name:	pr_status
Refines:	
Instances:	PR
SelfName:	pr
Translation:	Enumeration
Elaborates:	Container / Name / Comment
Comment:	

Figure 6.20: Properties Overview of the Physical Resource State Machine

The iUML-B translation to Event-B will generate an implicit context to define the different states of the statemachine according to the required translation type. The generated sets are: *pr_status_STATES*, *pr_available_sm_STATES*, *pr_allocated_sm_STATES*, *pr_free_sm_STATES*. These sets represent the different statemachines in Figure 6.19, where we have used four statemachines, the overall statemachine *pr_status_STATES*, which contains another statemachine *pr_available_sm_STATES*, which in turn consists of two statemachines *pr_allocated_sm_STATES* and *pr_free_sm_STATES*. We have used multiple statemachines to make the diagram simpler compared to having lots of transitions from each state.

The different states of the statemachines are defined using the following axioms which are generated from the iUML-B statemachines in Figure 6.19. Each state is defined as a constant in the related statemachine set, and the states ending with the keyword *NULL* corresponds to states outside the current statemachine:

axm_sm1 $partition(pr\_status\_STATES, \{Available\}, \{Unavailable\})$

axm_sm2 $partition(pr\_available\_sm\_STATES, \{pr\_allocated\}, \{pr\_free\},$
$\{pr\_available\_sm\_NULL\})$

axm_sm3 $partition(pr\_allocated\_sm\_STATES, \{Mobilised\}, \{InAttendance\},$
$\{pr\_allocated\_sm\_NULL\})$

axm_sm4 $partition(pr\_free\_sm\_STATES, \{Returning\}, \{StationAvailable\},$
$\{pr\_free\_sm\_NULL\})$

In Figure 6.19, the transitions from one state to another are marked with the event names that update the state of the physical resource. Some of the events are common to the fire dispatch system presented in the ERS diagrams. The events common with the ERS diagrams will be extended with status update (Figure 6.21(a)), whereas the uncommon events are newly added events to the model that are only related to the physical resources, but **not** related to the incidents (Figure 6.21(b)). In both cases the state of the resource before the event execution will be added as a guard, while the status update will be done using action(s) as shown in Figure 6.21.

```
event HandleDupAlloc extends HandleDupAlloc
  where
    @isin_pr_allocated pr_available_sm(pr) = pr_allocated
  then
    @leave_pr_allocated_sm pr_allocated_sm(pr) ≔ pr_allocated_sm_NULL
    @enter_Returning pr_free_sm(pr) ≔ Returning
    @enter_pr_free pr_available_sm(pr) ≔ pr_free
end
```

(a) An ERS Event Extended With Resource Status (*M9*)

```
event BookResHome
  any pr
  where
    @isin_Returning pr_free_sm(pr) = Returning
  then
    @enter_StationAvailable pr_free_sm(pr) ≔ StationAvailable
end
```

(b) New Event Related to the Resource Status Only (*M9*)

Figure 6.21: Event-B Specification of Some of the Events Updating the Resource Status

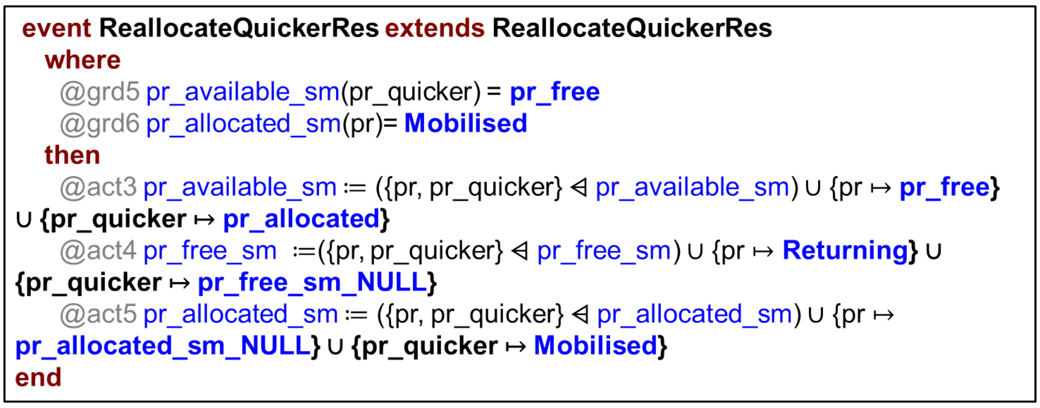
In Figure 6.21, the variables *pr_available_sm*, *pr_allocated_sm* and *pr_free_sm* are automatically defined as total functions relating the physical resources (PR) with the corresponding statemachine set. In addition to the automatically generated invariants, we have manually added invariants to relate the status of a physical resource with *alloc*. These invariants are:

inv_pr_alloc $\forall pr \cdot pr\_available\_sm(pr) = pr\_allocated \Leftrightarrow pr \in dom(alloc)$

inv_pr_alloc_2 $\forall pr \cdot pr\_available\_sm(pr) \in \{pr\_free,\ pr\_available\_sm\_NULL\} \Leftrightarrow pr \notin dom(alloc)$

The first invariant (*inv_pr_alloc*) will ensure that allocated resources are available, while the second invariant (*inv_pr_alloc_2*) will ensure that all free resources and unavailable ones are **not** allocated to an incident.

Only one of the events in the ERS diagram that involve physical resource change was not shown in the statemachine (Figure 6.19), this event is *ReallocateQuickerRes*. The reason for this is that *ReallocateQuickerRes* involves the state change of two different physical resources, the initially allocated physical resource and the quicker physical resource to be allocated. We have done the status change manually and we followed the same approach followed by the statemachine translation, so we extended *ReallocateQuickerRes* as shown in Figure 6.22.



```
event ReallocateQuickerRes extends ReallocateQuickerRes
  where
    @grd5 pr_available_sm(pr_quicker) = pr_free
    @grd6 pr_allocated_sm(pr)= Mobilised
  then
    @act3 pr_available_sm ≔ ({pr, pr_quicker} ⩤ pr_available_sm) ∪ {pr ↦ pr_free}
∪ {pr_quicker ↦ pr_allocated}
    @act4 pr_free_sm  ≔({pr, pr_quicker} ⩤ pr_free_sm) ∪ {pr ↦ Returning} ∪
{pr_quicker ↦ pr_free_sm_NULL}
    @act5 pr_allocated_sm ≔ ({pr, pr_quicker} ⩤ pr_allocated_sm) ∪ {pr ↦
pr_allocated_sm_NULL} ∪ {pr_quicker ↦ Mobilised}
end
```

Figure 6.22: *ReallocateQuickerRes* Extended Manually with the Resource Status Update

The *ReallocateQuickerRes* will only allocate a quicker resource, if that resource is free, i.e., not allocated to another incident, and the status of the current allocated resource is *Mobilised*. When reallocating the quicker resource, the status of the previously allocated resource (*pr*) will be changed to free and that of the quicker resource (*pr_quicker*) will be changed to allocated as shown by *act3*. Moreover in *act4* the status of *pr* will be *Returning* in the sub-state machine *pr_free_sm*, while that of *pr_quicker* will be removed and assigned to the null state (pr_free_sm_NULL). In *act5*, *pr_quicker* will be updated to *Mobilised* while that of *pr* will be removed from the *pr_allocated_sm* state machine.

6.3.6 Evaluation of the Updated Fire Dispatch Case Study

Modelling the fire dispatch system with the new exception handling combinators (Chapter 6), made it possible to model the requirements related to interruptions and cancellation, such requirements were difficult to model previously due to the impact they have on the workflow course. That is why having additional explicit combinators, like the exception handling combinators, is important to show the effect of one sub-flow on the other, whereas in previous combinators all the sub-flows of the combinator were treated the same way.

We showed how applying the *interrupt-combinator* managed to represent how a late duplicate incident discovery can stop future allocations to the incident, and how we handled any allocated resources to the duplicate incident. Additionally, we successfully applied the *retry-combinator* to attempt reallocating a new resource replacing a lost physical resource due to reallocation to higher priority incident or breakdown. Using these explicit combinators showed the possibility of such exceptions and how they are handled, without complicating the diagrams.

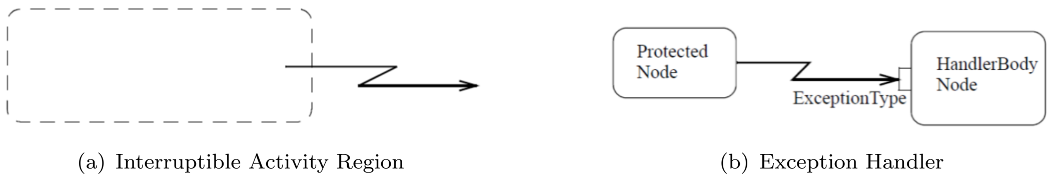
In this model we have also combined a statemachine with the ERS diagrams, this shows how ERS can be easily extended with other control flow approaches, as well as any additional Event-B data.

6.4 Comparison with Other Work on Exception Handling

In this section we compare our ERS exception handling approach to other workflow modelling approaches, such as activity diagrams and BPMN.

6.4.1 UML Activity Diagrams

UML activity diagrams support exception handling mechanisms using Interruptible Activity Regions and Exception Handlers. Interruptible Activity Regions, Figure 6.23(a), are a group of activities where execution can terminate once a certain behaviour connected to an interrupting edge is accepted. Once the region is interrupted the execution will transfer to a target activity outside the interruptible region but connected to the interrupting edge. An interrupting edge will have its source within the interruptible activity region and its target outside the region. Activity diagrams also have explicit exception handlers similar to the "try catch" statement used in programming, Figure 6.23(b). When the execution of the protected node completes whether the exception handler was triggered or not, execution will continue to the actions following the protected node as normal.



(a) Interruptible Activity Region (b) Exception Handler

Figure 6.23: Activity Diagrams Exception Handling Mechanisms (Object Management Group (OMG), 2015)

The ERS exception handling combinators, *interrupt* and *retry*, can be compared to the interruptible activity region due to the way an exception is triggered through external events. Similar to the interruptible activity region, the *interrupt* and the *retry* combinators abort the execution of the normal flow, which in an activity diagram is within the interruptible activity region, and execution is transferred to the interrupting event. Unlike the *retry-combinator*, the interruptible activity region does not retry the execution of the interrupted region until it succeeds. However, it is possible to represent this behaviour if we connect the interrupting event and its handler back to the main flow.

Consider the example of the order workflow using the interruptible activity region, Figure 6.24 which is taken from Object Management Group (OMG) (2015). In Figure 6.24, if *Cancel Order* is triggered, according to activity diagrams the payment flow starting with *Send Invoice* will not be interrupted. However, the join-node needs both *Ship Order* and *Accept Payment* to execute, hence the join-node will never be executed and will be implicitly interrupted. This kind of behaviour cannot be modelled directly with our ERS combinators as all of the normal flow (including payment) is interruptible. Given that the join-node of Figure 6.24 is not executed if the order is cancelled, it is not clear that the behaviour allowed by interruptible activity regions is desirable.

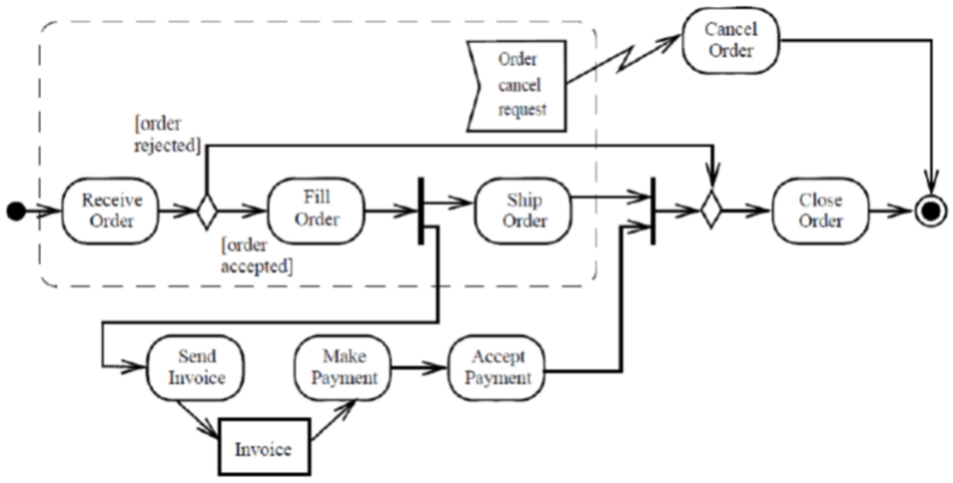


Figure 6.24: Interruptible Activity Region (Object Management Group (OMG), 2015)

6.4.2 BPMN

BPMN supports exception handling and compensation through different types of intermediate events. Intermediate events (Object Management Group, 2011) are represented by a circle which can be attached to the boundary of an activity or task to represent exception handling. There are different types of intermediate events, some of them are interrupting and others are not. We are only interested in interrupting intermediate events like the error intermediate event, represented by a flash symbol, and the cancel event, represented by an X marker. The error intermediate event is used to catch an error and always interrupts the activity it is attached to and diverts the flow according to what the error event is connected to. Therefore, it is similar to the *interrupt-combinator*, and if it is connected back to the same activity, its behaviour will be similart to the *retry-combinator*. Similarly, the cancel event can also interrupt processes but it can be only attached to transaction sub-processes.

BPMN also has a compensation intermediate event, represented by a rewind symbol, which when triggered will roll back completed activities. In compensation events, the interrupting property does not hold as only completed activities can be compensated. However, the compensation will be typically raised by an error handler as part of a cancellation process (Object Management Group, 2011). In compensation, activity roll-back will be done in reverse order of execution and can be performed concurrently if no sequencing is applied. Unlike the ERS *retry* which will only reset the control variables of all the activities in the interrupting region whether completed or not and this will be done concurrently. Compensating other manually added data is the responsibility of the modeller, as we have seen in Figure 6.18, where we refined the interrupting event *IsDuplicate* and added additional events to manually compensate for allocated resources.

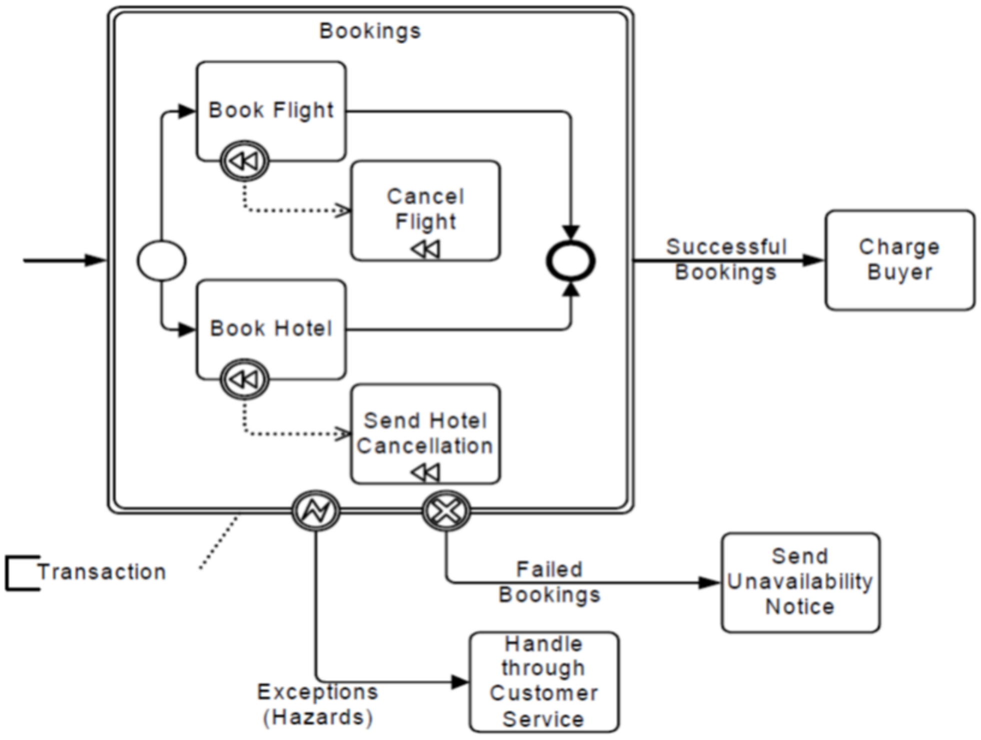


Figure 6.25: Exception Handling Example Using BPMN (Object Management Group, 2011)

Figure 6.25 (Object Management Group, 2011) shows a transaction for flight and hotel bookings, where each transaction has an associated compensation which will be triggered in case of failed bookings, represented by the cancel intermediate event, undoing all successful bookings. Exceptions and hazards are represented by the error intermediate event, which when triggered will interrupt the bookings and divert the flow to be handled by customer service.

In Figure 6.26, we show a possible ERS representation of Figure 6.25. In Figure 6.26, we have represented the transaction for a single instance case. We have also introduced two data variables *ReservedF* and *ReservedH* to represent the successful flights and hotels bookings respectively. The booking cancellation is done manually using Event-B as we can see by the manually added actions in *CancelFlight* and *CancelHotel* events. We have

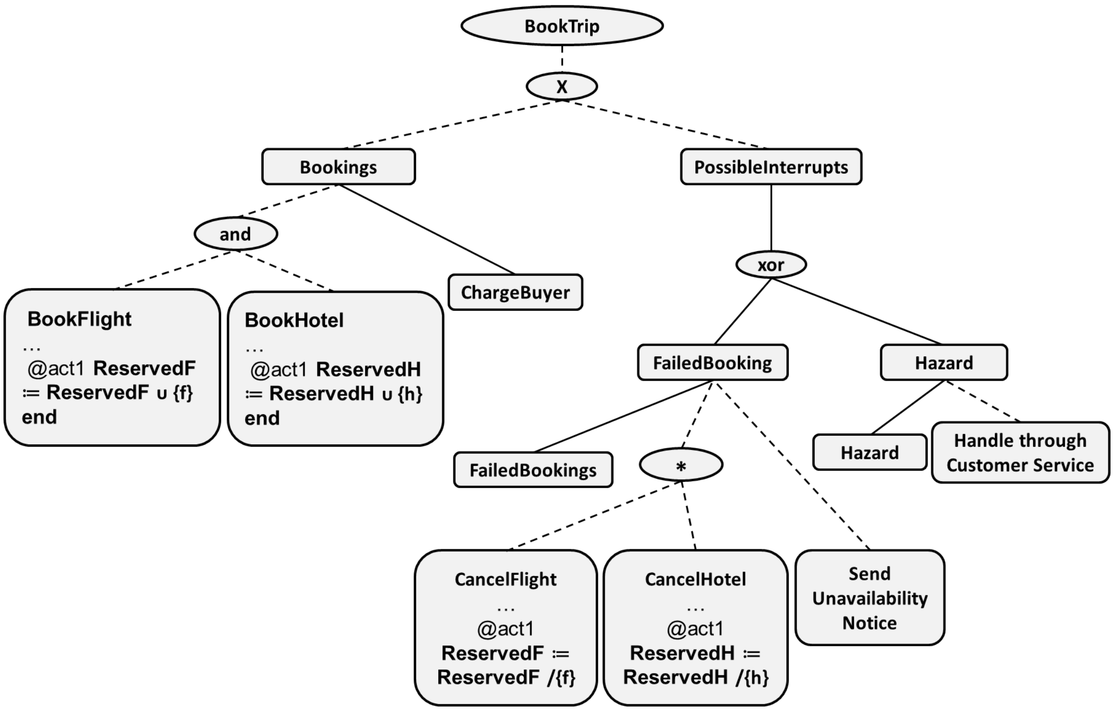


Figure 6.26: Possible ERS Representation of Figure 6.25

decided to model the cancellation events using a *loop* to cover the case where there are no successful bookings.

Regarding multiple instances, compensation in BPMN will be done for all instances, while in ERS as we have seen in Figure 6.15 we can apply it to a single instance. In ERS if we wanted to apply the *retry-consruct* of Figure 6.15 to all the instances then we could have introduced it at a level higher than the *par-replicator*.

6.4.3 Little JIL

Little JIL (Wise et al., 2000) is a formal language, yet graphical, that supports process coordination in hierarchical steps. Figure 6.27 shows the graphical representation of a Little-JIL step, showing how Little-JIL represents control flow and exception handling. Regarding control flow which is specified by the parent step, at the leftmost of the step, Little-JIL supports sequential, parallel, choice and try flows of its sub-steps. In a choice flow, exactly one sub-step is executed and in a try flow, execution of sub-steps starts from left to right and stops when a sub-step completes successfully.

Exception handling which is the main focus of this chapter is represented by the "X" symbol at the rightmost of the step, which can be connected to the exception handler. If the exception did not find a matching handler, it goes up the decomposition tree until finding a match. After the exception handler completes, a continuation badge connected

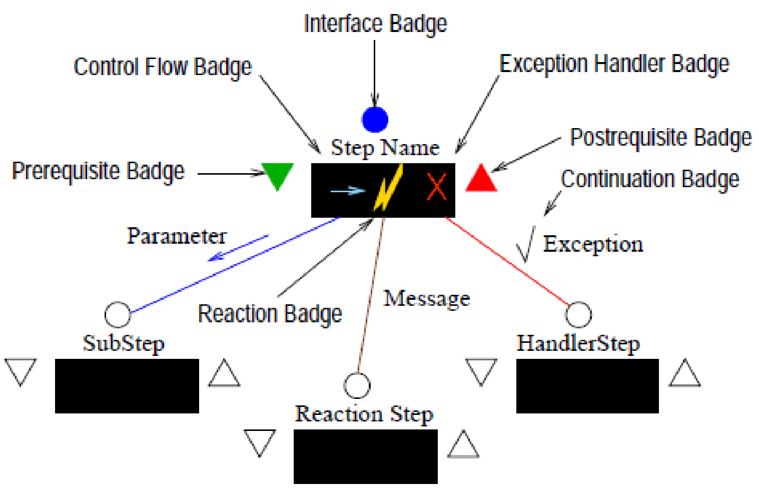


Figure 6.27: Little-JIL Step (Wise et al., 2000)

to the handler will determine whether the step will continue execution, successfully complete, restart execution or rethrow the exception (Group, 2006; Wise et al., 2000).

The ERS *retry-combinator* will be similar to an exception with a restart execution as the continuation badge of the handler, while the *interrupt-combinator* will be like an exception with a successfully complete badge. If there is an event following the *interrupt-combinator*, it will execute if either the normal flow completed execution, or the interrupt flow completed execution. Continuing the execution of the step, even if an exception occurs, is not represented by the ERS exception handling combinators. Rethrowing the exception means after handling the exception, execution will terminate, this can be only represented if there is no follow-on event after the *interrupt-combinator*, such as the *IsDuplicate* interrupt example of the fire dispatch case study in Section 6.3.1. Another difference between the exception handlers and the ERS interrupts is that ERS will inherit the interrupt and the handler which can be introduced as a refinement down the tree, so there is no need to propagate up the tree like little-JIL to look for a matching handler, which in ERS can be only associated to one interrupting flow.

In the Little-JIL step, there is also a lightning bolt symbol in the middle of the step, which represents received messages that can arrive during the execution of the step. The messages are connected to reactive steps which should be acted on immediately once received. The difference with the ERS interrupt is that the reactive steps can execute in parallel with the normal steps, while in ERS the messages which can be represented as interrupting events will stop the execution of the normal flow. Reactive messages unlike exceptions are local to the step and do not propagate up the tree.

6.4.4 Exception Handling and Fault Tolerance Patterns

The workflow initiative (Russell et al., 2006b,c) have also identified patterns for exception handling, which take into consideration four important aspects:

1. What is the exception type.
2. How the work item that captured the exception will be handled.
3. How other work items in the case will be handled
4. What recovery action will be taken.

These four aspects combined together resulted in many patterns. They identified five exception types for expected exceptions, which are work item failure, deadline expiry, resource unavailability, external trigger and constraint violation. They also identify fifteen strategies for exception handling at work item level such as: restart (SRS), force fail (SFF). For handling at the case level they have three cases: continue (CWC), remove current case (RCC) and remove all cases (RAC). Recovery actions are three: no action (NIL), rollback (RBK) and compensate (COM).

The patterns are given abbreviated names related to the last three considerations and grouped according to the exception type. An example pattern is "SFF-CWC-COM" which is one of the patterns specified for the exception type work item failure. The first part of the name relates to work item level, the second part to case level and the last part to recovery action.

In ERS we do not have this large number of patterns and we do not differentiate between exception types. In ERS we give the mechanism to capture the exception and leave it to the user to define the handling process. Regarding *retry* we provide the mechanism to support approaches like rollback and restart because the user should not modify control variables, whereas handling data variables is left to the user, which can be introduced during the refinement of the interrupting events.

Another exception handling pattern classification is done in Lerner et al. (2010) which classifies the exception patterns into three generic approaches as follows:

1. Trying other alternatives: Ordered Alternatives and Unordered Alternatives patterns
2. Inserting behaviour: Immediate Fixing, Deferred Fixing, Retry, and Exception-Driven Rework patterns
3. Canceling behaviour: Reject and Compensate patterns

They also attempted to represent the exception handling patterns using Little-JIL, BPMN and activity diagrams. The three modelling languages provide support for exception handling, some have advantages in certain areas, e.g., Little-JIL provides better support to ordered and unordered alternatives due to the choice and try combinators, while BPMN has a compensate combinator providing better support to compensate patterns.

In ERS there is no direct support for the ordered alternatives pattern as found in Little-JIL with the *try* combinator, but similar to activity diagrams this can be represented by chaining the alternatives using different exception handlers. Unordered alternatives are represented using the *choice* combinator in little-JIL and in activity diagrams using a combination of conditional nodes, a loop and exception handler. In ERS and Event-B the events are atomic so that they cannot be interrupted unless we consider the alternative involves more than one event, in which case it could be interrupted by an exception. If the alternatives consist of one event, i.e., the alternative either execute or not, then this can be done using the *xor-combinator*. Otherwise, it is possible to represent this pattern indirectly using a combination of an *or-combinator* which allows the unordered execution of its branches, the *interrupt-combinator* and additional Event-B guards to check the execution of the *or-combinator* branches and the *interrupt-combinator* branches.

The patterns immediate fixing and retry patterns can be represented by the ERS *interrupt-combinator* and the *retry-combinator* respectively. Immediate fixing requires adding extra work before joining back the normal flow, this can be represented by the *interrupt-combinator*, since we can add additional actions after interrupting the normal flow, and as explained earlier if there is a flow following the *interrupt-combinator*, it can execute if either the normal flow completed execution or the interrupting flow completed execution. The *retry* pattern allows trying the activity again if it fails by possibly doing some context update before attempting the activity again, this behaviour can be achieved by the *retry-combinator*, which allows trying the activity again by resetting the control variables of the ERS diagram.

On the other hand deferred fixing and exception-driven rework are other versions of the immediate fixing and retry but allowing time to pass between the occurrence of the exception and its handling. There is no direct support for deferred fixing in ERS. Regarding rework and the ERS *retry-combinator* this is not possible because the *retry-interrupt* will reset the control variables of the normal flow once it occurs and the follow-on event depends on the normal flow execution, hence we cannot postpone the retrying of the normal flow.

The intent of the reject pattern is that some intended behaviour can become undesirable and the agent must be notified allowing the agent to take action or make changes and try again, so that the reject pattern can be used to abort a process or part of a process. In this case this pattern can be represented by either the *interrupt* or the *retry* combinator of ERS depending on the required handling as both combinators can interrupt the normal flow which is the intended behaviour.

Regarding the compensate pattern, which is part of the cancelling behaviour as described by Lerner et al. (2010), it provides a mechanism to determine what is the successfully completed work and undo it with the associated compensation actions, when cancellation of the activity occurs. If we compare the compensate patten to the ERS *retry-combinator*, *retry* will only reset the control variables regardless if they have executed or not, and it is left to the modeller to undo other data related activities. Hence in order to undo only completed activities using either the *retry* or *interrupt* combinator of ERS, additional checks are needed, here we are talking about undoing data related actions. Hence the compensate pattern is not directly supported by ERS as there is no direct mechanism to only undo completed activities.

In Ball and Butler (2007), the authors define fault tolerance patterns for multi agent systems in Event-B, which are *timeout*, *refuse*, *cancel*, *failure* and *not-understood* pattern. These patterns are related to communication messages between different agents. Of those patterns, the cancel pattern is the only one that can be compared to the ERS exception handling combinators. The cancel pattern can be represented using the ERS *interrupt-combinator*, where an agent no longer requires to complete an action. The refuse pattern can be represented by the *xor-combinator* as the agent have the choice to accept or refuse to perform an action. It is also possible to represent the fail pattern using the *interrupt-combinator* or the *xor-combinator*, as the failure pattern is similar to the refuse pattern but it happens after commitment of an agent to an action while refuse happens before commitment. The timeout and the not-understood patterns are not represented explicitly by ERS.

6.4.5 Other Work on Compensation and Formal Languages

Compensation has also been studied by integrating formal methods with other approaches. In Butler and Ferreira (2003); Butler et al. (2005a), the authors use StAC (Structured Activity Compensation), which is a business process modelling language that was introduced by Butler and Ferreira (2000) to support parallel and sequential behaviour and most importantly to support compensation. Similar to our approach where we use ERS for the coordination of events and the Event-B formal method for data handling, they use StAC to represent the orchestration of business transactions while the B method is used for data manipulation.

Compensation in StAC is done by defining compensation pairs, one representing the normal process and the other representing the compensating process. The execution of the compensating process is done in reverse order of the normal process. The compensation process is remembered until it is no longer required, for this two operators are introduced one called *reverse* which refers for the invocation of compensation or its execution and the other is called *accept* which indicates that the compensation process is no longer required and can be forgotten. They also provide a mechanism for compensation nesting using scoping brackets. They also extend StAC by introducing indexing for compensation which is used to support multiple compensations. An important operator in indexed StAC is the merge operator which allows merging all the compensating tasks belonging to the same task in parallel.

Indexing in StAC also allows the modelling of named compensation in BPEL, where in Butler et al. (2005a) they show how StAC can be used to give BPEL formal semantics. In Chessell et al. (2002), they also use indexed StAC to define two forms of compensation called selective and alternative. In selective compensation, the reversal chooses which activities should be compensated, while in alternative compensation, the reversal can choose which compensation task to execute among several alternatives.

Compared to our approach, we only provide the interrupting mechanism which the user can refine and define their own exception handling mechanism, which could also include different compensation alternatives or applying interleaving of different compensating tasks depending on what ERS combinators the modeller choose and the activities they define. In this sense, our ERS exception handling combinators are similar to the StAC *reverse* operator but without explicitly defining the compensation mechanism, with *retry* having an additional feature to support compensation by facilitating the activity redo. Regarding scoping, the ERS interruption will not be applicable once the normal flow completes execution, but we also allow the nesting of the different exception handling operators, as seen in Section 6.3.

In Butler et al. (2005b), they extend CSP (Hoare et al., 1985) with compensating operators, where they also use the concept of compensation pairs. Compensation pairs are similar to the StAC compensation pairs, where they define the compensation task for a successfully completed task. In Compensating CSP, they also define the operator *THROW* for throwing interrupts and a special operator for handling interrupts. Our ERS *interrupt-combinator* can be similar to the combination of interrupt *Throw* and interrupt handler.

6.5 Conclusion

The *Interrupt* combinator can be used whenever it is possible to abort a flow or a sub-flow at any time. The *Retry* combinator can be used whenever there is a need to stop the flow or sub-flow and redo it again.

Having a combinator like the *Interrupt* combinator has the advantage of avoiding multiple *xor* branches, since the interrupting event can start at any time before the completion of the normal flow. The *retry* combinator will reset the control variables of the interrupted flow, allowing the flow to be retried again thus allowing behaviour such as rollback. Any compensations can be added to the interrupting events or can be added through refinement following the interrupting event. By this ERS can support both forward and backward recovery actions.

Compared to other approaches, the *interrupt* and *retry* combinators capture similar exception handling mechanisms as other approaches, but in ERS we do not differentiate between the exception types. Moreover ERS does not support transaction compensation behaviour as found in languages such as BPMN, but supports a generic handling approach which is decided by the modeller according to the application.

Chapter 7

Travel Agency Case Study

7.1 Introduction

In this chapter, we model the on-line booking system of a travel agency. In modelling the case study, we apply some of the extended ERS approach, presented in Chapter 4 and Chapter 6, and Event-B. The model represents the on-line booking system of a trip, where a user can book different flights, hotels and/or cars.

In modelling this case study, we show the importance of having an explicit *interrupt* combinator in ERS, which allows the user to quit the entire booking process at any time. In addition to the importance of having a flexible replicator like the *par-replicator*, whenever the number of instances to be executed cannot be predetermined.

7.2 Requirements Overview of the Travel Agency Case Study

The on-line booking system of a travel agency allows a user to select and book multiple flights, cars and/or hotels. If a user selected a flight/car/hotel and had them in their itinerary for a long time without booking, the flight/hotel/car will be removed from their itinerary. In addition, it is possible for the user to cancel the whole trip at any time before checking out.

Many other approaches have modelled the travel agency example using compensations, such as Chessell et al. (2002), where they used StAC and Wong and Gibbons (2011) where they used BPMN.

7.3 Incremental Development of the Travel Agency Case Study

The Event-B model of the travel agency case study consists of two contexts representing the static part of the model, and one abstract and four refinement levels representing the dynamic part of the model as follows:

- **Abstract Level:** Models starting a trip, getting the itinerary and the possibility of quitting.
- **First Refinement:** Refines the trip itinerary to more detailed flight, car and hotel itinerary.
- **Second Refinement:** Introduces booking flights, cars and hotels.
- **Third Refinement:** Introduces the possibility of removing a flight, car and/or hotel if not booked.
- **Fourth Refinement:** Introduces deadline to remove flight, car and/or hotel from itinerary if not booked within a certain time.

In the following sections, we only show the refinement and the Event-B specifications of the flight itinerary, but the same applies to car and hotel itineraries. The complete version of the model is presented in Appendix C.

7.3.1 Abstract and First Refinement

We model the travel agency booking system for multiple instances, where at the context we define the carrier set *TRIP* representing all the possible trips. The user can quit the trip at any time before checkout, that is why we introduce *Quit* using the *interrupt-combinator*, as shown in Figure 7.1.

In Event-B the effect of *Quit* appears in the generated guard $t \notin Quit$ in all the leaf events of *TripItinerary*, which are in this case: *FlightItinerary*, *CarItinerary*, *HotelItinerary* and *Checkout*. The user cannot quit the trip if *Checkout* is completed and the *interrupt-combinator* will ensure this by the guard $t \notin Checkout$, which is added to *Quit*, in addition to the invariant: $partition((Checkout \cup Quit), Checkout, Quit)$.

We can see in Figure 7.1, that the last event of the normal sub-flow of the *interrupt-combinator* must be always the solid line, which is in this case *Checkout*. As explained in Section 6.2.1 of Chapter 6, the importance of the *interrupt-combinator* appears after refining the normal child of the *interrupt-combinator*, to show that the interrupting event (*Quit*) can stop the execution of the normal sub-flow at any point before completion.

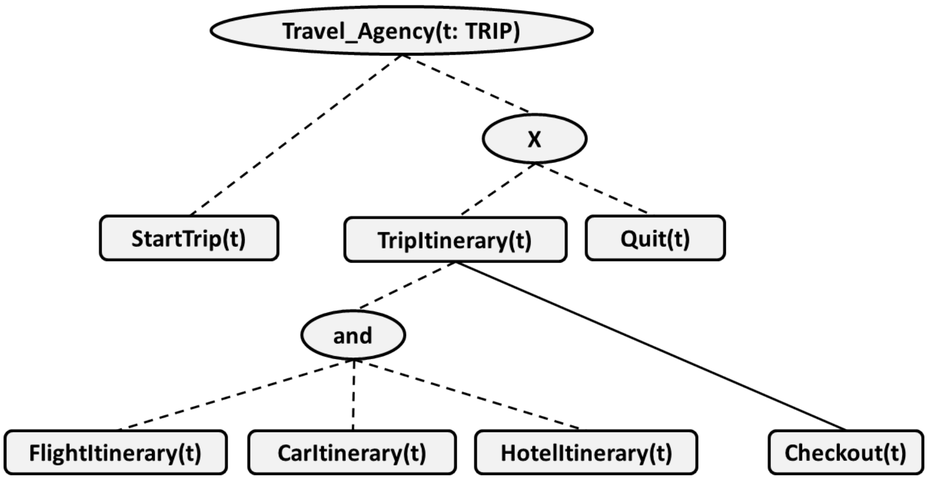


Figure 7.1: ERS Diagram for the Abstract and First Refinement Levels of the Travel Agency System

7.3.2 Second and Third Refinement

In the second refinement we extend the context to introduce the sets *FLIGHT*, *CAR* and *HOTEL*, representing the possible flights, cars, hotels that can be added to the trip itinerary respectively. In Figure 7.2, we show the refinement of *FlightItinerary*, where *CarItinerary* and *Hotelitinerary* will be refined in a similar way but when applying the *par-replicator*, we use the parameter sets *CAR* and *HOTEL* respectively.

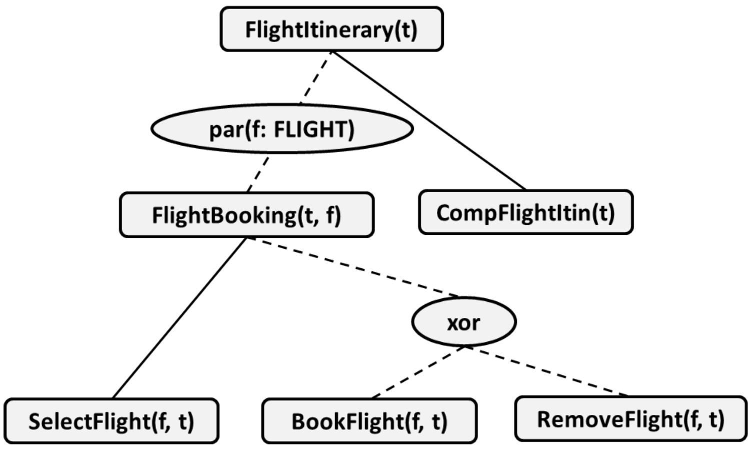
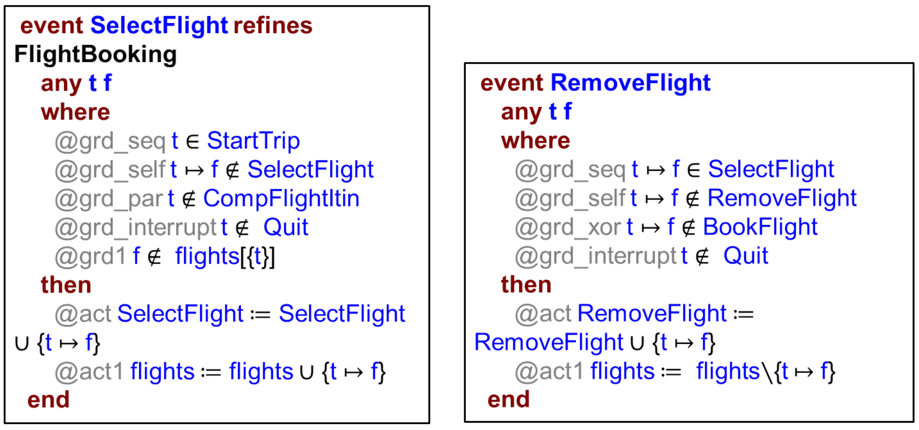


Figure 7.2: ERS Diagram for the Second and Third Refinement Levels of the Travel Agency System

The *FlightItinerary* is refined to book or remove the selected flights from the flight itinerary of the trip for individual flights. When selecting flights, similarly for hotels and cars, we apply the *par-replicator* because the number of flights to be selected is unknown. In the third refinement, we also introduce three data variables *flights*, *cars* and *hotels* representing those flights, cars and hotels added to the trip itinerary. The data variables are defined manually using the following invariants, which relate trips to flights, cars and hotels. We use a relation in defining these variables because the same flight can be booked by different trips and a trip can book different flights. Here, the carrier set *FLIGHT* represents the flight number that is why it can be booked by several trips, the same applies to hotels and cars.

$$\textbf{inv\_flights}\ flights \in TRIP \leftrightarrow FLIGHT$$
$$\textbf{inv\_cars}\ cars \in TRIP \leftrightarrow CAR$$
$$\textbf{inv\_hotels}\ hotels \in TRIP \leftrightarrow HOTEL$$

In the model once a flight is selected (*SelectFlight*), it is added to the flight itinerary which is represented by the variable flights (Figure 7.3(a)), and if *RemoveFlight* is executed, it is removed from the set $flights[\{t\}]$ (Figure 7.3(b)), which represents all the flights of trip t. In Figure 7.3, only the data related guard *grd1* and the action *act1* of both events are added manually, while the rest are generated from the ERS diagram in Figure 7.2.



```
 event SelectFlight refines
FlightBooking
   any t f
   where
    @grd_seq t ∈ StartTrip
    @grd_self t ↦ f ∉ SelectFlight
    @grd_par t ∉ CompFlightItin
    @grd_interrupt t ∉  Quit
    @grd1 f ∉  flights[{t}]
   then
    @act SelectFlight ≔ SelectFlight
∪ {t ↦ f}
    @act1 flights ≔ flights ∪ {t ↦ f}
 end
```

(a) *SelectFlight* Event

```
event RemoveFlight
   any t f
   where
    @grd_seq t ↦ f ∈ SelectFlight
    @grd_self t ↦ f ∉ RemoveFlight
    @grd_xor t ↦ f ∉ BookFlight
    @grd_interrupt t ∉  Quit
   then
    @act RemoveFlight ≔
RemoveFlight ∪ {t ↦ f}
    @act1 flights ≔  flights\{t ↦ f}
 end
```

(b) *RemoveFlight* Event

Figure 7.3: Event-B Specification of the Events Updating the Flight Itinerary

We have represented the relation between *SelectFlight*, *RemoveFlight* and *flights* using the following invariants:

$$\textbf{inv\_flights\_select}\ flights \subseteq SelectFlight$$
$$\textbf{inv\_flights\_remove}\ RemoveFlight \cap flights = \phi$$

Similar invariants are also added manually relating *hotels* and *cars* with their corresponding control variables. The first invariant ensures that *flights* cannot be added to the itinerary unless selected, while the second invariant ensures that a flight cannot be in the itinerary, if it is removed.

The *par-replicator* refinement rules, Section 4.5.1 of Chapter 4, will ensure that the flight itinerary cannot be completed (*CompFlightItin*), unless all the selected flights are either booked or removed from the itinerary, as shown by *grd_par* of *CompFlightItin* in Figure 7.4, which is also ensured by an invariant.

```
event CompFlightItin refines CompFlightItin
  any t
  where
    @grd_seq_ t ∈ StartTrip
    @grd_self t ∉ CompFlightItin
    @grd_par SelectFlight[{t}] = BookFlight[{t}] ∪  RemoveFlight[{t}]
    @grd_interrupt t ∉  Quit
  then
    @act CompFlightItin ≔ CompFlightItin ∪ {t}
 end
```

Figure 7.4: *CompFlightItin* Event

If a user decides to *Quit*, all the flights corresponding to the trip will be removed from the flight itinerary, similarly for cars and hotels. In this case we decided to remove all the flights, cars, and hotels related to the quitted trip in one step as shown by actions *act1*, *act2* and *act3* of Figure 7.5. However it is possible to remove each booked flight, car and hotel separately using an approach similar to the refinement of *IsDuplicate* when handling allocated resources (Figure 6.18)

```
event Quit refines Quit
  any t
  where
    @grd_seq t ∈ StartTrip
    @grd_self t ∉  Quit
    @grd_interrupt t ∉ Checkout
  then
    @act Quit ≔ Quit ∪ {t}
    @act1 flights ≔ {t} ⩤ flights
    @act2 hotels ≔ {t} ⩤ hotels
    @act3 cars ≔ {t} ⩤ cars
 end
```

Figure 7.5: Refinement of Quit to Remove Booked Flights, Hotels and Cars from Itinerary

In Figure 7.6, we show another possible approach to deal with booked flights, hotels and cars, by refining the *Quit* event and handling each booked item in the itinerary individually. However in the model, we have done it on one step as shown in Figure 7.5, to show different possible ways depending on the case study.

7.3.3 Fourth Refinement

In the last refinement level, we do not use ERS diagrams, we only extend the model with time constraints, to remove an item from the itinerary, if it is not booked after a certain time. We follow a similar approach as presented in Sarshogh and Butler (2011)

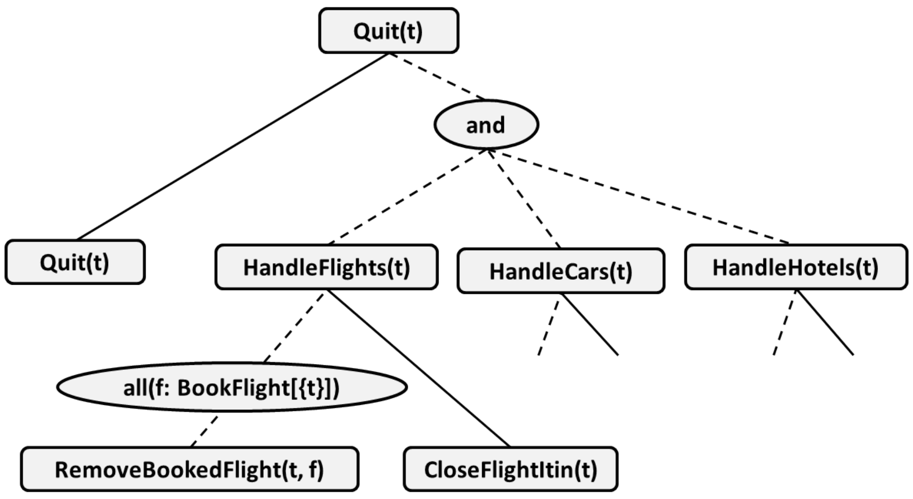


Figure 7.6: Another Possible Handling of Booked Itinerary

for modelling the expiry pattern, where we first record when the item was selected and then add a guard to the booking event to check if it passed a certain time.

We introduce the data variables *time* and *deadline* to respectively record the current time and define a time limit an item can be reserved for a trip before being removed. The data variables *tFlight*, *tCar* and *tHotel* will record the time an item is selected. These data variables are defined manually using the following invariants:

inv_time $time \in \mathbb{N}$

inv_deadline $deadline \in \mathbb{N}$

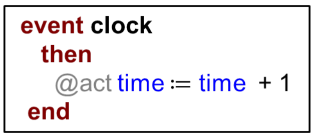
inv_tFlight $tFlight \in flights \rightarrow \mathbb{N}$

inv_tCar $tCar \in cars \rightarrow \mathbb{N}$

inv_tHotel $tHotel \in hotels \rightarrow \mathbb{N}$

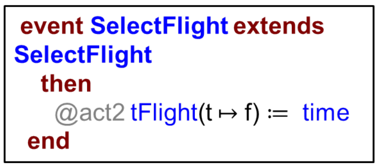
The data variables, *time* and *deadline*, are defined as natural numbers, while *tFlight*, *tCar* and *tHotel* are total functions assigning each flight, car, and hotel in the itinerary to a natural number representing the selection time. In Figure 7.7, we show how the time is introduced to the model. First in Figure 7.7(a), we introduce a new event *Clock* which only increments time, then once a flight is selected, similarly for cars and hotels, we set its time to the current time as shown in Figure 7.7(b).

In Figure 7.8, we extend the flight booking and removal with time where we do the same for cars and hotels. When booking a flight, we need to check that it did **not** pass the deadline as shown in *grd1* of Figure 7.8(a). When removing a flight from an itinerary, we also remove its time tracking, as it is not required any more as shown by *act2* of Figure 7.8(b). When removing a flight we did not add a guard to check that it passed the deadline, to give the user the flexibility to remove unwanted flights.



```
event clock
  then
    @act time ≔ time + 1
 end
```

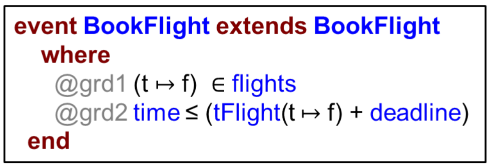
(a) *Clock* Event



```
event SelectFlight extends
SelectFlight
  then
    @act2 tFlight(t ↦ f) ≔ time
 end
```

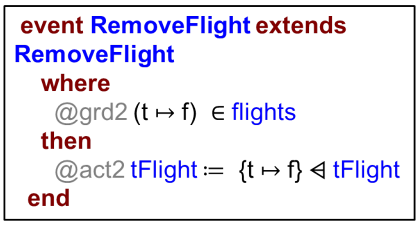
(b) Extended *SelectFlight* Event

Figure 7.7: Setting the Time of an Itinerary Item



```
event BookFlight extends BookFlight
  where
    @grd1 (t ↦ f) ∈ flights
    @grd2 time ≤ (tFlight(t ↦ f) + deadline)
 end
```

(a) Extended *BookFlight* Event



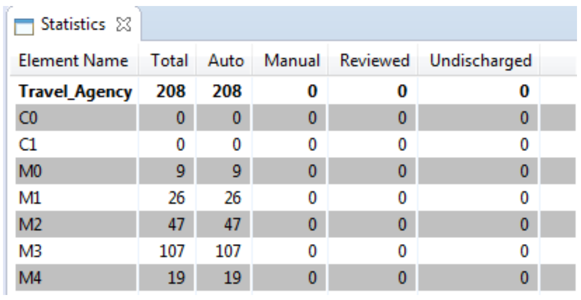
```
event RemoveFlight extends
RemoveFlight
  where
    @grd2 (t ↦ f) ∈ flights
  then
    @act2 tFlight ≔ {t ↦ f} ⩤ tFlight
 end
```

(b) Extended *RemoveFlight* Event

Figure 7.8: Extending the Flight Events with Time

7.4 Evaluation of the Travel Agency Case Study

All the proof obligations of the travel agency case study were discharged automatically by the Rodin provers as shown in Figure 7.9, which shows the simplicity of the invariants generated by the ERS diagrams.



Statistics

Element Name	Total	Auto	Manual	Reviewed	Undischarged
Travel_Agency	**208**	**208**	**0**	**0**	**0**
C0	0	0	0	0	0
C1	0	0	0	0	0
M0	9	9	0	0	0
M1	26	26	0	0	0
M2	47	47	0	0	0
M3	107	107	0	0	0
M4	19	19	0	0	0

Figure 7.9: Statistics of the Travel Agency Model

In the model, when a flight is removed ($RemoveFlight$) then it is not possible to book the same flight again, due to the disabling guard ($t \mapsto f \notin SelectFlight$) in the $SelectFlight$ event. In this case it was fine because the requirements are not clear about that situation. However, a better solution to the model is applying the *retry* combinator, which allows the flight to be added again to the itinerary as shown in Figure 7.10. In this case, it would be possible to select the removed flight again because the interrupting event ($RemoveFlight$) will reset the control variables $SelectFlight$, $CustomiseFlight$ and $BookFlight$. In Figure 7.10, we have added the event $CustomiseFlight$, which is **not** added in Figure 7.2 to show how in this case the *retry* combinator can appear clearly as a more

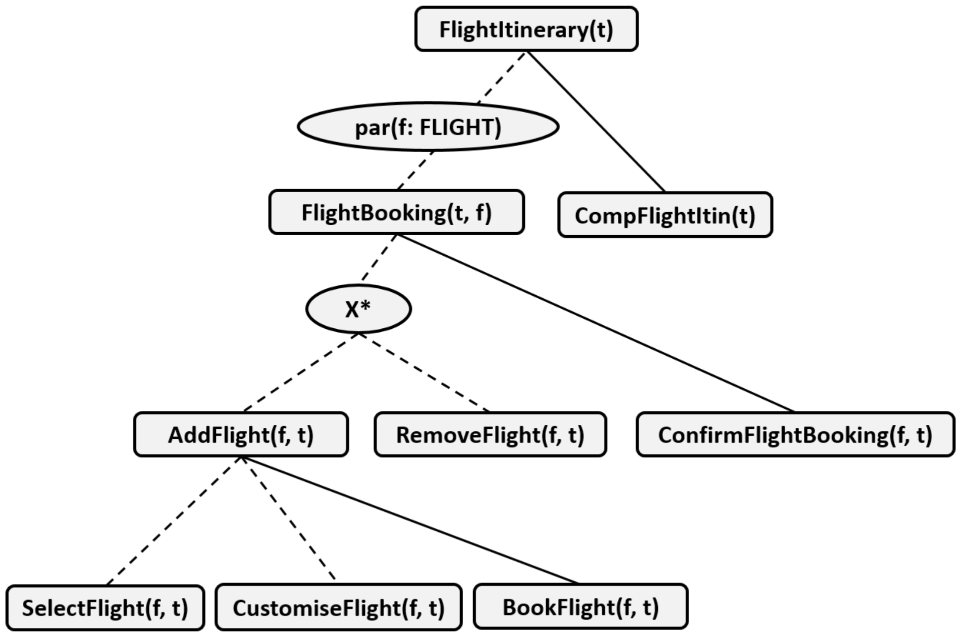


Figure 7.10: Flight Itinerary Using the *Retry* Combinator

suitable combinator than the *xor-combinator* to model this interrupting behaviour. The same also applies to the car and hotel bookings.

Another possibility would be using the generalised *loop*. All the different options in modelling the same behaviour show how the details needed in the requirements can make one combinator more suitable than the other. In addition to the flexibility the new ERS extensions gave us the ability to cover more cases.

In this model, we have also shown how we related the data variables to the control variables like invariants *inv_flights_select* and *inv_flight_remove* in Section 7.3.2.

In Section 7.3.3, we have shown how we can easily extend the ERS models with timing patterns like the expiry pattern. Such timing patterns combined with exception handling combinators like in Figure 7.10 seems very compatible and can be studied further in the future.

In addition to that, we have seen in Section 7.3.2, how the guard generated by ERS as a result of the *par-replicator* refinement has ensured the proper behaviour of the model, where in this case an itinerary cannot be completed if a selection is not dealt with, by either booking or removing it form the itinerary (*grd_par* in Figure 7.4).

7.5 Conclusion

In the travel agency case study, we have shown how we applied some of the ERS extensions introduced in chapters 4 and 6. Using these extensions, we successfully managed to model the requirements. The new ERS extensions have proved to be necessary where in some cases it was not possible to model the behaviour using the original ERS alone.

We have also shown how we can extend the models generated by ERS with additional patterns like the expiry pattern described in Section 7.3.3, where ERS controlled the flow of the events and using Event-B we handled the data changes.

Such distinction between the control flow and the data handling resulted in simpler diagrams that can be easily understood. The distinction between the control flow and the data handling did not prevent us from explicitly relating the control variables and the data variables, as we have seen in Section 7.3.2.

Chapter 8

ERS: Formal Specification and Tool Support

8.1 Introduction

In this chapter we will formally describe the ERS pattern extensions presented in chapters 4 and 6, using "Augmented Backus–Naur Form" (ABNF) (Overell and Crocker, 2008). The new ERS extensions involve some changes to the existing ERS translation rules which were defined by Salehi (Salehi Fathabadi, 2012; Salehi Fathabadi et al., 2012; Fathabadi et al., 2014) to include the new pattern changes, in addition to some new rules that are only related to the new patterns.

Regarding tool support, the ERS extensions are partly developed in the ERS plug-in (Dghaym et al., 2016), a tool supporting the ERS approach in Rodin (Section 3.3.4). However, as future work, the plug-in is extending to support all of the extensions.

8.2 ERS Language Formal: Specification

In formalising the ERS extensions, we follow the same approach used to formalise the existing ERS language which adopts "Augmented Backus–Naur Form" (ABNF) (Overell and Crocker, 2008). ABNF specifications are a set of derivation rules written as "$rule = definition$". The following ABNF (Overell and Crocker, 2008) operators are used to descibe the current and the extended ERS language:

- Elements enclosed in parenthesis are treated as one element and their order is important.
- Elements enclosed in square brackets are optional.

- Terminal values which represent string values are enclosed between double quotes.
- Alternatives are separated by forward slash ("/"), i.e., A/B means A or B.
- A star operator ("*") preceding an element indicates repetition. We can also specify upper and/or lower bounds by $(m{*}n)$, where m represents the lower bound and n represents the upper bound, e.g., $*A$ indicates the possible repetition of A zero or more times, $2{*}A$ indicates that A can be repeated 2 or more times etc.

The syntax of the original ERS language is presented in Table 8.1, while the extended ERS syntax is in Table 8.2, where we highlight the extensions in red.

par	=	(name)(type)
flow	=	"flow"(name, *par, sw)(1*child(ref))
child	=	"leaf"(name)/constructor/1*flow
cons-child	=	"leaf"(name)/1*flow
constructor	=	("and"/"or"/"xor")(2*cons-child)/("all"/"some"/"one")(par) (cons-child)/"loop"(cons-child)

Table 8.1: Original ERS Syntax

par-exp	=	(name)(type)[range-expression]
flow	=	"flow"(name, *par-exp, sw)(1*child(ref))
child	=	"leaf"(name)/constructor/1*flow
cons-child	=	"leaf"(name)/1*flow
constructor	=	("and"/"or"/"xor")(2*cons-child)/("all"/"some"/ "one"/"par-rep")(par-exp)(cons-child)/"loop" (1*cons-child)/exception-combinator
exception-combinator	=	("interrupt"/"retry")(normal-child)(interrupting-child)
normal-child	=	cons-child
interrupting-child	=	cons-child

Table 8.2: Extended ERS Syntax

The ERS syntax in Tables 8.1 and 8.2 can be described as follows:

- "*par*" represents a parameter, which can be added to an event to indicate multiple instances modelling. It is updated to "*par-exp*" in order to include the property "*range_expression*" to allow using variables and not only carrier sets and constants. The properties "*name*" and "*type*" remain the same describing the name and the type of the parameter.
- A "*flow*" represents a single root of an ERS diagram, the only difference in the updated version is the use of "*par-exp*" instead of "*par*". A flow has a name, zero or more parameters and the boolean property "*sw*". The boolean property "*sw*" is one in case of strong sequencing and zero in case of weak sequencing. A flow has

also one or more children with the boolean property "*ref*" referring to whether the child is refining or not. "*ref*" will be one in case of a solid line and zero in case of a dashed line.

- The "*child*" and the "*cons-child*" remain the same. A "*child*" can be a "*leaf*", a "*constructor*", or one or more "*flows*" when it is further decomposed during a refinement. A "*cons-child*" is similar to a "*child*", but limited to be a "*leaf*" or "*flow(s)*" and not a "*constructor*".
- A "*constructor*" can be a non-replicator ("*and*", "*or*", "*xor*") with two or more "*cons-child*". Alternatively, a "*constructor*" can be a replicator with one parameter and one "*cons-child*". A replicator can be "*all*", "*some*", "*one*", or a "*par-rep*" which is added in the updated version; or a "*constructor*" can be a "*loop*" which is updated to permit one or more "*cons-child*". Otherwise, a constructor can be a an "*exception-combinator*" which is introduced in Chapter 6.
- An "*exception-combinator*" has two child nodes, both are "*cons-child*" nodes, but since its child nodes are not symmetrical, we give them different names ("*normal-child*", "*interrupting-child*") to distinguish between them.

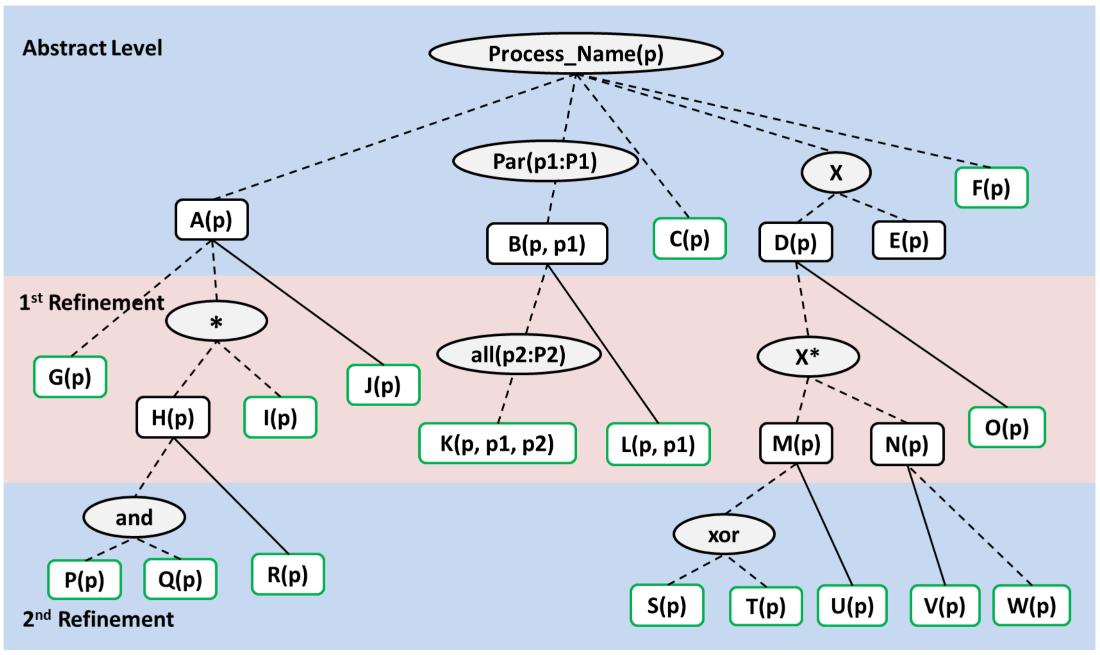


Figure 8.1: An Example of an ERS Diagram

In order to aid understanding of the extended syntax (Table 8.2), we present an example of ERS diagram in Figure 8.1 and the associated syntax in Table 8.3. This example will be gradually explained to present the translation rules in the following sections.

Table 8.3: ABNF Representation of the ERS Example in Figure 8.1

Abstract Level:

```
flow(process_Name, p, 1)
         (leaf(A)(0),
          par-rep(p1)(leaf(B))(0),
          leaf(C)(0),
          interrupt(leaf(D))(leaf(E))(0),
          leaf(F)(0))
```

1st Refinement:

```
flow(process_name, p, 1)
          (flow(A, p, 1)
                    (leaf(G)(0),
                     loop(leaf(H), leaf(I))(0),
                     leaf(J)(1)),
             par-rep(p1)(flow(B, (p, p1), 1)
                                  (all(p2)(leaf(K))(0),
                                   leaf(L)(1)))(0),
             leaf(C)(0),
             interrupt(flow(D, p, 1)
                              (retry(leaf(M))(leaf(N))(0),
                               leaf(O)(1)))
                           (leaf(E))(0),
             leaf(F)(0))
```

2nd Refinement:

```
flow(process_name, p, 1)
          (flow(A, p, 1)
                    (leaf(G)(0),
                     loop(flow(H, p, 1)
                              (and(leaf(P), leaf(Q))(0),
                               leaf(R)(1)),
                            leaf(I))(0),
                     leaf(J)(1)),
             par-rep(p1)(flow(B, (p, p1), 1)
                                  (all(p2)(leaf(K))(0),
                                   leaf(L)(1)))(0),
             leaf(C)(0),
             interrupt(flow(D, p, 1)
                              (retry(flow(M, p, 1)(xor(leaf(S), leaf(T))(0), leaf(U)(1)))
                                       (flow(N, p, 1)(leaf(V)(1), leaf(W)(0))))(0),
                               leaf(O)(1)))
                           (leaf(E))(0),
             leaf(F)(0))
```

8.3 Auxiliary Functions

In defining the new translation rules, we are reusing some of the auxiliary functions defined in Salehi Fathabadi (2012), which are updated to include the new extensions in chapters 4 and 6, in addition to some newly defined functions. The auxiliary functions are used to help in defining invariants and guards in the output Event-B element of the translation rules.

A more detailed definition of the auxiliary functions reused from Salehi Fathabadi (2012) and how they are updated to include the new ERS extensions can be found in Appendix D, but now they can be summarised as follows:

- **list_of_leaves:** Is a recursive function used to traverse down the sub-tree of a child to get the leaves at the final refinement level of the sub-tree. The result of this function is a list of leaf events with their names and parameters. We have not used this function in our translation rules but used two modified versions, *list_first_child_leaves* and *list_child_leaves*, which will be introduced in Section 8.3.1. For example, if we apply *list_of_leaves* to the sub-flow $A(p)$ in Figure 8.1, the result is $G(p)$ in a strong sequencing interpretation and $J(p)$ in a weak sequencing interpretation. If we apply the function to $D(p)$, the result is $S(p), T(p)$ in a strong sequencing interpretation and $O(p)$ in a weak sequencing interpretation. However, the function that we use *list_first_child_leaves* (which will be introduced in Section 8.3.1) returns the same result in both strong and weak sequencing.
- **predecessor:** Is used to find the previous node of a leaf. This function is needed when defining the sequencing between two leaves. The *predecessor* of the i^{th} child of a sub-tree which is not the first child of the sub-flow is ($i^{th} - 1$ child of the sub-tree, whereas, that of the first child of a sub-tree is its parent *predecessor*. In the case the *predecessor* is a *loop* or a *par-replicator*, we find their *predecessor* again due to the zero execution case in both combinators (Figure D.1 in Appendix D).
- **successor:** Is used to find the next node of a *loop*. The successor of the i^{th} child of a flow is its right sub-tree which is the $i^{th} + 1$ child of a flow. A *loop* cannot be the first or the last child of a flow, so the i^{th} child of a flow is never the last child. This function is updated to include the *par-replicator*, which also cannot be the last child of a flow. In case the $i^{th} + 1$ child of a flow is a *loop* or a *par-replicator* then we find its successor and so on. Regarding the *loop* and the *par-rep*, we also add another restriction that they **cannot** be followed by an exception handler combinator (*interrupt*, *retry*), therefore the successor of a *par-rep* and a *loop* cannot be an exception handler combinator (Figure D.2 in Appendix D).
- **conjunction_of_leaves:** The output of this recursive function is a predicate representing the conjunction of the guards of the leaf events. The output of *conjunction_of_leaves* applied to a leaf with no parameters is *leaf-name* $= FALSE$. If the leaf has parameters the output is *leaf-name* $= \varnothing$. This function is updated to include the *par-replicator* and the exception handling combinators (*interrupt*, *retry*). In the case of a *par-replicator*, the conjunction is applied to the *par-replicator* child

and its *successor* child due to the zero execution case. The conjunction is also applied to the *normal-child* and the *interrupting-child* of the exception handling combinators (Figure D.3 in Appendix D).

- **disjunction_of_leaves:** The output of this recursive function is a predicate representing the disjunction of the invariants of the leaf events. The output of *disjunction_of_leaves* applied to a leaf with no parameters is simply the *leaf-name*, while in the case the leaf has parameters, the output is *leaf-name* $\neq \varnothing$. This function is updated similar to the *conjunction_of_leaves:* to include the *par-replicator* and the exception handling combinators (*interrupt*, *retry*), with the difference of applying disjunction instead of conjunction (Figure D.4 in Appendix D).

- **union_of_leaves:** The output of this recursive function is a predicate representing the union of the control variables of the leaf events. The domain function may be applied to each leaf event, according to the number of times a replicator exists in the traversing steps. This function is also updated to introduce the *par-replicator* and the exception handling combinators in a way similar to the *conjunction_of_leaves*, but with applying union instead of conjunction (Figure D.5 in Appendix D).

- **build_seq_grd:** The result of this recursive function is a guard predicate specifying the sequencing between two leaf events, where sequencing depends on the *predecessor* of a leaf. This function is updated to include the exception handling combinators, where in the case the predecessor is *retry*, we only take into account the *normal-child*, whereas in the case the *predecessor* is an *interrupt*, we require the execution of either the *normal-child* or the *interrupting-child*. We have not included the *par-replicator* because the *par-replicator* will be ignored due to the zero execution case as explained by the *predecessor* function (Figure D.6 in Appendix D).

8.3.1 New Auxiliary Functions

In addition to the previously updated auxiliary functions reused from Salehi Fathabadi (2012), we introduce some new auxiliary functions that help us in formalising the newly added translation rules.

These functions take into account the previous restriction that a *loop* cannot be a first or last child of a flow because its leaf events can have no associated variable names (Salehi Fathabadi, 2012). The *par-replicator* cannot be the last child of a flow, since it needs an event to stop its execution. In addition to these restrictions, we have also introduced additional restrictions to ERS related to the translation rules and the tool. These restrictions are required to ensure the proper behaviour of all the combinators.

These restrictions are mainly related to the combinators that depend on the follow-on or the successor events, which can be described as follows:

1. The *loop* and the *par-replicator* **cannot** be directly followed by an exception handling combinator (*interrupt*, *retry*).
2. If the *loop* and the *par-replicator* are followed by a flow (i.e., not a leaf), then the first child of the flow must be always a solid child.
3. If the *interrupting-child* of the *interrupt* and the *retry* combinators is a flow (i.e., not a leaf), then the first child of the interrupting flow must be a solid child.
4. If the *normal-child* of a an *interrupt* and *retry* is a flow, then the last child must be a solid child.

The last restriction is explained in Section 6.2.1 of Chapter 6. The simple justification for the rest of the restrictions is to simplify the translation rules. These restrictions can be also justified by the execution of the *loop* and the *par-replicator* which is disabled by its follow-on event ensuring that these combinators do not execute forever, hence we added the restriction that these combinators cannot be directly followed by an exception handling combinator (*interrupt*, *retry*), since the normal flows of these combinators can be interrupted. In addition if the follow-on child of the *loop* and the *par-replicator* is a flow then we will need the first child to disable their execution, in which case having the first child as the solid child makes sense, since the solid child should simulate the main behaviour of its parent. Similarly for the interrupting child of the *interrupt* and the *retry* combinators, their execution will be disabled by the completion of the normal sub-flow, hence having the first child as the solid child to simulate the behaviour of its parent is logical.

In addition, having the first child as the solid child will ensure that the first child cannot be a combinator like the *loop* for example with a zero execution case, and there will be no difference between strong and weak sequencing.

The newly added auxiliary functions takes into consideration the previously mentioned restrictions. Following the approach of Salehi Fathabadi (2012) in defining the auxiliary functions, we show the function definition, the output operation and the traversing steps which show how the function applies pattern-matching to extract the various sub-terms of its source, which could also rely on other functions. In all the auxiliary functions, the letter f is an abbreviation of the function name.

The auxiliary functions *list_child_leaves* (Figure 8.2) and *list_first_child_leaves* (Figure 8.3), are used similar to the function *list_of_leaves* to traverse down the last refinement level of the sub-tree. The difference is the output of these functions, where *list_child_leaves* returns every leaf of a flow, while *list_first_child_leaves* only returns the leaves of the

list_child_leaves (ch: child/cons-child, *par: parameter list of ch)
Output Operation:
list_child_leaves (leaf (name), *par) = **leaf (name, *par)**
Pattern Matching:
• **f** (**constructor** (c_1 , …, c_n), *par) = **f** (c_1, *par) , … , **f** (c_n, *par) where constructor = and/or/xor/loop • **f** (**exception-combinator** ($c_{normal\text{-}child}$, $c_{interrupting\text{-}child}$), *par) = **f** ($c_{normal\text{-}child}$, *par) , **f** ($c_{interrupting\text{-}child}$, *par) where exception-combinator = interrupt/retry • **f** (**replicator** (p, c), *par) = **f** (c, (*par, p)) where replicator = all/some/one/par-rep • **f** (**1* flow**, *par) = **f** ($flow_1$, *par), …, **f** ($flow_n$, *par) • **f** (**flow** (name, *par, sw), *par) = **f** ($child_1$, *par), …, **f** ($child_n$, *par)

Figure 8.2: *list_child_leaves* Function

first child of a flow. Both will result in a list of leaf names with their parameter lists. The main difference between *list_first_child_leaves* and *list_of_leaves* is that, *list_of_leaves* does **not** always return the first child of the flow, depending on the sequencing type of the flow.

Notice that in Figure 8.2, we take into consideration the *loop* case, whereas in Figure 8.3, we do not because the *loop* cannot be the first child of a flow.

The auxiliary function *build_par_ref_grd* (Figure 8.4) is used in the *par-replicator* translation rules when the *par-replicator* child is a flow and not a simple leaf. This is to ensure that the follow-on child of the *par-replicator* cannot interrupt the *par-replicator* child in the middle of its execution. This function uses another function *par_ref_grd* which is defined in Figure 8.5.

The function *build_succ_SI_inv* (Figure 8.6) is used in building invariants for the *par-replicator* and the *loop* in the single instance case. These invariants are related to the successor of the *loop* and the *par-replicator*, which are needed in the case the child of these combinators are flows and not simple leaves.

list_first_child_leaves (ch: child/cons-child, *par: parameter list of ch)
Output Operation:
list_first_child_leaves (leaf (name), *par) = **leaf (name, *par)**
Pattern Matching:
• f (constructor (c_1 , ..., c_n), *par) = f (c_1, *par) , ... , f (c_n, *par) where constructor = and/or/xor   • f (exception-combinator (**c**$_{\text{normal-child}}$, **c**$_{\text{interrupting-child}}$), *par) = f (c$_{\text{normal-child}}$, *par) , f (**c**$_{\text{interrupting-child}}$, *par) where exception-combinator = interrupt/retry   • f (replicator (p, c), *par) = f (c, (*par, p)) where replicator = all/some/one   • f (par (p, c), *par) = f (c, (*par, p)), f (succ, (*par)) where succ is the successor child of par(p,c)   • f (1* flow, *par) = f ($flow_1$, *par), ..., f ($flow_n$, *par)   • f (flow (name, *par, sw), *par) = f ($child_1$, *par) where $child_1$ is the first child of the flow

Figure 8.3: *list_first_child_leaves* Function

build_par_ref_grd (ch: child) =		
• f (leaf) =	“ ”	
• f (one (c)) =	f (c)	
• f (xor (c_1 , ..., c_n)) =	f (c_1) ∧ ... ∧ f(c_n)	where f (c_i) ≠ “ ”
• f (1* flow) =	f ($flow_1$) ∧ ... ∧ f ($flow_n$)	where f ($flow_i$) ≠ “ ”
• f (flow (c_1 , ..., c_n)) =	f (c_1)	where c_1 = c_n
• f (flow(c_1 , ..., c_n)) =	par_ref_grd (c_1, true) = par_ref_grd (c_n , false)	where c_1 ≠ c_n

Figure 8.4: *build_par_ref_grd* Function

8.4 Translation Rules

The original ERS patterns, presented in Chapter 3 are described formally using twenty three translation rules in Salehi Fathabadi (2012). The rules are summarised in Tables 8.4 and 8.5. The tables present the rule name (first column) and the rule description (second column) presenting the ERS element and its corresponding element in Event-B. The ERS elements described include a solid leaf, which is a leaf connected to its parent

par_ref_grd (ch: child, firstChild: Boolean) =

- f (**leaf**(name), firstChild) = **name**
- f (**and**(c_1 , …, c_n), firstChild) = f (c_1, firstChild) ∩ … ∩ f (c_n, firstChild)
- f (**or** / **xor** (c_1 , …, c_n), firstChild) = f (c_1, firstChild) ∪ … ∪ f (c_n, firstChild)
- f (**interrupt** ($c_{normal\text{-}child}$, $c_{interrupting\text{-}child}$), firstChild) =
  f ($c_{normal\text{-}child}$, firstChild) ∪ f ($c_{interrupting\text{-}child}$, firstChild)
- f (**retry** ($c_{normal\text{-}child}$, $c_{interrupting\text{-}child}$), firstChild) = f ($c_{normal\text{-}child}$, firstChild)
- f (**all** / **some** / **one** (c), firstChild) = dom (f(c, firstChild))
- f (**par**(c), true) = dom (f(c, firstChild)) ∪ f(succ, firstChild)
  where succ is the successor child of par(c)
- f (**1* flow**, firstChild) = f ($flow_1$, firstChild) ∪ … ∪ f ($flow_n$, firstChild)
- f(**flow**(c_1 , …, c_n), firstChild) = f (c_1, true) **where** firstChild = true
- f(**flow**(c_1 , …, c_n), firstChild) = f (c_n, false) **where** firstChild = false

Figure 8.5: *par_ref_grd* Function

build_succ_SI_inv (ch: child, parnum: int)

Output Operation:

f (**leaf** (leaf-name)) =	**leaf-name = TRUE**	parnum = 0
f (**leaf** (leaf-name)) =	**leaf-name ≠ ∅**	parnum > 0

Pattern Matching:

- f (**all** / **some** / **one** (c), parnum) = f (c, parnum + 1)
- f (**and**/ **or** / **xor** (c_1 , …, c_n), parnum) = f (c_1 , parnum) $\vee$ … $\vee$ f(c_n, parnum)
- f (**1* flow**, parnum) = f ($flow_1$, parnum) $\vee$ … $\vee$ f ($flow_n$,
  parnum)
- f (**flow** (name, *par, sw), parnum) = f ($child_1$, parnum)
  where $child_1$ is the first child of the flow

Figure 8.6: *build_succ_SI_inv* Function

by a solid line, and a dashed leaf, which is a leaf connected to its parent by a dashed line. A replicator leaf is a leaf connected to a replicator combinator, which can be one of the following (*all*, *some*, *one*). On the other hand, a non-replicator leaf is a leaf that is **not** connected to a replicator. A solid xor is a *xor* combinator connected to its parent using a solid line. Similarly, a solid one is a *one* combinator connected to its parent using a solid line.

Table 8.4 presents the rules related to an ERS leaf, these rules describe how a leaf is translated into variables, events, guards, actions and invariants in Event-B, while Table 8.5 presents the rules that are specific to the applied combinator like the *xor*, *loop* and *one*, in addition to the weak sequencing flow rules.

Rule Name	**Rule Signature: ERS $\Longrightarrow$ Event-B**
TR_leaf1	leaf $\Longrightarrow$ variable
TR_leaf2	non-replicator leaf $\Longrightarrow$ typing invariant
TR_leaf3	replicator leaf $\Longrightarrow$ typing invariant
TR_leaf4	leaf $\Longrightarrow$ sequencing invariant
TR_leaf5	solid leaf $\Longrightarrow$ gluing invariant
TR_leaf6	solid leaf $\Longrightarrow$ refining event
TR_leaf7	dashed leaf $\Longrightarrow$ new event
TR_leaf8	leaf $\Longrightarrow$ sequencing guard
TR_leaf9	non-replicator leaf $\Longrightarrow$ guard
TR_leaf10	replicator leaf $\Longrightarrow$ guard
TR_leaf11	non-replicator leaf $\Longrightarrow$ action
TR_leaf12	replicator leaf $\Longrightarrow$ action

Table 8.4: ERS Leaf-Related Translation Rules

These rules have different roles, for example the typing invariant rules will define the type of the control variable, while sequencing invariant rules will define the execution order of the control variable with respect to previously defined control variables, which in most cases are the ones directly to their left in the ERS diagram. The gluing invariant will relate the refining control variable to its parent in the more abstract level. The sequencing guard rules will add a guard to the event to check the ordering of its corresponding control variable. Other guards such as the replicator and non-replicator leaf guards in Table 8.4 will add a guard to the event to check for previous execution of the event, while the action rules will add an action to the event to record its execution. Other guards and invariants in Table 8.5 are specific for the combinator.

Rule Name	**Rule Signature: ERS $\Longrightarrow$ Event-B**
TR_xor1	solid xor $\Longrightarrow$ gluing invariant
TR_xor2	xor $\Longrightarrow$ invariant
TR_xor3	xor $\Longrightarrow$ guards
TR_one1	solid one $\Longrightarrow$ gluing invariant
TR_one2	one $\Longrightarrow$ invariant(s)
TR_one3	one $\Longrightarrow$ guard(s)
TR_loop1	loop $\Longrightarrow$ guard(s)
TR_loop2	loop $\Longrightarrow$ loop successor event(s) guard(s)
TR_loop3	loop $\Longrightarrow$ resetting event
TR_weak1	weak sequencing flow $\Longrightarrow$ invariant(s)
TR_weak2	weak sequencing flow $\Longrightarrow$ guard(s)

Table 8.5: ERS Combinator-Related Translation Rules

In the following sub-sections we present the translation rules related to the extended ERS approach as presented in chapters 4 and 6. In defining the translation rules we follow the same tabular structure used in defining the original ERS rules. The structure is presented in Figure 8.7, where the first row represents rule signature which is of the form:

Rule Name: ABNF Element(s) (*Source*) $\longrightarrow$ Event-B Element(s) (*Target*)

The second row, represents the ABNF element which is the source element. The third row uses some of the auxiliary functions defined in Section 8.3 to help in specifying the Event-B target element. The auxiliary functions will in turn apply pattern matching to extract the various sub-terms of the translation source, as shown in Section 8.3.1. The last row defines the target element in Event-B, i.e., the output. The last row can sometimes be divided into two rows to differentiate between the single instance case and the multiple instances case, if they are different.

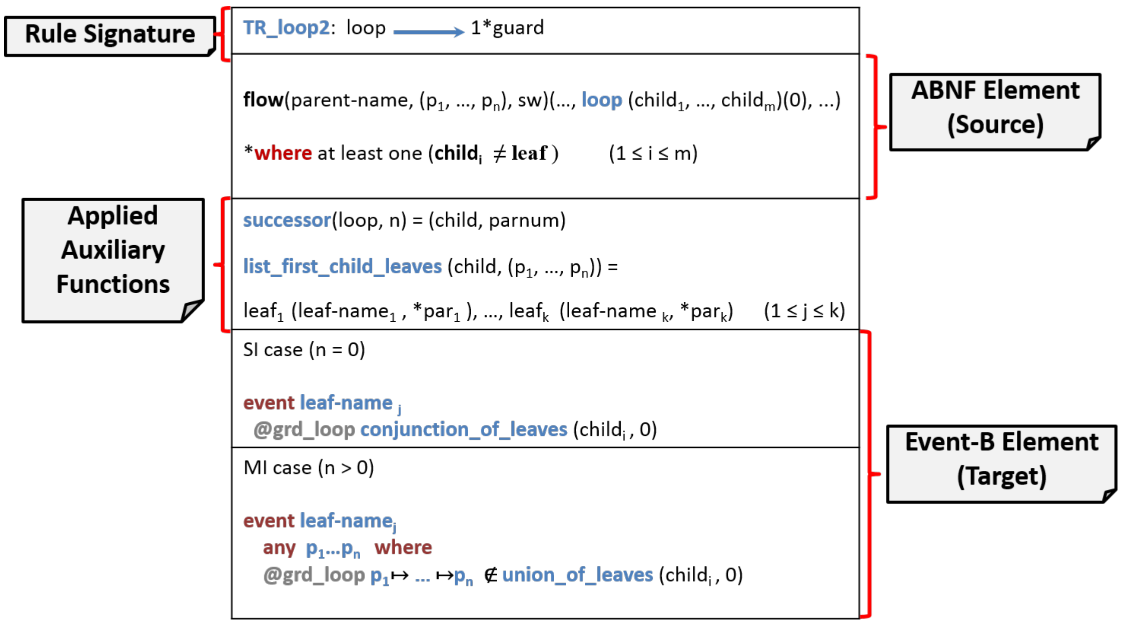

Rule Signature
TR_loop2: loop ⟶ 1*guard
ABNF Element (Source)
flow(parent-name, (p_1, ..., p_n), sw)(..., loop (child$_1$, ..., child$_m$)(0), ...)
*where at least one (child$_i$ ≠ leaf) (1 ≤ i ≤ m)
Applied Auxiliary Functions
successor(loop, n) = (child, parnum)
list_first_child_leaves (child, (p_1, ..., p_n)) =
leaf$_1$ (leaf-name$_1$, *par$_1$), ..., leaf$_k$ (leaf-name $_k$, *par$_k$) (1 ≤ j ≤ k)
Event-B Element (Target)
SI case (n = 0)
event leaf-name $_j$
@grd_loop conjunction_of_leaves (child$_i$, 0)
MI case (n > 0)
event leaf-name$_j$
any p_1...p_n where
@grd_loop p_1↦ ... ↦p_n ∉ union_of_leaves (child$_i$, 0)

Figure 8.7: Translation Rules Tabular Format

An example ERS diagram is presented in Figure 8.1, which we will use to show how the new translation rules of the extended ERS are applied. The ERS diagram Figure 8.1 consists of an abstract level and two refinements, where the green nodes represent leaf events that are not decomposed.

8.4.1 Parameter Extension Translation Rules

In addition to changing the syntax of a parameter to include a *range_expression* as shown in Table 8.2, we introduce two new translation rules related to adding the *range_expression* property. The first is *TR_rangeExpression* which adds a guard to a replicator leaf. The second is *TR_rep* which maps a replicator to (an) invariant(s).

8.4.1.1 TR_rangeExpression: Mapping a replicator to (a) guard(s)

TR_rangeExpression presented in Figure 8.8, adds a guard (*grd_Exp*) to the first child leaves of a replicator (*all*, *some*, *one*, *par-rep*). This guard checks that the replicator parameter (p_i) remains in the range of the expression (*range_exp*), if it exists.

TR_rangeExpression: replicator $\longrightarrow$ 1* guard(s)

flow(parent-name, (p_1,...,p_n), sw)(..., **all** (p_i **(range_exp)**, child)(0),...)
flow(parent-name, (p_1,...,p_n), sw)(..., **some** (p_i**(range_exp)**, child)(0),...)
flow(parent-name, (p_1,...,p_n), sw)(..., **one**(p_i **(range_exp)**, child)(ref),...)
flow(parent-name, (p_1,...,p_n), sw)(..., **par-rep** (p_i **(range_exp)**, child)(0),...)

*where p_i has a **range_expression**

list_first_child_leaves (child, (p_1,...,p_n , p_i)) =
leaf_1 (leaf-name_1 ,*par_1)... leaf_m (leaf-name_m,*par_m)

event **leaf-name_j** ($1 \leq j \leq m$)
  @grd_Exp p_i ∈ **range_exp**

Figure 8.8: TR_rangeExpression: Mapping a replicator to (a) guard(s)

The rule applies the *list_first_child_leaves* function (Figure 8.3) to get the list of leaves of the first child of the replicator. The output of the rule adds a guard to each leaf event resulting from the *list_first_child_leaves* function. The added guard checks that the replicator parameter is an element of the range expression of the applied replicator. This rule is only applied when the replicator parameter has a defined range expression.

In Figure 8.9, we apply the rule *TR_rangeExpression*, to both the *par-rep* and the *all* at the second refinement level of Figure 8.1. In this case the first child of both replicators is the leaf *K*, therefore two guards will be added to the event *K*. The first guard is to ensure the *par-replicator* parameter (*p1*) is in the range expression *P1*, and the second is to ensure that the *all* replicator parameter (*p2*) is in the range expression *P2*. The guards are the same in both the single and the multiple instances cases.

8.4.1.2 TR_rep: Mapping a replicator leaf to (an) invariant(s)

The translation rule *TR_rep* in Figure 8.10, maps a replicator to one or more invariant(s) if the replicator defines a range expression. The replicator invariant ensures that the range of the replicator leaf variable remains within the range expression.

Application of TR_rangeExpression

```
flow(process_name, p, 1)
          (flow(A, p, 1)
                    (leaf(G)(0),
                     loop(flow(H, p, 1)
                                   (and(leaf(P), leaf(Q))(0),
                                    leaf(R)(1)),
                            leaf(I)(0)),
                     leaf(J)(1)),
              par-rep(p1)(flow(B, (p, p1), 1)
                                     (all(p2)(leaf(K))(0),
                                      leaf(L)(1))(0),
              leaf(C)(0),
              interrupt(flow(D, p, 1)
                                   (retry(flow(M, p, 1)(xor(leaf(S), leaf(T))(0), leaf(U)(1)))
                                          (flow(N, p, 1)(leaf(V)(1), leaf(W)(0)))(0),
                                    leaf(O)(1)))
                      (leaf(E))(0),
              leaf(F)(0))
```

Applied Auxiliary Functions:

1. *par-rep* Case:
- list_first_child_leaves (flow(B, (p, p1), 1), (p, p1))= **leaf(K, (p, p1, p2))**

2. *all* Case:
- list_first_child_leaves (leaf(K), (p, p1, p2)) = **leaf(K, (p, p1, p2))**

Output:

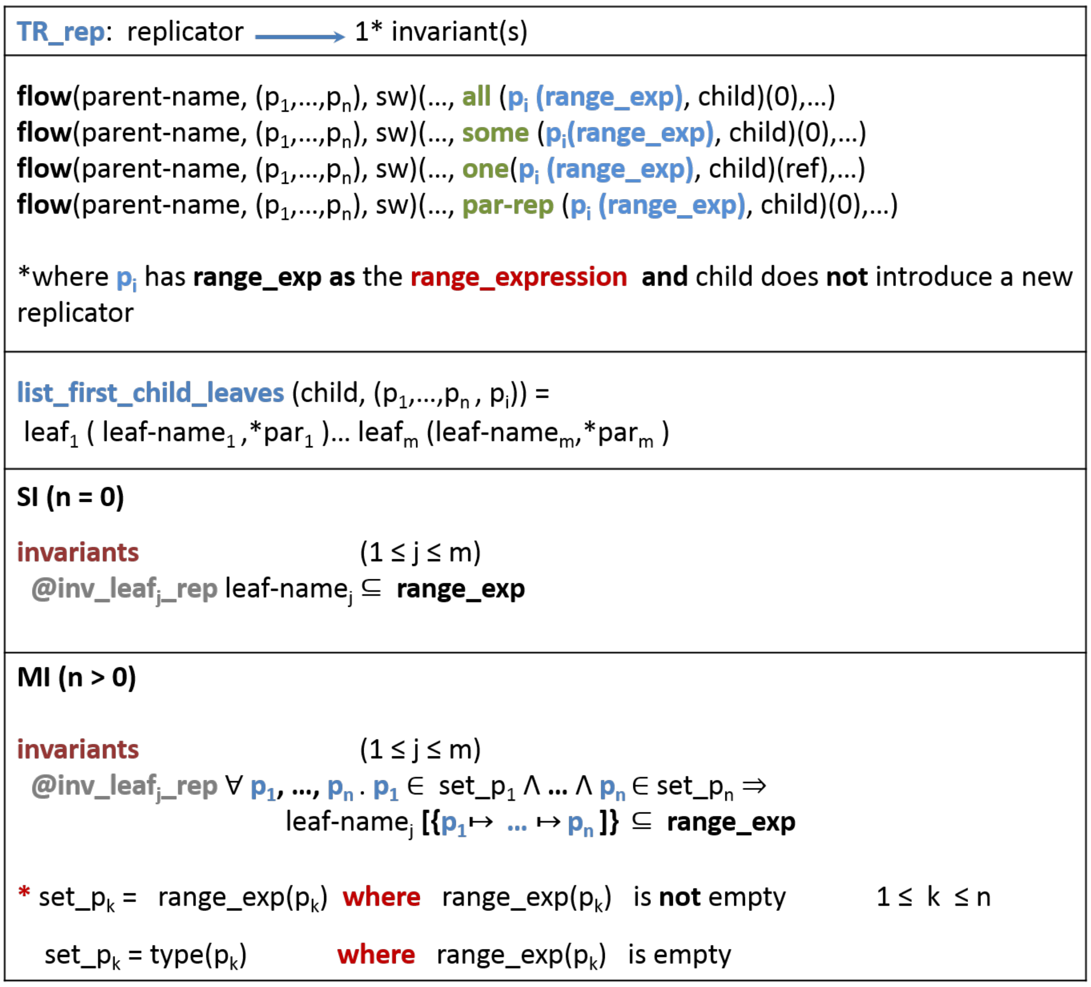
```
event K
 any p p1 p2 where
  @grd_Exp1 p1 ∈ P1
  @grd_Exp2 p2 ∈ P2
```

Figure 8.9: Application of TR_rangeExpression to the *par-rep* and *all*

TR_rep adds an invariant for the leaf variables of the first child of the replicator. In the single instance case, where the replicator leaves resulting from *list_first_child_leaves* function, have only the replicator parameter (p_i), the invariant is simply that the leaf control variable is subset or equal to the range expression of the replicator ($range\_exp$).

In the multiple instances case where the leaf replicator leaves inherit the parent flow parameters ($p_1 ... p_n$), the invariant specifies that each parameter is an element of its range expression if defined or its type otherwise, and finally it ensures that the leaf variable of the parent flow parameter maplets ($p_1 \mapsto .. \mapsto p_n$) is subset or equal to the replicator range expression.

In Figure 8.11 we apply *TR_rep* to the *all* replicator in the second refinement level showing the MI case, we also apply *TR_rep* to the *par-replicator* but in the first refinement



TR_rep: replicator $\longrightarrow$ 1* invariant(s)

flow(parent-name, (p_1,…,p_n), sw)(…, **all** (p_i **(range_exp)**, child)(0),…)
flow(parent-name, (p_1,…,p_n), sw)(…, **some** (p_i**(range_exp)**, child)(0),…)
flow(parent-name, (p_1,…,p_n), sw)(…, **one**(p_i **(range_exp)**, child)(ref),…)
flow(parent-name, (p_1,…,p_n), sw)(…, **par-rep** (p_i **(range_exp)**, child)(0),…)

*where p_i has **range_exp as** the **range_expression and** child does **not** introduce a new replicator

list_first_child_leaves (child, (p_1,…,p_n , p_i)) =
leaf$_1$ (leaf-name$_1$,*par$_1$)… leaf$_m$ (leaf-name$_m$,*par$_m$)

SI (n = 0)

invariants ($1 \le j \le m$)
@inv_leaf$_j$_rep leaf-name$_j$ $\subseteq$ **range_exp**

MI (n > 0)

invariants ($1 \le j \le m$)
@inv_leaf$_j$_rep $\forall$ **p_1, …, p_n . p_1** $\in$ set_p_1 $\wedge$ **…** $\wedge$ **p_n** $\in$ set_p_n $\Rightarrow$
leaf-name$_j$ **[{p_1**$\mapsto$ **…** $\mapsto$ **p_n]}** $\subseteq$ **range_exp**

* set_p_k = range_exp(p_k) **where** range_exp(p_k) is **not** empty $1 \le k \le n$

set_p_k = type(p_k) **where** range_exp(p_k) is empty

Figure 8.10: TR_rep: Mapping a replicator leaf to an invariant

level where we assume the ERS diagram does **not** introduce the parameter p at the root of the diagram to show the result of applying this rule in the single instance case.

8.4.2 Par-Replicator Translation Rules

The *par-replicator* combinator is one of the replicator patterns as shown in the ERS language syntax (Table 8.2). Extending the ERS combinators with the *par-replicator*, results in a set of minor changes and a set of major changes. Minor changes include the modification of few translation rules (*TR_leaf3*, *TR_leaf10*, *TR_leaf12*) and some auxiliary functions. Here we only describe the major changes, since the minor ones only include the case of adding the *par-replicator* and no major changes take place.

The major changes include the introduction of three new translation rules: *TR_par1*, *TR_par2* and *TR_par3*. *TR_par1* maps a *par-replicator* to a guard that stops the execution of the *par-replicator* child once the next child of the *par-replicator* executes. *TR_par2* maps a refined *par-replicator* to a guard in its next child, to ensure that the

Application of TR_rep

```
flow(process_name, p, 1)
        (flow(A, p, 1)
                (leaf(G)(0),
                 loop(flow(H, p, 1)
                               (and(leaf(P), leaf(Q))(0),
                                leaf(R)(1)),
                      leaf(I)(0)),
                 leaf(J)(1)),
           par-rep(p1)(flow(B, (p, p1), 1)
                                  (all(p2)(leaf(K))(0),
                                  leaf(L)(1))(0),
           leaf(C)(0),
           interrupt(flow(D, p, 1)
                               (retry(flow(M, p, 1)(xor(leaf(S), leaf(T))(0), leaf(U)(1)))
                                      (flow(N, p, 1)(leaf(V)(1), leaf(W)(0)))(0),
                                leaf(O)(1)))
                      (leaf(E))(0),
           leaf(F)(0))
```

Applied Auxiliary Functions:

- list_first_child_leaves (leaf(K), (p, p1, p2)) = **leaf(K, (p, p1, p2))**

Output:

invariants
@inv_K_rep $\forall$**p, p1**. **p**$\in$ **Type(p)** $\wedge$ **p1** $\in$ **P1**$\Rightarrow$ **K[{p$\mapsto$p1}]** $\subseteq$**P2**

Output in SI: where we assume **Process_Name** does **not** introduce parameter **p**
The SI case applies to the par-rep at the first refinement level

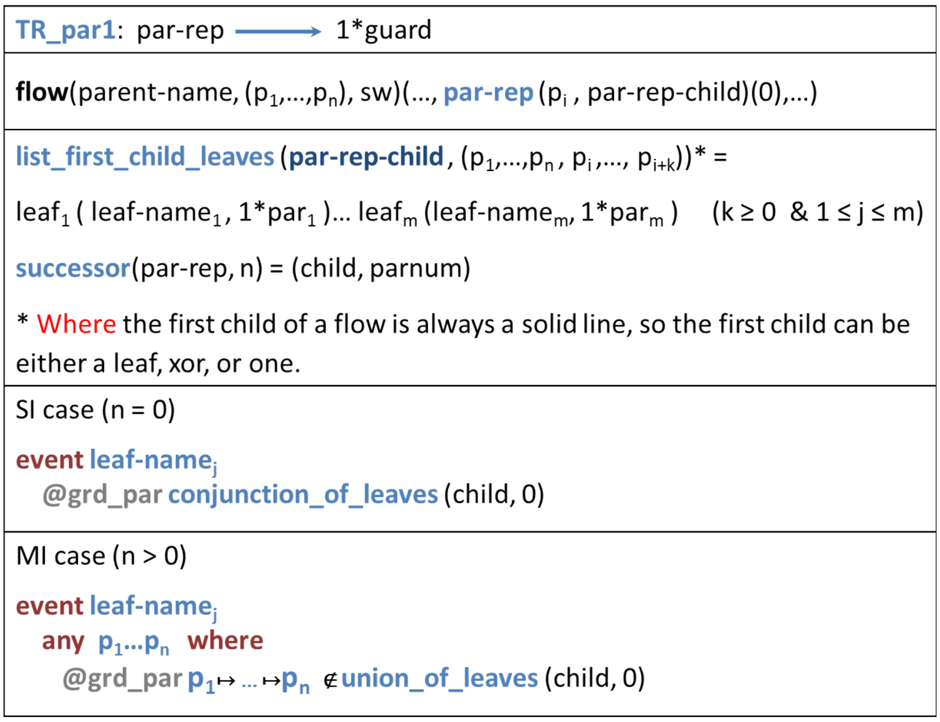
invariants
@inv_B_rep **B** $\subseteq$ **P1**

Figure 8.11: Application of TR_rep

next child does not stop the *par-replicator* sub-events in the middle of their execution. *TR_par3* maps a refined *par-replicator* child to (an) invariant(s) to enforce the guard(s) generated by *TR_par2*.

8.4.2.1 TR_par1: Mapping a par-rep to (a) guard(s)

The child of a *par-replicator* can execute zero or more times until the *par-replicator* successor child executes. In Event-B, this can be represented by adding a guard to the *par-replicator* child to check that the successor or the next child has not executed yet. Figure 8.12 presents the translation rule *TR_par1* which adds (a) guard(s) to the *par-replicator* child.


TR_par1: par-rep $\longrightarrow$ 1*guard

flow(parent-name, (p_1,…,p_n), sw)(…, par-rep (p_i , par-rep-child)(0),…)

list_first_child_leaves (**par-rep-child**, (p_1,…,p_n, p_i,…, p_{i+k}))* =

$leaf_1$ ($leaf\text{-}name_1$, $1{*}par_1$)… $leaf_m$ ($leaf\text{-}name_m$, $1{*}par_m$) ($k \geq 0$ & $1 \leq j \leq m$)

successor(par-rep, n) = (child, parnum)

* Where the first child of a flow is always a solid line, so the first child can be either a leaf, xor, or one.

SI case (n = 0)

event $leaf\text{-}name_j$
  @grd_par **conjunction_of_leaves** (child, 0)

MI case (n > 0)

event $leaf\text{-}name_j$
  any p_1…p_n **where**
    @grd_par $p_1 \mapsto … \mapsto p_n$ $\notin$ **union_of_leaves** (child, 0)

Figure 8.12: TR_par1: Mapping a par-rep to (a) guard(s)

TR_par1 uses the auxiliary functions, *list_first_child_leaves* and *successor* in the traversing steps, where a guard is added to each leaf event resulting from *list_first_child_leaves* function. The guard is generated by applying the *conjunction_of_leaves* in the single instance case and *union_of_leaves* in the multiple instances case to the result of the *successor* function. Note that we use zero for the parameter number when applying both the *conjunction_of_leaves* and *union_of_leaves*.

In Figure 8.13, we apply *TR_par1* to the *par-replicator* in the second refinement level of 8.1. which results in adding a guard, *grd_par*, to the leaf event *K*.

8.4.2.2 TR_par2: Mapping a refined par-rep to (a) guard(s) in the successor event(s)

When the child of a *par-replicator* is not a leaf, *TR_par2* will be applied. This rule will add a guard to the *par-replicator* successor first child leaves to disable them from interrupting the execution of the *par-replicator* sub-events if required. We do this by checking that both the first child and the last child of the *par-replicator* have executed. Figure 8.14 presents the translation rule *TR_par2* which requires introducing two new recursive functions *build_par_ref_grd* in Figure 8.4 and its inner function *par_ref_grd* in Figure 8.5.

When the child of a *par-replicator* is not a leaf, *TR_par2* will execute by first getting the successor child of the *par-replicator*. Then, we apply *list_first_child_leaves* function

Application of TR_par1

```
flow(process_name, p, 1)
          (flow(A, p, 1)
                    (leaf(G)(0),
                     loop(flow(H, p, 1)
                                      (and(leaf(P), leaf(Q))(0),
                                       leaf(R)(1)),
                            leaf(I)(0)),
                     leaf(J)(1)),
             par-rep(p1)(flow(B, (p, p1), 1)
                                 (all(p2)(leaf(K))(0),
                                  leaf(L)(1))(0),
             leaf(C)(0),
             interrupt(flow(D, p, 1)
                                      (retry(flow(M, p, 1)(xor(leaf(S), leaf(T))(0), leaf(U)(1)))
                                               (flow(N, p, 1)(leaf(V)(1), leaf(W)(0)))(0),
                                       leaf(O)(1)))
                           (leaf(E))(0),
             leaf(F)(0))
```

Applied Auxiliary Functions:

- list_first_child_leaves (flow(B, (p, p1), 1), (p, p1)) = **leaf(K, (p, p1, p2))**
- sucessor(par-rep(p1, flow(B, (p, p1), 1)), 2) = **leaf(C, 1)**

Output:

```
event K
 any p p1 p2 where
  @grd_par  p ∉ C
```

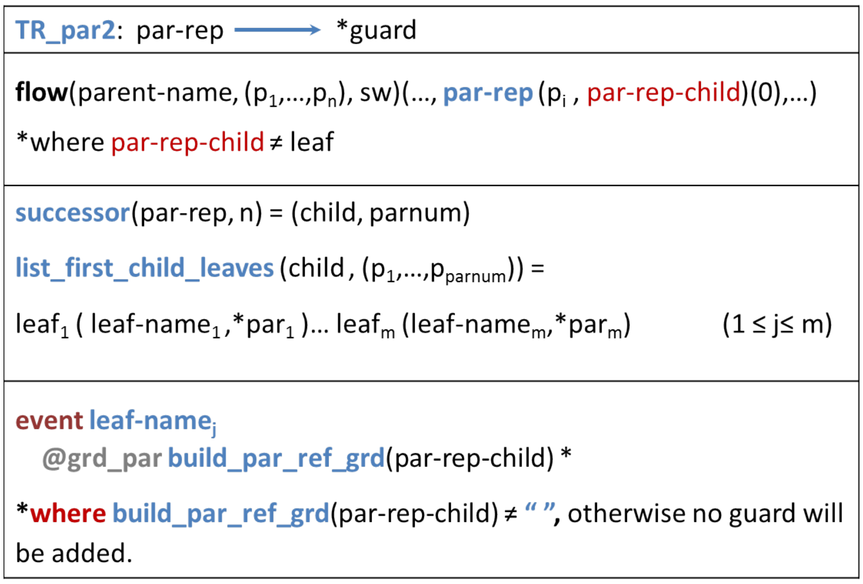
Output in SI: where we assume **Process_Name** does **not** introduce parameter **p**

```
event K
 any p1 p2 where
  @grd_par  C = FALSE
```

Figure 8.13: Application of TR_par1

to the *par-replicator* successor child, to get the leaves of its first child which must be always a solid line. The output of this translation rule will be the addition of a guard *grd_par* to the leaf events, if the result of *build_par_ref_grd* is not an empty string.

TR_par2 does not always result in adding a guard to the successor child leaves, that is why we say "**guard*" which means zero or more times. The cases that do not always require adding a guard to the *par-replicator* successor child is when refining the *par-replicator* child to only one solid *leaf*, or one solid *xor* or a one solid *one*. In these cases the successor of the *par-replicator* cannot interrupt them, that is why there is no need to add a guard.



TR_par2: par-rep $\longrightarrow$ *guard

flow(parent-name, (p_1,…,p_n), sw)(…, **par-rep** (p_i , par-rep-child)(0),…)

*where par-rep-child ≠ leaf

successor(par-rep, n) = (child, parnum)

list_first_child_leaves (child , (p_1,…,p_{parnum})) =

$leaf_1$ ($leaf\text{-}name_1$,$*par_1$)… $leaf_m$ ($leaf\text{-}name_m$,$*par_m$) (1 ≤ j≤ m)

event **$leaf\text{-}name_j$**
@grd_par **build_par_ref_grd**(par-rep-child) *

***where** **build_par_ref_grd**(par-rep-child) ≠ **“ ”,** otherwise no guard will be added.

Figure 8.14: TR_par2: Mapping a refined par-rep to (a) guard(s) in the successor event(s)

In Figure 8.15, we show how we applied *TR_par2*, which resulted in the addition of a guard (*grd_par*) to the successor leaf C of the *par-replicator*.

8.4.2.3 TR_par3: Mapping a refined par-rep to an invariant

TR_par3 generates an invariant related to *TR_par2*. This invariant will enforce the guard generated by *TR_par2*, by ensuring that if the leaf event of the successor child has executed, then the refined *par-replicator* must have executed for the same number of instances for both its first and last child events.

In this translation rule we use the auxiliary function *build_succ_SI_inv* of Figure 8.6 to build the first part of the invariant in the single instance case, while in the multiple instances case, we use the *union_of_leaves* function. The second part of the invariant is similar to *TR_par2*, where we use *build_par_ref_grd* function.

In Figure 8.17, we apply *TR_par3*, which results in the addition of the invariant *inv_par*. This invariant will be only added if the result of applying *build_par_ref_grd* to the *par-replicator* child is **not** an empty string.

8.4.3 Loop Generalisation Translation Rules

Loop generalisation allows a loop to be applied to more than one child executed non-deterministically. Each loop child is independent of the other, so the loop translation rules (*TR_loop1*, *TR_loop2*, *TR_loop3*) are updated to include more than one child of

Application of TR_par2

```
flow(process_name, p, 1)
        (flow(A, p, 1)
                (leaf(G)(0),
                 loop(flow(H, p, 1)
                               (and(leaf(P), leaf(Q))(0),
                                leaf(R)(1)),
                        leaf(I)(0)),
                 leaf(J)(1)),
          par-rep(p1)(flow(B, (p, p1), 1)
                               (all(p2)(leaf(K))(0),
                               leaf(L)(1))(0),
          leaf(C)(0),
          interrupt(flow(D, p, 1)
                               (retry(flow(M, p, 1)(xor(leaf(S), leaf(T))(0), leaf(U)(1)))
                                      (flow(N, p, 1)(leaf(V)(1), leaf(W)(0)))(0),
                                leaf(O)(1)))
                     (leaf(E))(0),
          leaf(F)(0))
```

Applied Auxiliary Functions:

- sucessor(par-rep(p1, flow(B, (p, p1), 1)), 2) = **leaf(C, 1)**
- list_first_child_leaves (leaf(C), p) = **leaf(C, p)**

Output:

```
event C
 any p  where
  @grd_par  dom(K) = L
```

Figure 8.15: Application of TR_par2

TR_par3: par-rep $\longrightarrow$ invariant

flow(parent-name, (p_1,…,p_n), sw)(…, par-rep (p_i , par-child)(0),…)

*where par-child ≠ leaf and build_par_ref_grd (par-child) ≠ “”

successor(par-rep, n) = (child, parnum)

SI case (n = 0)

invariants
 @inv_par build_succ_SI_inv (child, 0) $\Rightarrow$ build_par_ref_grd (par-child)

MI case (n = 0)

invariants
 @inv_par $\forall p_1, \ldots, p_n \cdot p_1 \mapsto \ldots \mapsto p_n \in$ union_of_leaves(child, 0) $\Rightarrow$ build_par_ref_grd(par-child)

Figure 8.16: TR_par3: Mapping a refined par-rep to an invariant

Application of TR_par3

```
flow(process_name, p, 1)
        (flow(A, p, 1)
                (leaf(G)(0),
                 loop(flow(H, p, 1)
                              (and(leaf(P), leaf(Q))(0),
                               leaf(R)(1)),
                      leaf(I)(0)),
                 leaf(J)(1)),
          par-rep(p1)(flow(B, (p, p1), 1)
                              (all(p2)(leaf(K))(0),
                               leaf(L)(1))(0),
          leaf(C)(0),
          interrupt(flow(D, p, 1)
                              (retry(flow(M, p, 1)(xor(leaf(S), leaf(T))(0), leaf(U)(1)))
                                     (flow(N, p, 1)(leaf(V)(1), leaf(W)(0)))(0),
                               leaf(O)(1)))
                      (leaf(E))(0),
          leaf(F)(0))
```

Applied Auxiliary Functions:

- **sucessor(**par-rep(p1, flow(B, (p, p1), 1)), 2**)** = **leaf(C, 1)**
- **build_par_ref_grd(**flow(B, (p, p1), 1)**)** = **dom(K) = L**

Output:

invariants
 @inv_par ∀**p**. **p**∈ **C** ⇒ **dom(K) = L**

Output in SI: where we assume **Process_Name** does **not** introduce parameter **p**

invariants
 @inv_par **C** = **TRUE** ⇒ **dom(K) = L**

Figure 8.17: Application of TR_par3

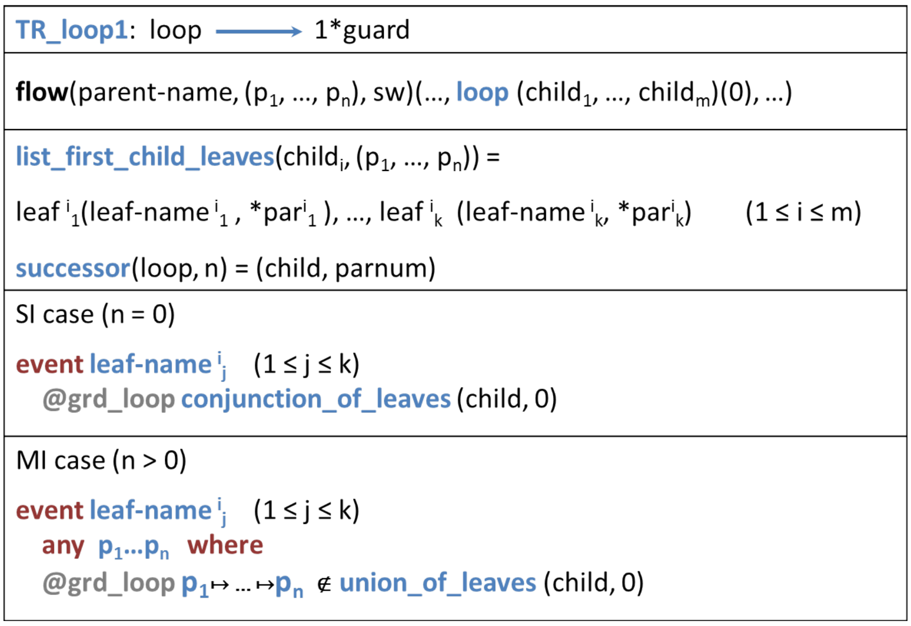
the loop. We also add a new translation rule *TR_loop4* which generates (an) invariant enforcing the guard(s) generated by *TR_loop2*.

Some of the possible event traces of the loop in Figure 8.1 at the second refinement level are:

$< G(p), J(p), ... >$
$< G(p), P(p), Q(p), R(p), J(p), ...) >$
$< G(p), I(p), J(p), ... >$
$< G(p), I(p), I(p), P(p), I(p), Q(p), R(p), J(p), ... >$
$< G(p), P(p), Q(p), R(p), P(p), Q(p), I(p), I(p), R(p), I(p), J(p), ... >$

8.4.3.1 TR_loop1: Mapping a loop to (a) guard(s)

TR_loop1, like *TR_par1*, adds a guard to the loop children to ensure that the loop next child has not executed yet. *TR_loop1* is applied every time a *loop* exists and results in adding one or more guards. A *loop* can never be a refining event due to the *single solid line rule*, so the property *ref* is always zero. Figure 8.18 updates the original rule by allowing more than one child to a *loop*. This rule applies the *list_first_child_leaves* to each child of the loop, to get the leaf events of the first child of each *loop*. Then, the *loop* guard (*grd_loop*) is added to each leaf event resulting from the function *list_first_child_leaves*.



TR_loop1: loop $\longrightarrow$ 1*guard

flow(parent-name, (p_1, …, p_n), sw)(…, loop (child$_1$, …, child$_m$)(0), …)

list_first_child_leaves(child$_i$, (p_1, …, p_n)) =

leaf$^i_1$(leaf-name$^i_1$, *par$^i_1$), …, leaf^i_k (leaf-name^i_k, *par^i_k) $(1 \le i \le m)$

successor(loop, n) = (child, parnum)

SI case (n = 0)

event **leaf-name^i_j** $(1 \le j \le k)$
  @grd_loop **conjunction_of_leaves** (child, 0)

MI case (n > 0)

event **leaf-name^i_j** $(1 \le j \le k)$
  any **$p_1 \ldots p_n$** **where**
  @grd_loop **$p_1 \mapsto \ldots \mapsto p_n$** $\notin$ **union_of_leaves** (child, 0)

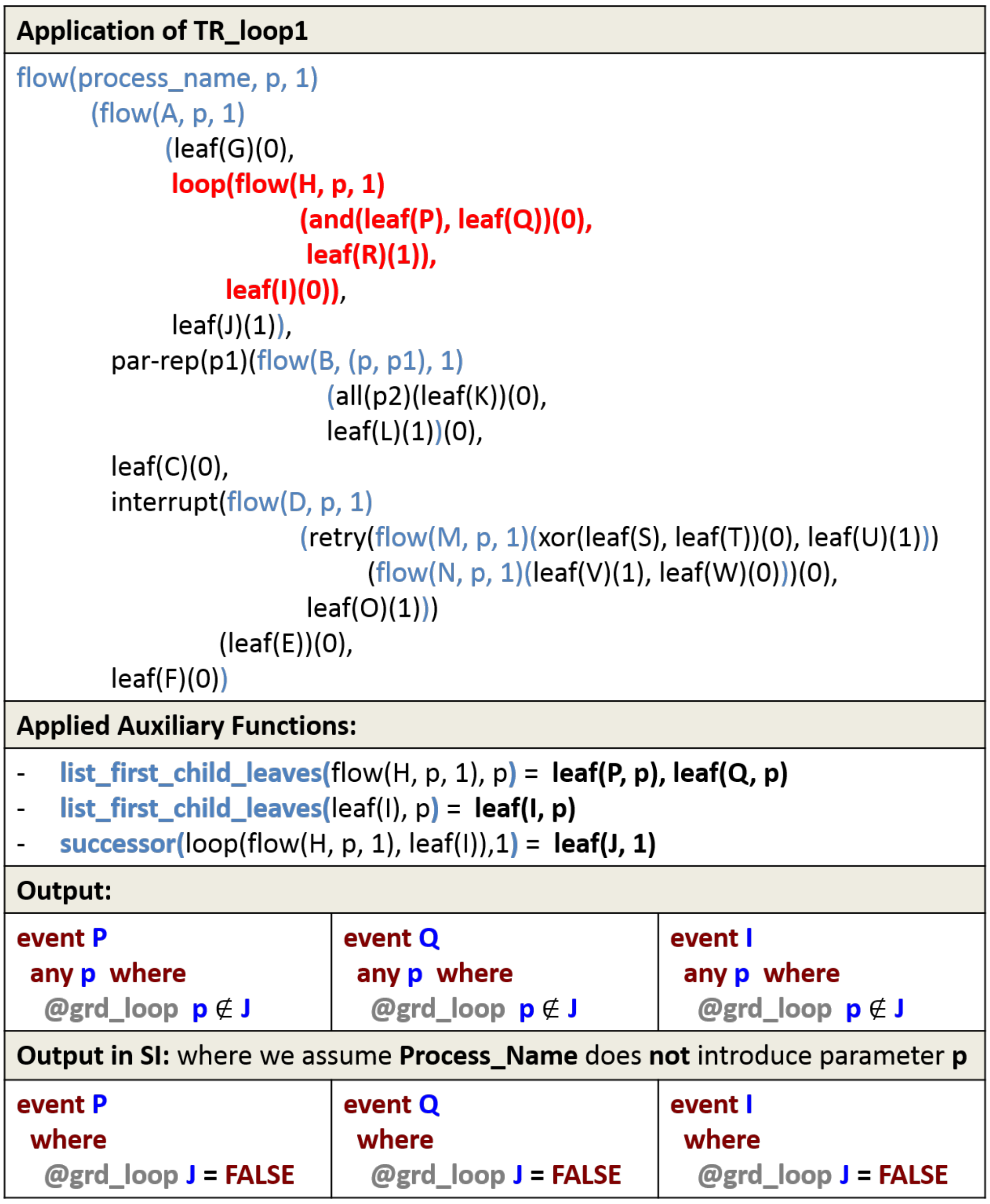
Figure 8.18: TR_loop1: Mapping a loop to (a) guard(s)

The guard *grd_loop* is defined in the same manner for each leaf event, first finding the next child of the *loop* by applying the *successor* function to the *loop*. In a single instance case, where the parent of the loop has no parameters (n=0), apply the *conjunction_of_leaves* function (Figure D.3) to the successor child. Otherwise, for a multiple instances case, apply the *union_of_leaves* function (Figure D.5) to the successor child and check that the maplet of the loop's parent parameters ($p_1 \mapsto .. \mapsto p_n$) are not in the result of the *union_of_leaves* function.

Figure 8.19 shows an application of *TR_loop1* in the second refinement level of Figure 8.1, resulting in adding the guard (*grd_loop*) to each first child leaf events, checking that the *successor* event J has not executed.

8.4.3.2 TR_loop2: Mapping a loop to (a) guard(s) in the successor child

TR_loop2 is applied when at least one of the loop children is not a leaf due to refinement. The rule ensures that the next child of the loop does not interrupt the execution of the



Application of TR_loop1

```
flow(process_name, p, 1)
        (flow(A, p, 1)
                (leaf(G)(0),
                 loop(flow(H, p, 1)
                              (and(leaf(P), leaf(Q))(0),
                               leaf(R)(1)),
                      leaf(I)(0)),
                 leaf(J)(1)),
        par-rep(p1)(flow(B, (p, p1), 1)
                            (all(p2)(leaf(K))(0),
                             leaf(L)(1))(0),
        leaf(C)(0),
        interrupt(flow(D, p, 1)
                          (retry(flow(M, p, 1)(xor(leaf(S), leaf(T))(0), leaf(U)(1)))
                                 (flow(N, p, 1)(leaf(V)(1), leaf(W)(0)))(0),
                           leaf(O)(1)))
                  (leaf(E))(0),
        leaf(F)(0))
```

Applied Auxiliary Functions:

- list_first_child_leaves(flow(H, p, 1), p) = **leaf(P, p), leaf(Q, p)**
- list_first_child_leaves(leaf(I), p) = **leaf(I, p)**
- successor(loop(flow(H, p, 1), leaf(I)),1) = **leaf(J, 1)**

Output:

event P **any p where** **@grd_loop p ∉ J**	**event Q** **any p where** **@grd_loop p ∉ J**	**event I** **any p where** **@grd_loop p ∉ J**

Output in SI: where we assume **Process_Name** does **not** introduce parameter **p**

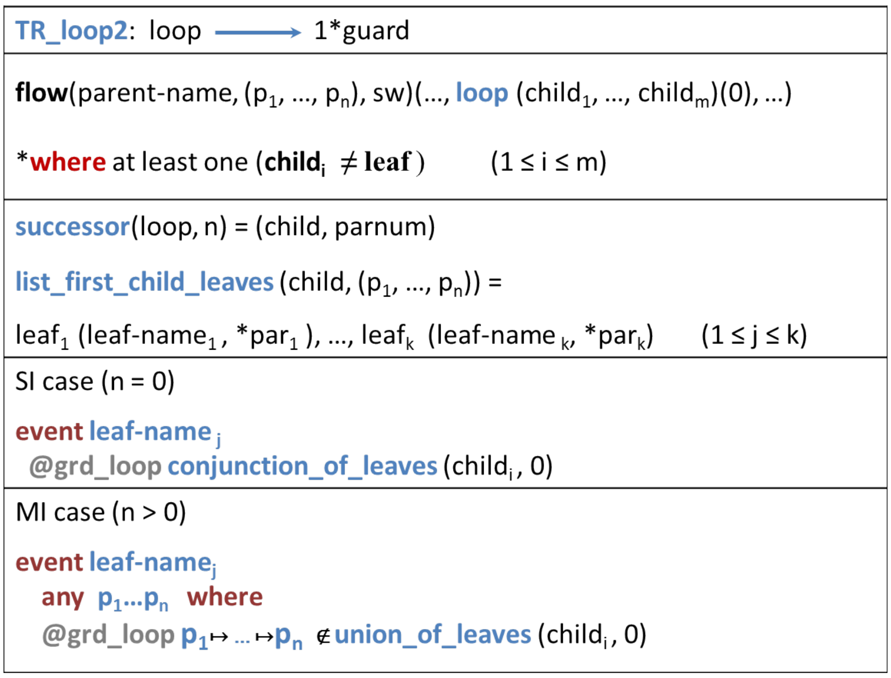
event P **where** **@grd_loop J = FALSE**	**event Q** **where** **@grd_loop J = FALSE**	**event I** **where** **@grd_loop J = FALSE**

Figure 8.19: Application of TR_loop1

refined loop children. *TR_loop2* is updated as shown in Figure 8.20 to include the case of more than one loop child. The output of applying *TR_loop2* is one or more guards added to the next leaf event(s) of the loop.

When applying *TR_loop2*, we find the next *loop* child by applying the *successor* function. Then *list_first_child_leaves* function is applied to the successor child of the *loop*, to get its leaf events. The guard *grd_loop* will then be added to each leaf event.

In a single instance case, if more than one child of the *loop* is not just a single leaf, then the loop guard (loop_grd) is the result of applying the *conjunction_of_leaves* function to the non-leaf child of the *loop*. In a multiple instances case, it is the result of applying



TR_loop2: loop $\longrightarrow$ 1*guard

flow(parent-name, (p_1, ..., p_n), sw)(..., loop (child$_1$, ..., child$_m$)(0), ...)

*where at least one (**child**$_i$ $\neq$ **leaf**) $(1 \leq i \leq m)$

successor(loop, n) = (child, parnum)

list_first_child_leaves (child, (p_1, ..., p_n)) =

leaf$_1$ (leaf-name$_1$, *par$_1$), ..., leaf$_k$ (leaf-name$_k$, *par$_k$) $(1 \leq j \leq k)$

SI case (n = 0)

event leaf-name$_j$
@grd_loop conjunction_of_leaves (child$_i$, 0)

MI case (n > 0)

event leaf-name$_j$
any $p_1 ... p_n$ where
@grd_loop $p_1 \mapsto ... \mapsto p_n \notin$ union_of_leaves (child$_i$, 0)

Figure 8.20: TR_loop2: Mapping a loop to (a) guard(s) in the successor child

the *union_of_leaves* function to the non-leaf child of the loop. The same will be repeated to each non-leaf child of the *loop*.

Applying *TR_loop2* to the second refinement level of Figure 8.1 is shown in Figure 8.21. The *loop* child (*H*) is not a single leaf, therefore the rule *TR_loop2* can be applied, resulting in the addition of the guard (*grd_loop*) to the successor event *J*.

8.4.3.3 TR_loop3: Mapping a loop to (a) resetting event(s)

Similar to *TR_loop2*, *TR_loop3* is applied when at least one of the loop children is not a single leaf. Refining a loop child results in adding control variables to ensure the proper sequencing, so *TR_loop3* adds a resetting event for each child that is not a single leaf to enable the multiple execution of the loop child. The updated version of this rule is shown in Figure 8.22. The only change in the rule is, the possibility of generating more than one resetting event depending on the number of the loop children that are not single leaves.

The resetting event of the refined loop child is executed when its last leaf/leaves completes execution, this is ensured by the sequencing guard of the corresponding reset event. The sequencing guard of the *reset* event is the output of applying *build_seq_grd* function (Figure D.6) where the *predecessor* (Figure D.1) is the corresponding refined loop child.

The purpose of the resetting event is to allow multiple executions of the loop child, and this is ensured by resetting the control variables of the *loop* events. So, the action of

Application of TR_loop2

```
flow(process_name, p, 1)
       (flow(A, p, 1)
              (leaf(G)(0),
               loop(flow(H, p, 1)
                           (and(leaf(P), leaf(Q))(0),
                            leaf(R)(1)),
                    leaf(I)(0)),
               leaf(J)(1)),
         par-rep(p1)(flow(B, (p, p1), 1)
                              (all(p2)(leaf(K))(0),
                              leaf(L)(1))(0),
         leaf(C)(0),
         interrupt(flow(D, p, 1)
                            (retry(flow(M, p, 1)(xor(leaf(S), leaf(T))(0), leaf(U)(1)))
                                  (flow(N, p, 1)(leaf(V)(1), leaf(W)(0)))(0),
                            leaf(O)(1)))
                   (leaf(E))(0),
         leaf(F)(0))
```

Applied Auxiliary Functions:

- successor(loop(flow(H, p, 1), leaf(I)),1) = **leaf(J, 1)**
- list_first_child_leaves(leaf(J), p) = **leaf(J, p)**

Output:

```
event J
 any p  where
  @grd_loop  p ∉ P ∪ Q
```

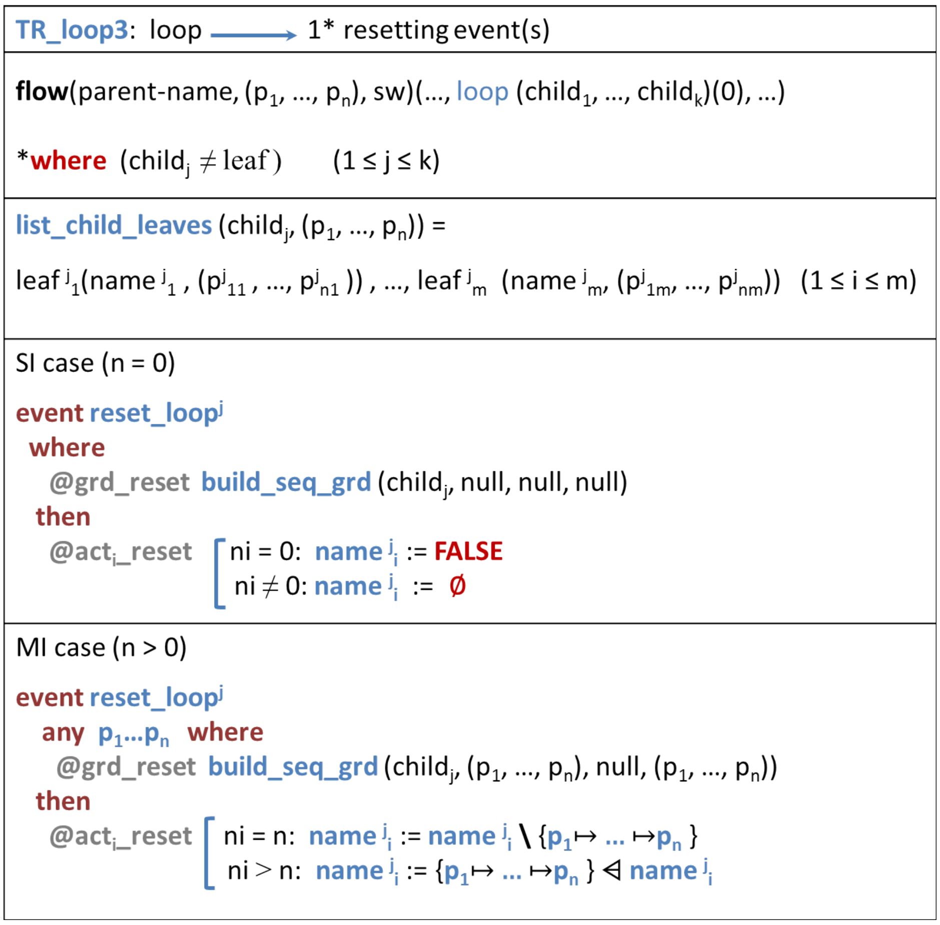
Output in SI: where we assume **Process_Name** does **not** introduce parameter **p**

```
event J
 where
  @grd_loop P = FALSE ∧ Q = FALSE
```

Figure 8.21: Application of TR_loop2

the reset event in a single instance case, will set each leaf of the refined child to false. This is the case when no new parameters are added during refinement ($ni = 0$). In case a parameter is added during refinement as a result of applying a replicator ($ni \neq 0$), it sets each leaf of the refined child to the empty set.

In a multiple instances case, in case the parameter list is the same as the loop ($ni = n$), the resetting event removes the maplet of the loop parent parameters ($p_1 \mapsto .. \mapsto p_n$) from each leaf. Otherwise, if the parameter list is longer ($ni > n$), due to applying a replicator during refinement, the resetting is done using the domain subtraction of the loop parameters from each leaf control variable.



TR_loop3: loop $\longrightarrow$ 1* resetting event(s)

flow(parent-name, (p_1, …, p_n), sw)(…, loop (child$_1$, …, child$_k$)(0), …)

***where** (child$_j$ $\neq$ leaf) (1 $\leq$ j $\leq$ k)

list_child_leaves (child$_j$, (p_1, …, p_n)) =

leaf$^j{}_1$(name$^j{}_1$, ($p^j{}_{11}$, …, $p^j{}_{n1}$)) , …, leaf$^j{}_m$ (name$^j{}_m$, ($p^j{}_{1m}$, …, $p^j{}_{nm}$)) (1 $\leq$ i $\leq$ m)

SI case (n = 0)

```
event reset_loop^j
 where
   @grd_reset build_seq_grd (child_j, null, null, null)
 then
   @act_i_reset [ ni = 0:  name^j_i := FALSE
                [ ni ≠ 0:  name^j_i :=  ∅
```

MI case (n > 0)

```
event reset_loop^j
  any p_1…p_n  where
   @grd_reset build_seq_grd (child_j, (p_1, …, p_n), null, (p_1, …, p_n))
 then
   @act_i_reset [ ni = n:  name^j_i := name^j_i \ {p_1↦ … ↦p_n }
                [ ni > n:  name^j_i := {p_1↦ … ↦p_n } ⩤ name^j_i
```

Figure 8.22: TR_loop3: Mapping a loop to (a) resetting event(s)

In Figure 8.23, the loop child (H) is not a single leaf, in this case applying *TR_loop3* at the second refinement level, will result in the resetting event *reset_loop*.

8.4.3.4 TR_loop4: Mapping a loop to (an) invariant(s)

TR_loop4, which is presented in Figure 8.24, is a new translation rule that is added to enforce the guard(s) resulting from *TR_loop2* using invariant(s). This rule, similar to *TR_loop2* only applies to the refined child of the *loop*.

This rule will generate (an) invariant(s) that ensures that the *loop* successor can only occur if the first child of the refined *loop* has not occurred, i.e., the refined child of the *loop* must be reset first to ensure that its execution is not interrupted.

The application of *TR_loop3* is shown in Figure 8.25, which results in the addition of the invariant (*inv_loop*).

Application of TR_loop3

```
flow(process_name, p, 1)
        (flow(A, p, 1)
                (leaf(G)(0),
                 loop(flow(H, p, 1)
                               (and(leaf(P), leaf(Q))(0),
                                leaf(R)(1)),
                       leaf(I)(0)),
                 leaf(J)(1)),
           par-rep(p1)(flow(B, (p, p1), 1)
                                 (all(p2)(leaf(K))(0),
                                  leaf(L)(1))(0),
           leaf(C)(0),
           interrupt(flow(D, p, 1)
                               (retry(flow(M, p, 1)(xor(leaf(S), leaf(T))(0), leaf(U)(1)))
                                      (flow(N, p, 1)(leaf(V)(1), leaf(W)(0)))(0),
                                leaf(O)(1)))
                      (leaf(E))(0),
           leaf(F)(0))
```

Applied Auxiliary Functions:

- list_child_leaves(flow(H, p, 1), p) = **leaf(P, p), leaf(Q, p), leaf(R, p)**

Output:

```
event reset_loop
 any p  where
  @grd_reset  p ∈ R
 then
  @act1_reset  P := P\{p}
  @act2_reset  Q := Q\{p}
  @act3_reset  R := R\{p}
```

Output in SI: where we assume **Process_Name** does **not** introduce parameter **p**

```
event reset_loop
 where
  @grd_reset  R = TRUE
 then
  @act1_reset  P := FALSE
  @act2_reset  Q := FALSE
  @act3_reset  R := FALSE
```

Figure 8.23: Application of TR_loop3

8.4.4 Translation Rules for Exception Handling Combinators

As explained earlier the child nodes of the exception handling combinators, unlike the other combinators, are **not** symmetrical, because the interrupting child can block the normal child execution at any time before completion.

TR_loop4: loop $\longrightarrow$ 1*invariant(s)
flow(parent-name, $(p_1, \dots, p_n)$, sw)(..., loop $(child_1, \dots, child_m)(0)$, ...)   ***where** at least one ($\textbf{child}_i \neq$ **leaf**) $(1 \leq i \leq m)$
successor(loop, n) = (child, parnum)
SI case (n = 0)   **invariants**   @inv_loop **build_succ_SI_inv** (child, 0) $\Rightarrow$ **conjunction_of_leaves** $(child_i, 0)$
MI case (n > 0)   **invariants**   @inv_loop $\forall p_1, \dots, p_n \,.\, p_1 \mapsto \dots \mapsto p_n \in$ **union_of_leaves** (child, 0) $\Rightarrow$ $p_1 \mapsto \dots \mapsto p_n \notin$ **union_of_leaves** $(child_i, 0)$

Figure 8.24: TR_loop4: Mapping a loop to (an) invariant(s)

The new translation rules *TR_interrupt*, *TR_disable_interrupt* and *TR_interrupt_inv* apply to both the *interrupt* and *retry* combinators, whereas *TR_retry* and *TR_retry_reset* only apply to the *retry* combinator.

8.4.4.1 TR_interrupt: Mapping the Interrupt and Retry Combinators to Disabling Guard(s) in the Normal Flow

Both the *Interrupt* and the *Retry* combinators will add a disabling guard to each leaf event of the normal flow. This guard will disable the execution of the normal flow, once the interrupting event executes.

Figure 8.26 presents the rule to get the Event-B output, which is in this case one or more guards. The rule will be applied to any *interrupt* or *retry* combinator, which will always has the refining property set to zero. The traversing steps of the rule requires getting every leaf event of the normal sub-flow of the exception handling combinator, which is done by applying the *list_child_leaves* function (Figure 8.2) to the *normal-child*. We also need the first child of the interrupting sub-flow.

The output will be a guard (*grd_interrupt*) added to each leaf event of the normal sub-flow, which is computed by applying the *conjunction_of_leaves* function to the first child ($child_1$) of the interrupting sub-flow in the single instance case. In the multiple instances case the guard is the flow parameters not in the *union_of_leaves* of the first child of the interrupting sub-flow.

In Figure 8.27, we show the application of *TR_interrupt* to both the *interrupt* and *retry* combinators at the second refinement level of Figure 8.1. The *interrupt-combinator* is applied at a higher level of the *retry-combinator*, that is why we can see the interrupting

Application of TR_loop4

```
flow(process_name, p, 1)
        (flow(A, p, 1)
                (leaf(G)(0),
                 loop(flow(H, p, 1)
                               (and(leaf(P), leaf(Q))(0),
                                leaf(R)(1)),
                       leaf(I)(0)),
                 leaf(J)(1)),
            par-rep(p1)(flow(B, (p, p1), 1)
                                (all(p2)(leaf(K))(0),
                                 leaf(L)(1))(0),
            leaf(C)(0),
            interrupt(flow(D, p, 1)
                               (retry(flow(M, p, 1)(xor(leaf(S), leaf(T))(0), leaf(U)(1)))
                                      (flow(N, p, 1)(leaf(V)(1), leaf(W)(0)))(0),
                                leaf(O)(1)))
                      (leaf(E))(0),
            leaf(F)(0))
```

Applied Auxiliary Functions:

- successor(flow(H, p, 1), p) = **leaf(J, p)**

Output:

invariants
@inv_par $\forall p.\ p \in J \Rightarrow p \notin P \cup Q$

Output in SI: where we assume **Process_Name** does **not** introduce parameter **p**

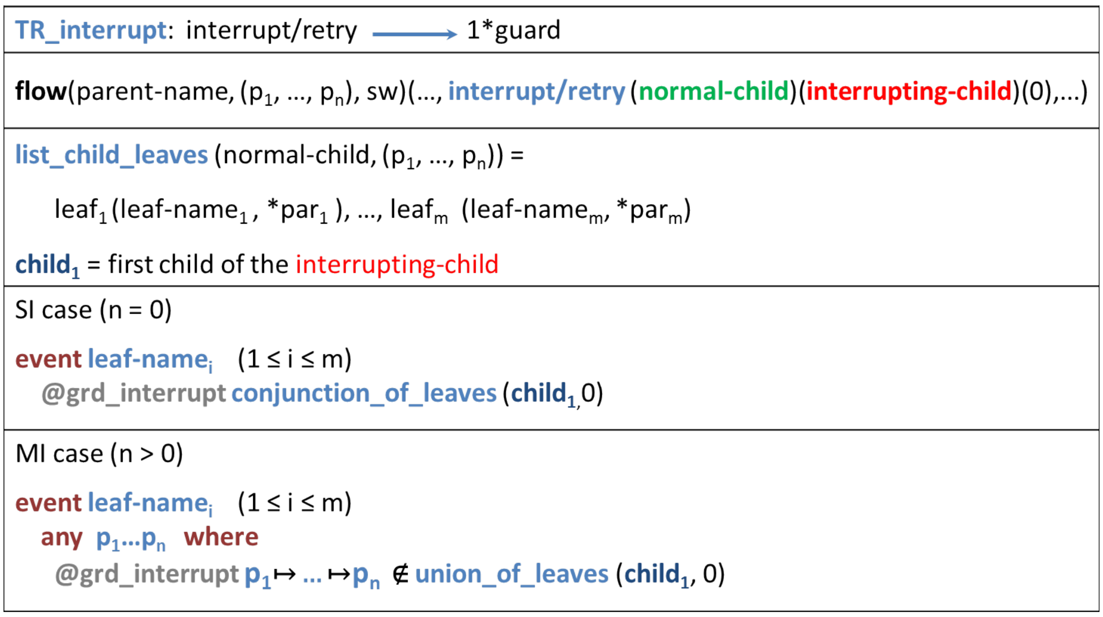
invariants
@inv_par $J = TRUE \Rightarrow P = FALSE \land Q = FALSE$

Figure 8.25: Application of TR_loop4

leaves of the *retry-combinator* (V and W) have the interrupting guard of the *interrupt-combinator*.

8.4.4.2 TR_disable_interrupt: Mapping the Interrupt and Retry Combinators to a Disabling Guard(s) in the Interrupting Event

This is to define the scope of the interrupting sub-flow of both the *interrupt* and *retry* combinators, where the interrupting sub-flow will not be applicable when the normal sub-flow completes successfully. As explained earlier in the refinement of *interrupt* and *retry* in sections 6.2.1 and 6.2.2, the refining child of the normal sub-flow must be always the last child of the normal sub-flow, while the solid child of the interrupting sub-flow must be the first child.



TR_interrupt: interrupt/retry $\longrightarrow$ 1*guard

flow(parent-name, (p_1, …, p_n), sw)(…, **interrupt/retry** (**normal-child**)(**interrupting-child**)(0),…)

list_child_leaves (normal-child, (p_1, …, p_n)) =

leaf$_1$ (leaf-name$_1$, *par$_1$), …, leaf$_m$ (leaf-name$_m$, *par$_m$)

child$_1$ = first child of the interrupting-child

SI case (n = 0)

event **leaf-name$_i$** $(1 \le i \le m)$
@grd_interrupt **conjunction_of_leaves** (**child$_1$**, 0)

MI case (n > 0)

event **leaf-name$_i$** $(1 \le i \le m)$
any **p_1…p_n** **where**
@grd_interrupt **$p_1 \mapsto … \mapsto p_n$** $\notin$ **union_of_leaves** (**child$_1$**, 0)

Figure 8.26: TR_interrupt: Mapping the *interrupt/retry* to Guard(s) in the Normal Flow

Figure 8.28 presents *TR_disable_interrupt* translation rule. In the traversing steps of the rule, we need to get the first child leaves by applying the *list_first_child_leaves* function to the *interrupting-child*. Then the guard will be added to each leaf event resulting from *list_first_child_leaves*, which is formed by applying *conjunction_of_leaves* in the single instance case, or the *union_of_leaves* in the multiple instances case to the last child ($child_n$) of the normal sub-flow.

In Figure 8.29 we apply *TR_disable_interrupt* to both the *interrupt* and *retry* combinators of Figure 8.1. The *interrupt-combinator* will result in the addition of the guard (*grd_interrupt*) to event E, while the *retry-combinator* will add the guard to event V.

8.4.4.3 TR_interrupt_inv: Mapping the Interrupt and Retry Combinators to an Invariant

The output of this rule is an invariant, which ensures that the last child of the normal flow and the first child of interrupting flow are mutually exclusive. The guards generated by *TR_interrupt* and *TR_disable_interrupt* translation rules, when applied to the last child of the normal flow and the first child of the interrupting flow respectively, ensures the mutual exclusive property.

Figure 8.30 presents the translation rule, where in the single instance case, we apply the *disjunction_of_leaves* function (Figure D.4), while in the multiple instances case we apply the *union_of_leaves* function. The *xor* operator in the single instance invariant is **not** implemented yet in the Event-B language, we use it here to show the need of a

Application of TR_interrupt

```
flow(process_name, p, 1)
        (flow(A, p, 1)
                (leaf(G)(0),
                loop(flow(H, p, 1)
                                (and(leaf(P), leaf(Q))(0),
                                leaf(R)(1)),
                        leaf(I)(0)),
                leaf(J)(1)),
        par-rep(p1)(flow(B, (p, p1), 1)
                                (all(p2)(leaf(K))(0),
                                leaf(L)(1))(0),
        leaf(C)(0),
        interrupt(flow(D, p, 1)
                                (retry(flow(M, p, 1)(xor(leaf(S), leaf(T))(0), leaf(U)(1)))
                                        (flow(N, p, 1)(leaf(V)(1), leaf(W)(0)))(0),
                                leaf(O)(1)))
                        (leaf(E))(0),
        leaf(F)(0))
```

Applied Auxiliary Functions:

1. ***interrupt-combinator* Case:**
- **list_child_leaves** (flow(D, p, 1), p) = leaf(S, p), leaf(T, p), leaf(U, p), leaf(V, p), leaf(W, p), leaf(O, p)
- **child$_1$** = leaf(E)

2. ***retry-combinator* Case:**
- **list_child_leaves** (flow(M, p, 1), p) = leaf(S, p), leaf(T, p), leaf(U, p)
- **child$_1$** = leaf(V)

Output:

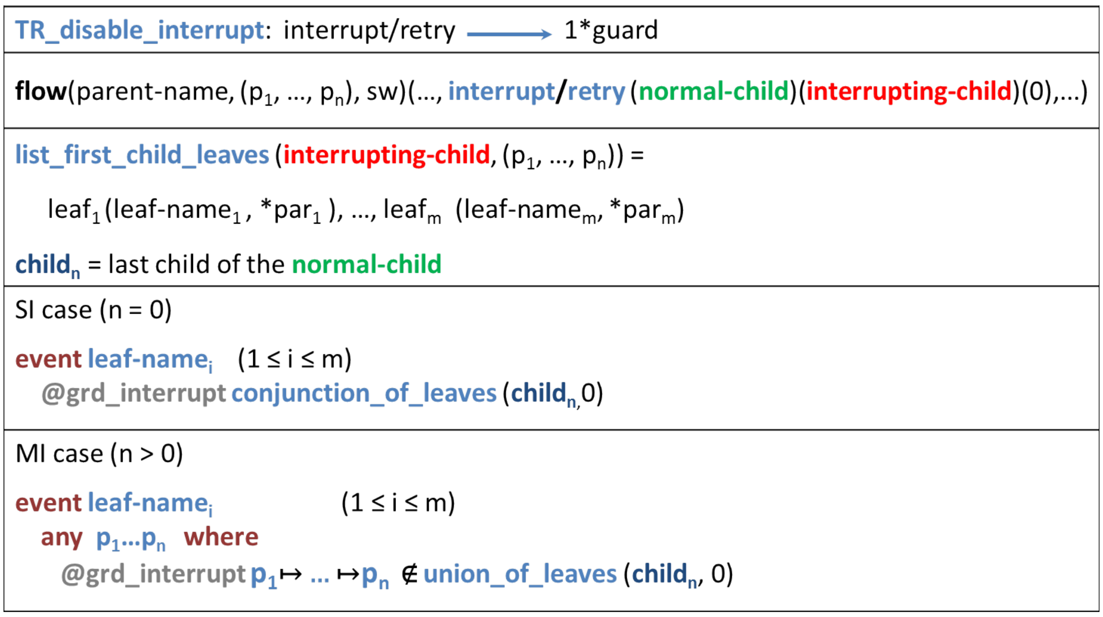
event S **any p where** @grd_interrupt1 **p** $\notin$ **E** @grd_interrupt2 **p** $\notin$ **V**	**event T** **any p where** @grd_interrupt1 **p** $\notin$ **E** @grd_interrupt2 **p** $\notin$ **V**	**event U** **any p where** @grd_interrupt1 **p** $\notin$ **E** @grd_interrupt2 **p** $\notin$ **V**
event V **any p where** @grd_interrupt **p** $\notin$ **E**	**event W** **any p where** @grd_interrupt **p** $\notin$ **E**	**event O** **any p where** @grd_interrupt **p** $\notin$ **E**

Output in SI: where we assume **Process_Name** does **not** introduce parameter **p**

event S @grd_interrupt1 **E** = **FALSE** @grd_interrupt2 **V** = **FALSE**	**event T** @grd_interrupt1 **E** = **FALSE** @grd_interrupt2 **V** = **FALSE**	**event U** @grd_interrupt1 **E** = **FALSE** @grd_interrupt2 **V** = **FALSE**
event V @grd_interrupt **E** = **FALSE**	**event W** @grd_interrupt **E** = **FALSE**	**event O** @grd_interrupt **E** = **FALSE**

Figure 8.27: Application of TR_interrupt to the *interrupt-combinator*

mutual exclusive operator in the single instance case. While the *partition* operator in Event-B can describe the mutual exclusive relationship in the multiple instances case.



TR_disable_interrupt: interrupt/retry $\longrightarrow$ 1*guard

flow(parent-name, (p_1, ..., p_n), sw)(..., interrupt/retry (normal-child)(interrupting-child)(0),...)

list_first_child_leaves (interrupting-child, (p_1, ..., p_n)) =

leaf$_1$ (leaf-name$_1$, *par$_1$), ..., leaf$_m$ (leaf-name$_m$, *par$_m$)

child$_n$ = last child of the normal-child

SI case (n = 0)

event leaf-name$_i$ $(1 \leq i \leq m)$
@grd_interrupt conjunction_of_leaves (child$_n$,0)

MI case (n > 0)

event leaf-name$_i$ $(1 \leq i \leq m)$
any $p_1 \ldots p_n$ where
@grd_interrupt $p_1 \mapsto \ldots \mapsto p_n \notin$ union_of_leaves (child$_n$, 0)

Figure 8.28: TR_disable_interrupt: Mapping the *interrupt/retry* to Guard(s) in the Interrupting Flow

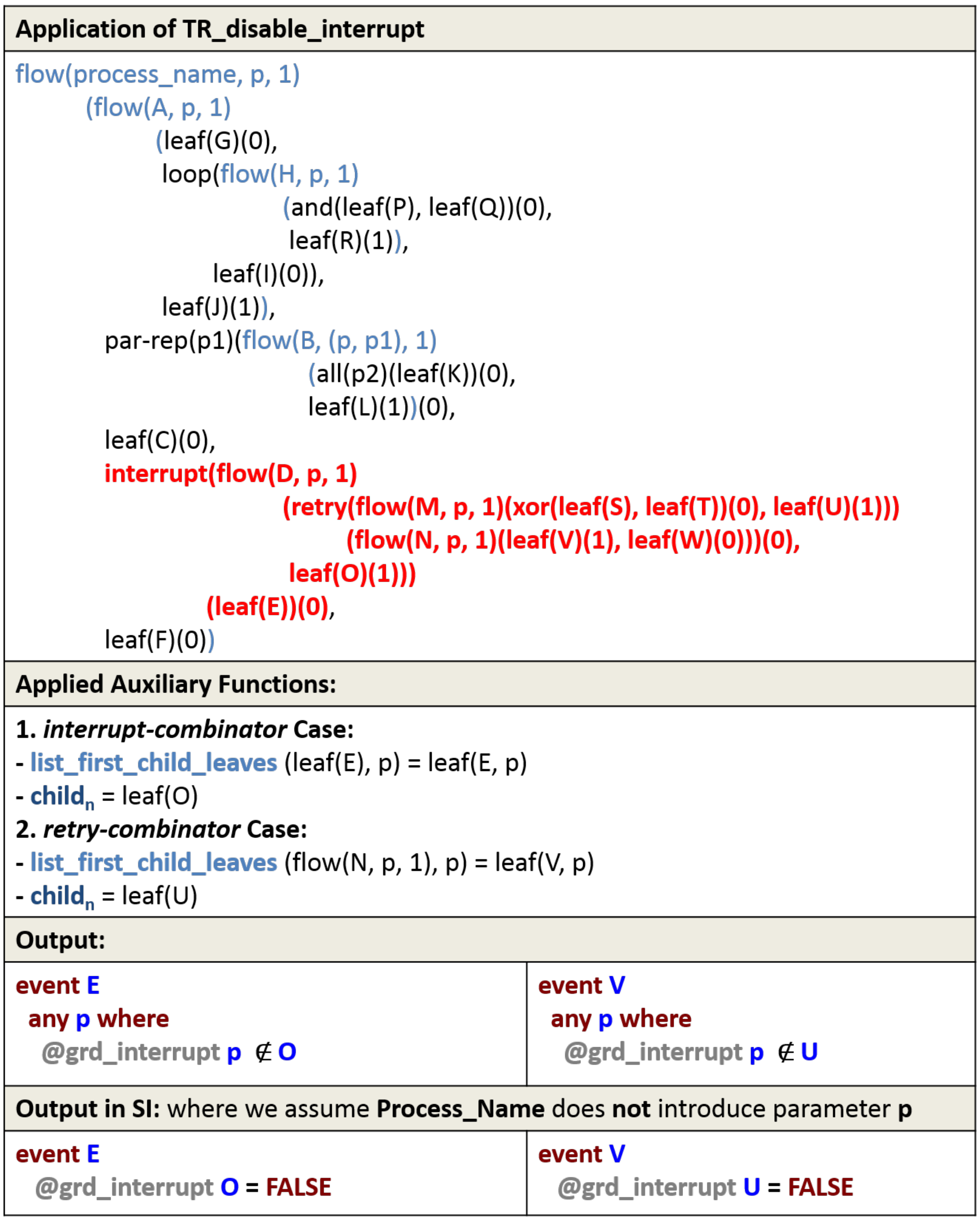
The application of *TR_interrupt_inv* is done in Figure 8.31, resulting in *interrupt1_inv* invariant for the *interrupt-combinator*, and *interrupt2_inv* invariant for the *retry-combinator*.

8.4.4.4 TR_retry: Mapping the Retry Combinator to Resetting Actions

The translation rule, *TR_retry* (Figure 8.32), will reset the control variables for each leaf event in the normal sub-flow of the *retry* combinator, this is to enable the execution of the normal sub-flow again.

The resetting actions of the control variables of the normal sub-flow will be added to the first child leaf events of the interrupting sub-flow. Similar to the resetting actions in the loop resetting event, in the single instance case where no parameters are introduced due to a replicator, the resetting action will set the leaf control variable to *false*, however if there are parameters added due to an application of a replicator, the leaf control variable will be reset to the empty set. In the multiple instances case, if the number of parameters of the leaf control variable is the same as its parent flow, then we just remove the flow parameters from the control variable, whereas if its parameters are more due to an application of a replicator, then the parent parameters will be removed from the domain of the leaf control variable.

In Figure 8.33, we apply *TR_retry* to the *retry-combinator* at the second refinement level. In this case the number of parameters is the same as the parent flow, because we have **not** introduced a replicator for any leaf event of the normal-child.



Application of TR_disable_interrupt

```
flow(process_name, p, 1)
      (flow(A, p, 1)
            (leaf(G)(0),
             loop(flow(H, p, 1)
                        (and(leaf(P), leaf(Q))(0),
                         leaf(R)(1)),
                  leaf(I)(0)),
             leaf(J)(1)),
       par-rep(p1)(flow(B, (p, p1), 1)
                        (all(p2)(leaf(K))(0),
                         leaf(L)(1))(0),
       leaf(C)(0),
       interrupt(flow(D, p, 1)
                        (retry(flow(M, p, 1)(xor(leaf(S), leaf(T))(0), leaf(U)(1)))
                              (flow(N, p, 1)(leaf(V)(1), leaf(W)(0)))(0),
                         leaf(O)(1)))
                (leaf(E))(0),
       leaf(F)(0))
```

Applied Auxiliary Functions:

1. *interrupt-combinator* Case:
- **list_first_child_leaves** (leaf(E), p) = leaf(E, p)
- **child$_n$** = leaf(O)

2. *retry-combinator* Case:
- **list_first_child_leaves** (flow(N, p, 1), p) = leaf(V, p)
- **child$_n$** = leaf(U)

Output:

event E **any p where** **@grd_interrupt p $\notin$ O**	**event V** **any p where** **@grd_interrupt p $\notin$ U**

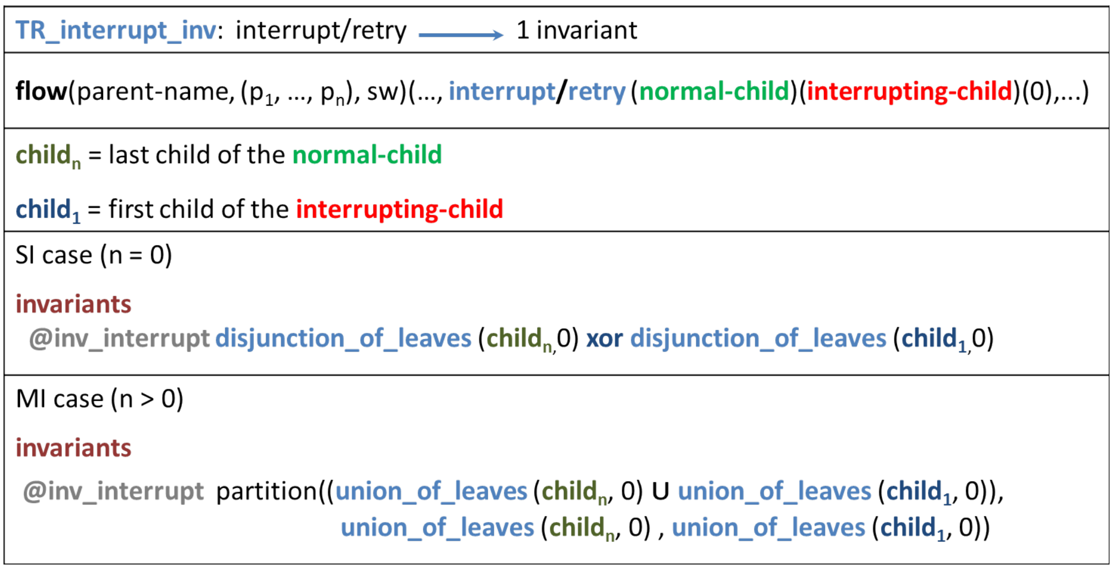
Output in SI: where we assume **Process_Name** does **not** introduce parameter **p**

event E **@grd_interrupt O = FALSE**	**event V** **@grd_interrupt U = FALSE**

Figure 8.29: Application of TR_disable_interrupt

8.4.4.5 TR_retry_reset: Mapping the Retry Combinator to a Reset Event

In the *TR_retry_reset* translation rule (Figure 8.34), the *retry-combinator* will be mapped to an event, that resets the control variables of the interrupting child, after it completes execution. This is to enable the normal sub-flow to execute again. As explained in Chapter 6, the *normal-child* of the *retry-combinator* can only complete execution once, while the *interrupting-child* can complete execution more than once.



TR_interrupt_inv: interrupt/retry $\longrightarrow$ 1 invariant

flow(parent-name, (p_1, …, p_n), sw)(…, **interrupt/retry** (**normal-child**)(**interrupting-child**)(0),…)

child$_n$ = last child of the **normal-child**

child$_1$ = first child of the **interrupting-child**

SI case (n = 0)

invariants
@inv_interrupt **disjunction_of_leaves** (**child**$_n$,0) **xor** **disjunction_of_leaves** (**child**$_1$,0)

MI case (n > 0)

invariants
@inv_interrupt partition((**union_of_leaves** (**child**$_n$, 0) U **union_of_leaves** (**child**$_1$, 0)),
union_of_leaves (**child**$_n$, 0) , **union_of_leaves** (**child**$_1$, 0))

Figure 8.30: TR_interrupt_inv: Mapping the *interrupt/retry* to an Invariant

The sequencing guard of this resetting event, will be after the interrupting sub-flow completes execution, that is why we use the *build_seq_grd* function where the *predecessor* child is the *interrupting-child* of the *retry-combinator*. The resetting actions of this event are done in the same way as in *TR_retry*, but this time the resetting is done for the *interrupting-child* control variables.

In Figure 8.35, we apply *TR_retry_reset*, where a new resetting event (*reset_retry*) is generated with the appropriate sequencing guard and resetting actions.

8.5 ERS Tool Support

The openness of the Event-B tool, Rodin (Section 3.3.4), allows us to support the ERS approach by a plug-in that provides an automatic generation of part of the Event-B model related to the ordering of events and their relationships at different refinement levels.

The Eclipse Modelling Framework (EMF) (Steinberg et al., 2008) provides a modelling and code generation facility based on structured data models, where an emf meta-model can be used to define the different entities, attributes and relations between the entities of a model. The ERS language, introduced in Section 8.2, is defined using the Eclipse Modelling Framework (EMF) meta-model, and then transformed into an Event-B EMF meta-model.

In the earlier version of the tool (Fathabadi et al., 2014), an ERS diagram was defined in an EMF tree structure. The transformation from the ERS language to Event-B was performed using the Epsilon Transformation Language (ETL)(Kolovos et al., 2014).

Application of TR_interrupt_inv

```
flow(process_name, p, 1)
        (flow(A, p, 1)
                (leaf(G)(0),
                 loop(flow(H, p, 1)
                               (and(leaf(P), leaf(Q))(0),
                                leaf(R)(1)),
                       leaf(I)(0)),
                 leaf(J)(1)),
          par-rep(p1)(flow(B, (p, p1), 1)
                                  (all(p2)(leaf(K))(0),
                                   leaf(L)(1))(0),
          leaf(C)(0),
          interrupt(flow(D, p, 1)
                               (retry(flow(M, p, 1)(xor(leaf(S), leaf(T))(0), leaf(U)(1)))
                                      (flow(N, p, 1)(leaf(V)(1), leaf(W)(0)))(0),
                                leaf(O)(1)))
                       (leaf(E))(0),
          leaf(F)(0))
```

Applied Auxiliary Functions:

1. ***interrupt-combinator*** **Case:**
- **$child_n$** = leaf(O)
- **$child_1$** = leaf(E)

2. ***retry-combinator*** **Case:**
- **$child_n$** = leaf(U)
- **$child_1$** = leaf(V)

Output:

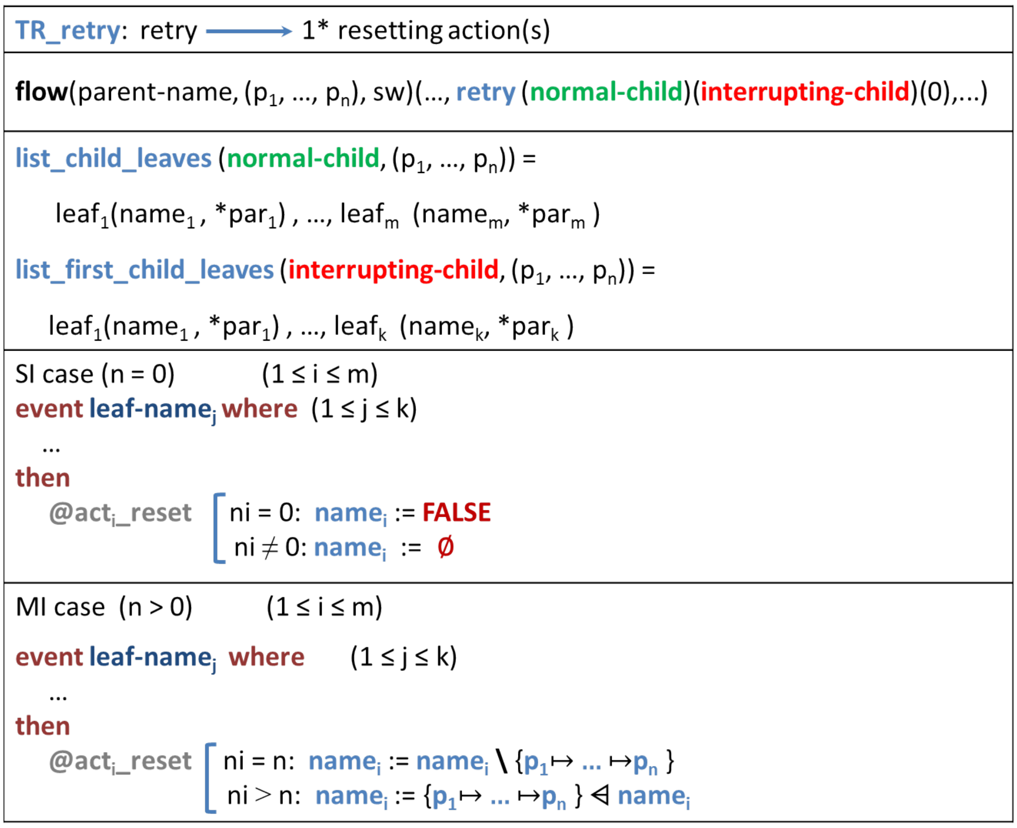
```
invariants
  @interrupt1_inv partition((O ∪ E), O, E)
  @interrupt2_inv partition((U ∪ V), U, V)
```

Output in SI: where we assume **Process_Name** does **not** introduce parameter **p**

```
invariants
  @interrupt1_inv O xor E
  @interrupt2_inv U xor V
```

Figure 8.31: Application of TR_interrupt_inv

In the updated version of the ERS plug-in (Dghaym et al., 2016), we provide a graphical environment for ERS, and we apply a different approach to transform from the ERS language to Event-B. The new followed approach is based on the generic Diagram Extensions framework for Event-B (Savicks and Snook, 2012). The framework used, is built specifically to support Event-B providing lots of helpful functionalities. In addition to supporting model transformation to Event-B, the generic Diagram Extensions framework provides graphical and validation support.



TR_retry: retry $\longrightarrow$ 1* resetting action(s)

flow(parent-name, (p_1, …, p_n), sw)(…, retry (normal-child)(interrupting-child)(0),…)

list_child_leaves (normal-child, (p_1, …, p_n)) =

leaf$_1$(name$_1$, *par$_1$) , …, leaf$_m$ (name$_m$, *par$_m$)

list_first_child_leaves (interrupting-child, (p_1, …, p_n)) =

leaf$_1$(name$_1$, *par$_1$) , …, leaf$_k$ (name$_k$, *par$_k$)

SI case (n = 0) $(1 \leq i \leq m)$
event leaf-name$_j$ where $(1 \leq j \leq k)$
…
then
@act$_i$_reset
ni = 0: name$_i$:= **FALSE**
ni $\neq$ 0: name$_i$:= $\emptyset$

MI case (n > 0) $(1 \leq i \leq m)$
event leaf-name$_j$ where $(1 \leq j \leq k)$
…
then
@act$_i$_reset
ni = n: name$_i$:= name$_i$ $\setminus$ $\{p_1 \mapsto … \mapsto p_n\}$
ni > n: name$_i$:= $\{p_1 \mapsto … \mapsto p_n\} \lhd$ name$_i$

Figure 8.32: TR_retry: Mapping the *retry* Combinator to Resetting Actions of the Normal Flow

Most of the new ERS translation rules described in this chapter and the updates to the original translation rules to include the ERS extensions, are added to the improved ERS plug-in. Generating the Event-B elements from the ERS diagram is based on the generator framework which is part of the generic Diagram Extension framework (Savicks and Snook, 2012; Wiki.event-b.org, 2015). Each rule transforming an ERS element to an Event-B element implements the *IRule* interface of the generator framework and defines the methods *enabled*, *dependenciesOK*, and *fire*. The *enabled* method checks when the translation rule should be applied, e.g., the rule transforming a leaf to a variable is enabled if the ERS source element is a leaf that is not a child of a loop. The *dependenciesOK* method checks if there are any dependencies required like other elements that need to be generated first, e.g., generating a sequencing guard to an event requires the event to be generated first or already existing in the machine, so if dependencies are not satisfied firing the rule will be postponed until all dependencies are satisfied. The *fire* method is the main method where the mapping of ERS elements to Event-B elements takes place. The *fire* method returns a list of *GenerationDescriptors* describing what should be generated, e.g., an ERS *leaf* is mapped to an *event* in Event-B.

In figures 8.36 and 8.37, we show how the translation rule *TR_par1* (Section 8.4.2.1) is implemented. The *enabled* method (Figure 8.36) checks that the rule can be only enabled when the source is a *leaf* with a *par-replicator* ancestor. The *dependenciesOK*

Application of TR_retry

```
flow(process_name, p, 1)
        (flow(A, p, 1)
                (leaf(G)(0),
                 loop(flow(H, p, 1)
                               (and(leaf(P), leaf(Q))(0),
                                leaf(R)(1)),
                      leaf(I)(0)),
                 leaf(J)(1)),
         par-rep(p1)(flow(B, (p, p1), 1)
                              (all(p2)(leaf(K))(0),
                               leaf(L)(1))(0),
         leaf(C)(0),
         interrupt(flow(D, p, 1)
                            (retry(flow(M, p, 1)(xor(leaf(S), leaf(T))(0), leaf(U)(1)))
                                   (flow(N, p, 1)(leaf(V)(1), leaf(W)(0)))(0),
                             leaf(O)(1)))
                      (leaf(E))(0),
         leaf(F)(0))
```

Applied Auxiliary Functions:

- list_child_leaves(flow(M, p, 1), p) = **leaf(S, p), leaf(T, p), leaf(U, p)**
- list_first_child_leaves(flow(N, p, 1), p) = **leaf(V, p)**

Output:

```
event V  where
  …
then
  @act1_reset S := S\{p}
  @act2_reset T := T\{p}
  @act3_reset U := U\{p}
```

Output in SI: where we assume **Process_Name** does **not** introduce parameter **p**

```
event V  where
  …
then
  @act1_reset S := FALSE
  @act2_reset T := FALSE
  @act3_reset U := FALSE
```

Figure 8.33: Application of TR_retry

checks that all the required elements are satisfied, in this case the leaf event to which the guard will be added should be already generated before firing the rule, otherwise the application of the rule will be postponed.

Figure 8.37 is the main part of the rule where the mapping between the ERS elements and the Event-B elements takes place. The function *getAncestorsAncestorsOfClass*, which is

TR_retry_reset: retry $\longrightarrow$ 1 resetting event
flow(parent-name, $(p_1, \dots, p_n)$, sw)(…, retry (normal-child)(interrupting-child)(0),…)
list_child_leaves (interrupting-child, $(p_1, \dots, p_n)$) = $\text{leaf}_1(\text{name}_1, {*}\text{par}_1), \dots, \text{leaf}_m(\text{name}_m, {*}\text{par}_m)$
SI case (n = 0) $(1 \le i \le m)$ event reset_retry   where   @grd_reset build_seq_grd (interrupting-child, null, null, null)   then $@\text{act}_i\text{\_reset}$ $\left[\begin{array}{l} ni = 0:\ \text{name}_i := \text{FALSE} \\ ni \neq 0:\ \text{name}_i := \emptyset \end{array}\right.$
MI case (n > 0) $(1 \le i \le m)$ event reset_retry   any $p_1 \dots p_n$ where   @grd_reset build_seq_grd (interrupting-child, $(p_1, \dots, p_n)$, null, $(p_1, \dots, p_n)$)   then $@\text{act}_i\text{\_reset}$ $\left[\begin{array}{l} ni = n:\ \text{name}_i := \text{name}_i \setminus \{p_1 \mapsto \dots \mapsto p_n\} \\ ni > n:\ \text{name}_i := \{p_1 \mapsto \dots \mapsto p_n\} \lhd \text{name}_i \end{array}\right.$

Figure 8.34: TR_retry_reset: Mapping the *retry* Combinator to a Resetting Event to Reset the Interrupting Flow

used in both the *enabled* and the *fire* methods, traverses down the tree to get a list of the first child elements of the specified class. In this case *getAncestorsAncestorsOfClass* has the *Par* class which represents the *par-replicator* as an argument, and we also check that the source is a leaf that is not decomposed, therefore, this will play the role of *get_first_child_leaves* which is used in Figure 8.12. The *index*, used in naming the *par-replicator* guard, is needed in the case more than one *par-replicator* is in the leaf parent chain. The *fire* method returns a generation descriptor, which will be used by the *Generator Framework* to generate the *par-replicator* guard(s).

All the generated Event-B elements from the ERS diagram are read only so that the modeller cannot change the control flow described by the ERS diagram, except the events in order to allow the user to add data related guards and actions.

8.5.1 User Interface

Creating an ERS Diagram in the ERS plug-in requires an existing Event-B machine, where the user can right click and add an ERS flow-diagram. The graphical environment of the ERS plug-in (Figure 8.38) consists of a diagram editor where we can add the ERS elements, a palette which contains the different ERS elements, which are the different

Application of TR_retry_reset

```
flow(process_name, p, 1)
        (flow(A, p, 1)
                (leaf(G)(0),
                 loop(flow(H, p, 1)
                               (and(leaf(P), leaf(Q))(0),
                                leaf(R)(1)),
                       leaf(I)(0)),
                 leaf(J)(1)),
            par-rep(p1)(flow(B, (p, p1), 1)
                                  (all(p2)(leaf(K))(0),
                                   leaf(L)(1))(0),
            leaf(C)(0),
            interrupt(flow(D, p, 1)
                               (retry(flow(M, p, 1)(xor(leaf(S), leaf(T))(0), leaf(U)(1)))
                                      (flow(N, p, 1)(leaf(V)(1), leaf(W)(0)))(0),
                                leaf(O)(1)))
                      (leaf(E))(0),
            leaf(F)(0))
```

Applied Auxiliary Functions:

- list_child_leaves(flow(N, p, 1), p) = **leaf(V, p), leaf(W, p)**

Output:

```
event reset_retry  any p  where
   @grd_reset  p ∈ W
 then
   @act1_reset  V :=  V\{p}
   @act2_reset  W :=  W\{p}
```

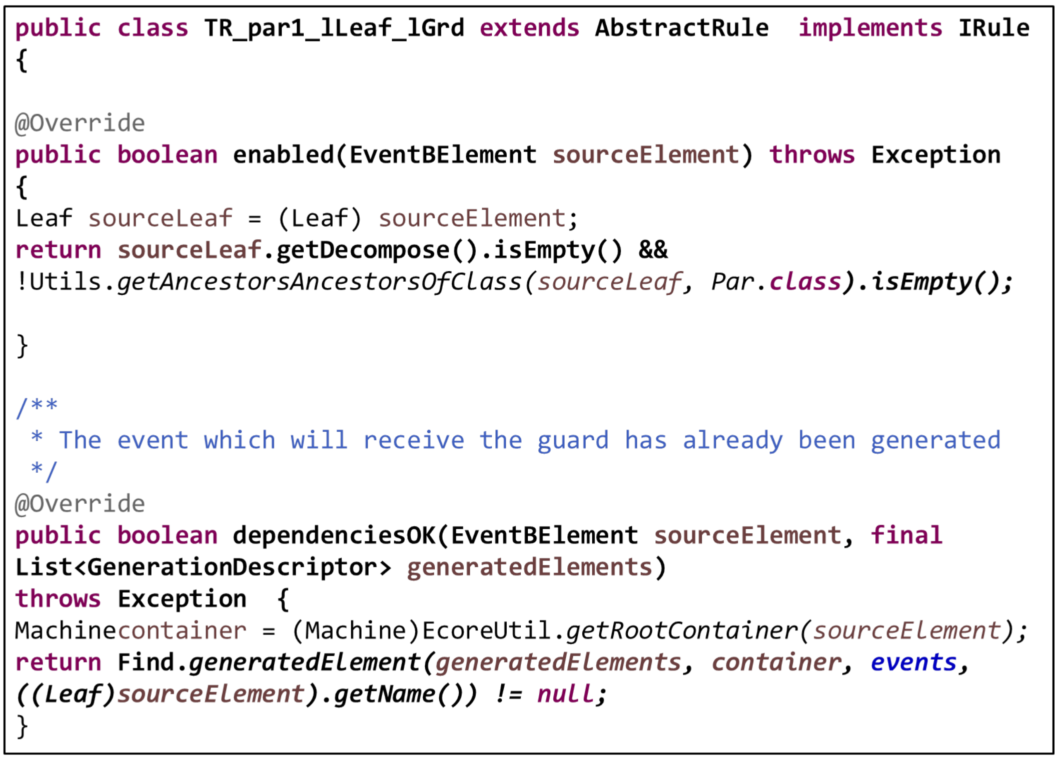
Output in SI: where we assume **Process_Name** does **not** introduce parameter **p**

```
event reset_retry
  where
   @grd_reset  W = TRUE
 then
   @act1_reset  V := FALSE
   @act2_reset  W := FALSE
```

Figure 8.35: Application of TR_retry_reset

combinators (*all*, *xor*,...), leaf and the connections, in addition to the toolbar buttons which allows the user to validate the diagram and generate Event-B.

Refining an ERS diagram, requires refining the machine first, the ERS diagram will be automatically copied to the refined machine, the user then can refine the leaves of the new copied ERS diagram by right clicking the leaf and selecting refine, as shown in Figure 8.39. The manually added specification elements, which in our case are any



```
public class TR_par1_lLeaf_lGrd extends AbstractRule  implements IRule
{

@Override
public boolean enabled(EventBElement sourceElement) throws Exception
{
Leaf sourceLeaf = (Leaf) sourceElement;
return sourceLeaf.getDecompose().isEmpty() &&
!Utils.getAncestorsAncestorsOfClass(sourceLeaf, Par.class).isEmpty();

}

/**
 * The event which will receive the guard has already been generated
 */
@Override
public boolean dependenciesOK(EventBElement sourceElement, final
List<GenerationDescriptor> generatedElements)
throws Exception  {
Machinecontainer = (Machine)EcoreUtil.getRootContainer(sourceElement);
return Find.generatedElement(generatedElements, container, events,
((Leaf)sourceElement).getName()) != null;
}
```

Figure 8.36: TR_par1 Rule: *enabled* and *dependenciesOK* Methods

requirement specifications that are not related to control flow, are automatically copied to the solid child after refining a leaf.

8.6 Conclusion

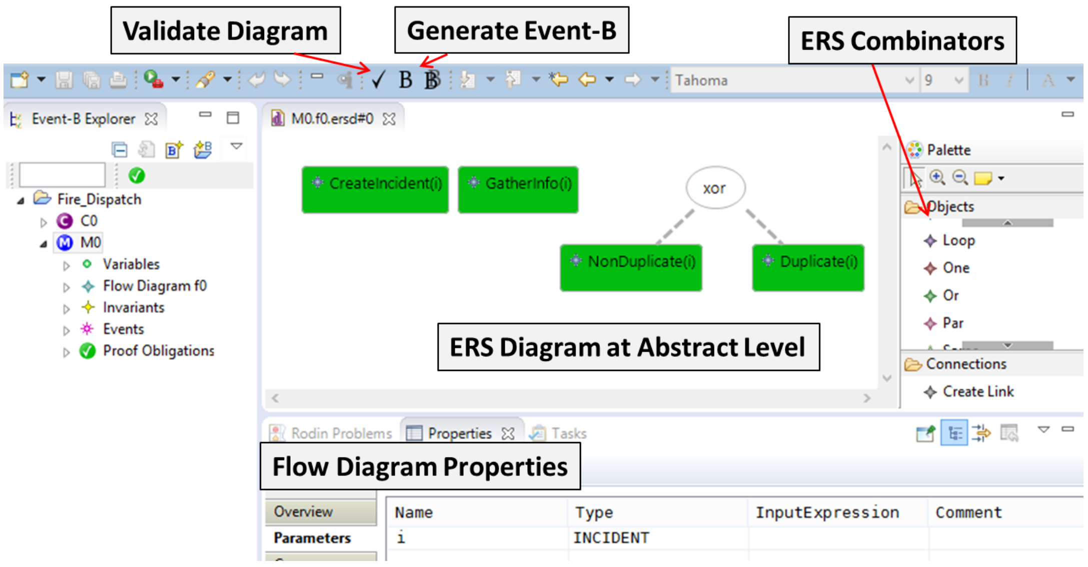
In addition to the original twenty three translation rules (Salehi Fathabadi, 2012), we added eleven new translation rules, in addition to updating some of the translation rules (e.g. *TR_loop1*) and functions. The newly added rules are summarised in Table 8.6.

Rule Name	**Rule Signature: ERS $\Longrightarrow$ Event-B**
TR_rangeExpression	replicator $\Longrightarrow$ guard(s)
TR_rep	replicator $\Longrightarrow$ invariant(s)
TR_par1	par-rep $\Longrightarrow$ guard(s)
TR_par2	par-rep $\Longrightarrow$ par-rep successor event(s) guard(s)
TR_par3	par-rep $\Longrightarrow$ invariant
TR_loop4	loop $\Longrightarrow$ invariant(s)
TR_interrupt	interrupt/retry $\Longrightarrow$ guard(s)
TR_disable_interrupt	interrupt/retry $\Longrightarrow$ guard(s) in interrupting sub-flow
TR_interrupt_inv	interrupt/retry $\Longrightarrow$ invariant
TR_retry	retry $\Longrightarrow$ action(s)
TR_retry_reset	retry $\Longrightarrow$ resetting event

Table 8.6: New ERS Translation Rules

```
/**
 * TR_par1, Transform a proper par-rep leaf to a guard in the equivalent event
(ensures that next child has not executed yet)
 */
@Override
public List<GenerationDescriptor> fire(EventBElement sourceElement,
List<GenerationDescriptor> generatedElements) throws Exception {
List<GenerationDescriptor> ret = new ArrayList<GenerationDescriptor>();
Leaf sourceLeaf = (Leaf) sourceElement;
Machinecontainer = (Machine)EcoreUtil.getRootContainer(sourceElement);
Event equivalent = (Event) Find.generatedElement(generatedElements, container,
events, ((Leaf)sourceElement).getName());

int index = 0;
for(EventBElement ee : Utils.getAncestorsAncestorsOfClass(sourceLeaf, Par.class)){
Par p = (Par)ee;
String name = Strings.GRD + index + Strings._PAR;
List<TypedParameterExpression> pars =
((FlowDiagram)p.eContainer()).getParameters();

List<Object> suc = Utils.successor(p, pars.size());

String predicate = "";
//SI case
if(pars.isEmpty())
predicate = Utils.conjunction_of_leaves((Child) suc.get(0), 0);
//MI case
else
predicate = Utils.getParMaplet(pars) + Strings.B_NOTIN +
Utils.union_of_leaves((Child) suc.get(0), 0);


ret.add(Make.descriptor(equivalent, guards, Make.guard(name, predicate), 5));
}

return ret;
}
```

Figure 8.37: TR_par1 Rule: *fire* Method



Name	Type	InputExpression	Comment
i	INCIDENT		

Figure 8.38: An Example ERS Diagram at Abstract Level in the ERS Tool

The formal definition of the ERS syntax and its translation rules have made it easier to

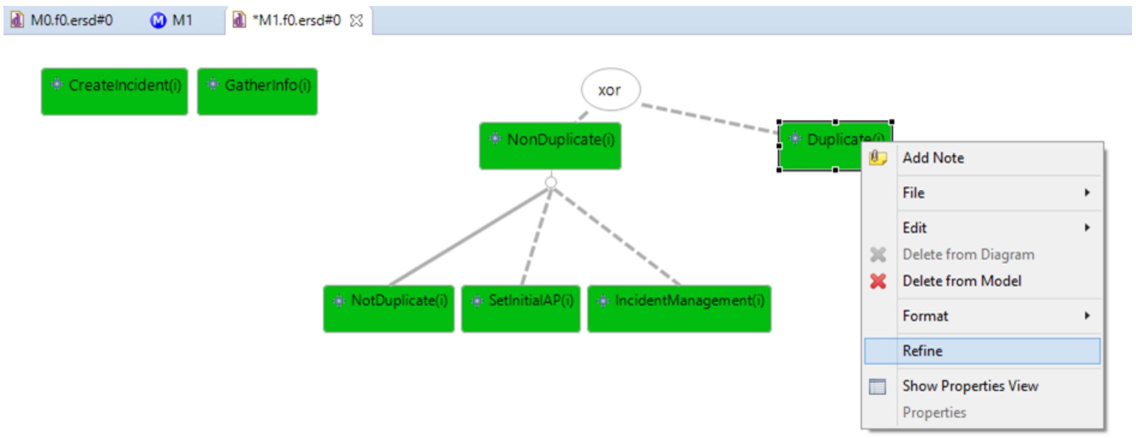


Figure 8.39: Refinement in the ERS Tool

automate the translation of the ERS language to Event-B language using the ERS plug-in, which extends the Event-B tool, Rodin. Having a tool that automatically generates Event-B models from ERS diagrams can encourage applying ERS for workflow modelling and also reduce the human error.

Moreover, the way the ERS tool can be used, which starts from an existing Event-B machine, makes it easier to combine ERS with other approaches based on Event-B semantics, such as the iUML-B statemachines as we have seen in Chapter 6. In addition to allowing the user to manually extend the model with manually added specification elements, which as explained earlier are not lost but will be reserved, even if they are added to an ERS element, they will be still copied to the solid child if refinement is done, and the user can still edit the manually added specification elements.

Chapter 9

Conclusions

In this chapter we present a summary of our main contributions and the lessons learned during the development of our case study, and identify the limitations in our work. We also evaluate our approach against other approaches that focus on workflow modelling and integration of diagrammatic approaches with formal modelling. Finally we present some directions for future work.

9.1 Contributions

The main contributions of this thesis can be summarised as follows:

- Defining a workflow modelling approach that combines both formal and semi-formal (graphical) modelling. This approach uses the formal method, Event-B for data handling and the Event Refinement Structures (ERS) diagrams for control flow.
- Extending the ERS approach to provide better support for workflow modelling, these extensions are divided into two types:
  - **Dynamic Modelling:** Includes defining a new combinator (*par-replicator*) that provides flexibility in modelling dynamic changes of data-dependent workflows. This new combinator allows us to model behaviours like the *parallel producer-consumer* pattern. We have also extended the parameter definition to include sets that can be defined dynamically as well as statically. Finally, we generalised the *loop* definition allowing the non-deterministic execution of the *loop* child events.
  - **Exception Handling:** Introduces two new combinators (*interrupt*, *retry*) that provide a mechanism to support exception handling. Both combinators allow one sub-flow (representing the exception) to stop the execution of

another sub-flow (the normal behaviour) before completion. The *retry* combinator, in addition to the interruption feature, resets the control variables of the normal sub-flow in an attempt to execute the normal sub-flow again.

- Validation and evaluation of our workflow modelling approach and our ERS extensions through application to two case studies:
  - Fire dispatch workflow
  - On-line booking workflow of the travel agency
- Formal specification of our extensions through translation rules that map the ERS elements to Event-B. In addition to updating the original ERS translation rules and functions to include our extensions, we define an additional eleven translation rules that are exclusively related to our ERS extensions.
- Tool support: Extending the ERS language and adding the ERS tranlation rules to an improved version of the ERS plug-in that provides graphical support in addition to some validation of the diagrams.

9.2 Limitations

In Chapter 3 and Chapter 8, we have explained some of the restrictions of the ERS diagrams. Some of these restrictions are necessary to ensure the proper behaviour of the ERS diagrams like the case of the solid last child of the normal sub-flow of an exception handling combinator. Other restrictions are related to technical issues and it might be possible to remove them if we find cases that requires so, such as:

- A *loop* cannot be the first child of a flow.
- The *loop* and the *par-replicator* cannot be directly followed by an exception handling combinator
- The *loop* and the *par-replicator* if followed by a flow (i.e. a refined leaf) then the first child must be a solid child.
- The first child of the interrupting sub-flow of a an exception handling combinator must be a solid child.

Other restrictions such as the *single solid line rule* and applying a combinator to a leaf only have proved through different case studies, to be necessary to simplify the ERS diagrams and their generated models. Having simpler models will in turn facilitate the automation of the proofs. Similarly the restriction that does not allow the same leaf name in different branches is important due to dependency between the events, which if removed could lead to a deadlock. Take the simplest case which is leaf A followed by leaf A. The second leaf event A requires the execution of A and at the same time checks that A has not executed. If the modeller requires the same event at different branches, then the solution is to define similar events, but with different names.

9.3 Lessons Learned

In this section we present the lessons learned while using ERS in modelling the case studies:

9.3.1 Separation of Control Flow and Data Handling

In our workflow modelling approach, we adopt the separation of control flow from data handling, where we model control flow using ERS diagrams and leave the data handling to be done manually using Event-B. However, in our case the separation does not mean the independence of the control flow from the data flow, this is because the ERS diagrams semantics are defined by transforming the diagrams into Event-B. Moreover defining explicit control flow variables in Event-B, have made it possible to relate the data variables to the control flow variables.

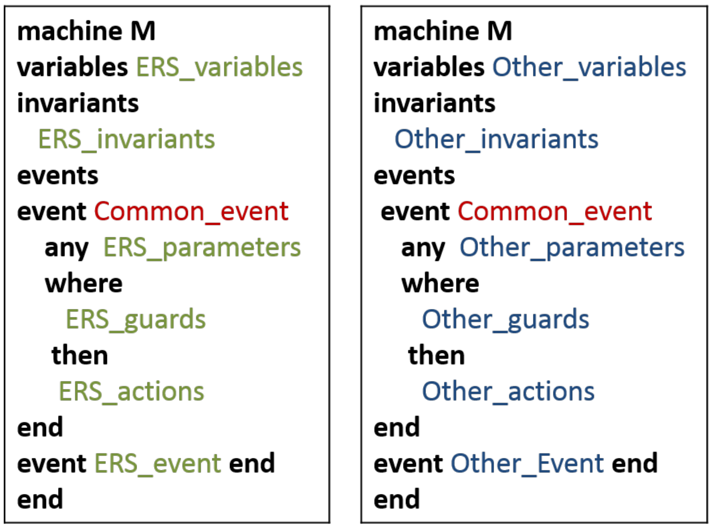
In both case studies, we have explicitly defined, using invariants, the relation between manually defined data variables and the ERS automatically generated control variables. By defining such invariants, we managed to define the scope of the data variables, i.e., which events they are relevant to. In addition to that, in many cases we used a combination of data variables and control variables to model the requirements, e.g., the requirements related to closing an incident in the fire dispatch case study.

An advantage of the separation is having simpler diagrams without complicating them with lots of specification elements concerned with data. The simpler the ERS diagrams are, the simpler the Event-B models are; hence easier verification as we have seen most of the proof obligations of our case studies were discharged automatically.

9.3.2 Integration of ERS with Other Event-B Approaches

Using the ERS plug-in we have seen how we used ERS with iUML-B statemachines, where we used ERS to model the overall dispatch workflow related to an incident and used statemachines to model the state changes of a physical resource. The common events between the ERS diagram and the statemachine were extended by the additional guards and actions resulting from the statemachine. We have also extended the ERS diagram in the travel agency case study with timing patterns.

The integration of ERS with other approaches generating Event-B models, such as iUML-B statemachines, is similar to the shared event composition approach (Silva and Butler, 2010), where the composed machine includes variables, invariants and events from both approaches. The common events are synchronised by the conjunction of the guards and the combination of their actions. Consider, for example Figure 9.1(a) representing a model generated by ERS and Figure 9.1(b) representing a model generated by a different approach or even a different ERS diagram. The integration of both models is represented in Figure 9.2 showing how the common event is synchronised.



```
machine M
variables ERS_variables
invariants
  ERS_invariants
events
event Common_event
   any  ERS_parameters
   where
      ERS_guards
    then
     ERS_actions
end
event ERS_event end
end
```

(a) ERS

```
machine M
variables Other_variables
invariants
  Other_invariants
events
 event Common_event
   any  Other_parameters
   where
      Other_guards
    then
      Other_actions
end
event Other_Event end
end
```

(b) Other Approach

Figure 9.1: Machine *M* Generated by Applying Different Approaches

Sometimes the integration may cause a deadlock if the shared events constraints generated by the different approaches or diagrams are inconsistent, that is why it is always required to check for deadlock freedom using ProB (Leuschel and Butler, 2003; Hallerstede and Leuschel, 2011). Consider the two ERS flow diagrams in Figure 9.3, integrating such diagrams will cause a deadlock, because *B* requires the execution of *A* in *ERS1*, and *A* requires the execution of *B* in *ERS2*. However, such deadlock can be easily detected using ProB model checking.

9.3.3 Proper Behaviour of the Model

In addition to supporting control flow, in many cases we have seen how the ERS translation rules, which are related to refinement have ensured that requirements are satisfied by the model. For example, the *par-replicator* refinement rule ensures that the same number of instances have executed for both the first and last events of the *par-replicator*

```
machine M
variables ERS_variables, Other_variables
invariants
  ERS_invariants
  Other_invariants
events
event Common_event
   any  ERS_parameters, Other_parameters
   where
     ERS_guards
     Other_guards
   then
     ERS_actions
     Other_actions
end
event ERS_event end
event Other_event end
end
```

Figure 9.2: Integration of ERS with Other Approaches

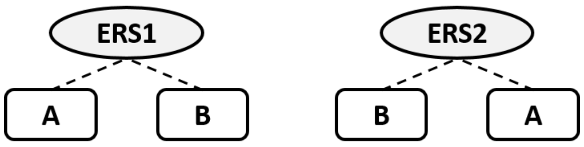


Figure 9.3: Example of Inconsistent Integration of Diagrams

sub-flow, before it can be interrupted by the follow-on event. In the fire dispatch case study (Chapter 5), the *par-replicator* refinement rule ensured that any allocated resource to the incident must be deallocated before considering the incident as managed. Likewise, in the travel agency case study (Chapter 7) the *par-replicator* refinement rule ensured that any selected flight/car/hotel must be either booked or removed from the itinerary before completing the itinerary.

9.3.4 Modelling New Behaviours and Features

The addition of the new ERS extensions made it possible to model some behaviours that were difficult or even not possible to represent in the original ERS. The *parallel producer-consumer* pattern is one of these behaviours that could not be represented by the original ERS, while the addition of the *par-replicator* and the extension of the parameter definition made it possible to explicitly model this behaviour using ERS.

Before introducing the exception handling combinators, it was difficult to model one flow interrupting another, as we have seen in the duplicate case of the fire dispatch case study (Chapter 5), where we had to change the requirement or else using the original ERS would have ended with a complex ERS diagram with many alternative branches. After introducing the *interrupt* combinator we managed to model the original requirement (Chapter 6) in a simple ERS diagram.

The new *retry* combinator, has made it possible to model cancellation behaviours. Although this combinator only resets control variables, we have shown in both case studies how we can manually undo data related actions. For example, in the case of a lost allocation of a physical resource, we used the *retry* combinator to make it possible to get a replacement, where we manually cancelled the allocation of the physical resource to the incident.

9.3.5 Role of Invariants

The ERS approach generates invariants to ensure the behaviour of the model is maintained all the time. In refinement some of the ERS generated invariants also help to discharge proof obligation by avoiding having repeated guards in events. This is in particular useful when the first child is not the solid child, and normally in ERS we add the guards to the first child of the flow in refinement, therefore by having explicit invariants we can discharge *GRD* proof obligations and avoid having unnecessarily repeated guards.

Take for example the refinement of *IncidentHandling* in Chapter 5. As shown in Figure 9.4, the refining event of *IncidentHandling* is the last event *DeallocateRes*. Applying the *par-replicator* to *IncidentHandling* will result in the guard $i \notin IncidentManaged$ added to the event *IncidentHandling*. After refining the *IncidentHandling* event, the *par-replicator* guard will be added to the first event *AllocateRes* to interrupt the execution of the *par-replicator* flow, once the *IncidentManaged* event occur. However *DeallocateRes* is the refining event of *IncidentHandling*, hence according to the *GRD* proof obligation we need the *par-replicator* guard in *DeallocateRes*. In this case the following invariant resulting from the *par-replicator* refinement will ensure that the *GRD* proof obligation is satisfied:

$$\forall i \cdot i \in IncidentManaged \implies AllocateRes[\{i\}] = DeallocateRes[\{i\}]$$

9.3.6 Role of ERS Diagrams in Refinements and Proofs

In ERS, applying more than one combinator to the same event at the same refinement level is not permitted. Such restriction has the advantage of imposing smaller changes at each refinement level. Some studies have shown that such approach, although resulting in more refinements, helps in reducing the proving effort and help achieve more automation

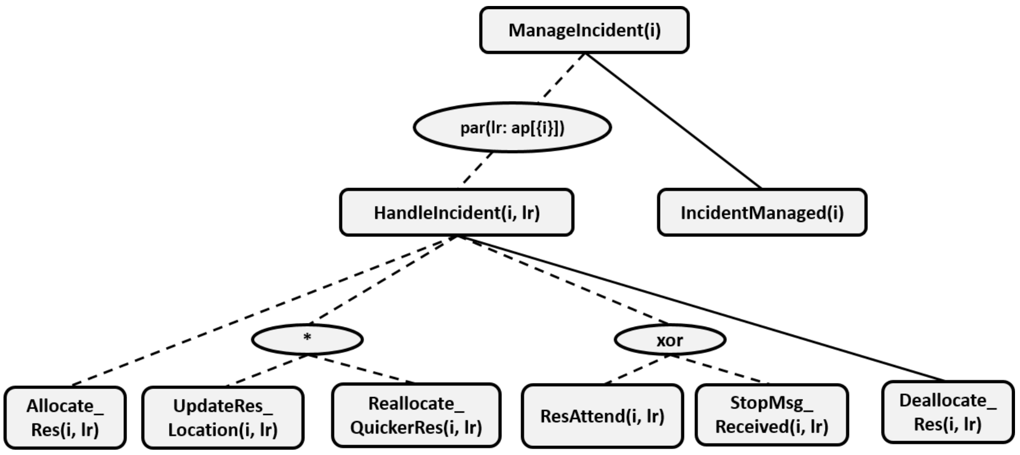


Figure 9.4: ERS Refinement Example from Fire Dispatch Workflow

of the proofs (Butler and Yadav, 2008; Butler, 2009b), and we have seen in both case studies that most of the proofs were done automatically.

9.4 Comparing ERS with Other Workflow Modelling Approaches

9.4.1 Workflow Control Flow Patterns

Russell et al. (2006a) describe forty three control flow patterns for business process modelling. This is an extension to their work which initially described twenty control flow patterns in van der Aalst et al. (2003). They use coloured Petri-Nets to describe the patterns and they do an evaluation to different control flow offerings including UML activity diagrams.

One of the important features of ERS is multiple instance modelling, which can be done using the different replicators (*all*, *some*, *one*, *par-replicator*), among the forty three patterns, they identify seven multiple instance patterns. The patterns "*Multiple Instances with a Priori Design-Time Knoweldege*" and "*Multiple Instances with a Priori Run-Time Knoweldege*", as defined by the *Workflow Initiative* allow the concurrent execution of multiple instances, where any subsequent tasks cannot start before the completion of all the task instances. The different between the two patterns is that, the number of instances is in the former pattern is known at design time, while in the latter it is known during runtime. Both patterns can be represented by the ERS *all-replicator*, where in the first one the type of the set is constant, while in the second one, the type of the set is a variable.

The pattern "*Multiple Instances without a Priori Run-Time Knoweldege*" allows the concurrent execution of multiple instances where the number of instances is not known beforehand and new instances can be initiated during execution. The pattern "*Dynamic Partial Join for Multiple Instances*" is a variant of "*Multiple Instances without a Priori Run-Time Knoweldege*" pattern, but does not require the completion of all created instances to trigger the next task. Both patterns can be represented by the *par-replicator* and the *parallel producer-consumer* pattern, by defining the required completion criteria which is determined by the *par-replicator* next event.

The remaining multiple instances patterns are indirectly captured by the *some* and *one* replicators. The pattern "*Multiple Instances without Synchronisation*", allows the creation of multiple instances of an activity that can run concurrently and do not require synchronisation upon completion. This could possibly be captured by the *some-replicator*. However, in the case of the *some-replicator*, if there is an event following the *some-replicator*, then its execution requires the execution of at least one instance of the *some-replicator*. One possibility to avoid the one instance execution requirement, can be achieved by applying weak sequencing when introducing the *some-replicator*.

The pattern "*Static Partial Join for Multiple Instances*", allows the concurrent execution of multiple instances of a task, where m instances are created. Once n instances ($n < m$) have completed, the next task is triggered. The pattern "*Static Partial Join for Multiple Instances*" is similar to the *some-replicator*, where $n = 1$ is always the case for the *some-replicator*. If we add an additional guard to the *some-replecator* follow-on event specifying the required (n) number using Event-B, then this pattern can be captured by ERS.

The pattern "*Cancelling Partial Join for Multiple Instances*" is similar to "*Static Partial Join for Multiple Instances*". The only difference is that once the required n instances have executed, the "*Cancelling Partial Join for Multiple Instances*" pattern will cancel the execution of the remaining $m - n$ instances. The *one-replicator* represents a special case of the pattern "*Cancelling Partial Join for Multiple Instances*", where $n = 1$ is always the case. However, adding an additional guard using Event-B to the *some-replicator* to disable the execution of the *some-replicator* after a specified number (n) can cover this pattern. The *par-replicator* can also represent this pattern.

This shows how the ERS four replicators captured the behaviour of all the seven multiple instances patterns. The same applies to most of the other control flow patterns of the *workflow patterns initiative*. Here we focused on the multiple instances patterns to show the need for a dynamic pattern like the *par-replicator* which in this case managed to capture the behaviour of three of the multiple instances patterns, and also the addition of *range-expression* helped to cover these patterns.

The long list of patterns defined by Russell et al. (2006a) is criticised by Börger (Börger, 2007a,b), where the patterns are simplified into eight general patterns, described using abstract state machines (ASM) (Börger, 2005), which is a rule-based modelling approach. The rest of the patterns can be derived from the basic eight, using parameter instantiation, refinement and composition. ASM and the eight control flow patterns are described in more detail in chapters 2 and 3.

In Table 9.1, we evaluate the extended ERS patterns against the eight generic patterns, in the same way Russell et al. (2006a) evaluated different workflow approaches against the forty three patterns. The first four patterns represent the sequential category, while the last four represent the parallel category. The "+" score means the pattern is explicitly supported by ERS, while the "−" scoring means, it is not explicitly supported by ERS.

Table 9.1: Evaluation of ERS Patterns Against the Generic Control Flow Patterns

Pattern	**Score**	**Motivation**
Sequence	+/−	Directly supported by the left-to-right arrangement of events, the "−" is because of the related *milestone* pattern which can be supported by additional Event-B guards to check the specified state.
Iteration	+/−	Structured loops are supported, but recursion and arbitrary cycles, i.e., cycles having more than one entry or exit point, are not supported.
Begin/Termination	+/−	Can be explained by the enabling/disabling conditions in Table 4.1 and Table 6.1, where some combintarors have explicit termination like the loop and the par-replicator, while others termination can be implecitly described by their disabling conditions. The cancellation patterns are classified under termination, can now be supported by the *interrupt-combinator*. However, we still give a "−" because of the *CancelRegion* where *interrupt* cannot cancel any set of unconnected activities, which is still doable using Event-B.
Selection	+	Supported using *or*, *xor* combinators and their generalisations into *some*, *one* replicators.
Parallel Split	+	Supported using *and*, *or* combinators and the *all*, *some*, *par* replicators, where the *par-replicator* supports dynamic sets.
Merge	+/−	merging with synchronisation is supported by *and/all*, while merging without synchronisation is supported by *or/some*. The − because some variants of the *merge* pattern described by the *workflow patterns initiative* are not supported
Interleaving	+/−	No explicit ERS combinator, but parallelism in Event-B is only supported through interleaving. Concurrency supported by *and*, *all*, etc. can only be done through interleaving in Event-B
Trigger	+/−	Can be implicitly represented using monitored guards in Event-B, however an enabled event in Event-B is permitted to execute but cannot be forced to execute.

Regarding the interleaving pattern, there is no special ERS combinator that only supports interleaving, the ERS combinators (*and, or, all, some, par*) support concurrency in general including interleaving. On the other hand, Event-B only allows the non-deterministic execution of one event if more than one event is enabled, so the actual parallel execution of the Event-B events is done through the interleaving of events. Therefore since ERS semantics is based on Event-B, then concurrency can only be achieved by the interleaving of events. However, the author in Börger (2007b) mentions that interleaving might be better treated as a different form of parallelism rather than a pattern by itself, i.e., interleaving can be another form of the parallel split pattern. Similarly the trigger pattern can be a form of the start patterns.

In the merge pattern we gave ERS partial support, because of variants like the *Multi-Merge* are not supported by ERS. For example, the *Multi-Merge* pattern allows the convergence of multiple branches into a single branch. However, unlike the ERS *or-combinator*, the execution of each incoming branch will result in enabling the execution of the subsequent branch.

In Table 9.1, most of the patterns if not supported explicitly by ERS can be supported implicitly by adding some additional Event-B guards and actions to the existing ERS combinators. In most of the patterns we have seen partial support instead of a full support, due to the specific variants specified by the *workflow patterns initiative*. However, if we treat the patterns in general without getting into the detailed specification as the *workflow patterns initiative* does, most of the partial support can be changed into full support.

By comparing the ERS constructors with Börger's approach, we can see that ERS can be used for workflow modelling, but one of the main differences is our distinction between the constructors that introduce dynamic and static parameters like the difference between the *all-replicator* and the *par-replicator*, where the *all* parameter value, unlike the *par-replicator* parameter, should be static i.e. known before executing the *all* event and cannot be changed after its execution. The ASM eight patterns does not distinguish between the static and dynamic character of the parameter and leave it as a matter of how the parameter is declared.

Petri Nets (Murata, 1989) are also widely used as a graphical formal approach for workflow modelling. However, Petri Nets are criticised by van der Aalst and ter Hofstede (2005) for their limitations in supporting multiple instances patterns, cancellation and advanced synchronisation patterns, which is why they present YAWL (Section 2.6), inspired by Petri Nets but extended to overcome these limitations. ERS, on the other hand supports multiple instances through its different replicators, also cancellation is supported by the new extensions *interrupt* and *retry*, where cancellation in this sense means aborting the execution of subsequent tasks when cancellation happens, or in other words removing the tokens of execution. Advanced synchronisation is supported by the *or-combinator* of ERS.

9.4.2 UML Activity Diagram

Activity diagrams are one of the Unified Modelling Language (UML) diagrams (Rumbaugh et al., 1998), that describe the behavioural part of systems. Activity Diagrams are widely used to describe control flow (omg.org, 2012). In Dghaym et al. (2013), we have done a comparison between activity diagrams and ERS, and evaluated both approaches by applying them to a fire dispatch case study.

Figure 9.5 from Dghaym et al. (2013), represents the various ERS non-replicator combinators and their counterparts in activity diagram. As shown in the diagram all the non-replicator combinators of ERS (*and*, *xor*, *or*, *sequencing*, *loop*), can be represented using activity diagrams. The *and-combinator* in ERS is represented as a fork to describe the concurrent execution of *Eve1* and *Eve2*, and then they are joined or synchronised into one flow before *Eve3* can be executed. The *xor-combinator* is similar to an activity diagram decision-node with mutually exclusive guards.

The *or-combinator* means at least one of the sub-events is executed before the next event (*Eve3*) can be enabled, that is why we use a merge-node in the activity diagram which requires at least one flow to proceed and does not synchronise the flows like the join-node. However, in the '*or*' case in Figure 9.5, *Eve3* in ERS can be executed only once even if both *Eve1* and *Eve2* have executed, whereas in activity diagram according to the token flow translation, *Eve3* will be executed twice if both *Eve1* and *Eve2* have executed.

The left to right sequencing in ERS is shown by the direction of the arrow in activity diagrams. Finally, the star symbol, which indicates looping, can be described by using a decision-node and a merge-node that bring all flows together without synchronisation. Here the decision-node is added before the loop event (*Eve1*) to describe the zero iteration. UML 2.0 has introduced a structured node to represent a *loop*, but it is still a syntactic sugar that is equivalent to applying the decision and merge nodes as shown in Figure 9.6.

UML 2.0 has also introduced a new construct called expansion region. The expansion region, shown in Figure 9.7 (omg.org, 2012), is a structured activity node that takes collections of elements as input and can return output collections described in expansion nodes. The execution of elements can be described in three different modes (parallel, iterative, stream) according to a keyword at the top of the region. In the parallel mode, the elements execute concurrently and can overlap in time. In the iterative mode, execution is done in sequence, so the execution of one element can not start until the previous one completes its execution and if ordering is applied to the input elements, the

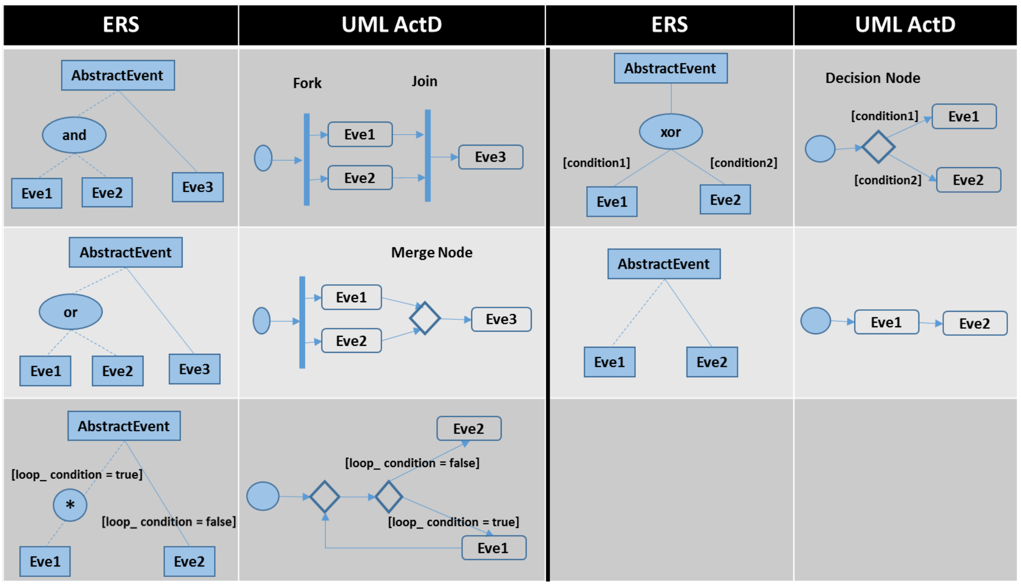


Figure 9.5: Graphical Comparison of the non-replicator Combinators of ERS and Activity Diagrams

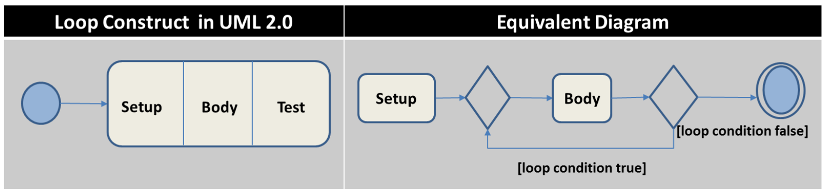


Figure 9.6: Structured Loop Node in UML 2.0 and Its Equivalent Representation

same order must be followed during execution. In a stream mode, there is exactly one execution in the region, where elements are offered to execution in a stream, preserving ordering if applied. That is, a stream mode execution is continuous and elements are processed as soon as one is available.

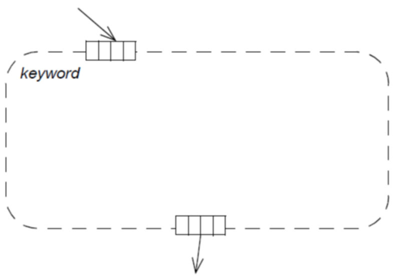


Figure 9.7: Expansion Region (omg.org, 2012)

Regarding the ERS replicators (*all, some, one*), there are no explicit counterparts for the *some* and the *one* repliactors, but the *all* replicator can be equivalent to the expansion region in the parallel mode. Similar to the ERS replicators (*all, some, one*), the expansion region can not change its input during its execution and the input nodes must be determined in advance, hence activity diagrams cannot support dynamic behaviour like the *par-replicator*. However, ERS does not have an equivalent representation of the expansion region in the iterative and the stream modes.

When it comes to exceptional behaviour, we have explained in Chapter 6, how the *interrupt* combinator can support behaviour similar to the interruptible activity regions of activity diagrams. However the main difference between the ERS *interrupt* and the activity diagram interruptible region is that ERS interrupted tasks must be within the same subactivity, whereas the interruptible region can interrupt arbitrary tasks that are not related or not connected.

Regarding the resource perspective, which is one of the four perspectives of workflow modelling (Control, Data, Exception Handling, Resource) as described by the *Workflow Patterns Initiative*, activity diagrams support activity partitions or swimlanes where actions and activities are grouped according to whom or what is performing them. In Dghaym et al. (2013), we have done the resource grouping in ERS using colour coding of the ERS events according to whom/what is performing the event.

9.4.3 iUML-B Statemachines

iUML-B (Snook, 2014; Wiki.event-b.org, 2014) is another form of UML-B (presented in Section 3.3.5), but with stronger integration with Event-B. Some of the features supported by iUML-B state-machines include junctions, forks, joins and multiple instances.

Junctions in iUML-B result in the addition of a disjunctive guard to the events. A new pseudo state called "*any*" is also introduced to iUML-B, which represents all the states at the same level of the statemachine, where a transition with source *any* can fire irrespective of the current state of the statemachine. Lifting is another feature introduced to iUML-B, where a state-machine can be lifted to multiple instances.

However, the main difference between ERS and iUML-B is what is the purpose of the diagram, where ERS diagrams focus on events, iUML-B statemachines focus on the state and the state transitions of the model. Therefore, choosing between the approaches depends on the nature of the problem, like in Chapter 6, we have shown how we used ERS to represent the dispatch system of the incident, but when it came to modelling the state changes of the resource, we used iUML-B statemachines, and both approaches complemented each other.

9.5 Other Work on Integration of Formal Methods and Graphical Representations

9.5.1 Tree-Like Structures Integrated with Formal Methods

Matoussi et al. (2011) describe a transformation of the KAOS goal model to the Event-B formal model. KAOS (Keep All Objectives Satisfied) is a goal based requirement engineering method, providing an overall representation of the system's goals, objects, agents, operations and behaviours (van Lamsweerde, 2008). Matoussi et al. focus on the goal KOAS model in their work, where each goal is translated into an event in the Event-B model, with actions describing the goal achievement and guards are the pre-conditions of the goal. Regarding refinement of the Event-B model, they follow the same approach of goal refinement where goals are refined into AND/OR constrained sub-goals. AND-refined means all the sub-goals need to be achieved in order to achieve the parent goal, whereas OR-refined means achieving one sub-goal is enough to achieve the parent goal (Darimont and van Lamsweerde, 1996). Regarding the transformation to Event-B, they describe the sequencing, the AND and the OR, which is described as mutually exclusive. There is no mention of the AND and OR generalisations into *some* and *all*, and unlike ERS the type, ordering and gluing invariants are not described and must be done manually by the modeller. Figure 9.8 shows a safe transportation example (Darimont and van Lamsweerde, 1996) using KAOS goal model. In this example AND-refinement of the sub-goals is used at the first level, while OR-refinement is applied in the refinement of "*No Collision*" goal at the second level.

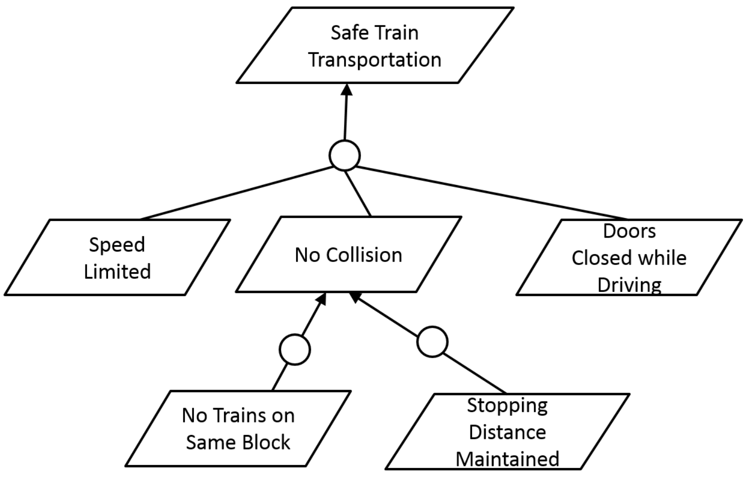

Safe Train Transportation
Speed Limited
No Collision
Doors Closed while Driving
No Trains on Same Block
Stopping Distance Maintained

Figure 9.8: Safe Transportation (van Lamsweerde, 2008): Example of KAOS Goal Model

Laleau et al. (2010) also use the KAOS goal model to reduce the gap between requirements and formal methods. They use the KAOS goal model to extend the SysML requirements model, and then map this model to the B method. Each goal is mapped to a B machine with an associated operation describing the goal, the refinement is then described similar to the sub-goals refinement AND/OR and milestone or sequence of sub-goals, in a way that each sub-goal is defined in a new B-machine with a corresponding operation, and the refinement is described by a new machine refining the abstract goal and including the sub-goals machines; the refinement behaviour (AND/OR/Sequence) is described within an operation in the refining B machine. This approach includes lots of machines in the B method and they also do not describe invariants related sequencing, typing and gluing like ERS, and there is no notion of multiple instances representations.

9.5.2 BPMN and Formal Methods

BPMN, which is adopted as the de-facto for business process modelling, has been extended with formal semantics using Event-B in Bryans and Wei (2010). In the translation of BPMN to Event-B, the interpretation of control flow is based on tokens, where a function is defined mapping each process instance to the number of tokens in that instance. Each flow is then guarded by the number of instances required to commence execution. The ERS approach uses subset relationships with events having corresponding control variables with the same names. The advantage of using subset relationships appears in the ability of defining control flow, not only using guards but also using invariants, ensuring that ordering is strongly specified and maintained by invariants. Moreover, the naming convention used by ERS, makes it easier to understand and track the ordering of events. Another key advantage of applying ERS is the explicit refinement relationships between events at different refinement levels

In Börger and Sörensen (2011), a formal semantics of BPMN is described using Abstract State Machines (ASM). Both previous works cover control and data flow of BPMN. In ERS data flow is left to be specified by Event-B. However, regarding the dynamic change of multiple instances, BPMN does not support the addition of new instances once a task has commenced execution, unlike the *par-replicator*, which gives the flexibility to support dynamic parameters. On the other hand, BPMN supports both structured and arbitrary cycles, whereas ERS only supports structured iteration and replication.

9.5.3 UML Activity Diagrams and Formal Methods

de Sousa et al. (2011) investigate the translation of IOD or interaction overview diagrams into Event-B, to describe the behavioural part of the model. IOD is a variant of UML activity diagrams, but with more restrictive syntax that does not allow the use of object nodes and object flows and instead of actions, IOD uses interactions. Regarding the static part, they extend the UML-B class diagram with BCE or boundary-control-entity class diagram. In their translation, they define a boolean variable for each control flow including decisions, they also define variables for each class and association in the class diagram. Then each interaction of the IOD is associated with an event, and each construct (join, fork, decision, merge) is associated with a special event. Compared to the ERS approach, the ERS combinators are not defined as separate events but their effect is shown by the sequencing invariants and guards in the events. They also associate a new boolean variable for each control flow edge, whereas ERS describes the control flow using the defined event control variables which can be easier to follow. As mentioned earlier, IOD does not allow the use of objects so there is no notion of multiple instances in their representation.

Refinement is described in de Sousa et al. (2012), but not in one type of diagrams like ERS. It is based on the ICONIX process that includes four types of UML diagrams (class, use cases, sequence and robustness which is a hybrid of the activity and class diagrams). They start with identifying the domain of the model which is mapped to sets and relations, then derive the use cases diagram which is mapped into events in the Event-B model. Then in the first refinement, they derive a robustness diagram from each use case which adds new events, invariants and new class associations to update the domain model. Lastly in the second refinement, they derive a sequence diagram from each robustness diagram updating the control, entity and boundary classes and the associations between them, however the translation of the sequence diagram into Event-B is not described yet. This process of refinement helps in understanding the system using different views of UML diagrams, but it does not describe exactly how events are refined and how they relate to abstract ones.

Ben Younes and Ben Ayed (2008) also describe the translation of UML activity diagram into Event-B. In their translation they describe refinement of Event-B machines in each nested activity, and they determine the ordering of events by a decreasing variant. Regarding their translation. They extend their work in Ben Younes et al. (2012) to describe the refinement patterns, where they apply a similar approach to goal refinement using sequence/AND/OR refinements. However, they do not describe how to handle new constructs of activity diagrams like expansion regions which is related to multiple instances modelling.

As mentioned in Section 2.3.1, version 2.0 of UML describes the semantics of activity diagrams based on token flow inspired by Petri nets. This resulted in many attempts to formalise the semantics of activity diagrams using Petri nets and its variants. However, most of these translations are based on earlier versions of UML activity diagram or only concentrated on the basic structures of UML 2.0 activity diagrams. Storrle (2004) is one of the few that concentrated on newly introduced structures of activity diagram like expansion regions. Storrle describes the semantics of structured nodes like loop nodes and conditional nodes as *syntactic sugar*, since they can be defined by other basic nodes. However, the situation is not that simple when it comes to expansion regions. So the author defines the formal semantics of expansion regions using procedural Petri nets. By this, he means translating the expansion regions as refined actions, where various calls to the same refined transitions are executed into its own space. The different modes of the expansion region implies different types of refinement nets.

Syriani and Ergin (2012) describe model transformation rules mapping constructs of UML 2.4.1 activity diagrams to Petri nets. In these transformation rules they describe the preconditions that must be found, and the negative application conditions (NACs) which are the precondition patterns that must not be found and prevent the application of the rules, and finally the postconditions which are the results of the transformation. These rules are divided into 11 phases to describe execution order. Each phase consists of a set of rules that can be applied at any order. An example of transformation rule, is the mapping of an action node into an entry transition, a processing transition and an exit transition. For expansion regions they introduce the concept of inhibitor arcs, that ensure all the inputs are processed before sending them to output. In this work, there is no notion of refinement.

9.5.4 Other UML Diagrams and Formal Methods

Idani et al. (2006) follow a reverse engineering approach where they derive a UML class diagram from B specifications, in order to help the customers understand the formal model and also help the modellers better understand the structure of their models. From each refinement of the B specifications, they derive a pertinent context where they ensure that each set and operation appears only once and then the contexts are automatically transformed into class diagrams, but this tool is still limited because it requires the interference of the modeller to select the sets. This approach helps understanding the formal model and provides good documentations of the formal model, but still cannot bridge the gap between requirements and formal methods, since it is applied after finishing the formal model.

Lausdahl et al. (2009) describe a mapping between UML and VDM++, which extends VDM, explained in Section 3.2.1, to support object oriented modelling and concurrency. They do two types of mappings, a bidirectional mapping between UML class diagrams and VDM++, and a one direction mapping from UML sequence diagrams to VDM++. The class diagrams describe the static part of the VDM++ model, whereas sequence diagrams describe the dynamics of the model. Here, they transform the sequence diagrams into trace definitions, in order to generate automatic test cases. While in our case, we use the ERS diagrams to build the dynamic part of the model.

Amálio et al. (2010) defines a framework for modelling using UML and Z. In this framework they use three types of UML diagrams, which are class, state and object diagrams that help later in generating and analysing a Z formal model. They start with a class diagram that describes the static part of the model. Then a UML state diagram to represent the dynamics of the model. These diagrammatic representations are then instantiated into templates (Amálio et al., 2007), which describe general structures and patterns of the formal language in this case Z, to generate the Z model. Finally they use object diagrams to validate the system against the requirements. They also apply object oriented structuring (Amálio et al., 2005, 2010) based on five different views, which are structural, intensional, extensional, relational and global, when generating the Z model from the instantiated templates. This framework helps in generating, analysing and validating models, but unlike ERS, it does not represent the refinement relations explicitly between the different operations, and do not describe explicitly how the different construct representations like *all*, *and* etc. are presented both diagrammatically and formally.

In Murali et al. (2015), they use UML use cases with Event-B. They define an accident use case which is an extension use case to model safety accidents which can result in an undesired loss. The accident use case is introduced through refinement describing the alternate flow if the accident is allowed to happen. This is similar to our approach where we introduce explicit exception handling combinators to explicitly define possible disruptions to the planned flow.

9.6 Overview of ERS in Comparison to Other Approaches

In this section we highlight the main strengths and weaknesses of the ERS approach in comparison to the previously discussed related work.

- One of the key strengths of the ERS approach is the support to multiple instance modelling through different combinators (*par*, *all*, *some*, *one*) and the *or-combinator* which is not supported by some of the workflow techniques. Moreover based on the evaluation with the control flow patterns, ERS supports most of these patterns either explicitly or implicitly when combined with additional Event-B guards, which is done using a reasonable number of combinators. However the ERS approach lacks the flexibility of modelling arbitrary jumps due to its structured nature.

- The models generated by ERS, are verified using the Event-B formal method, and we can also apply ProB for model checking and validation. This also applies to the models generated by iUML-B, however ERS provides a different viewpoint focusing on the events rather than the state changes.

- The diagrammatic nature of ERS showing the different refinement levels, also helps in making modelling decisions that facilitate the model verification such as the decision of when to introduce a data variable that can be modified by different events, in order to discharge *EQL* proof obligations generated during refinement.

- ERS rules, such as the *single solid line* rule, prompt doing smaller changes at each refinement level and hence facilitate the automation of discharging proof obligations.
- On the other hand, having lots of refinement levels can result in larger models and the hierarchical nature of ERS can result in additional events especially that the same event names cannot appear in different ERS branches.
- In addition to explicitly representing control flow, ERS structures the Event-B refinement and shows explicitly the relationship between events at different refinement levels.
- Most of the approaches that integrate formal modelling with graphical approaches aim at giving formal semantics to the graphical representations. ERS on the other hand extends the formal method Event-B to provide explicit support for control flow in Event-B and most importantly providing a systematic refinement approach.
- Control flow in ERS is not only expressed in terms of enabling conditions as most formalisms of workflow approaches do, but also as invariants that have to be maintained by the model.

9.7 Conclusion and Future Work

In this thesis we have presented a workflow modelling approach that combines formal modelling (Event-B) with a hierarchical graphical representation (ERS). This combination facilitates the incremental modelling of the workflows, providing explicit control flow representation which makes communication and validation easier. In addition to the advantages of formal modelling in verification.

Our approach extended the original ERS with combinators that facilitate dynamic modelling and support exception handling, thus expanding the usability of ERS to include more complex cases, as we have shown by the application to complex dispatch system of emergency services.

We have also presented a tool that generates Event-B models from ERS diagrams based on a well defined set of transformation rules. In the future we would like to improve our tool support by adding all the translation rules to the tool, providing more validations and trying to overcome some of the restrictions described in Section 9.2, if possible.

Another possible improvement that can be done to the tool is enhancing the graphical enviornment by allowing the user to add data in the ERS graphical editor without having to switch to the Event-B editor. We would like to focus on the improvement of the tool support which encourage users to apply and try our approach.

In addition to improving the tool support of our approach, our future work will include:

Application to Case Studies: Apply our approach to other industrial case studies, with special focus on exception handling and dynamic modelling.

Timing Patterns: In the travel agency case study, we have extended the ERS diagrams with the expiry timing pattern, we would also like to explore other timing patterns and how they can be used in conjunction with ERS diagrams.

Decomposition: The ERS approach helps in structuring the Event-B refinement which aims at tackling the modelling complexity. However having different refinement levels sometimes result in complex and large models and in some cases only parts of the model need to be refined, hence the need for model decomposition, which is another key concept in Event-B. Our approach transforms the ERS diagrams to Event-B models, therefore anything that applies to Event-B can be applied to the generated Event-B models. There are two approaches that support model decomposition in Event-B: shared variable (Abrial and Hallerstede, 2007; Abrial, 2009) and shared event (Butler, 2006, 2009a). Both approaches are supported by the Decomposition plug-in in the Rodin platform (Silva et al., 2010).

Applying model decomposition will generate new sub-models, although the new generated sub-models can be further refined independently using ERS, the original ERS diagrams will be lost and new ERS diagrams must be added to be able to apply the ERS approach in refinement. In the future we would like to integrate ERS with model decomposition by also allowing the ERS diagram decomposition based on the model decomposition approaches.

Such integration is feasible and in Butler (2009a) and Salehi Fathabadi et al. (2011), ERS and shared event decomposition were both applied, where ERS helped in preparing the model for decomposition. Model decomposition can benefit from the event visualisation provided by ERS when deciding how to split the events between the decomposed models.

For example the travel agency case study can be decomposed to three models: flight itinerary, hotel itinerary and car itinerary. Currently we can have three separate ERS diagrams for each itinerary and apply the shared event composition approach presented in Section 9.3.2, with the shared events *start itinerary* and *checkout*. However the three ERS diagrams will generate Event-B encodings in the same model, while decomposition will lead to three separate Event-B models.

Transformation Correctness: In Chapter 8 we have defined the translation rules which map the ERS elements to Event-B. The ERS translation rules are built using recursive functions to map the ERS operators and their possible combinations to Event-B. Currently the correctness of these transformations rely on the verification capabilities of Event-B, by trying to prove all the proof obligations generated by the Event-B models and validate the generated models using ProB. The difficulty in proving the correctness of the ERS transformations to Event-B is that the ERS operators are not compositional, like trace semantics for example, where in ERS, we need to traverse down the hierarchy to define the operators. As a future work, we would like to prove the correctness of the transformation rules generating Event-B from ERS.

One possibility is to define the semantics of both ERS and Event-B within one framework and prove the correspondence between the two semantic models of the source (ERS) and target (Event-B). A possible framework is the Prototype Verification System (PVS) (Owre et al., 1992), which provides a framework for formal specification and verification with a powerful theorem proving mechanisms. PVS is a good candidate where in Bodeveix et al. (1999), a formalisation of the B-method (an ancestor of Event-B) using PVS was done, and in Ripon and Butler (2009), the different semantics of compensating CSP (cCSP) and their equivalence were mechanised using PVS.

Another possibility that we can start with is to study the proof obligations generated by the different combinations of the ERS combinators, and provide proof tactics for the proofs that cannot be automatically proved.

Appendix A

Event-B Model of the Fire Dispatch Workflow

In this chapter we present the complete Event-B version of the fire dispatch system presented in Chapter 5. We also present all the ERS diagrams of the model, which are done by the improved version of the ERS plug-in. An interim update site of the ERS tool is available on `http://users.ecs.soton.ac.uk/dd4g12/`.

A.1 Abstract Level

A.1.1 Context C0

CONTEXT C0

SETS

    INCIDENT

END

A.1.2 ERS Diagram: M0

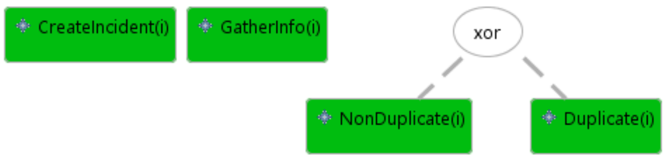


Figure A.1: ERS Diagram at Abstract Level: M0

A.1.3 Machine M0

MACHINE M0

SEES C0

VARIABLES

CreateIncident

GatherInfo

Duplicate

NonDuplicate

INVARIANTS

`inv_CreateIncident_type`: $CreateIncident \subseteq INCIDENT$

`inv_GatherInfo_fd0_seq`: $GatherInfo \subseteq CreateIncident$

`inv_Duplicate_fd0_seq`: $Duplicate \subseteq GatherInfo$

`inv_NonDuplicate_fd0_seq`: $NonDuplicate \subseteq GatherInfo$

`inv1_xorfd0`: $partition((Duplicate \cup NonDuplicate), Duplicate, NonDuplicate)$

EVENTS

Initialisation

begin

`act_CreateIncident`: $CreateIncident := \varnothing$

`act_GatherInfo`: $GatherInfo := \varnothing$

`act_Duplicate`: $Duplicate := \varnothing$

`act_NonDuplicate`: $NonDuplicate := \varnothing$

end

Event CreateIncident ⟨ordinary⟩ $\widehat{=}$

any

i

where

`grd_self`: $i \notin CreateIncident$

then

`act`: $CreateIncident := CreateIncident \cup \{i\}$

end

Event GatherInfo ⟨ordinary⟩ $\widehat{=}$

any

i

where

`grd_seq_fd0`: $i \in CreateIncident$

`grd_self`: $i \notin GatherInfo$

then

`act`: $GatherInfo := GatherInfo \cup \{i\}$

```
    end
Event Duplicate ⟨ordinary⟩ $\widehat{=}$
    any
        i
    where
        grd_seq_fd0:  $i \in GatherInfo$
        grd_self:  $i \notin Duplicate$
        grd1_xor:  $i \notin NonDuplicate$
    then
        act: $Duplicate := Duplicate \cup \{i\}$
    end
Event NonDuplicate ⟨ordinary⟩ $\widehat{=}$
    any
        i
    where
        grd_seq_fd0:  $i \in GatherInfo$
        grd_self:  $i \notin NonDuplicate$
        grd1_xor:  $i \notin Duplicate$
    then
        act: $NonDuplicate := NonDuplicate \cup \{i\}$
    end
END
```

A.2 First Refinement

A.2.1 ERS Diagrams: M1

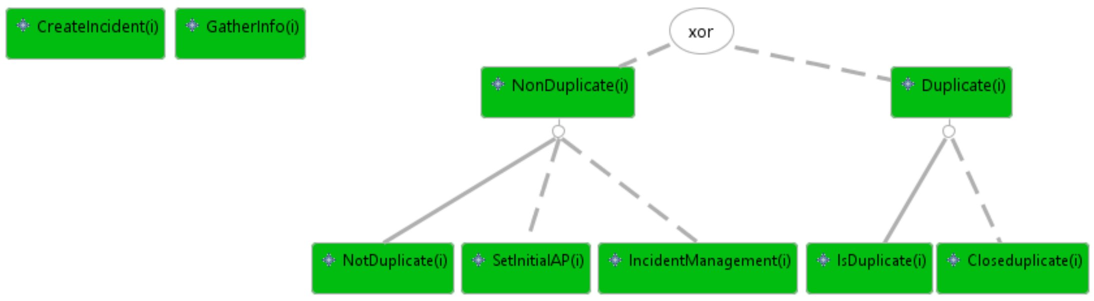


Figure A.2: ERS Diagram at First Refinement: M1

A.2.2 Machine M1

MACHINE M1

REFINES M0

SEES C0

VARIABLES

CreateIncident

GatherInfo

IsDuplicate

CloseDuplicate

NotDuplicate

SetInitialAP

IncidentManagement

INVARIANTS

`inv_CreateIncident_type`: $CreateIncident \subseteq INCIDENT$

`inv_GatherInfo_fd0_seq`: $GatherInfo \subseteq CreateIncident$

`inv_IsDuplicate_Duplicate_seq`: $IsDuplicate \subseteq GatherInfo$

`inv_IsDuplicate_glu`: $IsDuplicate = Duplicate$

`inv_CloseDuplicate_Duplicate_seq`: $CloseDuplicate \subseteq IsDuplicate$

`inv_NotDuplicate_NonDuplicate_seq`: $NotDuplicate \subseteq GatherInfo$

`inv_NotDuplicate_glu`: $NotDuplicate = NonDuplicate$

`inv_SetInitialAP_NonDuplicate_seq`: $SetInitialAP \subseteq NotDuplicate$

`inv_IncidentManagement_NonDuplicate_seq`: $IncidentManagement \subseteq SetInitialAP$

`inv1_xorfd0`: $partition((IsDuplicate \cup NotDuplicate), IsDuplicate, NotDuplicate)$

EVENTS

Initialisation

begin

`act_CreateIncident`: $CreateIncident := \varnothing$

`act_GatherInfo`: $GatherInfo := \varnothing$

`act_IsDuplicate`: $IsDuplicate := \varnothing$

`act_CloseDuplicate`: $CloseDuplicate := \varnothing$

`act_NotDuplicate`: $NotDuplicate := \varnothing$

`act_SetInitialAP`: $SetInitialAP := \varnothing$

`act_IncidentManagement`: $IncidentManagement := \varnothing$

end

Event CloseDuplicate ⟨ordinary⟩ $\widehat{=}$

any

i

where

`grd_seq_Duplicate`: $i \in IsDuplicate$

```
            grd_self:  $i \notin CloseDuplicate$
      then
            act: $CloseDuplicate := CloseDuplicate \cup \{i\}$
      end
Event SetInitialAP ⟨ordinary⟩ $\widehat{=}$
      any
            i
      where
            grd_seq_NonDuplicate:  $i \in NotDuplicate$
            grd_self:  $i \notin SetInitialAP$
      then
            act: $SetInitialAP := SetInitialAP \cup \{i\}$
      end
Event IncidentManagement ⟨ordinary⟩ $\widehat{=}$
      any
            i
      where
            grd_seq_NonDuplicate:  $i \in SetInitialAP$
            grd_self:  $i \notin IncidentManagement$
      then
            act: $IncidentManagement := IncidentManagement \cup \{i\}$
      end
Event CreateIncident ⟨ordinary⟩ $\widehat{=}$
refines CreateIncident
      any
            i
      where
            grd_self:  $i \notin CreateIncident$
      then
            act: $CreateIncident := CreateIncident \cup \{i\}$
      end
Event GatherInfo ⟨ordinary⟩ $\widehat{=}$
refines GatherInfo
      any
            i
      where
            grd_seq_fd0:  $i \in CreateIncident$
            grd_self:  $i \notin GatherInfo$
      then
            act: $GatherInfo := GatherInfo \cup \{i\}$
```

```
    end
Event IsDuplicate ⟨ordinary⟩ ≙
refines Duplicate
    any
        i
    where
        grd_seq_Duplicate: $i \in GatherInfo$
        grd_self: $i \notin IsDuplicate$
        grd1_xor: $i \notin NotDuplicate$
    then
        act: $IsDuplicate := IsDuplicate \cup \{i\}$
    end
Event NotDuplicate ⟨ordinary⟩ ≙
refines NonDuplicate
    any
        i
    where
        grd_seq_NonDuplicate: $i \in GatherInfo$
        grd_self: $i \notin NotDuplicate$
        grd1_xor: $i \notin IsDuplicate$
    then
        act: $NotDuplicate := NotDuplicate \cup \{i\}$
    end
END
```

A.3 Second Refinement

A.3.1 ERS Diagram: M2

A.3.2 Machine M2

```
MACHINE M2
REFINES M1
SEES C0
VARIABLES
    CreateIncident
    GatherInfo
    IsDuplicate
    CloseDuplicate
```

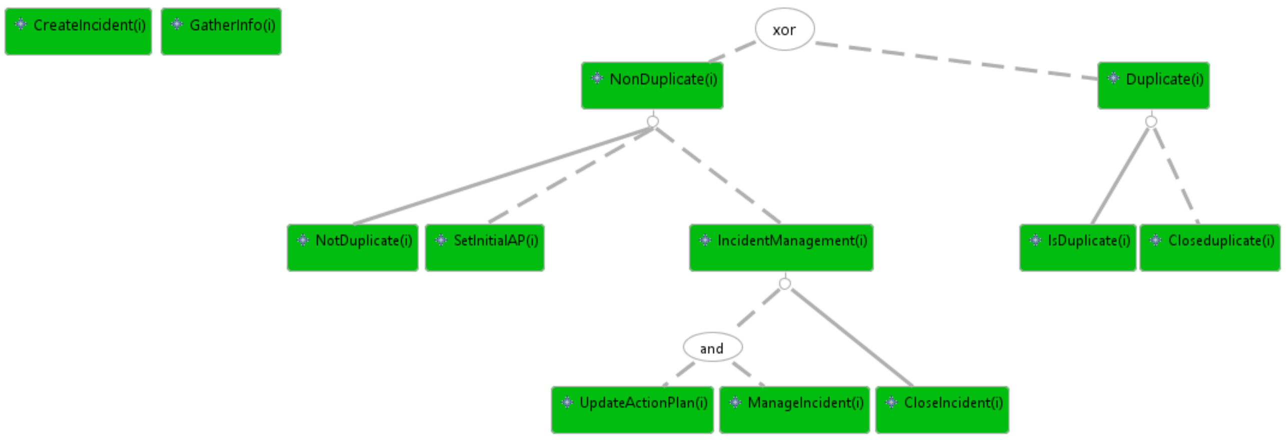


Figure A.3

NotDuplicate

SetInitialAP

UpdateActionPlan

ManageIncident

CloseIncident

INVARIANTS

`inv_CreateIncident_type`: $CreateIncident \subseteq INCIDENT$

`inv_GatherInfo_fd0_seq`: $GatherInfo \subseteq CreateIncident$

`inv_IsDuplicate_Duplicate_seq`: $IsDuplicate \subseteq GatherInfo$

`inv_CloseDuplicate_Duplicate_seq`: $CloseDuplicate \subseteq IsDuplicate$

`inv_NotDuplicate_NonDuplicate_seq`: $NotDuplicate \subseteq GatherInfo$

`inv_SetInitialAP_NonDuplicate_seq`: $SetInitialAP \subseteq NotDuplicate$

`inv_UpdateActionPlan__seq`: $UpdateActionPlan \subseteq SetInitialAP$

`inv_ManageIncident__seq`: $ManageIncident \subseteq SetInitialAP$

`inv_CloseIncident__seq`: $CloseIncident \subseteq (UpdateActionPlan \cap ManageIncident)$

`inv_CloseIncident_glu`: $CloseIncident = IncidentManagement$

`inv1_xorfd0`: $partition((IsDuplicate \cup NotDuplicate), IsDuplicate, NotDuplicate)$

EVENTS

Initialisation

begin

`act_CreateIncident`: $CreateIncident := \varnothing$

`act_GatherInfo`: $GatherInfo := \varnothing$

`act_IsDuplicate`: $IsDuplicate := \varnothing$

`act_CloseDuplicate`: $CloseDuplicate := \varnothing$

`act_NotDuplicate`: $NotDuplicate := \varnothing$

`act_SetInitialAP`: $SetInitialAP := \varnothing$

```
        act_UpdateActionPlan: $UpdateActionPlan := \varnothing$
        act_ManageIncident: $ManageIncident := \varnothing$
        act_CloseIncident: $CloseIncident := \varnothing$
    end
Event UpdateActionPlan ⟨ordinary⟩ $\widehat{=}$
    any
        i
    where
        grd_seq_:  $i \in SetInitialAP$
        grd_self:  $i \notin UpdateActionPlan$
    then
        act: $UpdateActionPlan := UpdateActionPlan \cup \{i\}$
    end
Event ManageIncident ⟨ordinary⟩ $\widehat{=}$
    any
        i
    where
        grd_seq_:  $i \in SetInitialAP$
        grd_self:  $i \notin ManageIncident$
    then
        act: $ManageIncident := ManageIncident \cup \{i\}$
    end
Event CreateIncident ⟨ordinary⟩ $\widehat{=}$
refines CreateIncident
    any
        i
    where
        grd_self:  $i \notin CreateIncident$
    then
        act: $CreateIncident := CreateIncident \cup \{i\}$
    end
Event GatherInfo ⟨ordinary⟩ $\widehat{=}$
refines GatherInfo
    any
        i
    where
        grd_seq_fd0:  $i \in CreateIncident$
        grd_self:  $i \notin GatherInfo$
    then
        act: $GatherInfo := GatherInfo \cup \{i\}$
    end
Event IsDuplicate ⟨ordinary⟩ $\widehat{=}$
refines IsDuplicate
    any
        i
    where
        grd_seq_Duplicate:  $i \in GatherInfo$
        grd_self:  $i \notin IsDuplicate$
        grd1_xor:  $i \notin NotDuplicate$
    then
        act: $IsDuplicate := IsDuplicate \cup \{i\}$
    end
Event CloseDuplicate ⟨ordinary⟩ $\widehat{=}$
refines CloseDuplicate
    any
        i
    where
        grd_seq_Duplicate:  $i \in IsDuplicate$
        grd_self:  $i \notin CloseDuplicate$
    then
        act: $CloseDuplicate := CloseDuplicate \cup \{i\}$
    end
Event NotDuplicate ⟨ordinary⟩ $\widehat{=}$
refines NotDuplicate
    any
        i
    where
        grd_seq_NonDuplicate:  $i \in GatherInfo$
        grd_self:  $i \notin NotDuplicate$
        grd1_xor:  $i \notin IsDuplicate$
    then
        act: $NotDuplicate := NotDuplicate \cup \{i\}$
    end
Event SetInitialAP ⟨ordinary⟩ $\widehat{=}$
refines SetInitialAP
    any
        i
    where
        grd_seq_NonDuplicate:  $i \in NotDuplicate$
        grd_self:  $i \notin SetInitialAP$
    then
        act: $SetInitialAP := SetInitialAP \cup \{i\}$
    end
Event CloseIncident ⟨ordinary⟩ ≙
refines IncidentManagement
    any
        i
    where
        grd_seq_:  $(i \in UpdateActionPlan \land i \in ManageIncident)$
        grd_self:  $i \notin CloseIncident$
    then
        act: $CloseIncident := CloseIncident \cup \{i\}$
    end
END
```

A.4 Third Refinement

A.4.1 Context C1

```
CONTEXT C1
EXTENDS C0
SETS
      LR
END
```

A.4.2 ERS Diagram: M3

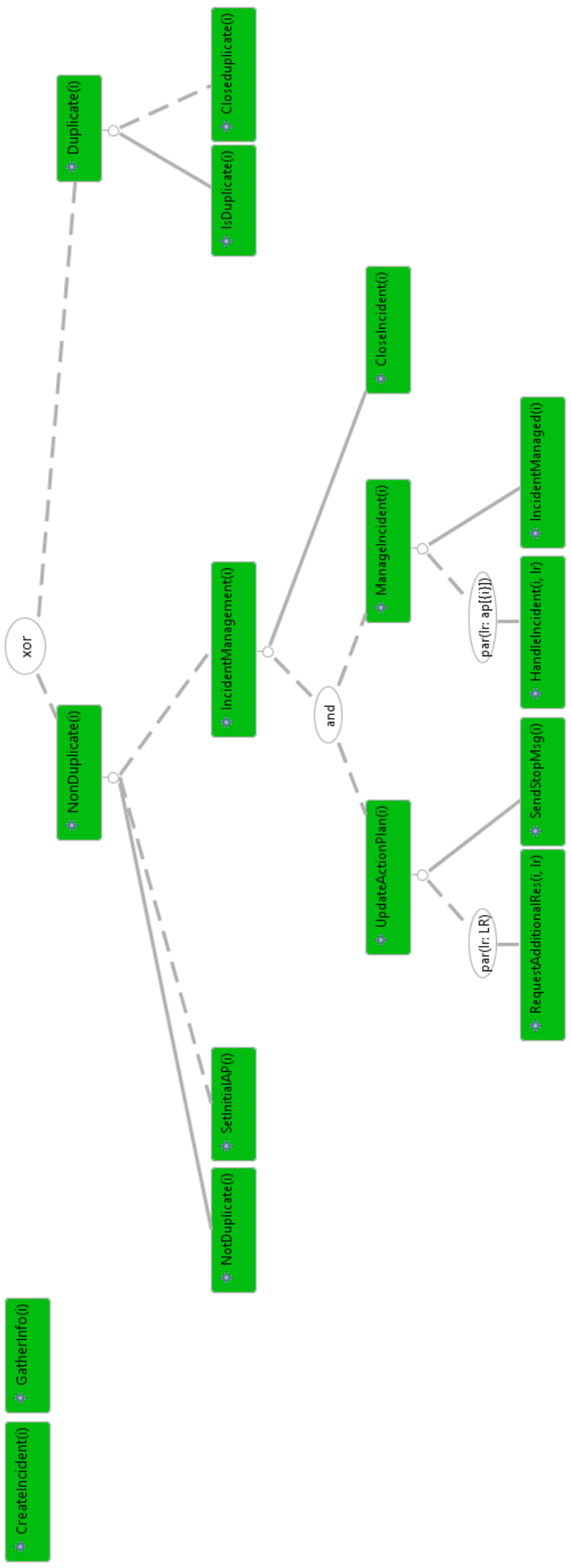


Figure A.4: ERS Diagram at Third Refinement: M3

A.4.3 Machine M3

MACHINE M3

REFINES M2

SEES C1

VARIABLES

CreateIncident

GatherInfo

IsDuplicate

CloseDuplicate

NotDuplicate

SetInitialAP

RequestAdditionalRes

SendStopMsg

HandleIncident

IncidentManaged

CloseIncident

ap added manually

notRequired_ap added manually

INVARIANTS

`inv_CreateIncident_type`: $CreateIncident \subseteq INCIDENT$

`inv_RequestAdditionalRes_type`: $RequestAdditionalRes \subseteq INCIDENT \times LR$

`inv_HandleIncident_type`: $HandleIncident \subseteq INCIDENT \times LR$

`inv_GatherInfo_fd0_seq`: $GatherInfo \subseteq CreateIncident$

`inv_IsDuplicate_Duplicate_seq`: $IsDuplicate \subseteq GatherInfo$

`inv_CloseDuplicate_Duplicate_seq`: $CloseDuplicate \subseteq IsDuplicate$

`inv_NotDuplicate_NonDuplicate_seq`: $NotDuplicate \subseteq GatherInfo$

`inv_SetInitialAP_NonDuplicate_seq`: $SetInitialAP \subseteq NotDuplicate$

`inv_RequestAdditionalRes__seq`: $dom(RequestAdditionalRes) \subseteq SetInitialAP$

`inv_SendStopMsg__seq`: $SendStopMsg \subseteq SetInitialAP$

`inv_SendStopMsg_glu`: $SendStopMsg = UpdateActionPlan$

`inv_HandleIncident__seq`: $dom(HandleIncident) \subseteq SetInitialAP$

`inv_IncidentManaged__seq`: $IncidentManaged \subseteq SetInitialAP$

`inv_IncidentManaged_glu`: $IncidentManaged = ManageIncident$

`inv_CloseIncident__seq`: $CloseIncident \subseteq (SendStopMsg \cap IncidentManaged)$

`inv1_xorfd0`: $partition((IsDuplicate \cup NotDuplicate), IsDuplicate, NotDuplicate)$

`inv1`: $ap \in INCIDENT \leftrightarrow LR$

`inv2`: $dom(ap) = SetInitialAP$

in_Handle_Rep: $\forall i \cdot i \in INCIDENT \Rightarrow HandleIncident[\{i\}] \subseteq ap[\{i\}]$

inv_ReuestAdditionalRes_rep: $\forall i \cdot i \in INCIDENT \Rightarrow RequestAdditionalRes[\{i\}] \subseteq LR$

inv_notRequired_ap_type: $notRequired\_ap \subseteq ap$

manually added data variable to describe the actions plan items that are not required after stop message is sent

EVENTS

Initialisation

begin

act_CreateIncident: $CreateIncident := \varnothing$

act_GatherInfo: $GatherInfo := \varnothing$

act_IsDuplicate: $IsDuplicate := \varnothing$

act_CloseDuplicate: $CloseDuplicate := \varnothing$

act_NotDuplicate: $NotDuplicate := \varnothing$

act_SetInitialAP: $SetInitialAP := \varnothing$

act_RequestAdditionalRes: $RequestAdditionalRes := \varnothing$

act_SendStopMsg: $SendStopMsg := \varnothing$

act_HandleIncident: $HandleIncident := \varnothing$

act_IncidentManaged: $IncidentManaged := \varnothing$

act_CloseIncident: $CloseIncident := \varnothing$

act1: $ap := \varnothing$

act_notRequired_ap: $notRequired\_ap := \varnothing$

end

Event RequestAdditionalRes ⟨ordinary⟩ $\widehat{=}$

any

i

lr

where

grd_seq_: $i \in SetInitialAP$

grd_self: $i \mapsto lr \notin RequestAdditionalRes$

grd_InputExpression1: $lr \in LR$

grd0_par: $i \notin SendStopMsg$

grd1: $lr \notin ap[\{i\}]$

then

act: $RequestAdditionalRes := RequestAdditionalRes \cup \{i \mapsto lr\}$

act1: $ap := ap \cup \{i \mapsto lr\}$

//manually added

end

Event HandleIncident ⟨ordinary⟩ $\widehat{=}$

any

```
        i
        lr
    where
        grd_self:  $i \mapsto lr \notin HandleIncident$
        grd_InputExpression1:  $lr \in ap[\{i\}]$
        grd0_par:  $i \notin IncidentManaged$
        grd_seq_:  $i \in SetInitialAP$
    then
        act: $HandleIncident := HandleIncident \cup \{i \mapsto lr\}$
    end
Event CreateIncident ⟨ordinary⟩ $\widehat{=}$
refines CreateIncident
    any
        i
    where
        grd_self:  $i \notin CreateIncident$
    then
        act: $CreateIncident := CreateIncident \cup \{i\}$
    end
Event GatherInfo ⟨ordinary⟩ $\widehat{=}$
refines GatherInfo
    any
        i
    where
        grd_self:  $i \notin GatherInfo$
        grd_seq_fd0:  $i \in CreateIncident$
    then
        act: $GatherInfo := GatherInfo \cup \{i\}$
    end
Event IsDuplicate ⟨ordinary⟩ $\widehat{=}$
refines IsDuplicate
    any
        i
    where
        grd_self:  $i \notin IsDuplicate$
        grd1_xor:  $i \notin NotDuplicate$
        grd_seq_Duplicate:  $i \in GatherInfo$
    then
        act: $IsDuplicate := IsDuplicate \cup \{i\}$
    end
```

```
Event CloseDuplicate ⟨ordinary⟩ $\widehat{=}$
refines CloseDuplicate
    any
        i
    where
        grd_self:  $i \notin CloseDuplicate$
        grd_seq_Duplicate:  $i \in IsDuplicate$
    then
        act: $CloseDuplicate := CloseDuplicate \cup \{i\}$
    end
Event NotDuplicate ⟨ordinary⟩ $\widehat{=}$
refines NotDuplicate
    any
        i
    where
        grd_self:  $i \notin NotDuplicate$
        grd1_xor:  $i \notin IsDuplicate$
        grd_seq_NonDuplicate:  $i \in GatherInfo$
    then
        act: $NotDuplicate := NotDuplicate \cup \{i\}$
    end
Event SetInitialAP ⟨ordinary⟩ $\widehat{=}$
refines SetInitialAP
    any
        i
        lrs
    where
        grd_self:  $i \notin SetInitialAP$
        grd1:  $lrs \subseteq LR$
        grd2:  $lrs \neq \varnothing$
        grd_seq_NonDuplicate:  $i \in NotDuplicate$
    then
        act: $SetInitialAP := SetInitialAP \cup \{i\}$
        act1: $ap := ap \cup (\{i\} \times lrs)$
            manually
    end
Event SendStopMsg ⟨ordinary⟩ $\widehat{=}$
refines UpdateActionPlan
    any
        i
```

```
            lrs
      where
            grd_self:  $i \notin SendStopMsg$
            grd1:  $lrs \subseteq ap[\{i\}]$
               //added manually
            grd_seq_:  $i \in SetInitialAP$
      then
            act: $SendStopMsg := SendStopMsg \cup \{i\}$
            act1: $notRequired\_ap := notRequired\_ap \cup (\{i\} \times lrs)$
               added manually
      end
Event IncidentManaged ⟨ordinary⟩ $\widehat{=}$
refines ManageIncident
      any
            i
      where
            grd_self:  $i \notin IncidentManaged$
            grd_seq_:  $i \in SetInitialAP$
      then
            act: $IncidentManaged := IncidentManaged \cup \{i\}$
      end
Event CloseIncident ⟨ordinary⟩ $\widehat{=}$
refines CloseIncident
      any
            i
      where
            grd_seq_:  $(i \in SendStopMsg \land i \in IncidentManaged)$
            grd_self:  $i \notin CloseIncident$
      then
            act: $CloseIncident := CloseIncident \cup \{i\}$
      end
END
```

A.5 Fourth Refinement

A.5.1 Context C2

```
CONTEXT C2
EXTENDS C1
SETS
```

INCIDENT_TYPE

LOCATION

LOCATION_TYPE

CONSTANTS

PDA set the action plan according to the type and location

AXIOMS

axm1: $PDA \in (INCIDENT\_TYPE \times LOCATION\_TYPE) \rightarrow \mathbb{P}(LR)$

axm2: $\forall it, loc \cdot it \in INCIDENT\_TYPE \land loc \in LOCATION\_TYPE \Rightarrow PDA(it \mapsto loc) \neq \varnothing$

END

A.5.2 ERS Diagram: M4

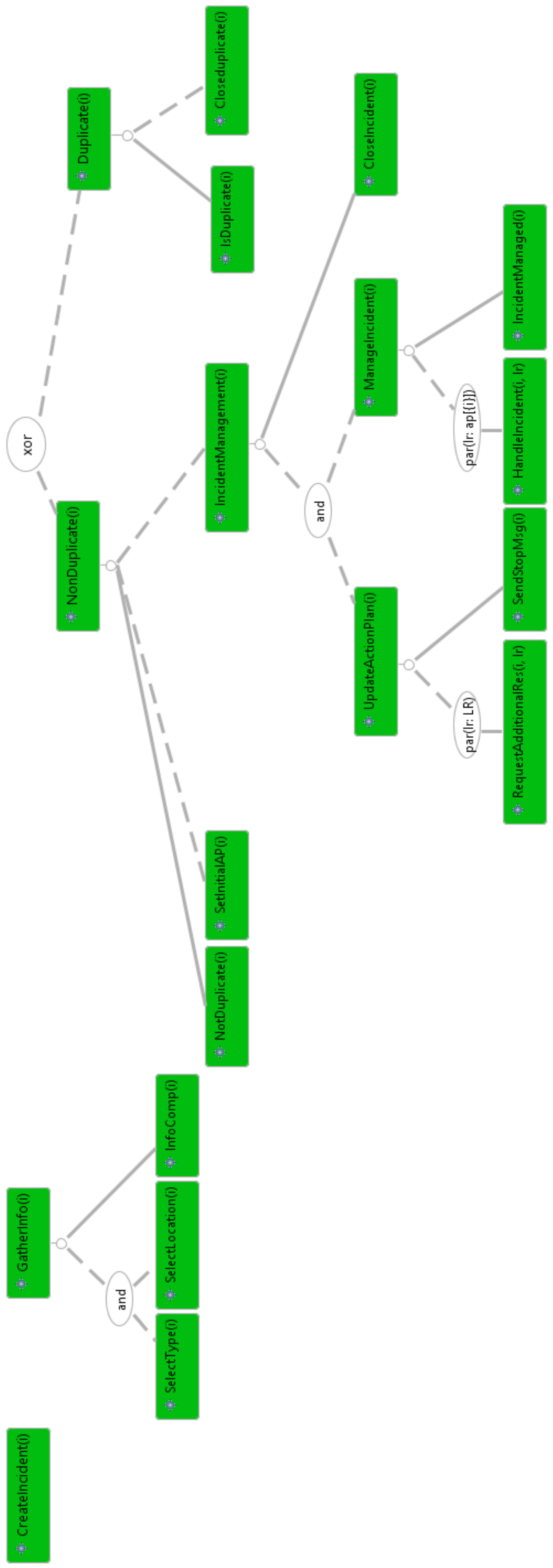


Figure A.5: ERS Diagram at Fourth Refinement: M4

A.5.3 Machine M4

MACHINE M4

REFINES M3

SEES C2

VARIABLES

CreateIncident

IsDuplicate

CloseDuplicate

NotDuplicate

SetInitialAP

SelectType

SelectLocation

InfoComp

RequestAdditionalRes

SendStopMsg

HandleIncident

IncidentManaged

CloseIncident

iType added manually

ap added manually

iLocation added manually

iLocType added manually

notRequired_ap added manually

INVARIANTS

`inv_CreateIncident_type`: $CreateIncident \subseteq INCIDENT$

`inv_RequestAdditionalRes_type`: $RequestAdditionalRes \subseteq INCIDENT \times LR$

`inv_HandleIncident_type`: $HandleIncident \subseteq INCIDENT \times LR$

`inv_SelectType__seq`: $SelectType \subseteq CreateIncident$

`inv_SelectLocation__seq`: $SelectLocation \subseteq CreateIncident$

`inv_InfoComp__seq`: $InfoComp \subseteq (SelectType \cap SelectLocation)$

`inv_InfoComp_glu`: $InfoComp = GatherInfo$

`inv_IsDuplicate_Duplicate_seq`: $IsDuplicate \subseteq InfoComp$

`inv_CloseDuplicate_Duplicate_seq`: $CloseDuplicate \subseteq IsDuplicate$

`inv_NotDuplicate_NonDuplicate_seq`: $NotDuplicate \subseteq InfoComp$

`inv_SetInitialAP_NonDuplicate_seq`: $SetInitialAP \subseteq NotDuplicate$

`inv_RequestAdditionalRes__seq`: $dom(RequestAdditionalRes) \subseteq SetInitialAP$

`inv_SendStopMsg__seq`: $SendStopMsg \subseteq SetInitialAP$

inv_HandleIncident__seq: $dom(HandleIncident) \subseteq SetInitialAP$

inv_IncidentManaged__seq: $IncidentManaged \subseteq SetInitialAP$

inv_CloseIncident__seq: $CloseIncident \subseteq (SendStopMsg \cap IncidentManaged)$

inv1_xorfd0: $partition((IsDuplicate \cup NotDuplicate), IsDuplicate, NotDuplicate)$

inv1: $iType \in SelectType \rightarrow INCIDENT\_TYPE$

added manually

inv2: $iLocation \in SelectLocation \rightarrow LOCATION$

added manually

inv_loc_type: $iLocType \in SelectLocation \rightarrow LOCATION\_TYPE$

data variable

inv3: $\forall i \cdot i \in IsDuplicate \Rightarrow (\exists i0 \cdot i0 \in NotDuplicate \wedge iLocation(i) = iLocation(i0))$

added manually

in_Handle_Rep: $\forall i \cdot i \in INCIDENT \Rightarrow HandleIncident[\{i\}] \subseteq ap[\{i\}]$

inv_ReuestAdditionalRes_rep: $\forall i \cdot i \in INCIDENT \Rightarrow RequestAdditionalRes[\{i\}] \subseteq LR$

EVENTS

Initialisation

begin

act_CreateIncident: $CreateIncident := \varnothing$

act_SelectType: $SelectType := \varnothing$

act_SelectLocation: $SelectLocation := \varnothing$

act_InfoComp: $InfoComp := \varnothing$

act_IsDuplicate: $IsDuplicate := \varnothing$

act_CloseDuplicate: $CloseDuplicate := \varnothing$

act_NotDuplicate: $NotDuplicate := \varnothing$

act_SetInitialAP: $SetInitialAP := \varnothing$

act_RequestAdditionalRes: $RequestAdditionalRes := \varnothing$

act_SendStopMsg: $SendStopMsg := \varnothing$

act_HandleIncident: $HandleIncident := \varnothing$

act_IncidentManaged: $IncidentManaged := \varnothing$

act_CloseIncident: $CloseIncident := \varnothing$

act1: $ap := \varnothing$

act2: $iLocation := \varnothing$

act_iLocType: $iLocType := \varnothing$

act3: $iType := \varnothing$

act_notRequired_ap: $notRequired\_ap := \varnothing$

end

Event SelectType ⟨ordinary⟩ $\widehat{=}$

any

i

```
        t
    where
        grd_seq_:   $i \in CreateIncident$
        grd_self:   $i \notin SelectType$
        grd1:   $t \in INCIDENT\_TYPE$
    then
        act: $SelectType := SelectType \cup \{i\}$
        act1: $iType(i) := t$
    end
Event SelectLocation ⟨ordinary⟩ ≙
    any
        i
        l
        lt
    where
        grd_seq_:   $i \in CreateIncident$
        grd_self:   $i \notin SelectLocation$
        grd1:   $l \in LOCATION$
        grd2:   $lt \in LOCATION\_TYPE$
    then
        act: $SelectLocation := SelectLocation \cup \{i\}$
        act1: $iLocation(i) := l$
        act2: $iLocType(i) := lt$
    end
Event CreateIncident ⟨ordinary⟩ ≙
refines CreateIncident
    any
        i
    where
        grd_self:   $i \notin CreateIncident$
    then
        act: $CreateIncident := CreateIncident \cup \{i\}$
    end
Event InfoComp ⟨ordinary⟩ ≙
refines GatherInfo
    any
        i
    where
        grd_seq_:   $(i \in SelectType \land i \in SelectLocation)$
        grd_self:   $i \notin InfoComp$
    then
```

```
        act: $InfoComp := InfoComp \cup \{i\}$
    end
Event IsDuplicate ⟨ordinary⟩ $\widehat{=}$
refines IsDuplicate
    any
        i
        i0
    where
        grd_seq_Duplicate:  $i \in InfoComp$
        grd_self:  $i \notin IsDuplicate$
        grd1_xor:  $i \notin NotDuplicate$
        grd_i0:  $i0 \in NotDuplicate \setminus CloseIncident$
            added manually
        grd_i0i:  $iLocation(i) = iLocation(i0)$
    then
        act: $IsDuplicate := IsDuplicate \cup \{i\}$
    end
Event CloseDuplicate ⟨ordinary⟩ $\widehat{=}$
refines CloseDuplicate
    any
        i
    where
        grd_seq_Duplicate:  $i \in IsDuplicate$
        grd_self:  $i \notin CloseDuplicate$
    then
        act: $CloseDuplicate := CloseDuplicate \cup \{i\}$
    end
Event NotDuplicate ⟨ordinary⟩ $\widehat{=}$
refines NotDuplicate
    any
        i
    where
        grd_seq_NonDuplicate:  $i \in InfoComp$
        grd_self:  $i \notin NotDuplicate$
        grd1_xor:  $i \notin IsDuplicate$
    then
        act: $NotDuplicate := NotDuplicate \cup \{i\}$
    end
Event SetInitialAP ⟨ordinary⟩ $\widehat{=}$
refines SetInitialAP
```

```
    any
        lrs
        i
        l
        t
    where
        grd_seq_NonDuplicate:  $i \in NotDuplicate$
        grd_self:  $i \notin SetInitialAP$
        grd1:  $t = iType(i)$
        grd2:  $l = iLocType(i)$
        grd3:  $lrs = PDA(t \mapsto l)$
    then
        act: $SetInitialAP := SetInitialAP \cup \{i\}$
        act1: $ap := ap \cup (\{i\} \times lrs)$
            manually
    end
Event RequestAdditionalRes ⟨ordinary⟩ $\widehat{=}$
refines RequestAdditionalRes
    any
        i
        lr
    where
        grd_seq_:  $i \in SetInitialAP$
        grd_self:  $i \mapsto lr \notin RequestAdditionalRes$
        grd_InputExpression1:  $lr \in (LR \setminus ap[\{i\}])$
        grd0_par:  $i \notin SendStopMsg$
    then
        act: $RequestAdditionalRes := RequestAdditionalRes \cup \{i \mapsto lr\}$
        act1: $ap := ap \cup \{i \mapsto lr\}$
            //manually added
    end
Event SendStopMsg ⟨ordinary⟩ $\widehat{=}$
refines SendStopMsg
    any
        i
        lrs
    where
        grd1:  $lrs \subseteq ap[\{i\}]$
            //added manually
        grd_seq_:  $i \in SetInitialAP$
        grd_self:  $i \notin SendStopMsg$
    then
        act: $SendStopMsg := SendStopMsg \cup \{i\}$
        act1: $notRequired\_ap := notRequired\_ap \cup (\{i\} \times lrs)$
            added manually
    end
Event HandleIncident ⟨ordinary⟩ $\widehat{=}$
refines HandleIncident
    any
        i
        lr
    where
        grd0_par: $i \notin IncidentManaged$
        grd_seq_: $i \in SetInitialAP$
        grd_self: $i \mapsto lr \notin HandleIncident$
        grd_InputExpression1: $lr \in ap[\{i\}]$
    then
        act: $HandleIncident := HandleIncident \cup \{i \mapsto lr\}$
    end
Event IncidentManaged ⟨ordinary⟩ $\widehat{=}$
refines IncidentManaged
    any
        i
    where
        grd_self: $i \notin IncidentManaged$
        grd_seq_: $i \in SetInitialAP$
    then
        act: $IncidentManaged := IncidentManaged \cup \{i\}$
    end
Event CloseIncident ⟨ordinary⟩ $\widehat{=}$
refines CloseIncident
    any
        i
    where
        grd_seq_: $(i \in SendStopMsg \land i \in IncidentManaged)$
        grd_self: $i \notin CloseIncident$
    then
        act: $CloseIncident := CloseIncident \cup \{i\}$
    end
END
```

A.6 Fifth Refinement

A.6.1 ERS Diagram: M5

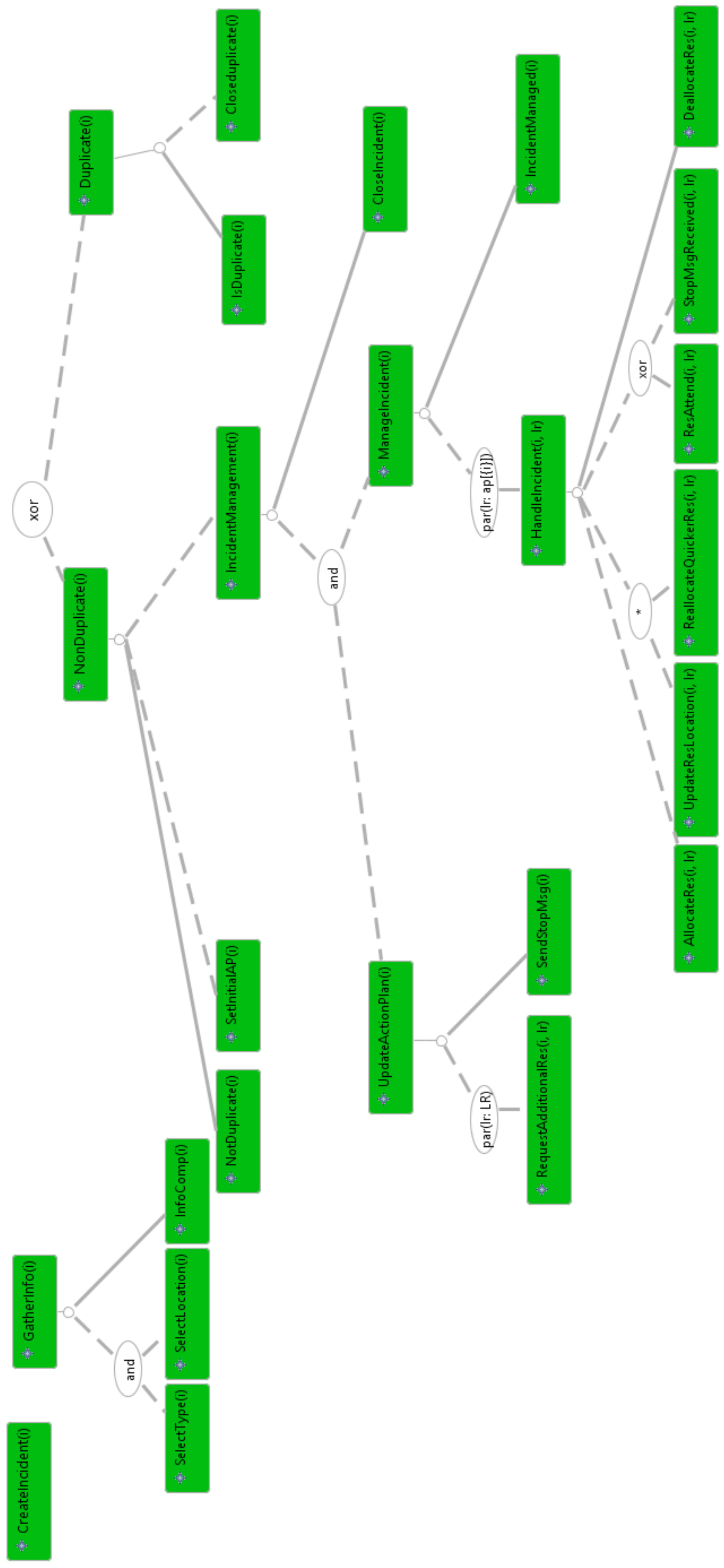


Figure A.6: ERS Diagram at Fifth Refinement Level: M5

A.6.2 Machine M5

MACHINE M5

REFINES M4

SEES C2

VARIABLES

CreateIncident

IsDuplicate

CloseDuplicate

NotDuplicate

SetInitialAP

SelectType

SelectLocation

InfoComp

RequestAdditionalRes

IncidentManaged

CloseIncident

SendStopMsg

AllocateRes

ResAttend

DeallocateRes

iType added manually

ap added manually

iLocation added manually

iLocType added manually

StopMsgReceived

notRequired_ap

INVARIANTS

`inv_CreateIncident_type`: $CreateIncident \subseteq INCIDENT$

`inv_RequestAdditionalRes_type`: $RequestAdditionalRes \subseteq INCIDENT \times LR$

`inv_SelectType__seq`: $SelectType \subseteq CreateIncident$

`inv_SelectLocation__seq`: $SelectLocation \subseteq CreateIncident$

`inv_InfoComp__seq`: $InfoComp \subseteq (SelectType \cap SelectLocation)$

`inv_IsDuplicate_Duplicate_seq`: $IsDuplicate \subseteq InfoComp$

`inv_CloseDuplicate_Duplicate_seq`: $CloseDuplicate \subseteq IsDuplicate$

`inv_NotDuplicate_NonDuplicate_seq`: $NotDuplicate \subseteq InfoComp$

`inv_SetInitialAP_NonDuplicate_seq`: $SetInitialAP \subseteq NotDuplicate$

`inv_RequestAdditionalRes__seq`: $dom(RequestAdditionalRes) \subseteq SetInitialAP$

inv_SendStopMsg__seq: $SendStopMsg \subseteq SetInitialAP$

inv_Allocate_Res_type: $AllocateRes \subseteq INCIDENT \times LR$

inv_AllocateRes__seq: $dom(AllocateRes) \subseteq SetInitialAP$

inv_ResAttend__seq: $ResAttend \subseteq AllocateRes$

inv_StopMsgReceived_seq: $StopMsgReceived \subseteq AllocateRes$

inv2_xorFD: $partition((ResAttend \cup StopMsgReceived), ResAttend, StopMsgReceived)$

inv_DeallocateRes__seq: $DeallocateRes \subseteq (ResAttend \cup StopMsgReceived)$

inv_DeallocateRes_glu: $DeallocateRes = HandleIncident$

inv_IncidentManaged__seq: $IncidentManaged \subseteq SetInitialAP$

inv_CloseIncident__seq: $CloseIncident \subseteq (SendStopMsg \cap IncidentManaged)$

inv1_xorfd0: $partition((IsDuplicate \cup NotDuplicate), IsDuplicate, NotDuplicate)$

inv_par_ref: $\forall i \cdot i \in IncidentManaged \Rightarrow AllocateRes[\{i\}] = DeallocateRes[\{i\}]$

inv_alloc_ap: $\forall i \cdot i \in INCIDENT \Rightarrow AllocateRes[\{i\}] \subseteq ap[\{i\}]$

inv_managed: $\forall i \cdot i \in CloseIncident \Rightarrow ap[\{i\}] \setminus notRequired\_ap[\{i\}] \subseteq DeallocateRes[\{i\}]$

manually added

EVENTS

Initialisation

begin

act_CreateIncident: $CreateIncident := \varnothing$

act_SelectType: $SelectType := \varnothing$

act_SelectLocation: $SelectLocation := \varnothing$

act_InfoComp: $InfoComp := \varnothing$

act_IsDuplicate: $IsDuplicate := \varnothing$

act_CloseDuplicate: $CloseDuplicate := \varnothing$

act_NotDuplicate: $NotDuplicate := \varnothing$

act_SetInitialAP: $SetInitialAP := \varnothing$

act_RequestAdditionalRes: $RequestAdditionalRes := \varnothing$

act_SendStopMsg: $SendStopMsg := \varnothing$

act_AllocateRes: $AllocateRes := \varnothing$

act_ResAttend: $ResAttend := \varnothing$

act_DeallocateRes: $DeallocateRes := \varnothing$

act_IncidentManaged: $IncidentManaged := \varnothing$

act_CloseIncident: $CloseIncident := \varnothing$

act1: $ap := \varnothing$

act2: $iLocation := \varnothing$

act_iLocType: $iLocType := \varnothing$

act3: $iType := \varnothing$

act4: $notRequired\_ap := \varnothing$

```
        act_StopMsgReceived: $StopMsgReceived := \varnothing$
    end
Event AllocateRes ⟨ordinary⟩ $\widehat{=}$
    any
        i
        lr
    where
        grd_seq_:  $i \in SetInitialAP$
        grd_self:  $i \mapsto lr \notin AllocateRes$
        grd_InputExpression1:  $lr \in ap[\{i\}]$
        grd0_par:  $i \notin IncidentManaged$
        grd1:  $i \mapsto lr \notin notRequired\_ap$
    then
        act: $AllocateRes := AllocateRes \cup \{i \mapsto lr\}$
    end
Event UpdateResLocation ⟨ordinary⟩ $\widehat{=}$
    any
        i
        lr
    where
        grd_seq_:  $i \mapsto lr \in AllocateRes$
        grd0_loop:  $i \mapsto lr \notin (ResAttend \cup StopMsgReceived)$
    then
        skip
    end
Event ReallocateQuickerRes ⟨ordinary⟩ $\widehat{=}$
    any
        i
        lr
    where
        grd_seq_:  $i \mapsto lr \in AllocateRes$
        grd0_loop:  $i \mapsto lr \notin (ResAttend \cup StopMsgReceived)$
        grd1:  $i \mapsto lr \notin notRequired\_ap$
    then
        skip
    end
Event ResAttend ⟨ordinary⟩ $\widehat{=}$
    any
        i
        lr
    where
```

grd_seq_: $i \mapsto lr \in AllocateRes$
grd_self: $i \mapsto lr \notin ResAttend$
grd_xor: $i \mapsto lr \notin StopMsgReceived$
@grd1 i ↦ lr ∉ notRequired_ap not necessary because it could be added but haven't received msg yet

then

act: $ResAttend := ResAttend \cup \{i \mapsto lr\}$

end

Event StopMsgReceived ⟨ordinary⟩ $\widehat{=}$

any

lr
i

where

grd_seq: $i \mapsto lr \in AllocateRes$
grd_self: $i \mapsto lr \notin StopMsgReceived$
grd_xor: $i \mapsto lr \notin ResAttend$
grd1: $i \mapsto lr \in notRequired\_ap$
in this case it is added because it can not be received unless not required

then

act: $StopMsgReceived := StopMsgReceived \cup \{i \mapsto lr\}$

end

Event CreateIncident ⟨ordinary⟩ $\widehat{=}$

refines CreateIncident

any

i

where

grd_self: $i \notin CreateIncident$

then

act: $CreateIncident := CreateIncident \cup \{i\}$

end

Event SelectType ⟨ordinary⟩ $\widehat{=}$

refines SelectType

any

t
i

where

grd_seq_: $i \in CreateIncident$
grd_self: $i \notin SelectType$
grd1: $t \in INCIDENT\_TYPE$

then

act: $SelectType := SelectType \cup \{i\}$

```
        act1: $iType(i) := t$
    end
Event SelectLocation ⟨ordinary⟩ $\widehat{=}$
refines SelectLocation
    any
        i
        l
        lt
    where
        grd_seq_: $i \in CreateIncident$
        grd_self: $i \notin SelectLocation$
        grd1: $l \in LOCATION$
        grd2: $lt \in LOCATION\_TYPE$
    then
        act: $SelectLocation := SelectLocation \cup \{i\}$
        act1: $iLocation(i) := l$
        act2: $iLocType(i) := lt$
    end
Event InfoComp ⟨ordinary⟩ $\widehat{=}$
refines InfoComp
    any
        i
    where
        grd_seq_: $(i \in SelectType \wedge i \in SelectLocation)$
        grd_self: $i \notin InfoComp$
    then
        act: $InfoComp := InfoComp \cup \{i\}$
    end
Event IsDuplicate ⟨ordinary⟩ $\widehat{=}$
refines IsDuplicate
    any
        i
        i0
    where
        grd_seq_Duplicate: $i \in InfoComp$
        grd_self: $i \notin IsDuplicate$
        grd1_xor: $i \notin NotDuplicate$
        grd_i0_i: $i0 \in NotDuplicate \setminus CloseIncident$
            added manually
        grd_i0i: $iLocation(i) = iLocation(i0)$
    then
        act: $IsDuplicate := IsDuplicate \cup \{i\}$
    end
Event CloseDuplicate ⟨ordinary⟩ $\widehat{=}$
refines CloseDuplicate
    any
        i
    where
        grd_seq_Duplicate:  $i \in IsDuplicate$
        grd_self:  $i \notin CloseDuplicate$
    then
        act: $CloseDuplicate := CloseDuplicate \cup \{i\}$
    end
Event NotDuplicate ⟨ordinary⟩ $\widehat{=}$
refines NotDuplicate
    any
        i
    where
        grd_seq_NonDuplicate:  $i \in InfoComp$
        grd1_xor:  $i \notin IsDuplicate$
        grd_self:  $i \notin NotDuplicate$
    then
        act: $NotDuplicate := NotDuplicate \cup \{i\}$
    end
Event SetInitialAP ⟨ordinary⟩ $\widehat{=}$
refines SetInitialAP
    any
        i
        lrs
        l
        t
    where
        grd3:  $lrs = PDA(t \mapsto l)$
        grd_seq_NonDuplicate:  $i \in NotDuplicate$
        grd_self:  $i \notin SetInitialAP$
        grd1:  $t = iType(i)$
        grd2:  $l = iLocType(i)$
    then
        act: $SetInitialAP := SetInitialAP \cup \{i\}$
        act1: $ap := ap \cup (\{i\} \times lrs)$
            manually
    end
Event RequestAdditionalRes ⟨ordinary⟩ $\widehat{=}$
refines RequestAdditionalRes
    any
        i
        lr
    where
        grd_seq_: $i \in SetInitialAP$
        grd_self: $i \mapsto lr \notin RequestAdditionalRes$
        grd_InputExpression1: $lr \in (LR \setminus ap[\{i\}])$
        grd0_par: $i \notin SendStopMsg$
    then
        act: $RequestAdditionalRes := RequestAdditionalRes \cup \{i \mapsto lr\}$
        act1: $ap := ap \cup \{i \mapsto lr\}$
            //manually added
    end
Event SendStopMsg ⟨ordinary⟩ $\widehat{=}$
refines SendStopMsg
    any
        i
        lrs
    where
        grd_seq_planF: $i \in SetInitialAP$
        grd_self: $i \notin SendStopMsg$
        grd_DuplicateInterrupt: $i \notin IsDuplicate$
        grd_lrs: $lrs = ap[\{i\}] \setminus ResAttend[\{i\}]$
    then
        act: $SendStopMsg := SendStopMsg \cup \{i\}$
        act1: $notRequired\_ap := notRequired\_ap \cup (\{i\} \times lrs)$
    end
Event DeallocateRes ⟨ordinary⟩ $\widehat{=}$
refines HandleIncident
    any
        i
        lr
    where
        grd_self: $i \mapsto lr \notin DeallocateRes$
        grd_seq_: $i \mapsto lr \in (ResAttend \cup StopMsgReceived)$
    then
        act: $DeallocateRes := DeallocateRes \cup \{i \mapsto lr\}$
```

end

Event IncidentManaged ⟨ordinary⟩ $\widehat{=}$

refines IncidentManaged

any

i

where

`grd_par`: $AllocateRes[\{i\}] = DeallocateRes[\{i\}]$

`grd_seq_`: $i \in SetInitialAP$

`grd_self`: $i \notin IncidentManaged$

then

`act`: $IncidentManaged := IncidentManaged \cup \{i\}$

end

Event CloseIncident ⟨ordinary⟩ $\widehat{=}$

refines CloseIncident

any

i

where

`grd_seq_`: $(i \in SendStopMsg \land i \in IncidentManaged)$

`grd_self`: $i \notin CloseIncident$

`gd1`: $ap[\{i\}] \setminus notRequired\_ap[\{i\}] \subseteq DeallocateRes[\{i\}]$

to ensure all action plan items are satisfied

then

`act`: $CloseIncident := CloseIncident \cup \{i\}$

end

END

A.7 Sixth Refinement

A.7.1 Context C3

CONTEXT C3

EXTENDS C2

SETS

R_TYPE type of the resource

PR set of physical resources

CONSTANTS

ltype

ptype

AXIOMS

axm1: $ltype \in LR \rightarrow R\_TYPE$

logical type of a resource

axm2: $ptype \in PR \rightarrow R\_TYPE$

physical type of a resource

axm3: $\forall lr \cdot lr \in LR \Rightarrow (\exists pr \cdot ptype(pr) = ltype(lr))$

END

A.7.2 Machine M6

MACHINE M6

REFINES M5

SEES C3

VARIABLES

CreateIncident

IsDuplicate

CloseDuplicate

NotDuplicate

SetInitialAP

SelectType

SelectLocation

InfoComp

RequestAdditionalRes

IncidentManaged

CloseIncident

SendStopMsg

AllocateRes

ResAttend

DeallocateRes

iType added manually

ap added manually

iLocation

iLocType added manually

alloc manually

alloc_LR manually

StopMsgReceived

notRequired_ap

INVARIANTS

inv_alloc_type: $alloc \in PR \nrightarrow INCIDENT$
  manually

inv_alloc_ran: $ran(alloc) \subseteq SetInitialAP$
  manually

inv_alloc_LR_type: $alloc\_LR \in PR \nrightarrow LR$
  manually

inv_alloc_LR_PR_TYp: $\forall pr, lr \cdot pr \in dom(alloc\_LR) \wedge alloc\_LR(pr) = lr \Rightarrow ptype(pr) = ltype(lr)$

inv_alloc_alloc_LR: $dom(alloc) = dom(alloc\_LR)$

inv_Close_Incident_alloc: $\forall i \cdot i \in (CloseIncident \cup CloseDuplicate) \Rightarrow i \notin ran(alloc)$

  manually

EVENTS

Initialisation

  begin

    act_CreateIncident: $CreateIncident := \varnothing$
    act_SelectType: $SelectType := \varnothing$
    act_SelectLocation: $SelectLocation := \varnothing$
    act_InfoComp: $InfoComp := \varnothing$
    act_IsDuplicate: $IsDuplicate := \varnothing$
    act_CloseDuplicate: $CloseDuplicate := \varnothing$
    act_NotDuplicate: $NotDuplicate := \varnothing$
    act_SetInitialAP: $SetInitialAP := \varnothing$
    act_RequestAdditionalRes: $RequestAdditionalRes := \varnothing$
    act_SendStopMsg: $SendStopMsg := \varnothing$
    act_AllocateRes: $AllocateRes := \varnothing$
    act_ResAttend: $ResAttend := \varnothing$
    act_DeallocateRes: $DeallocateRes := \varnothing$
    act_IncidentManaged: $IncidentManaged := \varnothing$
    act_CloseIncident: $CloseIncident := \varnothing$
    act1: $ap := \varnothing$
    act2: $iLocation := \varnothing$
    act_iLocType: $iLocType := \varnothing$
    act3: $iType := \varnothing$
    act_alloc: $alloc := \varnothing$
    act_alloc_LR: $alloc\_LR := \varnothing$
    act_notRequired: $notRequired\_ap := \varnothing$
    act_StopMsgReceived: $StopMsgReceived := \varnothing$

  end

Event AllocateRes ⟨ordinary⟩ $\widehat{=}$

```
refines AllocateRes
    any
        i
        lr
        pr
    where
        grd_seq_:   $i \in SetInitialAP$
        grd_self:   $i \mapsto lr \notin AllocateRes$
        grd_InputExpression1:   $lr \in ap[\{i\}]$
        grd0_par:   $i \notin IncidentManaged$
        grd1:   $pr \notin dom(alloc)$
            added manually
        grd2:   $ptype(pr) = ltype(lr)$
            added manually
        grd11:   $i \mapsto lr \notin notRequired\_ap$
    then
        act: $AllocateRes := AllocateRes \cup \{i \mapsto lr\}$
        act1: $alloc(pr) := i$
            added manually
        act2: $alloc\_LR(pr) := lr$
            added manually
    end
Event UpdateResLocation ⟨ordinary⟩ $\widehat{=}$
refines UpdateResLocation
    any
        i
        lr
    where
        grd_seq_:   $i \mapsto lr \in AllocateRes$
        grd0_loop:   $i \mapsto lr \notin (ResAttend \cup StopMsgReceived)$
    then
        skip
    end
Event ReallocateQuickerRes ⟨ordinary⟩ $\widehat{=}$
refines ReallocateQuickerRes
    any
        i
        lr
        pr
        pr0
    where
```

grd_seq_: $i \mapsto lr \in AllocateRes$

grd0_loop: $i \mapsto lr \notin (ResAttend \cup StopMsgReceived)$

grd1: $pr0 \in (alloc^{-1}[\{i\}] \cap alloc\_LR^{-1}[\{lr\}])$

added manually

grd2: $pr \notin dom(alloc)$

manually

grd3: $ptype(pr) = ltype(lr)$

manually

grd11: $i \mapsto lr \notin notRequired\_ap$

manually

then

act1: $alloc := (\{pr0\} \lhd alloc) \cup \{pr \mapsto i\}$

act2: $alloc\_LR := (\{pr0\} \lhd alloc\_LR) \cup \{pr \mapsto lr\}$

end

Event ResAttend ⟨ordinary⟩ $\widehat{=}$

refines ResAttend

any

i

lr

where

grd_self: $i \mapsto lr \notin ResAttend$

grd_seq_: $i \mapsto lr \in AllocateRes$

grd_xor: $i \mapsto lr \notin StopMsgReceived$

@grd1 i ↦ lr ∉ notRequired_ap not necessary because it could be added but haven't received msg yet

then

act: $ResAttend := ResAttend \cup \{i \mapsto lr\}$

end

Event CreateIncident ⟨ordinary⟩ $\widehat{=}$

refines CreateIncident

any

i

where

grd_self: $i \notin CreateIncident$

then

act: $CreateIncident := CreateIncident \cup \{i\}$

end

Event SelectType ⟨ordinary⟩ $\widehat{=}$

refines SelectType

any

```
        t
        i
    where
        grd_self:  $i \notin SelectType$
        grd1:  $t \in INCIDENT\_TYPE$
        grd_seq_:  $i \in CreateIncident$
    then
        act: $SelectType := SelectType \cup \{i\}$
        act1: $iType(i) := t$
    end
Event SelectLocation ⟨ordinary⟩ $\widehat{=}$
refines SelectLocation
    any
        i
        l
        lt
    where
        grd_seq_:  $i \in CreateIncident$
        grd_self:  $i \notin SelectLocation$
        grd1:  $l \in LOCATION$
        grd2:  $lt \in LOCATION\_TYPE$
    then
        act: $SelectLocation := SelectLocation \cup \{i\}$
        act1: $iLocation(i) := l$
        act2: $iLocType(i) := lt$
    end
Event InfoComp ⟨ordinary⟩ $\widehat{=}$
refines InfoComp
    any
        i
    where
        grd_self:  $i \notin InfoComp$
        grd_seq_:  $(i \in SelectType \land i \in SelectLocation)$
    then
        act: $InfoComp := InfoComp \cup \{i\}$
    end
Event IsDuplicate ⟨ordinary⟩ $\widehat{=}$
refines IsDuplicate
    any
        i0
```

```
      i
  where
      grd_self:  $i \notin IsDuplicate$
      grd1_xor:  $i \notin NotDuplicate$
      grd_i0:  $i0 \in NotDuplicate \setminus CloseIncident$
         manually
      grd_seq_Duplicate:  $i \in InfoComp$
      grd_i0i:  $iLocation(i) = iLocation(i0)$
  then
      act: $IsDuplicate := IsDuplicate \cup \{i\}$
  end
Event CloseDuplicate ⟨ordinary⟩ $\widehat{=}$
refines CloseDuplicate
  any
      i
  where
      grd_self:  $i \notin CloseDuplicate$
      grd_seq_Duplicate:  $i \in IsDuplicate$
  then
      act: $CloseDuplicate := CloseDuplicate \cup \{i\}$
  end
Event NotDuplicate ⟨ordinary⟩ $\widehat{=}$
refines NotDuplicate
  any
      i
  where
      grd1_xor:  $i \notin IsDuplicate$
      grd_self:  $i \notin NotDuplicate$
      grd_seq_NonDuplicate:  $i \in InfoComp$
  then
      act: $NotDuplicate := NotDuplicate \cup \{i\}$
  end
Event SetInitialAP ⟨ordinary⟩ $\widehat{=}$
refines SetInitialAP
  any
      i
      lrs
      l
      t
  where
```

```
            grd_self: $i \notin SetInitialAP$
            grd_seq_NonDuplicate: $i \in NotDuplicate$
            grd1: $t = iType(i)$
            grd2: $l = iLocType(i)$
            grd3: $lrs = PDA(t \mapsto l)$
      then
            act: $SetInitialAP := SetInitialAP \cup \{i\}$
            act1: $ap := ap \cup (\{i\} \times lrs)$
                manually
      end
Event RequestAdditionalRes ⟨ordinary⟩ $\widehat{=}$
refines RequestAdditionalRes
      any
            i
            lr
      where
            grd_self: $i \mapsto lr \notin RequestAdditionalRes$
            grd_InputExpression1: $lr \in (LR \setminus ap[\{i\}])$
            grd0_par: $i \notin SendStopMsg$
            grd_seq_: $i \in SetInitialAP$
      then
            act: $RequestAdditionalRes := RequestAdditionalRes \cup \{i \mapsto lr\}$
            act1: $ap := ap \cup \{i \mapsto lr\}$
                manually added
      end
Event SendStopMsg ⟨ordinary⟩ $\widehat{=}$
refines SendStopMsg
      any
            i
            lrs
      where
            grd_seq_planF: $i \in SetInitialAP$
            grd_self: $i \notin SendStopMsg$
            grd_DuplicateInterrupt: $i \notin IsDuplicate$
            grd_lrs: $lrs = ap[\{i\}] \setminus ResAttend[\{i\}]$
      then
            act: $SendStopMsg := SendStopMsg \cup \{i\}$
            act1: $notRequired\_ap := notRequired\_ap \cup (\{i\} \times lrs)$
      end
Event StopMsgReceived ⟨ordinary⟩ $\widehat{=}$
refines StopMsgReceived
```

any

lr

i

where

grd_seq: $i \mapsto lr \in AllocateRes$

grd_self: $i \mapsto lr \notin StopMsgReceived$

grd_xor: $i \mapsto lr \notin ResAttend$

grd1: $i \mapsto lr \in notRequired\_ap$

in this case it is added because it can not be received unless not required

then

act: $StopMsgReceived := StopMsgReceived \cup \{i \mapsto lr\}$

end

Event DeallocateRes ⟨ordinary⟩ $\widehat{=}$

refines DeallocateRes

any

i

lr

pr

where

grd_seq_: $i \mapsto lr \in (ResAttend \cup StopMsgReceived)$

grd1: $pr \in (alloc^{-1}[\{i\}] \cap alloc\_LR^{-1}[\{lr\}])$

grd_self: $i \mapsto lr \notin DeallocateRes$

then

act1: $alloc := \{pr\} \lhd alloc$

act2: $alloc\_LR := \{pr\} \lhd alloc\_LR$

act: $DeallocateRes := DeallocateRes \cup \{i \mapsto lr\}$

end

Event IncidentManaged ⟨ordinary⟩ $\widehat{=}$

refines IncidentManaged

any

i

where

grd_seq_: $i \in SetInitialAP$

grd_self: $i \notin IncidentManaged$

grd_par: $AllocateRes[\{i\}] = DeallocateRes[\{i\}]$

then

act: $IncidentManaged := IncidentManaged \cup \{i\}$

end

Event CloseIncident ⟨ordinary⟩ $\widehat{=}$

refines CloseIncident

```
    any
        i
    where
        grd_self:  $i \notin CloseIncident$
        grd_seq_:  $(i \in SendStopMsg \land i \in IncidentManaged)$
        gd1:  $ap[\{i\}] \setminus notRequired\_ap[\{i\}] \subseteq DeallocateRes[\{i\}]$
            to ensure all action plan items are satisfied
        grd2:  $i \notin ran(alloc)$
    then
        act: $CloseIncident := CloseIncident \cup \{i\}$
    end
END
```

A.8 Seventh Refinement

A.8.1 Context C4

```
CONTEXT C4
EXTENDS C3
SETS
        STATIONS
CONSTANTS
        s_location
        rl0
        duration0
AXIOMS
        axm1:  $s\_location \in STATIONS \to LOCATION$
        axm2:  $rl0 \in PR \to ran(s\_location)$
        duration0:  $duration0 \in (LOCATION \times LOCATION) \to \mathbb{N}$
        axm3:  $\forall l \cdot (l \in LOCATION \Rightarrow duration0(l \mapsto l) = 0)$
        axm4:  $\forall l0, l1 \cdot (l0 \in LOCATION \setminus \{l1\} \Rightarrow duration0(l0 \mapsto l1) \neq 0 \land duration0(l1 \mapsto l0) \neq 0)$
END
```

A.8.2 Machine M7

```
MACHINE M7
REFINES M6
```

SEES C4

VARIABLES

CreateIncident

IsDuplicate

CloseDuplicate

NotDuplicate

SetInitialAP

SelectType

SelectLocation

InfoComp

RequestAdditionalRes

IncidentManaged

CloseIncident

SendStopMsg

AllocateRes

ResAttend

DeallocateRes

iType added manually

ap added manually

iLocation

iLocType added manually

alloc manually

alloc_LR manually

rLocation location of a physical resource

duration duration from one location to another this is a variable because it depends on time and not distance which can be affected by traffic

StopMsgReceived

notRequired_ap

INVARIANTS

inv_r_loc_type: $rLocation \in PR \to LOCATION$

inv_duration_type: $duration \in (LOCATION \times LOCATION) \to \mathbb{N}$

EVENTS

Initialisation ⟨extended⟩

begin

act_CreateIncident: $CreateIncident := \varnothing$

act_SelectType: $SelectType := \varnothing$

act_SelectLocation: $SelectLocation := \varnothing$

act_InfoComp: $InfoComp := \varnothing$

act_IsDuplicate: $IsDuplicate := \varnothing$

act_CloseDuplicate: $CloseDuplicate := \varnothing$

act_NotDuplicate: $NotDuplicate := \varnothing$

act_SetInitialAP: $SetInitialAP := \varnothing$

act_RequestAdditionalRes: $RequestAdditionalRes := \varnothing$

act_SendStopMsg: $SendStopMsg := \varnothing$

act_AllocateRes: $AllocateRes := \varnothing$

act_ResAttend: $ResAttend := \varnothing$

act_DeallocateRes: $DeallocateRes := \varnothing$

act_IncidentManaged: $IncidentManaged := \varnothing$

act_CloseIncident: $CloseIncident := \varnothing$

act1: $ap := \varnothing$

act2: $iLocation := \varnothing$

act_iLocType: $iLocType := \varnothing$

act3: $iType := \varnothing$

act_alloc: $alloc := \varnothing$

act_alloc_LR: $alloc\_LR := \varnothing$

act_notRequired: $notRequired\_ap := \varnothing$

act_StopMsgReceived: $StopMsgReceived := \varnothing$

act_r_location: $rLocation := rl0$

act_duration: $duration := duration0$

end

Event AllocateRes ⟨ordinary⟩ $\widehat{=}$

extends AllocateRes

any

i

lr

pr

where

grd_seq_: $i \in SetInitialAP$

grd_self: $i \mapsto lr \notin AllocateRes$

grd_InputExpression1: $lr \in ap[\{i\}]$

grd0_par: $i \notin IncidentManaged$

grd1: $pr \notin dom(alloc)$

added manually

grd2: $ptype(pr) = ltype(lr)$

added manually

grd11: $i \mapsto lr \notin notRequired\_ap$

then

act: $AllocateRes := AllocateRes \cup \{i \mapsto lr\}$

act1: $alloc(pr) := i$
added manually
act2: $alloc\_LR(pr) := lr$
added manually
end

Event UpdateResLocation ⟨ordinary⟩ $\widehat{=}$
extends UpdateResLocation
any
i
lr
l
l0
pr
where
grd_seq_: $i \mapsto lr \in AllocateRes$
grd0_loop: $i \mapsto lr \notin (ResAttend \cup StopMsgReceived)$
grd1: $pr \in (alloc^{-1}[\{i\}] \cap alloc\_LR^{-1}[\{lr\}])$
grd_l0: $l0 = rLocation(pr)$
grd_l: $l \in LOCATION \setminus \{l0\}$
grd_dur: $duration(l \mapsto iLocation(i)) < duration(l0 \mapsto iLocation(i))$
then
act_loc: $rLocation(pr) := l$
end

Event ReallocateQuickerRes ⟨ordinary⟩ $\widehat{=}$
extends ReallocateQuickerRes
any
i
lr
pr
$pr0$
where
grd_seq_: $i \mapsto lr \in AllocateRes$
grd0_loop: $i \mapsto lr \notin (ResAttend \cup StopMsgReceived)$
grd1: $pr0 \in (alloc^{-1}[\{i\}] \cap alloc\_LR^{-1}[\{lr\}])$
added manually
grd2: $pr \notin dom(alloc)$
manually
grd3: $ptype(pr) = ltype(lr)$
manually
grd11: $i \mapsto lr \notin notRequired\_ap$
manually

```
        grd_quicker:  $duration(rLocation(pr) \mapsto iLocation(i)) < duration(rLocation(pr0) \mapsto iLocation(i))$
    then
        act1: $alloc := (\{pr0\} \lhd alloc) \cup \{pr \mapsto i\}$
        act2: $alloc\_LR := (\{pr0\} \lhd alloc\_LR) \cup \{pr \mapsto lr\}$
    end
Event ResAttend ⟨ordinary⟩ $\widehat{=}$
extends ResAttend
    any
        $i$
        $lr$
        pr
    where
        grd_self:  $i \mapsto lr \notin ResAttend$
        grd_seq_:  $i \mapsto lr \in AllocateRes$
        grd_xor:  $i \mapsto lr \notin StopMsgReceived$
            @grd1 i ↦ lr ∉ notRequired_ap not necessary because it could be added but
            haven't received msg yet
        grd1:  $pr \in (alloc^{-1}[\{i\}] \cap alloc\_LR^{-1}[\{lr\}])$
        grd_loc:  $rLocation(pr) = iLocation(i)$
    then
        act: $ResAttend := ResAttend \cup \{i \mapsto lr\}$
    end
Event CreateIncident ⟨ordinary⟩ $\widehat{=}$
extends CreateIncident
    any
        $i$
    where
        grd_self:  $i \notin CreateIncident$
    then
        act: $CreateIncident := CreateIncident \cup \{i\}$
    end
Event SelectType ⟨ordinary⟩ $\widehat{=}$
extends SelectType
    any
        $t$
        $i$
    where
        grd_self:  $i \notin SelectType$
        grd1:  $t \in INCIDENT\_TYPE$
        grd_seq_:  $i \in CreateIncident$
    then
        act: $SelectType := SelectType \cup \{i\}$
        act1: $iType(i) := t$
    end
Event SelectLocation ⟨ordinary⟩ $\widehat{=}$
extends SelectLocation
    any
        $i$
        $l$
        $lt$
    where
        grd_seq_:  $i \in CreateIncident$
        grd_self:  $i \notin SelectLocation$
        grd1:  $l \in LOCATION$
        grd2:  $lt \in LOCATION\_TYPE$
    then
        act: $SelectLocation := SelectLocation \cup \{i\}$
        act1: $iLocation(i) := l$
        act2: $iLocType(i) := lt$
    end
Event InfoComp ⟨ordinary⟩ $\widehat{=}$
extends InfoComp
    any
        $i$
    where
        grd_self:  $i \notin InfoComp$
        grd_seq_:  $(i \in SelectType \land i \in SelectLocation)$
    then
        act: $InfoComp := InfoComp \cup \{i\}$
    end
Event IsDuplicate ⟨ordinary⟩ $\widehat{=}$
extends IsDuplicate
    any
        $i0$
        $i$
    where
        grd_self:  $i \notin IsDuplicate$
        grd1_xor:  $i \notin NotDuplicate$
        grd_i0:  $i0 \in NotDuplicate \setminus CloseIncident$
            manually
        grd_seq_Duplicate:  $i \in InfoComp$
        grd_i0i:  $iLocation(i) = iLocation(i0)$
    then
        act: $IsDuplicate := IsDuplicate \cup \{i\}$
    end
Event CloseDuplicate ⟨ordinary⟩ ≙
extends CloseDuplicate
    any
        $i$
    where
        grd_self:  $i \notin CloseDuplicate$
        grd_seq_Duplicate:  $i \in IsDuplicate$
    then
        act: $CloseDuplicate := CloseDuplicate \cup \{i\}$
    end
Event NotDuplicate ⟨ordinary⟩ ≙
extends NotDuplicate
    any
        $i$
    where
        grd1_xor:  $i \notin IsDuplicate$
        grd_self:  $i \notin NotDuplicate$
        grd_seq_NonDuplicate:  $i \in InfoComp$
    then
        act: $NotDuplicate := NotDuplicate \cup \{i\}$
    end
Event SetInitialAP ⟨ordinary⟩ ≙
extends SetInitialAP
    any
        $i$
        $lrs$
        $l$
        $t$
    where
        grd_self:  $i \notin SetInitialAP$
        grd_seq_NonDuplicate:  $i \in NotDuplicate$
        grd1:  $t = iType(i)$
        grd2:  $l = iLocType(i)$
        grd3:  $lrs = PDA(t \mapsto l)$
    then
```

```
            act: $SetInitialAP := SetInitialAP \cup \{i\}$
            act1: $ap := ap \cup (\{i\} \times lrs)$
               manually
      end
Event RequestAdditionalRes ⟨ordinary⟩ $\widehat{=}$
extends RequestAdditionalRes
      any
            $i$
            $lr$
      where
            grd_self: $i \mapsto lr \notin RequestAdditionalRes$
            grd_InputExpression1: $lr \in (LR \setminus ap[\{i\}])$
            grd0_par: $i \notin SendStopMsg$
            grd_seq_: $i \in SetInitialAP$
      then
            act: $RequestAdditionalRes := RequestAdditionalRes \cup \{i \mapsto lr\}$
            act1: $ap := ap \cup \{i \mapsto lr\}$
               manually added
      end
Event SendStopMsg ⟨ordinary⟩ $\widehat{=}$
extends SendStopMsg
      any
            $i$
            $lrs$
      where
            grd_seq_planF: $i \in SetInitialAP$
            grd_self: $i \notin SendStopMsg$
            grd_DuplicateInterrupt: $i \notin IsDuplicate$
            grd_lrs: $lrs = ap[\{i\}] \setminus ResAttend[\{i\}]$
      then
            act: $SendStopMsg := SendStopMsg \cup \{i\}$
            act1: $notRequired\_ap := notRequired\_ap \cup (\{i\} \times lrs)$
      end
Event StopMsgReceived ⟨ordinary⟩ $\widehat{=}$
extends StopMsgReceived
      any
            $lr$
            $i$
      where
            grd_seq: $i \mapsto lr \in AllocateRes$
```

grd_self: $i \mapsto lr \notin StopMsgReceived$

grd_xor: $i \mapsto lr \notin ResAttend$

grd1: $i \mapsto lr \in notRequired\_ap$

in this case it is added because it can not be received unless not required

then

act: $StopMsgReceived := StopMsgReceived \cup \{i \mapsto lr\}$

end

Event DeallocateRes ⟨ordinary⟩ $\widehat{=}$

extends DeallocateRes

any

i

lr

pr

where

grd_seq_: $i \mapsto lr \in (ResAttend \cup StopMsgReceived)$

grd1: $pr \in (alloc^{-1}[\{i\}] \cap alloc\_LR^{-1}[\{lr\}])$

grd_self: $i \mapsto lr \notin DeallocateRes$

then

act1: $alloc := \{pr\} \lhd alloc$

act2: $alloc\_LR := \{pr\} \lhd alloc\_LR$

act: $DeallocateRes := DeallocateRes \cup \{i \mapsto lr\}$

end

Event IncidentManaged ⟨ordinary⟩ $\widehat{=}$

extends IncidentManaged

any

i

where

grd_seq_: $i \in SetInitialAP$

grd_self: $i \notin IncidentManaged$

grd_par: $AllocateRes[\{i\}] = DeallocateRes[\{i\}]$

then

act: $IncidentManaged := IncidentManaged \cup \{i\}$

end

Event CloseIncident ⟨ordinary⟩ $\widehat{=}$

extends CloseIncident

any

i

where

grd_self: $i \notin CloseIncident$

grd_seq_: $(i \in SendStopMsg \land i \in IncidentManaged)$

```
        gd1:  $ap[\{i\}] \setminus notRequired\_ap[\{i\}] \subseteq DeallocateRes[\{i\}]$
            to ensure all action plan items are satisfied
        grd2:  $i \notin ran(alloc)$
    then
        act:  $CloseIncident := CloseIncident \cup \{i\}$
    end
END
```

Appendix B

Fire Dispatch Model with Exception Handling

In this chapter we present the complete version of the fire dispatch workflow after introducing the exception handling combinators.

B.1 Static Part of the Model: Contexts

B.1.1 Context: C0

```
CONTEXT C0
SETS
        INCIDENT
END
```

B.1.2 Context: C1

```
CONTEXT C1
EXTENDS C0
SETS
        LR
END
```

B.1.3 Context: C2

```
CONTEXT C2
EXTENDS C1
```

SETS

INCIDENT_TYPE

LOCATION

LOCATION_TYPE

CONSTANTS

PDA set the action plan according to the type and location

AXIOMS

axm1: $PDA \in (INCIDENT\_TYPE \times LOCATION\_TYPE) \rightarrow \mathbb{P}(LR)$

axm2: $\forall it, loc \cdot it \in INCIDENT\_TYPE \land loc \in LOCATION\_TYPE \Rightarrow PDA(it \mapsto loc) \neq \varnothing$

END

B.1.4 Context: C3

CONTEXT C3

EXTENDS C2

SETS

R_TYPE type of the resource

CONSTANTS

ltype

default_priority

AXIOMS

axm1: $default\_priority \in INCIDENT\_TYPE \rightarrow 1 \,..\, 3$

axm2: $ltype \in LR \rightarrow R\_TYPE$

logical type of a resource

END

B.1.5 Context: C4

CONTEXT C4

EXTENDS C3

SETS

PR set of physical resources

CONSTANTS

ptype

AXIOMS

axm2: $ptype \in PR \rightarrow R\_TYPE$

physical type of a resource

axm3: $\forall lr \cdot lr \in LR \Rightarrow (\exists pr \cdot ptype(pr) = ltype(lr))$

END

B.1.6 Context: C5

CONTEXT C5

EXTENDS C4

SETS

STATIONS

CONSTANTS

s_location

rl0

duration0

AXIOMS

axm1: $s\_location \in STATIONS \rightarrow LOCATION$

axm2: $rl0 \in PR \rightarrow ran(s\_location)$

duration0: $duration0 \in (LOCATION \times LOCATION) \rightarrow \mathbb{N}$

axm3: $\forall l \cdot (l \in LOCATION \Rightarrow duration0(l \mapsto l) = 0)$

axm4: $\forall l0, l1 \cdot (l0 \in LOCATION \setminus \{l1\} \Rightarrow duration0(l0 \mapsto l1) \neq 0 \land duration0(l1 \mapsto l0) \neq 0)$

END

B.1.7 Context: M9_implicitContext

This context is generated by the iUML-B statemachine.

CONTEXT M9_implicitContext

SETS

pr_status_STATES

pr_available_sm_STATES

pr_allocated_sm_STATES

pr_free_sm_STATES

CONSTANTS

Available

pr_available_sm_NULL

pr_allocated

pr_allocated_sm_NULL

Mobilised

InAttendance

pr_free

pr_free_sm_NULL

Returning

StationAvailable

Unavailable

AXIOMS

typeof_pr_available_sm_NULL: $pr\_available\_sm\_NULL \in pr\_available\_sm\_STATES$

typeof_pr_allocated_sm_NULL: $pr\_allocated\_sm\_NULL \in pr\_allocated\_sm\_STATES$

typeof_pr_free_sm_NULL: $pr\_free\_sm\_NULL \in pr\_free\_sm\_STATES$

typeof_Available: $Available \in pr\_status\_STATES$

typeof_pr_allocated: $pr\_allocated \in pr\_available\_sm\_STATES$

typeof_Mobilised: $Mobilised \in pr\_allocated\_sm\_STATES$

typeof_InAttendance: $InAttendance \in pr\_allocated\_sm\_STATES$

typeof_pr_free: $pr\_free \in pr\_available\_sm\_STATES$

typeof_Returning: $Returning \in pr\_free\_sm\_STATES$

typeof_StationAvailable: $StationAvailable \in pr\_free\_sm\_STATES$

typeof_Unavailable: $Unavailable \in pr\_status\_STATES$

distinct_states_in_pr_status_STATES: $partition(pr\_status\_STATES, \{Available\}, \{Unavailable\})$

distinct_states_in_pr_available_sm_STATES:

$partition(pr\_available\_sm\_STATES, \{pr\_allocated\}, \{pr\_free\}, \{pr\_available\_sm\_NULL\})$

distinct_states_in_pr_allocated_sm_STATES:

$partition(pr\_allocated\_sm\_STATES, \{Mobilised\}, \{InAttendance\}, \{pr\_allocated\_sm\_NULL\})$

distinct_states_in_pr_free_sm_STATES:

$partition(pr\_free\_sm\_STATES, \{Returning\}, \{StationAvailable\}, \{pr\_free\_sm\_NULL\})$

END

B.2 Model Refinements

B.2.1 Abstract Level: M0

MACHINE M0

SEES C0

VARIABLES

CreateIncident

NotDuplicate

IsDuplicate

INVARIANTS

inv_CreateIncident_type: $CreateIncident \subseteq INCIDENT$

inv_NotDuplicate_seq: $NotDuplicate \subseteq CreateIncident$

inv_IsDuplicate_DuplicateInterrupt_seq: $IsDuplicate \subseteq CreateIncident$

inv_Interrupt: $partition((NotDuplicate \cup IsDuplicate), NotDuplicate, IsDuplicate)$

EVENTS

Initialisation

begin

act_CreateIncident: $CreateIncident := \varnothing$

act_NotDuplicate: $NotDuplicate := \varnothing$

act_IsDuplicate: $IsDuplicate := \varnothing$

end

Event CreateIncident ⟨ordinary⟩ $\widehat{=}$

any

i

where

grd_self: $i \notin CreateIncident$

then

act: $CreateIncident := CreateIncident \cup \{i\}$

end

Event NotDuplicate ⟨ordinary⟩ $\widehat{=}$

any

i

where

grd_DuplicateInterrupt: $i \notin IsDuplicate$

grd_seq: $i \in CreateIncident$

grd_self: $i \notin NotDuplicate$

then

act: $NotDuplicate := NotDuplicate \cup \{i\}$

end

Event IsDuplicate ⟨ordinary⟩ $\widehat{=}$

any

i

where

grd_seq_DuplicateInterrupt: $i \in CreateIncident$

grd_self: $i \notin IsDuplicate$

grd_Interrupt: $i \notin NotDuplicate$

```
    then
        act: $IsDuplicate := IsDuplicate \cup \{i\}$
    end
END
```

B.2.2 First Refinement: M1

```
MACHINE M1
REFINES M0
SEES C0
VARIABLES
        CreateIncident
        GatherInfo
        SetInitialAP
        ManageIncident
        UpdateAP
        CloseIncident
        IsDuplicate
INVARIANTS
        inv_CreateIncident_type: $CreateIncident \subseteq INCIDENT$
        inv_GatherInfo__seq: $GatherInfo \subseteq CreateIncident$
        inv_SetInitialAP__seq: $SetInitialAP \subseteq GatherInfo$
        inv_ManageIncident__seq: $ManageIncident \subseteq SetInitialAP$
        inv_UpdateAP__seq: $UpdateAP \subseteq SetInitialAP$
        inv_CloseIncident_FD_seq: $CloseIncident \subseteq (UpdateAP \cap ManageIncident)$
        inv_IsDuplicate_DuplicateInterrupt_seq: $IsDuplicate \subseteq CreateIncident$
        inv_CloseIncident_glu: $CloseIncident = NotDuplicate$
        inv_IsDuplicate_Interrupt: $partition((CloseIncident \cup IsDuplicate), CloseIncident, IsDuplicate)$

EVENTS
Initialisation
    begin
        act_CreateIncident: $CreateIncident := \varnothing$
        act_GatherInfo: $GatherInfo := \varnothing$
        act_SetInitialAP: $SetInitialAP := \varnothing$
        act_ManageIncident: $ManageIncident := \varnothing$
        act_CloseIncident: $CloseIncident := \varnothing$
        act_IsDuplicate: $IsDuplicate := \varnothing$
        act_UpdateAP: $UpdateAP := \varnothing$
    end
Event CreateIncident ⟨ordinary⟩ $\widehat{=}$
refines CreateIncident
    any
        i
    where
        grd_self:  $i \notin CreateIncident$
    then
        act: $CreateIncident := CreateIncident \cup \{i\}$
    end
Event GatherInfo ⟨ordinary⟩ $\widehat{=}$
    any
        i
    where
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd_seq_:  $i \in CreateIncident$
        grd_self:  $i \notin GatherInfo$
    then
        act: $GatherInfo := GatherInfo \cup \{i\}$
    end
Event SetInitialAP ⟨ordinary⟩ $\widehat{=}$
    any
        i
    where
        grd_seq_:  $i \in GatherInfo$
        grd_self:  $i \notin SetInitialAP$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
    then
        act: $SetInitialAP := SetInitialAP \cup \{i\}$
    end
Event UpdateAP ⟨ordinary⟩ $\widehat{=}$
    any
        i
    where
        grd_seq_:  $i \in SetInitialAP$
        grd_self:  $i \notin UpdateAP$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
    then
        act: $UpdateAP := UpdateAP \cup \{i\}$
    end
```

```
Event ManageIncident ⟨ordinary⟩ $\widehat{=}$
    any
        i
    where
        grd_seq_: $i \in SetInitialAP$
        grd_self: $i \notin ManageIncident$
        grd_DuplicateInterrupt: $i \notin IsDuplicate$
    then
        act: $ManageIncident := ManageIncident \cup \{i\}$
    end
Event CloseIncident ⟨ordinary⟩ $\widehat{=}$
refines NotDuplicate
    any
        i
    where
        grd_seq_FD: $(i \in UpdateAP \wedge i \in ManageIncident)$
        grd_self: $i \notin CloseIncident$
        grd_DuplicateInterrupt: $i \notin IsDuplicate$
    then
        act: $CloseIncident := CloseIncident \cup \{i\}$
    end
Event IsDuplicate ⟨ordinary⟩ $\widehat{=}$
refines IsDuplicate
    any
        i
    where
        grd_seq: $i \in CreateIncident$
        grd_self: $i \notin IsDuplicate$
        grd_DuplicateInterrupt: $i \notin CloseIncident$
    then
        act: $IsDuplicate := IsDuplicate \cup \{i\}$
    end
END
```

B.2.3 Second Refinement: M2

```
MACHINE M2
REFINES M1
SEES C1
VARIABLES
```

CreateIncident

GatherInfo

SetInitialAP

ManageIncident

RequestAdditionalRes

SendStopMsg

CloseIncident

IsDuplicate

ap

notRequired_ap

INVARIANTS

`inv_CreateIncident_type`: $CreateIncident \subseteq INCIDENT$

`inv_SetActionPlan_type`: $SetInitialAP \subseteq INCIDENT$

`inv_RequestAdditionalResources_type`: $RequestAdditionalRes \subseteq INCIDENT \times LR$

`inv_GatherInfo__seq`: $GatherInfo \subseteq CreateIncident$

`inv_SetActionPlan_glu`: $SetInitialAP = SetInitialAP$

`inv_SetActionPlan_inSeq`: $SetInitialAP \subseteq GatherInfo$

`inv_RequestAdditionalResources_planF_seq`: $dom(RequestAdditionalRes) \subseteq SetInitialAP$

`inv_RequestAdditionalRes_Rep`: $\forall i \cdot i \in INCIDENT \Rightarrow RequestAdditionalRes[\{i\}] \subseteq LR$

Replicator invariant

`inv_SendStopMsg_seq`: $SendStopMsg \subseteq SetInitialAP$

`inv_SendStopMsg_glu`: $SendStopMsg = UpdateAP$

`inv_ManageIncident__seq`: $ManageIncident \subseteq SetInitialAP$

`inv_CloseIncident_FD_seq`: $CloseIncident \subseteq ManageIncident \cap SendStopMsg$

changed weak seq to and refinement

`inv_IsDuplicate_DuplicateInterrupt_seq`: $IsDuplicate \subseteq CreateIncident$

`inv_ap_type`: $ap \in INCIDENT \leftrightarrow LR$

`inv_ap_seq`: $dom(ap) = SetInitialAP$

`inv_notRequired_ap_type`: $notRequired\_ap \subseteq ap$

manually added data variable to describe the actions plan items that are not required after stop message is sent

`inv_Interrupt`: $partition((CloseIncident \cup IsDuplicate), CloseIncident, IsDuplicate)$

EVENTS

Initialisation

```
    begin
        act_CreateIncident: $CreateIncident := \varnothing$
        act_GatherInfo: $GatherInfo := \varnothing$
        act_SetInitialAP: $SetInitialAP := \varnothing$
        act_ManageIncident: $ManageIncident := \varnothing$
        act_CloseIncident: $CloseIncident := \varnothing$
        act_IsDuplicate: $IsDuplicate := \varnothing$
        act_SendStopMsg: $SendStopMsg := \varnothing$
        act_ap: $ap := \varnothing$
        act_notRequired_ap: $notRequired\_ap := \varnothing$
        act_RequestAdditionalRes: $RequestAdditionalRes := \varnothing$
    end
Event CreateIncident ⟨ordinary⟩ $\widehat{=}$
refines CreateIncident
    any
        i
    where
        grd_self: $i \notin CreateIncident$
    then
        act: $CreateIncident := CreateIncident \cup \{i\}$
    end
Event GatherInfo ⟨ordinary⟩ $\widehat{=}$
refines GatherInfo
    any
        i
    where
        grd_self: $i \notin GatherInfo$
        grd_DuplicateInterrupt: $i \notin IsDuplicate$
        grd_seq_: $i \in CreateIncident$
    then
        act: $GatherInfo := GatherInfo \cup \{i\}$
    end
Event SetInitialAP ⟨ordinary⟩ $\widehat{=}$
refines SetInitialAP
    any
        i
        lrs
    where
        grd_DuplicateInterrupt: $i \notin IsDuplicate$
        grd_seq_: $i \in GatherInfo$
        grd_self:  $i \notin SetInitialAP$
        grd1:  $lrs \subseteq LR$
        grd2:  $lrs \neq \varnothing$
    then
        act: $SetInitialAP := SetInitialAP \cup \{i\}$
        act1: $ap := ap \cup (\{i\} \times lrs)$
    end
Event RequestAdditionalRes ⟨ordinary⟩ $\widehat{=}$
    any
        i
        lr
    where
        grd_seq_planF:  $i \in SetInitialAP$
        grd_self:  $i \mapsto lr \notin RequestAdditionalRes$
        grd_InputExpression1:  $lr \in LR \setminus ap[\{i\}]$
        grd0_par:  $i \notin SendStopMsg$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd1:  $lr \notin ap[\{i\}]$
            added manually to replace lr ∈ LR\ap[{i}]
    then
        act: $RequestAdditionalRes := RequestAdditionalRes \cup \{i \mapsto lr\}$
        act1: $ap := ap \cup \{i \mapsto lr\}$
    end
Event SendStopMsg ⟨ordinary⟩ $\widehat{=}$
refines UpdateAP
    any
        i
        lrs
    where
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd_seq_:  $i \in SetInitialAP$
        grd_self:  $i \notin SendStopMsg$
        grd_lrs:  $lrs \subseteq ap[\{i\}]$
    then
        act: $SendStopMsg := SendStopMsg \cup \{i\}$
        act1: $notRequired\_ap := notRequired\_ap \cup (\{i\} \times lrs)$
    end
Event ManageIncident ⟨ordinary⟩ $\widehat{=}$
refines ManageIncident
    any
        i
    where
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd_seq_:  $i \in SetInitialAP$
        grd_self:  $i \notin ManageIncident$
    then
        act: $ManageIncident := ManageIncident \cup \{i\}$
    end
Event CloseIncident ⟨ordinary⟩ $\widehat{=}$
refines CloseIncident
    any
        i
    where
        grd_seq_FD:  $(i \in SendStopMsg \wedge i \in ManageIncident)$
        grd_self:  $i \notin CloseIncident$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
    then
        act: $CloseIncident := CloseIncident \cup \{i\}$
    end
Event IsDuplicate ⟨ordinary⟩ $\widehat{=}$
refines IsDuplicate
    any
        i
    where
        grd_seq_DuplicateInterrupt:  $i \in CreateIncident$
        grd_self:  $i \notin IsDuplicate$
        grd1:  $i \notin CloseIncident$
    then
        act: $IsDuplicate := IsDuplicate \cup \{i\}$
    end
END
```

B.2.4 Third Refinement: M3

```
MACHINE M3
REFINES M2
SEES C2
VARIABLES
      CreateIncident
      SelectLocation
      SelectType
```

CompInfo

SetInitialAP

ManageIncident

RequestAdditionalRes

SendStopMsg

CloseIncident

IsDuplicate

ap

notRequired_ap

iLocation

iType

iLocType

INVARIANTS

`inv_CreateIncident_type`: $CreateIncident \subseteq INCIDENT$

`inv_SetActionPlan_type`: $SetInitialAP \subseteq INCIDENT$

`inv_RequestAdditionalResources_type`: $RequestAdditionalRes \subseteq INCIDENT \times LR$

`inv_SelectType__seq`: $SelectType \subseteq CreateIncident$

`inv_CompInfo__seq`: $CompInfo \subseteq (SelectLocation \cap SelectType)$

`inv_CompInfo_glu`: $CompInfo = GatherInfo$

`inv_SetActionPlan_glu`: $SetInitialAP = SetInitialAP$

`inv_SetActionPlan_inSeq`: $SetInitialAP \subseteq CompInfo$

`inv_RequestAdditionalResources_planF_seq`: $dom(RequestAdditionalRes) \subseteq SetInitialAP$

`inv_RequestAdditionalRes_Rep`: $\forall i \cdot i \in INCIDENT \Rightarrow RequestAdditionalRes[\{i\}] \subseteq LR$

Replicator invariant

`inv_SendStopMsg_seq`: $SendStopMsg \subseteq SetInitialAP$

`inv_ManageIncident__seq`: $ManageIncident \subseteq SetInitialAP$

`inv_CloseIncident_FD_seq`: $CloseIncident \subseteq ManageIncident \cap SendStopMsg$

`inv_IsDuplicate_DuplicateInterrupt_seq`: $IsDuplicate \subseteq CreateIncident$

`inv_Interrupt`: $partition((CloseIncident \cup IsDuplicate), CloseIncident, IsDuplicate)$

`inv_iLocation`: $iLocation \in SelectLocation \to LOCATION$

data variable

`inv_loc_type`: $iLocType \in SelectLocation \to LOCATION\_TYPE$

data variable

`inv_iType`: $iType \in SelectType \to INCIDENT\_TYPE$

data variable

EVENTS

Initialisation

  begin

    act_CreateIncident: $CreateIncident := \varnothing$

    act_SetInitialAP: $SetInitialAP := \varnothing$

    act_ManageIncident: $ManageIncident := \varnothing$

    act_CloseIncident: $CloseIncident := \varnothing$

    act_IsDuplicate: $IsDuplicate := \varnothing$

    act_SendStopMsg: $SendStopMsg := \varnothing$

    act_ap: $ap := \varnothing$

    act_notRequired_ap: $notRequired\_ap := \varnothing$

    act_RequestAdditionalRes: $RequestAdditionalRes := \varnothing$

    act2: $iLocation := \varnothing$

    act3: $iType := \varnothing$

    act_loc_type: $iLocType := \varnothing$

    act_SelectLocation: $SelectLocation := \varnothing$

    act_SelectType: $SelectType := \varnothing$

    act_CompInfo: $CompInfo := \varnothing$

  end

Event CreateIncident ⟨ordinary⟩ $\widehat{=}$

refines CreateIncident

  any

    i

  where

    grd_self: $i \notin CreateIncident$

  then

    act: $CreateIncident := CreateIncident \cup \{i\}$

  end

Event SelectLocation ⟨ordinary⟩ $\widehat{=}$

  any

    i

    l

    lt

  where

    grd_seq_: $i \in CreateIncident$

    grd_self: $i \notin SelectLocation$

    grd_DuplicateInterrupt: $i \notin IsDuplicate$

    grd1: $l \in LOCATION$

    grd2: $lt \in LOCATION\_TYPE$

  then

    act: $SelectLocation := SelectLocation \cup \{i\}$

```
        act1: $iLocation(i) := l$
        act2: $iLocType(i) := lt$
    end
Event SelectType ⟨ordinary⟩ $\widehat{=}$
    any
        i
        t
    where
        grd_seq_:  $i \in CreateIncident$
        grd_self:  $i \notin SelectType$
        grd1:  $t \in INCIDENT\_TYPE$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
    then
        act: $SelectType := SelectType \cup \{i\}$
        act1: $iType(i) := t$
    end
Event CompInfo ⟨ordinary⟩ $\widehat{=}$
refines GatherInfo
    any
        i
    where
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd_seq_:  $(i \in SelectLocation \land i \in SelectType)$
        grd_self:  $i \notin CompInfo$
    then
        act: $CompInfo := CompInfo \cup \{i\}$
    end
Event SetInitialAP ⟨ordinary⟩ $\widehat{=}$
refines SetInitialAP
    any
        i
        t
        l
        lrs
    where
        grd_seq:  $i \in CompInfo$
        grd_self:  $i \notin SetInitialAP$
        grd1:  $t = iType(i)$
        grd2:  $l = iLocType(i)$
        grd3:  $lrs = PDA(t \mapsto l)$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
    then
        act: $SetInitialAP := SetInitialAP \cup \{i\}$
        act1: $ap := ap \cup (\{i\} \times lrs)$
    end
Event RequestAdditionalRes ⟨ordinary⟩ $\widehat{=}$
refines RequestAdditionalRes
    any
        lr
        i
    where
        grd_seq_planF: $i \in SetInitialAP$
        grd_self: $i \mapsto lr \notin RequestAdditionalRes$
        grd_InputExpression1: $lr \in LR$
        grd0_par: $i \notin SendStopMsg$
        grd_DuplicateInterrupt: $i \notin IsDuplicate$
        grd1: $lr \notin ap[\{i\}]$
    then
        act1: $ap := ap \cup \{i \mapsto lr\}$
        act: $RequestAdditionalRes := RequestAdditionalRes \cup \{i \mapsto lr\}$
    end
Event SendStopMsg ⟨ordinary⟩ $\widehat{=}$
refines SendStopMsg
    any
        i
        lrs
    where
        grd_DuplicateInterrupt: $i \notin IsDuplicate$
        grd_seq_: $i \in SetInitialAP$
        grd_self: $i \notin SendStopMsg$
        grd_lrs: $lrs \subseteq ap[\{i\}]$
    then
        act: $SendStopMsg := SendStopMsg \cup \{i\}$
        act1: $notRequired\_ap := notRequired\_ap \cup (\{i\} \times lrs)$
    end
Event ManageIncident ⟨ordinary⟩ $\widehat{=}$
refines ManageIncident
    any
        i
    where
        grd_DuplicateInterrupt: $i \notin IsDuplicate$
```

```
          grd_seq_:   $i \in SetInitialAP$
          grd_self:   $i \notin ManageIncident$
     then
          act: $ManageIncident := ManageIncident \cup \{i\}$
     end
Event CloseIncident ⟨ordinary⟩ $\widehat{=}$
refines CloseIncident
     any
          i
     where
          grd_seq_FD:   $(i \in SendStopMsg \wedge i \in ManageIncident)$
          grd_self:   $i \notin CloseIncident$
          grd_DuplicateInterrupt:   $i \notin IsDuplicate$
     then
          act: $CloseIncident := CloseIncident \cup \{i\}$
     end
Event IsDuplicate ⟨ordinary⟩ $\widehat{=}$
refines IsDuplicate
     any
          i
     where
          grd_seq_DuplicateInterrupt:   $i \in CreateIncident$
          grd_self:   $i \notin IsDuplicate$
          grd1:   $i \notin CloseIncident$
     then
          act: $IsDuplicate := IsDuplicate \cup \{i\}$
     end
END
```

B.2.5 Fourth Refinement: M4

```
MACHINE M4
REFINES M3
SEES C3
VARIABLES
          CreateIncident
          SelectLocation
          SelectType
          CompInfo
```

SetInitialAP

ManageIncident

RequestAdditionalRes

SendStopMsg

CloseIncident

IsDuplicate

ap

notRequired_ap

iLocation

iType

DisablePriorityChange

priority

iLocType

INVARIANTS

`inv_CreateIncident_type`: $CreateIncident \subseteq INCIDENT$

`inv_SetActionPlan_type`: $SetInitialAP \subseteq INCIDENT$

`inv_RequestAdditionalResources_type`: $RequestAdditionalRes \subseteq INCIDENT \times LR$

`inv_SelectType__seq`: $SelectType \subseteq CreateIncident$

`inv_DisablePriorityChange_priority_seq`: $DisablePriorityChange \subseteq SelectType$

`inv_CompInfo__seq`: $CompInfo \subseteq (SelectLocation \cap SelectType)$

weak sequencing is used

`inv_SetActionPlan_glu`: $SetInitialAP = SetInitialAP$

`inv_SetActionPlan_inSeq`: $SetInitialAP \subseteq CompInfo$

`inv_RequestAdditionalResources_planF_seq`: $dom(RequestAdditionalRes) \subseteq SetInitialAP$

`inv_RequestAdditionalRes_Rep`: $\forall i \cdot i \in INCIDENT \Rightarrow RequestAdditionalRes[\{i\}] \subseteq LR$

Replicator invariant

`inv_SendStopMsg_seq`: $SendStopMsg \subseteq SetInitialAP$

`inv_ManageIncident__seq`: $ManageIncident \subseteq SetInitialAP$

`inv_CloseIncident_FD_seq`: $CloseIncident \subseteq ManageIncident \cap SendStopMsg$

`inv_IsDuplicate_DuplicateInterrupt_seq`: $IsDuplicate \subseteq CreateIncident$

`inv_Interrupt`: $partition((CloseIncident \cup IsDuplicate), CloseIncident, IsDuplicate)$

`inv_priority`: $priority \in SelectType \rightarrow 1 \,..\, 3$

EVENTS

Initialisation

```
    begin
        act_CreateIncident: $CreateIncident := \varnothing$
        act_SetInitialAP: $SetInitialAP := \varnothing$
        act_ManageIncident: $ManageIncident := \varnothing$
        act_CloseIncident: $CloseIncident := \varnothing$
        act_IsDuplicate: $IsDuplicate := \varnothing$
        act_SendStopMsg: $SendStopMsg := \varnothing$
        act_ap: $ap := \varnothing$
        act_notRequired_ap: $notRequired\_ap := \varnothing$
        act_RequestAdditionalRes: $RequestAdditionalRes := \varnothing$
        act2: $iLocation := \varnothing$
        act3: $iType := \varnothing$
        act_loc_type: $iLocType := \varnothing$
        act_SelectLocation: $SelectLocation := \varnothing$
        act_SelectType: $SelectType := \varnothing$
        act_CompInfo: $CompInfo := \varnothing$
        act_DisablePriorityChange: $DisablePriorityChange := \varnothing$
        act4: $priority := \varnothing$
    end
Event CreateIncident ⟨ordinary⟩ $\widehat{=}$
refines CreateIncident
    any
        i
    where
        grd_self: $i \notin CreateIncident$
    then
        act: $CreateIncident := CreateIncident \cup \{i\}$
    end
Event SelectLocation ⟨ordinary⟩ $\widehat{=}$
refines SelectLocation
    any
        i
        l
        lt
    where
        grd_seq_: $i \in CreateIncident$
        grd_self: $i \notin SelectLocation$
        grd_DuplicateInterrupt: $i \notin IsDuplicate$
        grd1: $l \in LOCATION$
        grd2: $lt \in LOCATION\_TYPE$
    then
```

```
        act: $SelectLocation := SelectLocation \cup \{i\}$
        act1: $iLocation(i) := l$
        act2: $iLocType(i) := lt$
    end
Event SelectType ⟨ordinary⟩ $\widehat{=}$
refines SelectType
    any
        i
        t
        p
    where
        grd_seq_: $i \in CreateIncident$
        grd_self: $i \notin SelectType$
        grd1: $t \in INCIDENT\_TYPE$
        grd_DuplicateInterrupt: $i \notin IsDuplicate$
        grd2: $p = default\_priority(t)$
    then
        act: $SelectType := SelectType \cup \{i\}$
        act1: $iType(i) := t$
        act2: $priority(i) := p$
    end
Event ChangePriority ⟨ordinary⟩ $\widehat{=}$
    any
        i
        p
    where
        grd_seq_priority: $i \in SelectType$
        grd0_loop: $i \notin DisablePriorityChange$
        grd1: $p \in 1 \,..\, 3$
        grd2: $p \neq priority(i)$
        grd3: $i \notin CloseIncident$
            we can also set DisablePriorityChange in CloseIncident
        grd_DuplicateInterrupt: $i \notin IsDuplicate$
    then
        act1: $priority(i) := p$
    end
Event DisablePriorityChange ⟨ordinary⟩ $\widehat{=}$
    any
        i
    where
        grd_DuplicateInterrupt: $i \notin IsDuplicate$
        grd_seq_priority:  $i \in SelectType$
        grd_self:  $i \notin DisablePriorityChange$
        grd1:  $i \notin CloseIncident$
            we can also set DisablePriorityChange in CloseIncident
    then
        act: $DisablePriorityChange := DisablePriorityChange \cup \{i\}$
    end
Event CompInfo ⟨ordinary⟩ $\widehat{=}$
refines CompInfo
    any
        i
    where
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd_seq_:  $(i \in SelectLocation \land i \in SelectType)$
        grd_self:  $i \notin CompInfo$
    then
        act: $CompInfo := CompInfo \cup \{i\}$
    end
Event SetInitialAP ⟨ordinary⟩ $\widehat{=}$
refines SetInitialAP
    any
        lrs
        i
        t
        l
    where
        grd_seq:  $i \in CompInfo$
        grd_self:  $i \notin SetInitialAP$
        grd1:  $t = iType(i)$
        grd2:  $l = iLocType(i)$
        grd3:  $lrs = PDA(t \mapsto l)$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
    then
        act: $SetInitialAP := SetInitialAP \cup \{i\}$
        act1: $ap := ap \cup (\{i\} \times lrs)$
    end
Event RequestAdditionalRes ⟨ordinary⟩ $\widehat{=}$
refines RequestAdditionalRes
    any
        i
        lr
    where
        grd_seq_planF:  $i \in SetInitialAP$
        grd_self:  $i \mapsto lr \notin RequestAdditionalRes$
        grd_InputExpression1:  $lr \in LR$
        grd0_par:  $i \notin SendStopMsg$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd1:  $lr \notin ap[\{i\}]$
    then
        act1: $ap := ap \cup \{i \mapsto lr\}$
        act: $RequestAdditionalRes := RequestAdditionalRes \cup \{i \mapsto lr\}$
    end
Event SendStopMsg ⟨ordinary⟩ $\widehat{=}$
refines SendStopMsg
    any
        i
        lrs
    where
        grd_lrs:  $lrs \subseteq ap[\{i\}]$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd_seq_:  $i \in SetInitialAP$
        grd_self:  $i \notin SendStopMsg$
    then
        act1: $notRequired\_ap := notRequired\_ap \cup (\{i\} \times lrs)$
        act: $SendStopMsg := SendStopMsg \cup \{i\}$
    end
Event ManageIncident ⟨ordinary⟩ $\widehat{=}$
refines ManageIncident
    any
        i
    where
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd_seq_:  $i \in SetInitialAP$
        grd_self:  $i \notin ManageIncident$
    then
        act: $ManageIncident := ManageIncident \cup \{i\}$
    end
Event CloseIncident ⟨ordinary⟩ $\widehat{=}$
refines CloseIncident
    any
```

```
        i
    where
        grd_seq_FD:  $(i \in SendStopMsg \land i \in ManageIncident)$
        grd_self:  $i \notin CloseIncident$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
    then
        act: $CloseIncident := CloseIncident \cup \{i\}$
    end
Event IsDuplicate ⟨ordinary⟩ $\widehat{=}$
refines IsDuplicate
    any
        i
    where
        grd_seq_DuplicateInterrupt:  $i \in CreateIncident$
        grd_self:  $i \notin IsDuplicate$
        grd1:  $i \notin CloseIncident$
    then
        act: $IsDuplicate := IsDuplicate \cup \{i\}$
    end
END
```

B.2.6 Fifth Refinement: M5

```
MACHINE M5
REFINES M4
SEES C3
VARIABLES
      CreateIncident
      SelectLocation
      SelectType
      CompInfo
      SetInitialAP
      ResHandleIncident
      IncidentManaged
      RequestAdditionalRes
      SendStopMsg
      CloseIncident
      IsDuplicate
      ap
```

notRequired_ap

iLocation

iType

DisablePriorityChange

priority

iLocType

INVARIANTS

inv_CreateIncident_type: $CreateIncident \subseteq INCIDENT$

inv_SetActionPlan_type: $SetInitialAP \subseteq INCIDENT$

inv_RequestAdditionalResources_type: $RequestAdditionalRes \subseteq INCIDENT \times LR$

inv_SelectType__seq: $SelectType \subseteq CreateIncident$

inv_DisablePriorityChange_priority_seq: $DisablePriorityChange \subseteq SelectType$

inv_CompInfo__seq: $CompInfo \subseteq (SelectLocation \cap SelectType)$

inv_SetActionPlan_glu: $SetInitialAP = SetInitialAP$

inv_SetActionPlan_inSeq: $SetInitialAP \subseteq CompInfo$

inv_RequestAdditionalResources_planF_seq: $dom(RequestAdditionalRes) \subseteq SetInitialAP$

inv_RequestAdditionalRes_Rep: $\forall i \cdot i \in INCIDENT \Rightarrow RequestAdditionalRes[\{i\}] \subseteq LR$
Replicator invariant

inv_SendStopMsg_seq: $SendStopMsg \subseteq SetInitialAP$

inv_HandleIncident_type: $ResHandleIncident \subseteq INCIDENT \times LR$

inv_HandleIncident__seq: $dom(ResHandleIncident) \subseteq SetInitialAP$

inv_IncidentManaged__seq: $IncidentManaged \subseteq SetInitialAP$

inv_IncidentManaged_glu: $IncidentManaged = ManageIncident$

inv_CloseIncident_FD_seq: $CloseIncident \subseteq IncidentManaged \cap SendStopMsg$

inv_IsDuplicate_DuplicateInterrupt_seq: $IsDuplicate \subseteq CreateIncident$

inv_Interrupt: $partition((CloseIncident \cup IsDuplicate), CloseIncident, IsDuplicate)$

inv_par_rep: $\forall i \cdot i \in INCIDENT \Rightarrow ResHandleIncident[\{i\}] \subseteq ap[\{i\}]$

EVENTS

Initialisation

begin

act_CreateIncident: $CreateIncident := \varnothing$

act_SetInitialAP: $SetInitialAP := \varnothing$

act_CloseIncident: $CloseIncident := \varnothing$

act_IsDuplicate: $IsDuplicate := \varnothing$

act_SendStopMsg: $SendStopMsg := \varnothing$

act_ap: $ap := \varnothing$

act_notRequired_ap: $notRequired\_ap := \varnothing$

act_RequestAdditionalRes: $RequestAdditionalRes := \varnothing$

act2: $iLocation := \varnothing$

act_loc_type: $iLocType := \varnothing$

act3: $iType := \varnothing$

act_SelectLocation: $SelectLocation := \varnothing$

act_SelectType: $SelectType := \varnothing$

act_CompInfo: $CompInfo := \varnothing$

act_DisablePriorityChange: $DisablePriorityChange := \varnothing$

act4: $priority := \varnothing$

act_HandleIncident: $ResHandleIncident := \varnothing$

act_IncidentManaged: $IncidentManaged := \varnothing$

end

Event CreateIncident ⟨ordinary⟩ $\widehat{=}$

refines CreateIncident

any

i

where

grd_self: $i \notin CreateIncident$

then

act: $CreateIncident := CreateIncident \cup \{i\}$

end

Event SelectLocation ⟨ordinary⟩ $\widehat{=}$

refines SelectLocation

any

i

l

lt

where

grd_seq_: $i \in CreateIncident$

grd_self: $i \notin SelectLocation$

grd_DuplicateInterrupt: $i \notin IsDuplicate$

grd1: $l \in LOCATION$

grd2: $lt \in LOCATION\_TYPE$

then

act: $SelectLocation := SelectLocation \cup \{i\}$

act1: $iLocation(i) := l$

act2: $iLocType(i) := lt$

end

```
Event SelectType ⟨ordinary⟩ $\widehat{=}$
refines SelectType
    any
        i
        t
        p
    where
        grd_seq_:  $i \in CreateIncident$
        grd_self:  $i \notin SelectType$
        grd1:  $t \in INCIDENT\_TYPE$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd2:  $p = default\_priority(t)$
    then
        act: $SelectType := SelectType \cup \{i\}$
        act1: $iType(i) := t$
        act2: $priority(i) := p$
    end
Event ChangePriority ⟨ordinary⟩ $\widehat{=}$
refines ChangePriority
    any
        i
        p
    where
        grd_seq_priority:  $i \in SelectType$
        grd0_loop:  $i \notin DisablePriorityChange$
        grd1:  $p \in 1\,..\,3$
        grd2:  $p \neq priority(i)$
        grd3:  $i \notin CloseIncident$
            we can also set DisablePriorityChange in CloseIncident
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
    then
        act1: $priority(i) := p$
    end
Event DisablePriorityChange ⟨ordinary⟩ $\widehat{=}$
refines DisablePriorityChange
    any
        i
    where
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd_seq_priority:  $i \in SelectType$
        grd_self:  $i \notin DisablePriorityChange$
        grd1:  $i \notin CloseIncident$
            we can also set DisablePriorityChange in CloseIncident
    then
        act: $DisablePriorityChange := DisablePriorityChange \cup \{i\}$
    end
Event CompInfo ⟨ordinary⟩ $\widehat{=}$
refines CompInfo
    any
        i
    where
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd_seq_:  $(i \in SelectLocation \land i \in SelectType)$
        grd_self:  $i \notin CompInfo$
    then
        act: $CompInfo := CompInfo \cup \{i\}$
    end
Event SetInitialAP ⟨ordinary⟩ $\widehat{=}$
refines SetInitialAP
    any
        lrs
        i
        t
        l
    where
        grd_seq:  $i \in CompInfo$
        grd_self:  $i \notin SetInitialAP$
        grd1:  $t = iType(i)$
        grd2:  $l = iLocType(i)$
        grd3:  $lrs = PDA(t \mapsto l)$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
    then
        act: $SetInitialAP := SetInitialAP \cup \{i\}$
        act1: $ap := ap \cup (\{i\} \times lrs)$
    end
Event RequestAdditionalRes ⟨ordinary⟩ $\widehat{=}$
refines RequestAdditionalRes
    any
        i
        lr
    where
```

```
        grd_seq_planF:  $i \in SetInitialAP$
        grd_self:  $i \mapsto lr \notin RequestAdditionalRes$
        grd_InputExpression1:  $lr \in LR$
        grd0_par:  $i \notin SendStopMsg$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd1:  $lr \notin ap[\{i\}]$
    then
        act1: $ap := ap \cup \{i \mapsto lr\}$
        act: $RequestAdditionalRes := RequestAdditionalRes \cup \{i \mapsto lr\}$
    end
Event SendStopMsg ⟨ordinary⟩ $\widehat{=}$
refines SendStopMsg
    any
        i
        lrs
    where
        grd_self:  $i \notin SendStopMsg$
        grd_lrs:  $lrs \subseteq ap[\{i\}]$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd_seq_:  $i \in SetInitialAP$
    then
        act1: $notRequired\_ap := notRequired\_ap \cup (\{i\} \times lrs)$
        act: $SendStopMsg := SendStopMsg \cup \{i\}$
    end
Event ResHandleIncident ⟨ordinary⟩ $\widehat{=}$
    any
        lr
        i
    where
        grd0_par:  $i \notin IncidentManaged$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd_seq_:  $i \in SetInitialAP$
        grd_self:  $i \mapsto lr \notin ResHandleIncident$
        grd_InputExpression1:  $lr \in ap[\{i\}]$
    then
        act: $ResHandleIncident := ResHandleIncident \cup \{i \mapsto lr\}$
    end
Event IncidentManaged ⟨ordinary⟩ $\widehat{=}$
refines ManageIncident
    any
        i
        where
              grd_seq_: $i \in SetInitialAP$
              grd_self: $i \notin IncidentManaged$
              grd_DuplicateInterrupt: $i \notin IsDuplicate$
        then
              act: $IncidentManaged := IncidentManaged \cup \{i\}$
        end
Event CloseIncident ⟨ordinary⟩ $\widehat{=}$
refines CloseIncident
        any
              i
        where
              grd_DuplicateInterrupt: $i \notin IsDuplicate$
              grd_seq_FD: $(i \in SendStopMsg \land i \in IncidentManaged)$
              grd_self: $i \notin CloseIncident$
        then
              act: $CloseIncident := CloseIncident \cup \{i\}$
        end
Event IsDuplicate ⟨ordinary⟩ $\widehat{=}$
refines IsDuplicate
        any
              i
        where
              grd1: $i \notin CloseIncident$
              grd_seq_DuplicateInterrupt: $i \in CreateIncident$
              grd_self: $i \notin IsDuplicate$
        then
              act: $IsDuplicate := IsDuplicate \cup \{i\}$
        end
END
```

B.2.7 Sixth Refinement: M6

```
MACHINE M6
REFINES M5
SEES C3
VARIABLES
        CreateIncident
        SelectLocation
        SelectType
```

CompInfo

SetInitialAP

HandleIncident

IncidentManaged

RequestAdditionalRes

SendStopMsg

CloseIncident

IsDuplicate

ap

notRequired_ap

iLocation

iType

DisablePriorityChange

priority

ResLost

ResCompTask

iLocType

INVARIANTS

`inv_CreateIncident_type`: $CreateIncident \subseteq INCIDENT$

`inv_SetActionPlan_type`: $SetInitialAP \subseteq INCIDENT$

`inv_RequestAdditionalResources_type`: $RequestAdditionalRes \subseteq INCIDENT \times LR$

`inv_SelectType__seq`: $SelectType \subseteq CreateIncident$

`inv_DisablePriorityChange_priority_seq`: $DisablePriorityChange \subseteq SelectType$

`inv_CompInfo__seq`: $CompInfo \subseteq (SelectLocation \cap SelectType)$

`inv_SetActionPlan_glu`: $SetInitialAP = SetInitialAP$

`inv_SetActionPlan_inSeq`: $SetInitialAP \subseteq CompInfo$

`inv_RequestAdditionalResources_planF_seq`: $dom(RequestAdditionalRes) \subseteq SetInitialAP$

`inv_RequestAdditionalRes_Rep`: $\forall i \cdot i \in INCIDENT \Rightarrow RequestAdditionalRes[\{i\}] \subseteq LR$

Replicator invariant

`inv_SendStopMsg_seq`: $SendStopMsg \subseteq SetInitialAP$

`inv_HandleIncident_type`: $HandleIncident \subseteq INCIDENT \times LR$

`inv_HandleIncident__seq`: $dom(HandleIncident) \subseteq SetInitialAP$

`inv_CompTask_glu`: $ResCompTask = ResHandleIncident$

`inv_CompTask_seq`: $ResCompTask \subseteq HandleIncident$

`inv_IncidentManaged__seq`: $IncidentManaged \subseteq SetInitialAP$

inv_CloseIncident_FD_seq: $CloseIncident \subseteq IncidentManaged \cap SendStopMsg$

inv_IsDuplicate_DuplicateInterrupt_seq: $IsDuplicate \subseteq CreateIncident$

inv_ResLost_type: $ResLost \subseteq INCIDENT \times LR$

inv_ResLost: $dom(ResLost) \subseteq SetInitialAP$

inv_ResLost_retry: $partition((HandleIncident \cup ResLost), HandleIncident, ResLost)$

inv_IncidentManaged_par: $\forall i \cdot i \in IncidentManaged \Rightarrow HandleIncident[\{i\}] = ResCompTask[\{i\}]$

inv_par_rep: $\forall i \cdot i \in INCIDENT \Rightarrow HandleIncident[\{i\}] \subseteq ap[\{i\}]$

EVENTS

Initialisation

begin

act_CreateIncident: $CreateIncident := \varnothing$

act_SetInitialAP: $SetInitialAP := \varnothing$

act_CloseIncident: $CloseIncident := \varnothing$

act_IsDuplicate: $IsDuplicate := \varnothing$

act_SendStopMsg: $SendStopMsg := \varnothing$

act_ap: $ap := \varnothing$

act_notRequired_ap: $notRequired\_ap := \varnothing$

act_RequestAdditionalRes: $RequestAdditionalRes := \varnothing$

act2: $iLocation := \varnothing$

act_loc_type: $iLocType := \varnothing$

act3: $iType := \varnothing$

act_SelectLocation: $SelectLocation := \varnothing$

act_SelectType: $SelectType := \varnothing$

act_CompInfo: $CompInfo := \varnothing$

act_DisablePriorityChange: $DisablePriorityChange := \varnothing$

act4: $priority := \varnothing$

act_ResLost: $ResLost := \varnothing$

act_HandleIncident: $HandleIncident := \varnothing$

act_IncidentManaged: $IncidentManaged := \varnothing$

act_ResCompTask: $ResCompTask := \varnothing$

end

Event CreateIncident ⟨ordinary⟩ $\widehat{=}$

refines CreateIncident

any

i

where

grd_self: $i \notin CreateIncident$

then

```
        act: $CreateIncident := CreateIncident \cup \{i\}$
    end
Event SelectLocation ⟨ordinary⟩ $\widehat{=}$
refines SelectLocation
    any
        i
        l
        lt
    where
        grd_seq_:  $i \in CreateIncident$
        grd_self:  $i \notin SelectLocation$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd1:  $l \in LOCATION$
        grd2:  $lt \in LOCATION\_TYPE$
    then
        act: $SelectLocation := SelectLocation \cup \{i\}$
        act1: $iLocation(i) := l$
        act2: $iLocType(i) := lt$
    end
Event SelectType ⟨ordinary⟩ $\widehat{=}$
refines SelectType
    any
        i
        t
        p
    where
        grd_seq_:  $i \in CreateIncident$
        grd_self:  $i \notin SelectType$
        grd1:  $t \in INCIDENT\_TYPE$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd2:  $p = default\_priority(t)$
    then
        act: $SelectType := SelectType \cup \{i\}$
        act1: $iType(i) := t$
        act2: $priority(i) := p$
    end
Event ChangePriority ⟨ordinary⟩ $\widehat{=}$
refines ChangePriority
    any
        i
        p
    where
        grd_seq_priority:   $i \in SelectType$
        grd0_loop:   $i \notin DisablePriorityChange$
        grd1:   $p \in 1 \,..\, 3$
        grd2:   $p \neq priority(i)$
        grd3:   $i \notin CloseIncident$
            we can also set DisablePriorityChange in CloseIncident
        grd_DuplicateInterrupt:   $i \notin IsDuplicate$
    then
        act1: $priority(i) := p$
    end
Event DisablePriorityChange ⟨ordinary⟩ $\widehat{=}$
refines DisablePriorityChange
    any
        i
    where
        grd_DuplicateInterrupt:   $i \notin IsDuplicate$
        grd_seq_priority:   $i \in SelectType$
        grd_self:   $i \notin DisablePriorityChange$
        grd1:   $i \notin CloseIncident$
            we can also set DisablePriorityChange in CloseIncident
    then
        act: $DisablePriorityChange := DisablePriorityChange \cup \{i\}$
    end
Event CompInfo ⟨ordinary⟩ $\widehat{=}$
refines CompInfo
    any
        i
    where
        grd_DuplicateInterrupt:   $i \notin IsDuplicate$
        grd_seq_:   $(i \in SelectLocation \land i \in SelectType)$
        grd_self:   $i \notin CompInfo$
    then
        act: $CompInfo := CompInfo \cup \{i\}$
    end
Event SetInitialAP ⟨ordinary⟩ $\widehat{=}$
refines SetInitialAP
    any
        lrs
```

```
            i
            t
            l
      where
            grd_seq:   $i \in CompInfo$
            grd_self:   $i \notin SetInitialAP$
            grd1:   $t = iType(i)$
            grd2:   $l = iLocType(i)$
            grd3:   $lrs = PDA(t \mapsto l)$
            grd_DuplicateInterrupt:   $i \notin IsDuplicate$
      then
            act: $SetInitialAP := SetInitialAP \cup \{i\}$
            act1: $ap := ap \cup (\{i\} \times lrs)$
      end
Event RequestAdditionalRes ⟨ordinary⟩ $\widehat{=}$
refines RequestAdditionalRes
      any
            i
            lr
      where
            grd_seq_planF:   $i \in SetInitialAP$
            grd_self:   $i \mapsto lr \notin RequestAdditionalRes$
            grd_InputExpression1:   $lr \in LR$
            grd0_par:   $i \notin SendStopMsg$
            grd_DuplicateInterrupt:   $i \notin IsDuplicate$
            grd1:   $lr \notin ap[\{i\}]$
      then
            act1: $ap := ap \cup \{i \mapsto lr\}$
            act: $RequestAdditionalRes := RequestAdditionalRes \cup \{i \mapsto lr\}$
      end
Event SendStopMsg ⟨ordinary⟩ $\widehat{=}$
refines SendStopMsg
      any
            i
            lrs
      where
            grd_self:   $i \notin SendStopMsg$
            grd_lrs:   $lrs \subseteq ap[\{i\}]$
            grd_DuplicateInterrupt:   $i \notin IsDuplicate$
            grd_seq_:   $i \in SetInitialAP$
      then
```

```
        act1: $notRequired\_ap := notRequired\_ap \cup (\{i\} \times lrs)$
        act: $SendStopMsg := SendStopMsg \cup \{i\}$
    end
Event HandleIncident ⟨ordinary⟩ $\widehat{=}$
    any
        lr
        i
    where
        grd_ResLostInterrupt: $i \mapsto lr \notin ResLost$
        grd_DuplicateInterrupt: $i \notin IsDuplicate$
        grd_seq_: $i \in SetInitialAP$
        grd_self: $i \mapsto lr \notin HandleIncident$
        grd_InputExpression1: $lr \in ap[\{i\}]$
        grd0_par: $i \notin IncidentManaged$
    then
        act: $HandleIncident := HandleIncident \cup \{i \mapsto lr\}$
    end
Event ResCompTask ⟨ordinary⟩ $\widehat{=}$
refines ResHandleIncident
    any
        lr
        i
    where
        grd_self: $i \mapsto lr \notin ResCompTask$
        grd_DuplicateInterrupt: $i \notin IsDuplicate$
        grd_seq_: $i \mapsto lr \in HandleIncident$
    then
        act: $ResCompTask := ResCompTask \cup \{i \mapsto lr\}$
    end
Event ResLost ⟨ordinary⟩ $\widehat{=}$
    any
        i
        lr
    where
        grd_retry: $i \mapsto lr \notin HandleIncident$
        grd_self: $i \mapsto lr \notin ResLost$
        grd_seq: $i \in SetInitialAP$
        grd_par: $i \notin IncidentManaged$
        grd_DuplicateInterrupt: $i \notin IsDuplicate$
    then
        act: $ResLost := ResLost \cup \{i \mapsto lr\}$
```

act1: $HandleIncident := HandleIncident \setminus \{i \mapsto lr\}$

end

Event Reset_ResLost ⟨ordinary⟩ $\widehat{=}$

any

i

lr

where

grd_seq: $i \mapsto lr \in ResLost$

then

act: $ResLost := ResLost \setminus \{i \mapsto lr\}$

end

Event IncidentManaged ⟨ordinary⟩ $\widehat{=}$

refines IncidentManaged

any

i

where

grd_par: $HandleIncident[\{i\}] = ResCompTask[\{i\}]$

grd_self: $i \notin IncidentManaged$

grd_DuplicateInterrupt: $i \notin IsDuplicate$

grd_seq_: $i \in SetInitialAP$

then

act: $IncidentManaged := IncidentManaged \cup \{i\}$

end

Event CloseIncident ⟨ordinary⟩ $\widehat{=}$

refines CloseIncident

any

i

where

grd_DuplicateInterrupt: $i \notin IsDuplicate$

grd_seq_FD: $(i \in SendStopMsg \wedge i \in IncidentManaged)$

grd_self: $i \notin CloseIncident$

then

act: $CloseIncident := CloseIncident \cup \{i\}$

end

Event IsDuplicate ⟨ordinary⟩ $\widehat{=}$

refines IsDuplicate

any

i

where

grd1: $i \notin CloseIncident$

```
            grd_seq_DuplicateInterrupt:  $i \in CreateIncident$
            grd_self:  $i \notin IsDuplicate$
      then
            act: $IsDuplicate := IsDuplicate \cup \{i\}$
      end
END
```

B.2.8 Seventh Refinement: M7

MACHINE M7

REFINES M6

SEES C3

VARIABLES

CreateIncident

SelectLocation

SelectType

CompInfo

SetInitialAP

AllocateRes

ResAttend

DeallocateRes

StopMsgReceived

IncidentManaged

RequestAdditionalRes

SendStopMsg

CloseIncident

IsDuplicate

ap

notRequired_ap

iLocation

iType

DisablePriorityChange

priority

RemoveAlloc

RemoveBrokenRes

ResCompTask

iLocType

INVARIANTS

`inv_CreateIncident_type`: $CreateIncident \subseteq INCIDENT$

`inv_SetActionPlan_type`: $SetInitialAP \subseteq INCIDENT$

`inv_RequestAdditionalResources_type`: $RequestAdditionalRes \subseteq INCIDENT \times LR$

`inv_SelectType__seq`: $SelectType \subseteq CreateIncident$

`inv_DisablePriorityChange_priority_seq`: $DisablePriorityChange \subseteq SelectType$

`inv_CompInfo__seq`: $CompInfo \subseteq (SelectLocation \cap SelectType)$

`inv_SetActionPlan_glu`: $SetInitialAP = SetInitialAP$

`inv_SetActionPlan_inSeq`: $SetInitialAP \subseteq CompInfo$

`inv_RequestAdditionalResources_planF_seq`: $dom(RequestAdditionalRes) \subseteq SetInitialAP$

`inv_RequestAdditionalRes_Rep`: $\forall i \cdot i \in INCIDENT \Rightarrow RequestAdditionalRes[\{i\}] \subseteq LR$

Replicator invariant

`inv_SendStopMsg_seq`: $SendStopMsg \subseteq SetInitialAP$

`inv_AllocateRes_type`: $AllocateRes \subseteq INCIDENT \times LR$

`inv_AllocateRes_allocation_seq`: $dom(AllocateRes) \subseteq SetInitialAP$

`inv_Attend_allocation_seq`: $ResAttend \subseteq AllocateRes$

`inv_StopMsgReceived_seq`: $StopMsgReceived \subseteq AllocateRes$

`inv_DeallocateRes__seq`: $DeallocateRes \subseteq (ResAttend \cup StopMsgReceived)$

`inv_DeallocateRes_glu`: $DeallocateRes = HandleIncident$

`inv2_xorFD`: $partition((ResAttend \cup StopMsgReceived), ResAttend, StopMsgReceived)$

`inv_CompTask_seq`: $ResCompTask \subseteq DeallocateRes$

`inv_IncidentManaged__seq`: $IncidentManaged \subseteq SetInitialAP$

`inv_CloseIncident_FD_seq`: $CloseIncident \subseteq IncidentManaged \cap SendStopMsg$

`inv_IsDuplicate_DuplicateInterrupt_seq`: $IsDuplicate \subseteq CreateIncident$

`inv_ResLost_gluing`: $partition(ResLost, RemoveAlloc, RemoveBrokenRes)$

`inv_RemoveAlloc_seq`: $dom(RemoveAlloc) \subseteq SetInitialAP$

`inv_RemoveBrokenRes_seq`: $dom(RemoveBrokenRes) \subseteq SetInitialAP$

`inv_ResLost_retry`: $partition((DeallocateRes \cup (RemoveAlloc \cup RemoveBrokenRes)), DeallocateRes, (Remove$
$RemoveBrokenRes))$

`in_rep`: $\forall i \cdot i \in INCIDENT \Rightarrow AllocateRes[\{i\}] \subseteq ap[\{i\}]$

`inv_par`: $\forall i \cdot i \in IncidentManaged \Rightarrow AllocateRes[\{i\}] = ResCompTask[\{i\}]$

`inv_managed`: $\forall i \cdot i \in CloseIncident \Rightarrow ap[\{i\}] \setminus notRequired\_ap[\{i\}] \subseteq DeallocateRes[\{i\}]$

manually added

EVENTS

Initialisation

begin

act_CreateIncident: $CreateIncident := \varnothing$

act_SetInitialAP: $SetInitialAP := \varnothing$

act_CloseIncident: $CloseIncident := \varnothing$

act_IsDuplicate: $IsDuplicate := \varnothing$

act_SendStopMsg: $SendStopMsg := \varnothing$

act_ap: $ap := \varnothing$

act_notRequired_ap: $notRequired\_ap := \varnothing$

act_RequestAdditionalRes: $RequestAdditionalRes := \varnothing$

act2: $iLocation := \varnothing$

act_loc_type: $iLocType := \varnothing$

act3: $iType := \varnothing$

act_SelectLocation: $SelectLocation := \varnothing$

act_SelectType: $SelectType := \varnothing$

act_CompInfo: $CompInfo := \varnothing$

act_DisablePriorityChange: $DisablePriorityChange := \varnothing$

act4: $priority := \varnothing$

act_StopMsgReceived: $StopMsgReceived := \varnothing$

act_AllocateRes: $AllocateRes := \varnothing$

act_Attend: $ResAttend := \varnothing$

act_DeallocateRes: $DeallocateRes := \varnothing$

act_IncidentManaged: $IncidentManaged := \varnothing$

act_RemoveAlloc: $RemoveAlloc := \varnothing$

act_RemoveBrokenRes: $RemoveBrokenRes := \varnothing$

act_ResCompTask: $ResCompTask := \varnothing$

end

Event CreateIncident ⟨ordinary⟩ $\widehat{=}$

refines CreateIncident

any

i

where

grd_self: $i \notin CreateIncident$

then

act: $CreateIncident := CreateIncident \cup \{i\}$

end

Event SelectLocation ⟨ordinary⟩ $\widehat{=}$

refines SelectLocation

any

i

l

```
        lt
    where
        grd_seq_:  $i \in CreateIncident$
        grd_self:  $i \notin SelectLocation$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd1:  $l \in LOCATION$
        grd2:  $lt \in LOCATION\_TYPE$
    then
        act: $SelectLocation := SelectLocation \cup \{i\}$
        act1: $iLocation(i) := l$
        act2: $iLocType(i) := lt$
    end
Event SelectType ⟨ordinary⟩ $\widehat{=}$
refines SelectType
    any
        i
        t
        p
    where
        grd_seq_:  $i \in CreateIncident$
        grd_self:  $i \notin SelectType$
        grd1:  $t \in INCIDENT\_TYPE$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd2:  $p = default\_priority(t)$
    then
        act: $SelectType := SelectType \cup \{i\}$
        act1: $iType(i) := t$
        act2: $priority(i) := p$
    end
Event ChangePriority ⟨ordinary⟩ $\widehat{=}$
refines ChangePriority
    any
        i
        p
    where
        grd_seq_priority:  $i \in SelectType$
        grd0_loop:  $i \notin DisablePriorityChange$
        grd1:  $p \in 1 .. 3$
        grd2:  $p \neq priority(i)$
        grd3:  $i \notin CloseIncident$
          we can also set DisablePriorityChange in CloseIncident
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
    then
        act1: $priority(i) := p$
    end
Event DisablePriorityChange ⟨ordinary⟩ $\widehat{=}$
refines DisablePriorityChange
    any
        i
    where
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd_seq_priority:  $i \in SelectType$
        grd_self:  $i \notin DisablePriorityChange$
        grd1:  $i \notin CloseIncident$
            we can also set DisablePriorityChange in CloseIncident
    then
        act: $DisablePriorityChange := DisablePriorityChange \cup \{i\}$
    end
Event CompInfo ⟨ordinary⟩ $\widehat{=}$
refines CompInfo
    any
        i
    where
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd_seq_:  $(i \in SelectLocation \land i \in SelectType)$
        grd_self:  $i \notin CompInfo$
    then
        act: $CompInfo := CompInfo \cup \{i\}$
    end
Event SetInitialAP ⟨ordinary⟩ $\widehat{=}$
refines SetInitialAP
    any
        lrs
        i
        t
        l
    where
        grd_seq:  $i \in CompInfo$
        grd_self:  $i \notin SetInitialAP$
        grd1:  $t = iType(i)$
        grd2:  $l = iLocType(i)$
        grd3:  $lrs = PDA(t \mapsto l)$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
    then
        act: $SetInitialAP := SetInitialAP \cup \{i\}$
        act1: $ap := ap \cup (\{i\} \times lrs)$
    end
Event RequestAdditionalRes ⟨ordinary⟩ $\widehat{=}$
refines RequestAdditionalRes
    any
        i
        lr
    where
        grd_seq_planF:  $i \in SetInitialAP$
        grd_self:  $i \mapsto lr \notin RequestAdditionalRes$
        grd_InputExpression1:  $lr \in LR \setminus ap[\{i\}]$
        grd0_par:  $i \notin SendStopMsg$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd1:  $lr \notin ap[\{i\}]$
            added manually
    then
        act1: $ap := ap \cup \{i \mapsto lr\}$
        act: $RequestAdditionalRes := RequestAdditionalRes \cup \{i \mapsto lr\}$
    end
Event SendStopMsg ⟨ordinary⟩ $\widehat{=}$
refines SendStopMsg
    any
        i
        lrs
    where
        grd_self:  $i \notin SendStopMsg$
        grd_lrs:  $lrs = ap[\{i\}] \setminus ResAttend[\{i\}]$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd_seq_:  $i \in SetInitialAP$
    then
        act1: $notRequired\_ap := notRequired\_ap \cup (\{i\} \times lrs)$
        act: $SendStopMsg := SendStopMsg \cup \{i\}$
    end
Event AllocateRes ⟨ordinary⟩ $\widehat{=}$
    any
        i
        lr
    where
        grdResLostRetry:   $i \mapsto lr \notin (RemoveAlloc \cup RemoveBrokenRes)$
        grd_seq_allocation:   $i \in SetInitialAP$
        grd_self:   $i \mapsto lr \notin AllocateRes$
        grd_InputExpression1:   $lr \in ap[\{i\}]$
        grd0_par:   $i \notin IncidentManaged$
        grd_DuplicateInterrupt:   $i \notin IsDuplicate$
        grd1:   $i \mapsto lr \notin notRequired\_ap$
    then
        act: $AllocateRes := AllocateRes \cup \{i \mapsto lr\}$
    end
Event UpdateResLocation ⟨ordinary⟩ $\widehat{=}$
    any
        i
        lr
    where
        grdResLostRetry:   $i \mapsto lr \notin (RemoveAlloc \cup RemoveBrokenRes)$
        grd_seq_allocation:   $i \mapsto lr \in AllocateRes$
        grd0_loop:   $i \mapsto lr \notin (ResAttend \cup StopMsgReceived)$
            @grdResLostRetry i ↦ lr ∉ ResLost
        grd_DuplicateInterrupt:   $i \notin IsDuplicate$
        grd1:   $i \mapsto lr \notin notRequired\_ap$
    then
        skip
    end
Event ReallocateQuickerRes ⟨ordinary⟩ $\widehat{=}$
    any
        i
        lr
    where
        grdResLostRetry:   $i \mapsto lr \notin (RemoveAlloc \cup RemoveBrokenRes)$
        grd_seq_allocation:   $i \mapsto lr \in AllocateRes$
        grd0_loop:   $i \mapsto lr \notin (ResAttend \cup StopMsgReceived)$
            @grdResLostRetry i ↦ lr ∉ ResLost
        grd_DuplicateInterrupt:   $i \notin IsDuplicate$
        grd1:   $i \mapsto lr \notin notRequired\_ap$
    then
        skip
    end
Event StopMsgReceived ⟨ordinary⟩ $\widehat{=}$
    any
```

```
        lr
        i
    where
        grd_InputExpression1:  $lr \in ap[\{i\}]$
        grd_seq_allocation:  $i \mapsto lr \in AllocateRes$
        grd_self:  $i \mapsto lr \notin StopMsgReceived$
            @grdResLostRetry i ↦ lr ∉ ResLost
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd_xor:  $i \mapsto lr \notin ResAttend$
        grd1:  $i \mapsto lr \in notRequired\_ap$
        grdResLostRetry:  $i \mapsto lr \notin (RemoveAlloc \cup RemoveBrokenRes)$
        grd_seq_:  $i \in SetInitialAP$
    then
        act:  $StopMsgReceived := StopMsgReceived \cup \{i \mapsto lr\}$
    end
Event ResAttend ⟨ordinary⟩ $\widehat{=}$
    any
        i
        lr
    where
        grdResLostRetry:  $i \mapsto lr \notin (RemoveAlloc \cup RemoveBrokenRes)$
        grd_InputExpression1:  $lr \in ap[\{i\}]$
        grd_seq_allocation:  $i \mapsto lr \in AllocateRes$
        grd_self:  $i \mapsto lr \notin ResAttend$
            @grdResLostRetry i ↦ lr ∉ ResLost
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd_xor:  $i \mapsto lr \notin StopMsgReceived$
        grd_seq_:  $i \in SetInitialAP$
        grd1:  $i \mapsto lr \notin notRequired\_ap$
    then
        act:  $ResAttend := ResAttend \cup \{i \mapsto lr\}$
    end
Event DeallocateRes ⟨ordinary⟩ $\widehat{=}$
refines HandleIncident
    any
        lr
        i
    where
        grdResLostRetry:  $i \mapsto lr \notin (RemoveAlloc \cup RemoveBrokenRes)$
        grd_seq_:  $i \mapsto lr \in (ResAttend \cup StopMsgReceived)$
        grd_self:   $i \mapsto lr \notin DeallocateRes$
            @grdResLostRetry i ↦ lr ∉ ResLost
        grd_DuplicateInterrupt:   $i \notin IsDuplicate$
            @grd1 lr ∈ ap[{i}] // GRD PO//@grdResLostRetry i ↦ lr ∉ ResLost
    then
        act:  $DeallocateRes := DeallocateRes \cup \{i \mapsto lr\}$
    end
Event IncidentManaged ⟨ordinary⟩ ≙
refines IncidentManaged
    any
        i
    where
        grd_par:   $AllocateRes[\{i\}] = ResCompTask[\{i\}]$
        grd_self:   $i \notin IncidentManaged$
        grd_DuplicateInterrupt:   $i \notin IsDuplicate$
        grd_seq_:   $i \in SetInitialAP$
    then
        act:  $IncidentManaged := IncidentManaged \cup \{i\}$
    end
Event RemoveAlloc ⟨ordinary⟩ ≙
refines ResLost
    any
        i
        lr
    where
        grd_DuplicateInterrupt:   $i \notin IsDuplicate$
        grd_self:   $i \mapsto lr \notin RemoveAlloc$
        grd_seq:   $i \in SetInitialAP$
        grd_par:   $i \notin IncidentManaged$
        grd_xor:   $i \mapsto lr \notin RemoveBrokenRes$
        grd1:   $i \mapsto lr \in AllocateRes$
        grd2:   $i \mapsto lr \notin DeallocateRes$
    then
        act:  $RemoveAlloc := RemoveAlloc \cup \{i \mapsto lr\}$
        act1:  $AllocateRes := AllocateRes \setminus \{i \mapsto lr\}$
        act2:  $ResAttend := ResAttend \setminus \{i \mapsto lr\}$
        act3:  $StopMsgReceived := StopMsgReceived \setminus \{i \mapsto lr\}$
        act4:  $DeallocateRes := DeallocateRes \setminus \{i \mapsto lr\}$
    end
Event RemoveBrokenRes ⟨ordinary⟩ ≙
refines ResLost
```

any

i

lr

where

grd_DuplicateInterrupt: $i \notin IsDuplicate$

grd_self: $i \mapsto lr \notin RemoveBrokenRes$

grd_seq: $i \in SetInitialAP$

grd_par: $i \notin IncidentManaged$

grd_xor: $i \mapsto lr \notin RemoveAlloc$

grd1: $i \mapsto lr \in AllocateRes$

grd2: $i \mapsto lr \notin DeallocateRes$

then

act: $RemoveBrokenRes := RemoveBrokenRes \cup \{i \mapsto lr\}$

act1: $AllocateRes := AllocateRes \setminus \{i \mapsto lr\}$

act2: $ResAttend := ResAttend \setminus \{i \mapsto lr\}$

act3: $StopMsgReceived := StopMsgReceived \setminus \{i \mapsto lr\}$

act4: $DeallocateRes := DeallocateRes \setminus \{i \mapsto lr\}$

end

Event Reset_ResLost ⟨ordinary⟩ $\widehat{=}$

refines Reset_ResLost

any

i

lr

where

grd_seq: $i \mapsto lr \in (RemoveBrokenRes \cup RemoveAlloc)$

then

act1: $RemoveAlloc := RemoveAlloc \setminus \{i \mapsto lr\}$

act2: $RemoveBrokenRes := RemoveBrokenRes \setminus \{i \mapsto lr\}$

end

Event ResCompTask ⟨ordinary⟩ $\widehat{=}$

refines ResCompTask

any

lr

i

where

grd_self: $i \mapsto lr \notin ResCompTask$

grd_DuplicateInterrupt: $i \notin IsDuplicate$

grd_seq_: $i \mapsto lr \in DeallocateRes$

then

act: $ResCompTask := ResCompTask \cup \{i \mapsto lr\}$

end

Event CloseIncident ⟨ordinary⟩ $\widehat{=}$
refines CloseIncident
    any
        i
    where
        `grd1`: $ap[\{i\}] \setminus notRequired\_ap[\{i\}] \subseteq DeallocateRes[\{i\}]$
            to ensure all action plan items are satisfied
        `grd_seq_FD`: $(i \in SendStopMsg \land i \in IncidentManaged)$
        `grd_self`: $i \notin CloseIncident$
        `grd_DuplicateInterrupt`: $i \notin IsDuplicate$
    then
        `act`: $CloseIncident := CloseIncident \cup \{i\}$
    end

Event IsDuplicate ⟨ordinary⟩ $\widehat{=}$
refines IsDuplicate
    any
        i
    where
        `grd1`: $i \notin CloseIncident$
        `grd_seq_DuplicateInterrupt`: $i \in CreateIncident$
        `grd_self`: $i \notin IsDuplicate$
    then
        `act`: $IsDuplicate := IsDuplicate \cup \{i\}$
    end
END

B.2.9 Eighth Refinement: M8

MACHINE M8
REFINES M7
SEES C4
VARIABLES
    CreateIncident
    SelectLocation
    SelectType
    CompInfo
    SetInitialAP
    AllocateRes
    ResAttend

DeallocateRes

StopMsgReceived

IncidentManaged

RequestAdditionalRes

SendStopMsg

CloseIncident

IsDuplicate

ap

notRequired_ap

iLocation

iLocType

iType

DisablePriorityChange

priority

RemoveAlloc

RemoveBrokenRes

alloc allocate physical resource

alloc_LR relate lr to pr

HandleDupAlloc

CloseDuplicate

ResCompTask

INVARIANTS

`inv_alloc_type`: $alloc \in PR \twoheadrightarrow INCIDENT$
manually

`inv_alloc_seq`: $ran(alloc) \subseteq SetInitialAP$

`inv_alloc_LR_type`: $alloc\_LR \in PR \twoheadrightarrow LR$
manually

`inv_alloc_LR_PR_TYp`: $\forall pr, lr \cdot pr \in dom(alloc\_LR) \wedge alloc\_LR(pr) = lr \Rightarrow ptype(pr) = ltype(lr)$

`inv_alloc_alloc_LR`: $dom(alloc) = dom(alloc\_LR)$

`inv_HandleAllocDup_type`: $HandleDupAlloc \subseteq INCIDENT \times LR$

`inv_HandleAllocDup_seq`: $dom(HandleDupAlloc) \subseteq IsDuplicate$

`inv_HandleAllocDup_rep`: $\forall i \cdot i \in INCIDENT \Rightarrow HandleDupAlloc[\{i\}] \subseteq AllocateRes[\{i\}]$

`inv_Close_Duplicate_seq`: $\forall i \cdot i \in CloseDuplicate \Rightarrow HandleDupAlloc[\{i\}] = AllocateRes[\{i\}]$

`inv_alloc_close`: $\forall i \cdot i \in (CloseIncident \cup CloseDuplicate) \Rightarrow i \notin ran(alloc)$

inv1: $CloseDuplicate \subseteq IsDuplicate$

Added manually to discharge some proofs

EVENTS

Initialisation

begin

act_CreateIncident: $CreateIncident := \varnothing$

act_SetInitialAP: $SetInitialAP := \varnothing$

act_CloseIncident: $CloseIncident := \varnothing$

act_IsDuplicate: $IsDuplicate := \varnothing$

act_SendStopMsg: $SendStopMsg := \varnothing$

act_ap: $ap := \varnothing$

act_notRequired_ap: $notRequired\_ap := \varnothing$

act_RequestAdditionalRes: $RequestAdditionalRes := \varnothing$

act2: $iLocation := \varnothing$

act3: $iType := \varnothing$

act_loc_type: $iLocType := \varnothing$

act_SelectLocation: $SelectLocation := \varnothing$

act_SelectType: $SelectType := \varnothing$

act_CompInfo: $CompInfo := \varnothing$

act_DisablePriorityChange: $DisablePriorityChange := \varnothing$

act4: $priority := \varnothing$

act_StopMsgReceived: $StopMsgReceived := \varnothing$

act_AllocateRes: $AllocateRes := \varnothing$

act_Attend: $ResAttend := \varnothing$

act_DeallocateRes: $DeallocateRes := \varnothing$

act_IncidentManaged: $IncidentManaged := \varnothing$

act_RemoveAlloc: $RemoveAlloc := \varnothing$

act_RemoveBrokenRes: $RemoveBrokenRes := \varnothing$

act_alloc: $alloc := \varnothing$

act_alloc_LR: $alloc\_LR := \varnothing$

act_HandleDupAlloc: $HandleDupAlloc := \varnothing$

act_CloseDuplicate: $CloseDuplicate := \varnothing$

act_ResCompTask: $ResCompTask := \varnothing$

end

Event CreateIncident ⟨ordinary⟩ $\widehat{=}$

refines CreateIncident

any

i

where

grd_self: $i \notin CreateIncident$

then

act: $CreateIncident := CreateIncident \cup \{i\}$

end

Event SelectLocation ⟨ordinary⟩ $\widehat{=}$

refines SelectLocation

any

i

l

lt

where

grd_seq_: $i \in CreateIncident$

grd_self: $i \notin SelectLocation$

grd_DuplicateInterrupt: $i \notin IsDuplicate$

grd1: $l \in LOCATION$

grd2: $lt \in LOCATION\_TYPE$

then

act: $SelectLocation := SelectLocation \cup \{i\}$

act1: $iLocation(i) := l$

act2: $iLocType(i) := lt$

end

Event SelectType ⟨ordinary⟩ $\widehat{=}$

refines SelectType

any

i

t

p

where

grd_seq_: $i \in CreateIncident$

grd_self: $i \notin SelectType$

grd1: $t \in INCIDENT\_TYPE$

grd_DuplicateInterrupt: $i \notin IsDuplicate$

grd2: $p = default\_priority(t)$

then

act: $SelectType := SelectType \cup \{i\}$

act1: $iType(i) := t$

act2: $priority(i) := p$

end

Event ChangePriority ⟨ordinary⟩ $\widehat{=}$

refines ChangePriority

any

i

```
        p
    where
        grd_seq_priority:  $i \in SelectType$
        grd0_loop:  $i \notin DisablePriorityChange$
        grd1:  $p \in 1\,..\,3$
        grd2:  $p \neq priority(i)$
        grd3:  $i \notin CloseIncident$
            we can also set DisablePriorityChange in CloseIncident
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
    then
        act1: $priority(i) := p$
    end
Event DisablePriorityChange ⟨ordinary⟩ $\widehat{=}$
refines DisablePriorityChange
    any
        i
    where
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd_seq_priority:  $i \in SelectType$
        grd_self:  $i \notin DisablePriorityChange$
        grd1:  $i \notin CloseIncident$
            we can also set DisablePriorityChange in CloseIncident
    then
        act: $DisablePriorityChange := DisablePriorityChange \cup \{i\}$
    end
Event CompInfo ⟨ordinary⟩ $\widehat{=}$
refines CompInfo
    any
        i
    where
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd_seq_:  $(i \in SelectLocation \land i \in SelectType)$
        grd_self:  $i \notin CompInfo$
    then
        act: $CompInfo := CompInfo \cup \{i\}$
    end
Event SetInitialAP ⟨ordinary⟩ $\widehat{=}$
refines SetInitialAP
    any
        lrs
        i
        t
        l
    where
        grd_seq:  $i \in CompInfo$
        grd_self:  $i \notin SetInitialAP$
        grd1:  $t = iType(i)$
        grd2:  $l = iLocType(i)$
        grd3:  $lrs = PDA(t \mapsto l)$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
    then
        act: $SetInitialAP := SetInitialAP \cup \{i\}$
        act1: $ap := ap \cup (\{i\} \times lrs)$
    end
Event RequestAdditionalRes ⟨ordinary⟩ ≙
refines RequestAdditionalRes
    any
        i
        lr
    where
        grd_seq_planF:  $i \in SetInitialAP$
        grd_self:  $i \mapsto lr \notin RequestAdditionalRes$
        grd_InputExpression1:  $lr \in LR$
        grd0_par:  $i \notin SendStopMsg$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd1:  $lr \notin ap[\{i\}]$
    then
        act1: $ap := ap \cup \{i \mapsto lr\}$
        act: $RequestAdditionalRes := RequestAdditionalRes \cup \{i \mapsto lr\}$
    end
Event SendStopMsg ⟨ordinary⟩ ≙
refines SendStopMsg
    any
        i
        lrs
    where
        grd_lrs:  $lrs = ap[\{i\}] \setminus ResAttend[\{i\}]$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd_seq_:  $i \in SetInitialAP$
        grd_self:  $i \notin SendStopMsg$
    then
```

```
        act1: $notRequired\_ap := notRequired\_ap \cup (\{i\} \times lrs)$
        act: $SendStopMsg := SendStopMsg \cup \{i\}$
    end
Event AllocateRes ⟨ordinary⟩ $\widehat{=}$
refines AllocateRes
    any
        i
        lr
        pr
    where
        grd_seq_allocation: $i \in SetInitialAP$
        grd_self: $i \mapsto lr \notin AllocateRes$
        grd_InputExpression1: $lr \in ap[\{i\}]$
        grd0_par: $i \notin IncidentManaged$
        grd_DuplicateInterrupt: $i \notin IsDuplicate$
        grd1: $i \mapsto lr \notin notRequired\_ap$
        grdResLostRetry: $i \mapsto lr \notin (RemoveAlloc \cup RemoveBrokenRes)$
        grd2: $ptype(pr) = ltype(lr)$
        grd3: $pr \in dom(alloc) \Rightarrow priority(i) < priority(alloc(pr))$
    then
        act: $AllocateRes := AllocateRes \cup \{i \mapsto lr\}$
        act1: $alloc(pr) := i$
        act2: $alloc\_LR(pr) := lr$
    end
Event UpdateResLocation ⟨ordinary⟩ $\widehat{=}$
refines UpdateResLocation
    any
        i
        lr
        pr
    where
        grd_seq_allocation: $i \mapsto lr \in AllocateRes$
        grd0_loop: $i \mapsto lr \notin (ResAttend \cup StopMsgReceived)$
          @grdResLostRetry i ↦ lr ∉ ResLost
        grd_DuplicateInterrupt: $i \notin IsDuplicate$
        grd1: $i \mapsto lr \notin notRequired\_ap$
        grdResLostRetry: $i \mapsto lr \notin (RemoveAlloc \cup RemoveBrokenRes)$
        grd2: $pr \in (alloc^{-1}[\{i\}] \cap alloc\_LR^{-1}[\{lr\}])$
    then
        skip
    end
```

Event ReallocateQuickerRes ⟨ordinary⟩ $\widehat{=}$
refines ReallocateQuickerRes
  any
    i
    lr
    pr
    pr_quicker
  where
    grd_seq_allocation: $i \mapsto lr \in AllocateRes$
    grd0_loop: $i \mapsto lr \notin (ResAttend \cup StopMsgReceived)$
      @grdResLostRetry i $\mapsto$ lr $\notin$ ResLost
    grd_DuplicateInterrupt: $i \notin IsDuplicate$
    grd1: $i \mapsto lr \notin notRequired\_ap$
    grdResLostRetry: $i \mapsto lr \notin (RemoveAlloc \cup RemoveBrokenRes)$
    grd2: $pr \in (alloc^{-1}[\{i\}] \cap alloc\_LR^{-1}[\{lr\}])$
    grd3: $pr\_quicker \notin dom(alloc)$
    grd4: $ptype(pr\_quicker) = ltype(lr)$
      @grd5 ptype(pr) = ltype(lr)
  then
    act1: $alloc := (\{pr\} \lhd alloc) \cup \{pr\_quicker \mapsto i\}$
    act2: $alloc\_LR := (\{pr\} \lhd alloc\_LR) \cup \{pr\_quicker \mapsto lr\}$
  end

Event StopMsgReceived ⟨ordinary⟩ $\widehat{=}$
refines StopMsgReceived
  any
    lr
    i
    pr
  where
    grd_seq_allocation: $i \mapsto lr \in AllocateRes$
    grd_self: $i \mapsto lr \notin StopMsgReceived$
      @grdResLostRetry i $\mapsto$ lr $\notin$ ResLost
    grd_DuplicateInterrupt: $i \notin IsDuplicate$
    grd_xor: $i \mapsto lr \notin ResAttend$
    grd1: $i \mapsto lr \in notRequired\_ap$
    grdResLostRetry: $i \mapsto lr \notin (RemoveAlloc \cup RemoveBrokenRes)$
    grd_seq_: $i \in SetInitialAP$
    grd_InputExpression1: $lr \in ap[\{i\}]$
    grd2: $pr \in (alloc^{-1}[\{i\}] \cap alloc\_LR^{-1}[\{lr\}])$
  then
    act: $StopMsgReceived := StopMsgReceived \cup \{i \mapsto lr\}$

```
        end
Event ResAttend ⟨ordinary⟩ $\widehat{=}$
refines ResAttend
        any
                i
                lr
                pr
        where
                grd_InputExpression1: $lr \in ap[\{i\}]$
                grd_seq_allocation: $i \mapsto lr \in AllocateRes$
                grd_self: $i \mapsto lr \notin ResAttend$
                    @grdResLostRetry i ↦ lr ∉ ResLost
                grd_DuplicateInterrupt: $i \notin IsDuplicate$
                grd_xor: $i \mapsto lr \notin StopMsgReceived$
                grd_seq_: $i \in SetInitialAP$
                grd1: $i \mapsto lr \notin notRequired\_ap$
                grdResLostRetry: $i \mapsto lr \notin (RemoveAlloc \cup RemoveBrokenRes)$
                grd2: $pr \in (alloc^{-1}[\{i\}] \cap alloc\_LR^{-1}[\{lr\}])$
        then
                act: $ResAttend := ResAttend \cup \{i \mapsto lr\}$
        end
Event DeallocateRes ⟨ordinary⟩ $\widehat{=}$
refines DeallocateRes
        any
                lr
                i
                pr
        where
                grd_seq_: $i \mapsto lr \in (ResAttend \cup StopMsgReceived)$
                grd_self: $i \mapsto lr \notin DeallocateRes$
                    @grdResLostRetry i ↦ lr ∉ ResLost
                grd_DuplicateInterrupt: $i \notin IsDuplicate$
                    @grd1 lr ∈ ap[{i}] // GRD PO//@grdResLostRetry i ↦ lr ∉ ResLost
                grdResLostRetry: $i \mapsto lr \notin (RemoveAlloc \cup RemoveBrokenRes)$
                grd1: $pr \in (alloc^{-1}[\{i\}] \cap alloc\_LR^{-1}[\{lr\}])$
                    manually
        then
                act: $DeallocateRes := DeallocateRes \cup \{i \mapsto lr\}$
                act1: $alloc := \{pr\} \lhd alloc$
                    manually
```

```
            act2: $alloc\_LR := \{pr\} \lhd alloc\_LR$
               manually
      end
Event IncidentManaged ⟨ordinary⟩ $\widehat{=}$
refines IncidentManaged
      any
            i
      where
            grd_self: $i \notin IncidentManaged$
            grd_DuplicateInterrupt: $i \notin IsDuplicate$
            grd_seq_: $i \in SetInitialAP$
            grd_par: $AllocateRes[\{i\}] = ResCompTask[\{i\}]$
      then
            act: $IncidentManaged := IncidentManaged \cup \{i\}$
      end
Event RemoveAlloc ⟨ordinary⟩ $\widehat{=}$
refines RemoveAlloc
      any
            i
            lr
      where
            grd2: $i \mapsto lr \notin alloc^{-1}; alloc\_LR$
               @grd3 (alloc$^{-1}$[{i}]∩ alloc_LR$^{-1}$[{lr}]) = ∅
            grd_self: $i \mapsto lr \notin RemoveAlloc$
            grd_seq: $i \in SetInitialAP$
            grd_par: $i \notin IncidentManaged$
            grd_xor: $i \mapsto lr \notin RemoveBrokenRes$
            grd_DuplicateInterrupt: $i \notin IsDuplicate$
            grd_Retry: $i \mapsto lr \notin DeallocateRes$
            grd1: $i \mapsto lr \in AllocateRes$
      then
            act: $RemoveAlloc := RemoveAlloc \cup \{i \mapsto lr\}$
            act1: $AllocateRes := AllocateRes \setminus \{i \mapsto lr\}$
            act2: $ResAttend := ResAttend \setminus \{i \mapsto lr\}$
            act3: $StopMsgReceived := StopMsgReceived \setminus \{i \mapsto lr\}$
            act4: $DeallocateRes := DeallocateRes \setminus \{i \mapsto lr\}$
      end
Event RemoveBrokenRes ⟨ordinary⟩ $\widehat{=}$
refines RemoveBrokenRes
      any
```

i

lr

pr

where

`grd_DuplicateInterrupt`: $i \notin IsDuplicate$

`grd_self`: $i \mapsto lr \notin RemoveBrokenRes$

`grd_seq`: $i \in SetInitialAP$

`grd_par`: $i \notin IncidentManaged$

`grd_xor`: $i \mapsto lr \notin RemoveAlloc$

`grd1`: $i \mapsto lr \in AllocateRes$

`grd2`: $i \mapsto lr \notin DeallocateRes$

`grd3`: $pr \in (alloc^{-1}[\{i\}] \cap alloc\_LR^{-1}[\{lr\}])$

then

`act`: $RemoveBrokenRes := RemoveBrokenRes \cup \{i \mapsto lr\}$

`act1`: $AllocateRes := AllocateRes \setminus \{i \mapsto lr\}$

`act2`: $ResAttend := ResAttend \setminus \{i \mapsto lr\}$

`act3`: $StopMsgReceived := StopMsgReceived \setminus \{i \mapsto lr\}$

`act4`: $DeallocateRes := DeallocateRes \setminus \{i \mapsto lr\}$

`act5`: $alloc := \{pr\} \lhd alloc$

manually

`act6`: $alloc\_LR := \{pr\} \lhd alloc\_LR$

manually

end

Event Reset_ResLost ⟨ordinary⟩ $\widehat{=}$

refines Reset_ResLost

any

i

lr

where

`grd_seq`: $i \mapsto lr \in (RemoveBrokenRes \cup RemoveAlloc)$

then

`act1`: $RemoveAlloc := RemoveAlloc \setminus \{i \mapsto lr\}$

`act2`: $RemoveBrokenRes := RemoveBrokenRes \setminus \{i \mapsto lr\}$

end

Event CloseIncident ⟨ordinary⟩ $\widehat{=}$

refines CloseIncident

any

i

where

`grd2`: $i \notin ran(alloc)$

`grd_seq_FD`: $(i \in SendStopMsg \land i \in IncidentManaged)$

grd_self: $i \notin CloseIncident$

grd_DuplicateInterrupt: $i \notin IsDuplicate$

grd1: $ap[\{i\}] \setminus notRequired\_ap[\{i\}] \subseteq DeallocateRes[\{i\}]$

to ensure all action plan items are satisfied

then

act: $CloseIncident := CloseIncident \cup \{i\}$

end

Event IsDuplicate ⟨ordinary⟩ $\widehat{=}$

refines IsDuplicate

any

i

where

grd_seq_DuplicateInterrupt: $i \in CreateIncident$

grd_self: $i \notin IsDuplicate$

grd1: $i \notin CloseIncident$

then

act: $IsDuplicate := IsDuplicate \cup \{i\}$

end

Event HandleDupAlloc ⟨ordinary⟩ $\widehat{=}$

any

i

lr

pr

where

grd_seq: $i \in IsDuplicate$

grd_self: $i \mapsto lr \notin HandleDupAlloc$

grd_exp: $lr \in AllocateRes[\{i\}]$

grd1: $pr \in (alloc^{-1}[\{i\}] \cap alloc\_LR^{-1}[\{lr\}])$

then

act: $HandleDupAlloc := HandleDupAlloc \cup \{i \mapsto lr\}$

act1: $alloc := \{pr\} \lhd\!\!\!- alloc$

act26: $alloc\_LR := \{pr\} \lhd\!\!\!- alloc\_LR$

end

Event CloseDuplicate ⟨ordinary⟩ $\widehat{=}$

any

i

where

grd_seq: $HandleDupAlloc[\{i\}] = AllocateRes[\{i\}]$

grd_self: $i \notin CloseDuplicate$

grd1: $i \in IsDuplicate$

grd2: $i \notin ran(alloc)$

```
    then
        act: $CloseDuplicate := CloseDuplicate \cup \{i\}$
    end
Event ResCompTask ⟨ordinary⟩ $\widehat{=}$
refines ResCompTask
    any
        lr
        i
    where
        grd_DuplicateInterrupt: $i \notin IsDuplicate$
        grd_seq_: $i \mapsto lr \in DeallocateRes$
        grd_self: $i \mapsto lr \notin ResCompTask$
    then
        act: $ResCompTask := ResCompTask \cup \{i \mapsto lr\}$
    end
END
```

B.2.10 Ninth Refinement: M9

```
MACHINE M9
REFINES M8
SEES M9_implicitContext,C4
VARIABLES
    pr_status
    pr_available_sm
    pr_allocated_sm
    pr_free_sm
    CreateIncident
    SelectLocation
    SelectType
    CompInfo
    SetInitialAP
    AllocateRes
    ResAttend
    DeallocateRes
    StopMsgReceived
    IncidentManaged
    RequestAdditionalRes
    SendStopMsg
```

CloseIncident

IsDuplicate

ap

notRequired_ap

iLocation

iType

DisablePriorityChange

priority

RemoveAlloc

RemoveBrokenRes

alloc allocate physical resource

alloc_LR relate lr to pr

HandleDupAlloc

CloseDuplicate

ResCompTask

iLocType

INVARIANTS

`typeof_pr_status`: $pr\_status \in PR \rightarrow pr\_status\_STATES$

`typeof_pr_available_sm`: $pr\_available\_sm \in PR \rightarrow pr\_available\_sm\_STATES$

`typeof_pr_allocated_sm`: $pr\_allocated\_sm \in PR \rightarrow pr\_allocated\_sm\_STATES$

`typeof_pr_free_sm`: $pr\_free\_sm \in PR \rightarrow pr\_free\_sm\_STATES$

`superstateof_pr_available_sm`: $dom(pr\_available\_sm \unrhd \{pr\_available\_sm\_NULL\}) = dom(pr\_status \rhd \{Available\})$

`superstateof_pr_allocated_sm`: $dom(pr\_allocated\_sm \unrhd \{pr\_allocated\_sm\_NULL\}) = dom(pr\_available\_sm \rhd \{pr\_allocated\})$

`superstateof_pr_free_sm`: $dom(pr\_free\_sm \unrhd \{pr\_free\_sm\_NULL\}) = dom(pr\_available\_sm \rhd \{pr\_free\})$

`inv1`: $\forall pr \cdot pr\_available\_sm(pr) = pr\_allocated \Leftrightarrow pr \in dom(alloc)$

`inv2`: $\forall pr \cdot pr\_available\_sm(pr) \in \{pr\_free, pr\_available\_sm\_NULL\} \Leftrightarrow pr \notin dom(alloc)$

EVENTS

Initialisation ⟨extended⟩

begin

`act_CreateIncident`: $CreateIncident := \varnothing$

`act_SetInitialAP`: $SetInitialAP := \varnothing$

`act_CloseIncident`: $CloseIncident := \varnothing$

`act_IsDuplicate`: $IsDuplicate := \varnothing$

`act_SendStopMsg`: $SendStopMsg := \varnothing$

```
        act_ap: $ap := \varnothing$
        act_notRequired_ap: $notRequired\_ap := \varnothing$
        act_RequestAdditionalRes: $RequestAdditionalRes := \varnothing$
        act2: $iLocation := \varnothing$
        act3: $iType := \varnothing$
        act_loc_type: $iLocType := \varnothing$
        act_SelectLocation: $SelectLocation := \varnothing$
        act_SelectType: $SelectType := \varnothing$
        act_CompInfo: $CompInfo := \varnothing$
        act_DisablePriorityChange: $DisablePriorityChange := \varnothing$
        act4: $priority := \varnothing$
        act_StopMsgReceived: $StopMsgReceived := \varnothing$
        act_AllocateRes: $AllocateRes := \varnothing$
        act_Attend: $ResAttend := \varnothing$
        act_DeallocateRes: $DeallocateRes := \varnothing$
        act_IncidentManaged: $IncidentManaged := \varnothing$
        act_RemoveAlloc: $RemoveAlloc := \varnothing$
        act_RemoveBrokenRes: $RemoveBrokenRes := \varnothing$
        act_alloc: $alloc := \varnothing$
        act_alloc_LR: $alloc\_LR := \varnothing$
        act_HandleDupAlloc: $HandleDupAlloc := \varnothing$
        act_CloseDuplicate: $CloseDuplicate := \varnothing$
        act_ResCompTask: $ResCompTask := \varnothing$
        init_pr_free_sm: $pr\_free\_sm := PR \times \{StationAvailable\}$
        init_pr_available_sm: $pr\_available\_sm := PR \times \{pr\_free\}$
        init_pr_status: $pr\_status := PR \times \{Available\}$
        init_pr_allocated_sm: $pr\_allocated\_sm := PR \times \{pr\_allocated\_sm\_NULL\}$
    end
Event CreateIncident ⟨ordinary⟩ $\widehat{=}$
extends CreateIncident
    any
        $i$
    where
        grd_self: $i \notin CreateIncident$
    then
        act: $CreateIncident := CreateIncident \cup \{i\}$
    end
Event SelectLocation ⟨ordinary⟩ $\widehat{=}$
extends SelectLocation
    any
        $i$
        l
        lt
    where
        grd_seq_:   i ∈ CreateIncident
        grd_self:   i ∉ SelectLocation
        grd_DuplicateInterrupt:   i ∉ IsDuplicate
        grd1:   l ∈ LOCATION
        grd2:   lt ∈ LOCATION_TYPE
    then
        act: SelectLocation := SelectLocation ∪ {i}
        act1: iLocation(i) := l
        act2: iLocType(i) := lt
    end
Event SelectType ⟨ordinary⟩ ≘
extends SelectType
    any
        i
        t
        p
    where
        grd_seq_:   i ∈ CreateIncident
        grd_self:   i ∉ SelectType
        grd1:   t ∈ INCIDENT_TYPE
        grd_DuplicateInterrupt:   i ∉ IsDuplicate
        grd2:   p = default_priority(t)
    then
        act: SelectType := SelectType ∪ {i}
        act1: iType(i) := t
        act2: priority(i) := p
    end
Event ChangePriority ⟨ordinary⟩ ≘
extends ChangePriority
    any
        i
        p
    where
        grd_seq_priority:   i ∈ SelectType
        grd0_loop:   i ∉ DisablePriorityChange
        grd1:   p ∈ 1 .. 3
        grd2:   p ≠ priority(i)
```

```
          grd3:  $i \notin CloseIncident$
               we can also set DisablePriorityChange in CloseIncident
          grd_DuplicateInterrupt:  $i \notin IsDuplicate$
     then
          act1: $priority(i) := p$
     end
Event DisablePriorityChange ⟨ordinary⟩ $\widehat{=}$
extends DisablePriorityChange
     any
          $i$
     where
          grd_DuplicateInterrupt:  $i \notin IsDuplicate$
          grd_seq_priority:  $i \in SelectType$
          grd_self:  $i \notin DisablePriorityChange$
          grd1:  $i \notin CloseIncident$
               we can also set DisablePriorityChange in CloseIncident
     then
          act: $DisablePriorityChange := DisablePriorityChange \cup \{i\}$
     end
Event CompInfo ⟨ordinary⟩ $\widehat{=}$
extends CompInfo
     any
          $i$
     where
          grd_DuplicateInterrupt:  $i \notin IsDuplicate$
          grd_seq_:  $(i \in SelectLocation \land i \in SelectType)$
          grd_self:  $i \notin CompInfo$
     then
          act: $CompInfo := CompInfo \cup \{i\}$
     end
Event SetInitialAP ⟨ordinary⟩ $\widehat{=}$
extends SetInitialAP
     any
          $lrs$
          $i$
          $t$
          $l$
     where
          grd_seq:  $i \in CompInfo$
          grd_self:  $i \notin SetInitialAP$
```

```
        grd1:  $t = iType(i)$
        grd2:  $l = iLocType(i)$
        grd3:  $lrs = PDA(t \mapsto l)$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
    then
        act: $SetInitialAP := SetInitialAP \cup \{i\}$
        act1: $ap := ap \cup (\{i\} \times lrs)$
    end
Event RequestAdditionalRes ⟨ordinary⟩ $\widehat{=}$
extends RequestAdditionalRes
    any
        $i$
        $lr$
    where
        grd_seq_planF:  $i \in SetInitialAP$
        grd_self:  $i \mapsto lr \notin RequestAdditionalRes$
        grd_InputExpression1:  $lr \in LR$
        grd0_par:  $i \notin SendStopMsg$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd1:  $lr \notin ap[\{i\}]$
    then
        act1: $ap := ap \cup \{i \mapsto lr\}$
        act: $RequestAdditionalRes := RequestAdditionalRes \cup \{i \mapsto lr\}$
    end
Event SendStopMsg ⟨ordinary⟩ $\widehat{=}$
extends SendStopMsg
    any
        $i$
        $lrs$
    where
        grd_lrs:  $lrs = ap[\{i\}] \setminus ResAttend[\{i\}]$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd_seq_:  $i \in SetInitialAP$
        grd_self:  $i \notin SendStopMsg$
    then
        act1: $notRequired\_ap := notRequired\_ap \cup (\{i\} \times lrs)$
        act: $SendStopMsg := SendStopMsg \cup \{i\}$
    end
Event AllocateRes ⟨ordinary⟩ $\widehat{=}$
extends AllocateRes
```

```
    any
        $i$
        $lr$
        $pr$
    where
        grd_seq_allocation:  $i \in SetInitialAP$
        grd_self:  $i \mapsto lr \notin AllocateRes$
        grd_InputExpression1:  $lr \in ap[\{i\}]$
        grd0_par:  $i \notin IncidentManaged$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd1:  $i \mapsto lr \notin notRequired\_ap$
        grdResLostRetry:  $i \mapsto lr \notin (RemoveAlloc \cup RemoveBrokenRes)$
        grd2:  $ptype(pr) = ltype(lr)$
        grd3:  $pr \in dom(alloc) \Rightarrow priority(i) < priority(alloc(pr))$
        isin_Mobilised_or_isin_pr_free:  $(pr\_allocated\_sm(pr) = Mobilised \lor pr\_available\_sm(pr) =$
            $pr\_free)$
    then
        act: $AllocateRes := AllocateRes \cup \{i \mapsto lr\}$
        act1: $alloc(pr) := i$
        act2: $alloc\_LR(pr) := lr$
        leave_pr_free_sm: $pr\_free\_sm(pr) := pr\_free\_sm\_NULL$
        enter_Mobilised: $pr\_allocated\_sm(pr) := Mobilised$
        enter_pr_allocated: $pr\_available\_sm(pr) := pr\_allocated$
    end
Event UpdateResLocation ⟨ordinary⟩ $\widehat{=}$
extends UpdateResLocation
    any
        $i$
        $lr$
        $pr$
    where
        grd_seq_allocation:  $i \mapsto lr \in AllocateRes$
        grd0_loop:  $i \mapsto lr \notin (ResAttend \cup StopMsgReceived)$
            @grdResLostRetry i ↦ lr ∉ ResLost
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd1:  $i \mapsto lr \notin notRequired\_ap$
        grdResLostRetry:  $i \mapsto lr \notin (RemoveAlloc \cup RemoveBrokenRes)$
        grd2:  $pr \in (alloc^{-1}[\{i\}] \cap alloc\_LR^{-1}[\{lr\}])$
        isin_Mobilised:  $pr\_allocated\_sm(pr) = Mobilised$
    then
        skip
```

end

Event ReallocateQuickerRes ⟨ordinary⟩ $\widehat{=}$

extends ReallocateQuickerRes

any

i

lr

pr

$pr\_quicker$

where

`grd_seq_allocation`: $i \mapsto lr \in AllocateRes$

`grd0_loop`: $i \mapsto lr \notin (ResAttend \cup StopMsgReceived)$

@grdResLostRetry i ↦ lr ∉ ResLost

`grd_DuplicateInterrupt`: $i \notin IsDuplicate$

`grd1`: $i \mapsto lr \notin notRequired\_ap$

`grdResLostRetry`: $i \mapsto lr \notin (RemoveAlloc \cup RemoveBrokenRes)$

`grd2`: $pr \in (alloc^{-1}[\{i\}] \cap alloc\_LR^{-1}[\{lr\}])$

`grd3`: $pr\_quicker \notin dom(alloc)$

`grd4`: $ptype(pr\_quicker) = ltype(lr)$

@grd5 ptype(pr) = ltype(lr)

`grd5`: $pr\_available\_sm(pr\_quicker) = pr\_free$

manually

`grd6`: $pr\_allocated\_sm(pr) = Mobilised$

manually

then

`act1`: $alloc := (\{pr\} \lhd alloc) \cup \{pr\_quicker \mapsto i\}$

`act2`: $alloc\_LR := (\{pr\} \lhd alloc\_LR) \cup \{pr\_quicker \mapsto lr\}$

`act3`: $pr\_available\_sm := (\{pr, pr\_quicker\} \lhd pr\_available\_sm) \cup \{pr \mapsto pr\_free\} \cup \{pr\_quicker \mapsto pr\_allocated\}$

manually

`act4`: $pr\_free\_sm := (\{pr, pr\_quicker\} \lhd pr\_free\_sm) \cup \{pr \mapsto Returning\} \cup \{pr\_quicker \mapsto pr\_free\_sm\_NULL\}$

manually

`act5`: $pr\_allocated\_sm := (\{pr, pr\_quicker\} \lhd pr\_allocated\_sm) \cup \{pr \mapsto pr\_allocated\_sm\_NULL\} \cup \{pr\_quicker \mapsto Mobilised\}$

manually

end

Event StopMsgReceived ⟨ordinary⟩ $\widehat{=}$

extends StopMsgReceived

any

lr

i

```
        pr
    where
        grd_seq_allocation:  $i \mapsto lr \in AllocateRes$
        grd_self:  $i \mapsto lr \notin StopMsgReceived$
            @grdResLostRetry i ↦ lr ∉ ResLost
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd_xor:  $i \mapsto lr \notin ResAttend$
        grd1:  $i \mapsto lr \in notRequired\_ap$
        grdResLostRetry:  $i \mapsto lr \notin (RemoveAlloc \cup RemoveBrokenRes)$
        grd_seq_:  $i \in SetInitialAP$
        grd_InputExpression1:  $lr \in ap[\{i\}]$
        grd2:  $pr \in (alloc^{-1}[\{i\}] \cap alloc\_LR^{-1}[\{lr\}])$
    then
        act:  $StopMsgReceived := StopMsgReceived \cup \{i \mapsto lr\}$
    end
Event ResAttend ⟨ordinary⟩ ≘
extends ResAttend
    any
        i
        lr
        pr
    where
        grd_InputExpression1:  $lr \in ap[\{i\}]$
        grd_seq_allocation:  $i \mapsto lr \in AllocateRes$
        grd_self:  $i \mapsto lr \notin ResAttend$
            @grdResLostRetry i ↦ lr ∉ ResLost
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd_xor:  $i \mapsto lr \notin StopMsgReceived$
        grd_seq_:  $i \in SetInitialAP$
        grd1:  $i \mapsto lr \notin notRequired\_ap$
        grdResLostRetry:  $i \mapsto lr \notin (RemoveAlloc \cup RemoveBrokenRes)$
        grd2:  $pr \in (alloc^{-1}[\{i\}] \cap alloc\_LR^{-1}[\{lr\}])$
        isin_Mobilised:  $pr\_allocated\_sm(pr) = Mobilised$
    then
        act:  $ResAttend := ResAttend \cup \{i \mapsto lr\}$
        enter_InAttendance:  $pr\_allocated\_sm(pr) := InAttendance$
    end
Event DeallocateRes ⟨ordinary⟩ ≘
extends DeallocateRes
    any
        lr
```

i

pr

where

grd_seq_: $i \mapsto lr \in (ResAttend \cup StopMsgReceived)$

grd_self: $i \mapsto lr \notin DeallocateRes$

@grdResLostRetry i ↦ lr ∉ ResLost

grd_DuplicateInterrupt: $i \notin IsDuplicate$

@grd1 lr ∈ ap[{i}] // GRD PO//@grdResLostRetry i ↦ lr ∉ ResLost

grdResLostRetry: $i \mapsto lr \notin (RemoveAlloc \cup RemoveBrokenRes)$

grd1: $pr \in (alloc^{-1}[\{i\}] \cap alloc\_LR^{-1}[\{lr\}])$

manually

isin_pr_allocated: $pr\_available\_sm(pr) = pr\_allocated$

then

act: $DeallocateRes := DeallocateRes \cup \{i \mapsto lr\}$

act1: $alloc := \{pr\} \lhd alloc$

manually

act2: $alloc\_LR := \{pr\} \lhd alloc\_LR$

manually

leave_pr_allocated_sm: $pr\_allocated\_sm(pr) := pr\_allocated\_sm\_NULL$

enter_Returning: $pr\_free\_sm(pr) := Returning$

enter_pr_free: $pr\_available\_sm(pr) := pr\_free$

end

Event IncidentManaged ⟨ordinary⟩ $\widehat{=}$

extends IncidentManaged

any

i

where

grd_self: $i \notin IncidentManaged$

grd_DuplicateInterrupt: $i \notin IsDuplicate$

grd_seq_: $i \in SetInitialAP$

grd_par: $AllocateRes[\{i\}] = ResCompTask[\{i\}]$

then

act: $IncidentManaged := IncidentManaged \cup \{i\}$

end

Event RemoveAlloc ⟨ordinary⟩ $\widehat{=}$

extends RemoveAlloc

any

i

lr

where

grd2: $i \mapsto lr \notin alloc^{-1}; alloc\_LR$

@grd3 (alloc^{-1}[{i}]∩ alloc_LR^{-1}[{lr}]) = ∅

grd_self: $i \mapsto lr \notin RemoveAlloc$

grd_seq: $i \in SetInitialAP$

grd_par: $i \notin IncidentManaged$

grd_xor: $i \mapsto lr \notin RemoveBrokenRes$

grd_DuplicateInterrupt: $i \notin IsDuplicate$

grd_Retry: $i \mapsto lr \notin DeallocateRes$

grd1: $i \mapsto lr \in AllocateRes$

then

act: $RemoveAlloc := RemoveAlloc \cup \{i \mapsto lr\}$

act1: $AllocateRes := AllocateRes \setminus \{i \mapsto lr\}$

act2: $ResAttend := ResAttend \setminus \{i \mapsto lr\}$

act3: $StopMsgReceived := StopMsgReceived \setminus \{i \mapsto lr\}$

act4: $DeallocateRes := DeallocateRes \setminus \{i \mapsto lr\}$

end

Event RemoveBrokenRes ⟨ordinary⟩ $\widehat{=}$

extends RemoveBrokenRes

any

i

lr

pr

where

grd_DuplicateInterrupt: $i \notin IsDuplicate$

grd_self: $i \mapsto lr \notin RemoveBrokenRes$

grd_seq: $i \in SetInitialAP$

grd_par: $i \notin IncidentManaged$

grd_xor: $i \mapsto lr \notin RemoveAlloc$

grd1: $i \mapsto lr \in AllocateRes$

grd2: $i \mapsto lr \notin DeallocateRes$

grd3: $pr \in (alloc^{-1}[\{i\}] \cap alloc\_LR^{-1}[\{lr\}])$

isin_pr_allocated: $pr\_available\_sm(pr) = pr\_allocated$

then

act: $RemoveBrokenRes := RemoveBrokenRes \cup \{i \mapsto lr\}$

act1: $AllocateRes := AllocateRes \setminus \{i \mapsto lr\}$

act2: $ResAttend := ResAttend \setminus \{i \mapsto lr\}$

act3: $StopMsgReceived := StopMsgReceived \setminus \{i \mapsto lr\}$

act4: $DeallocateRes := DeallocateRes \setminus \{i \mapsto lr\}$

act5: $alloc := \{pr\} \lhd alloc$

manually

act6: $alloc\_LR := \{pr\} \lhd alloc\_LR$

manually

```
        leave_pr_allocated_sm: $pr\_allocated\_sm(pr) := pr\_allocated\_sm\_NULL$
        leave_pr_available_sm: $pr\_available\_sm(pr) := pr\_available\_sm\_NULL$
        leave_pr_free_sm: $pr\_free\_sm(pr) := pr\_free\_sm\_NULL$
        enter_Unavailable: $pr\_status(pr) := Unavailable$
    end
Event Reset_ResLost ⟨ordinary⟩ $\widehat{=}$
extends Reset_ResLost
    any
        $i$
        $lr$
    where
        grd_seq: $i \mapsto lr \in (RemoveBrokenRes \cup RemoveAlloc)$
    then
        act1: $RemoveAlloc := RemoveAlloc \setminus \{i \mapsto lr\}$
        act2: $RemoveBrokenRes := RemoveBrokenRes \setminus \{i \mapsto lr\}$
    end
Event CloseIncident ⟨ordinary⟩ $\widehat{=}$
extends CloseIncident
    any
        $i$
    where
        grd2: $i \notin ran(alloc)$
        grd_seq_FD: $(i \in SendStopMsg \land i \in IncidentManaged)$
        grd_self: $i \notin CloseIncident$
        grd_DuplicateInterrupt: $i \notin IsDuplicate$
        grd1: $ap[\{i\}] \setminus notRequired\_ap[\{i\}] \subseteq DeallocateRes[\{i\}]$
            to ensure all action plan items are satisfied
    then
        act: $CloseIncident := CloseIncident \cup \{i\}$
    end
Event IsDuplicate ⟨ordinary⟩ $\widehat{=}$
extends IsDuplicate
    any
        $i$
    where
        grd_seq_DuplicateInterrupt: $i \in CreateIncident$
        grd_self: $i \notin IsDuplicate$
        grd1: $i \notin CloseIncident$
    then
        act: $IsDuplicate := IsDuplicate \cup \{i\}$
```

end

Event HandleDupAlloc ⟨ordinary⟩ $\widehat{=}$

extends HandleDupAlloc

any

i

lr

pr

where

`grd_seq`: $i \in IsDuplicate$

`grd_self`: $i \mapsto lr \notin HandleDupAlloc$

`grd_exp`: $lr \in AllocateRes[\{i\}]$

`grd1`: $pr \in (alloc^{-1}[\{i\}] \cap alloc\_LR^{-1}[\{lr\}])$

`isin_pr_allocated`: $pr\_available\_sm(pr) = pr\_allocated$

then

`act`: $HandleDupAlloc := HandleDupAlloc \cup \{i \mapsto lr\}$

`act1`: $alloc := \{pr\} \lhd alloc$

`act26`: $alloc\_LR := \{pr\} \lhd alloc\_LR$

`leave_pr_allocated_sm`: $pr\_allocated\_sm(pr) := pr\_allocated\_sm\_NULL$

`enter_Returning`: $pr\_free\_sm(pr) := Returning$

`enter_pr_free`: $pr\_available\_sm(pr) := pr\_free$

end

Event CloseDuplicate ⟨ordinary⟩ $\widehat{=}$

extends CloseDuplicate

any

i

where

`grd_seq`: $HandleDupAlloc[\{i\}] = AllocateRes[\{i\}]$

`grd_self`: $i \notin CloseDuplicate$

`grd1`: $i \in IsDuplicate$

`grd2`: $i \notin ran(alloc)$

then

`act`: $CloseDuplicate := CloseDuplicate \cup \{i\}$

end

Event ResCompTask ⟨ordinary⟩ $\widehat{=}$

extends ResCompTask

any

lr

i

where

`grd_DuplicateInterrupt`: $i \notin IsDuplicate$

grd_seq_: $i \mapsto lr \in DeallocateRes$

grd_self: $i \mapsto lr \notin ResCompTask$

then

act: $ResCompTask := ResCompTask \cup \{i \mapsto lr\}$

end

Event FixRes ⟨ordinary⟩ $\widehat{=}$

any

pr

where

isin_Unavailable: $pr\_status(pr) = Unavailable$

grd1: $pr \notin dom(alloc)$

then

enter_StationAvailable: $pr\_free\_sm(pr) := StationAvailable$

enter_pr_free: $pr\_available\_sm(pr) := pr\_free$

enter_Available: $pr\_status(pr) := Available$

end

Event FreeResBreakdown ⟨ordinary⟩ $\widehat{=}$

any

pr

where

isin_pr_free: $pr\_available\_sm(pr) = pr\_free$

then

leave_pr_free_sm: $pr\_free\_sm(pr) := pr\_free\_sm\_NULL$

leave_pr_available_sm: $pr\_available\_sm(pr) := pr\_available\_sm\_NULL$

leave_pr_allocated_sm: $pr\_allocated\_sm(pr) := pr\_allocated\_sm\_NULL$

enter_Unavailable: $pr\_status(pr) := Unavailable$

end

Event BookResHome ⟨ordinary⟩ $\widehat{=}$

any

pr

where

isin_Returning: $pr\_free\_sm(pr) = Returning$

then

enter_StationAvailable: $pr\_free\_sm(pr) := StationAvailable$

end

Event UpdateLoc_RetRes ⟨ordinary⟩ $\widehat{=}$

any

pr

where

isin_Returning: $pr\_free\_sm(pr) = Returning$

then

```
        skip
    end
END
```

B.2.11 Tenth Refinement: M10

```
MACHINE M10
REFINES M9
SEES C5,M9_implicitContext
VARIABLES
    pr_status
    pr_available_sm
    pr_allocated_sm
    pr_free_sm
    CreateIncident
    SelectLocation
    SelectType
    CompInfo
    SetInitialAP
    AllocateRes
    ResAttend
    DeallocateRes
    StopMsgReceived
    IncidentManaged
    RequestAdditionalRes
    SendStopMsg
    CloseIncident
    IsDuplicate
    ap
    notRequired_ap
    iLocation
    iType
    DisablePriorityChange
    priority
    RemoveAlloc
    RemoveBrokenRes
    alloc allocate physical resource
    alloc_LR relate lr to pr
```

HandleDupAlloc

CloseDuplicate

ResCompTask

rLocation location of a physical resource

duration duration from one location to another this is a variable because it depends on time and not distance which can be affected by traffic

iLocType

INVARIANTS

`inv_r_loc_type`: $rLocation \in PR \rightarrow LOCATION$

`inv_duration_type`: $duration \in (LOCATION \times LOCATION) \rightarrow \mathbb{N}$

EVENTS

Initialisation ⟨extended⟩

begin

`act_CreateIncident`: $CreateIncident := \varnothing$

`act_SetInitialAP`: $SetInitialAP := \varnothing$

`act_CloseIncident`: $CloseIncident := \varnothing$

`act_IsDuplicate`: $IsDuplicate := \varnothing$

`act_SendStopMsg`: $SendStopMsg := \varnothing$

`act_ap`: $ap := \varnothing$

`act_notRequired_ap`: $notRequired\_ap := \varnothing$

`act_RequestAdditionalRes`: $RequestAdditionalRes := \varnothing$

`act2`: $iLocation := \varnothing$

`act3`: $iType := \varnothing$

`act_loc_type`: $iLocType := \varnothing$

`act_SelectLocation`: $SelectLocation := \varnothing$

`act_SelectType`: $SelectType := \varnothing$

`act_CompInfo`: $CompInfo := \varnothing$

`act_DisablePriorityChange`: $DisablePriorityChange := \varnothing$

`act4`: $priority := \varnothing$

`act_StopMsgReceived`: $StopMsgReceived := \varnothing$

`act_AllocateRes`: $AllocateRes := \varnothing$

`act_Attend`: $ResAttend := \varnothing$

`act_DeallocateRes`: $DeallocateRes := \varnothing$

`act_IncidentManaged`: $IncidentManaged := \varnothing$

`act_RemoveAlloc`: $RemoveAlloc := \varnothing$

`act_RemoveBrokenRes`: $RemoveBrokenRes := \varnothing$

`act_alloc`: $alloc := \varnothing$

`act_alloc_LR`: $alloc\_LR := \varnothing$

`act_HandleDupAlloc`: $HandleDupAlloc := \varnothing$

`act_CloseDuplicate`: $CloseDuplicate := \varnothing$

act_ResCompTask: $ResCompTask := \varnothing$
init_pr_free_sm: $pr\_free\_sm := PR \times \{StationAvailable\}$
init_pr_available_sm: $pr\_available\_sm := PR \times \{pr\_free\}$
init_pr_status: $pr\_status := PR \times \{Available\}$
init_pr_allocated_sm: $pr\_allocated\_sm := PR \times \{pr\_allocated\_sm\_NULL\}$
act_r_location: $rLocation := rl0$
act_duration: $duration := duration0$

end

Event CreateIncident ⟨ordinary⟩ $\widehat{=}$
extends CreateIncident

any
i
where
grd_self: $i \notin CreateIncident$
then
act: $CreateIncident := CreateIncident \cup \{i\}$
end

Event SelectLocation ⟨ordinary⟩ $\widehat{=}$
extends SelectLocation

any
i
l
lt
where
grd_seq_: $i \in CreateIncident$
grd_self: $i \notin SelectLocation$
grd_DuplicateInterrupt: $i \notin IsDuplicate$
grd1: $l \in LOCATION$
grd2: $lt \in LOCATION\_TYPE$
then
act: $SelectLocation := SelectLocation \cup \{i\}$
act1: $iLocation(i) := l$
act2: $iLocType(i) := lt$
end

Event SelectType ⟨ordinary⟩ $\widehat{=}$
extends SelectType

any
i
t
p

```
        where
            grd_seq_:  $i \in CreateIncident$
            grd_self:  $i \notin SelectType$
            grd1:  $t \in INCIDENT\_TYPE$
            grd_DuplicateInterrupt:  $i \notin IsDuplicate$
            grd2:  $p = default\_priority(t)$
        then
            act: $SelectType := SelectType \cup \{i\}$
            act1: $iType(i) := t$
            act2: $priority(i) := p$
        end
Event ChangePriority ⟨ordinary⟩ $\widehat{=}$
extends ChangePriority
        any
            $i$
            $p$
        where
            grd_seq_priority:  $i \in SelectType$
            grd0_loop:  $i \notin DisablePriorityChange$
            grd1:  $p \in 1 .. 3$
            grd2:  $p \neq priority(i)$
            grd3:  $i \notin CloseIncident$
                we can also set DisablePriorityChange in CloseIncident
            grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        then
            act1: $priority(i) := p$
        end
Event DisablePriorityChange ⟨ordinary⟩ $\widehat{=}$
extends DisablePriorityChange
        any
            $i$
        where
            grd_DuplicateInterrupt:  $i \notin IsDuplicate$
            grd_seq_priority:  $i \in SelectType$
            grd_self:  $i \notin DisablePriorityChange$
            grd1:  $i \notin CloseIncident$
                we can also set DisablePriorityChange in CloseIncident
        then
            act: $DisablePriorityChange := DisablePriorityChange \cup \{i\}$
        end
Event CompInfo ⟨ordinary⟩ $\widehat{=}$
```

```
extends CompInfo
    any
        $i$
    where
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd_seq_:  $(i \in SelectLocation \land i \in SelectType)$
        grd_self:  $i \notin CompInfo$
    then
        act: $CompInfo := CompInfo \cup \{i\}$
    end
Event SetInitialAP ⟨ordinary⟩ $\widehat{=}$
extends SetInitialAP
    any
        $lrs$
        $i$
        $t$
        $l$
    where
        grd_seq:  $i \in CompInfo$
        grd_self:  $i \notin SetInitialAP$
        grd1:  $t = iType(i)$
        grd2:  $l = iLocType(i)$
        grd3:  $lrs = PDA(t \mapsto l)$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
    then
        act: $SetInitialAP := SetInitialAP \cup \{i\}$
        act1: $ap := ap \cup (\{i\} \times lrs)$
    end
Event RequestAdditionalRes ⟨ordinary⟩ $\widehat{=}$
extends RequestAdditionalRes
    any
        $i$
        $lr$
    where
        grd_seq_planF:  $i \in SetInitialAP$
        grd_self:  $i \mapsto lr \notin RequestAdditionalRes$
        grd_InputExpression1:  $lr \in LR$
        grd0_par:  $i \notin SendStopMsg$
        grd_DuplicateInterrupt:  $i \notin IsDuplicate$
        grd1:  $lr \notin ap[\{i\}]$
    then
```

act1: $ap := ap \cup \{i \mapsto lr\}$

act: $RequestAdditionalRes := RequestAdditionalRes \cup \{i \mapsto lr\}$

end

Event SendStopMsg ⟨ordinary⟩ $\widehat{=}$

extends SendStopMsg

any

i

lrs

where

grd_lrs: $lrs = ap[\{i\}] \setminus ResAttend[\{i\}]$

grd_DuplicateInterrupt: $i \notin IsDuplicate$

grd_seq_: $i \in SetInitialAP$

grd_self: $i \notin SendStopMsg$

then

act1: $notRequired\_ap := notRequired\_ap \cup (\{i\} \times lrs)$

act: $SendStopMsg := SendStopMsg \cup \{i\}$

end

Event AllocateRes ⟨ordinary⟩ $\widehat{=}$

extends AllocateRes

any

i

lr

pr

where

grd_seq_allocation: $i \in SetInitialAP$

grd_self: $i \mapsto lr \notin AllocateRes$

grd_InputExpression1: $lr \in ap[\{i\}]$

grd0_par: $i \notin IncidentManaged$

grd_DuplicateInterrupt: $i \notin IsDuplicate$

grd1: $i \mapsto lr \notin notRequired\_ap$

grdResLostRetry: $i \mapsto lr \notin (RemoveAlloc \cup RemoveBrokenRes)$

grd2: $ptype(pr) = ltype(lr)$

grd3: $pr \in dom(alloc) \Rightarrow priority(i) < priority(alloc(pr))$

isin_Mobilised_or_isin_pr_free: $(pr\_allocated\_sm(pr) = Mobilised \vee pr\_available\_sm(pr) = pr\_free)$

then

act: $AllocateRes := AllocateRes \cup \{i \mapsto lr\}$

act1: $alloc(pr) := i$

act2: $alloc\_LR(pr) := lr$

leave_pr_free_sm: $pr\_free\_sm(pr) := pr\_free\_sm\_NULL$

enter_Mobilised: $pr\_allocated\_sm(pr) := Mobilised$

```
        enter_pr_allocated: $pr\_available\_sm(pr) := pr\_allocated$
    end
Event UpdateResLocation ⟨ordinary⟩ $\widehat{=}$
extends UpdateResLocation
    any
        $i$
        $lr$
        $pr$
        l
        l0
    where
        grd_seq_allocation: $i \mapsto lr \in AllocateRes$
        grd0_loop: $i \mapsto lr \notin (ResAttend \cup StopMsgReceived)$
            @grdResLostRetry i ↦ lr ∉ ResLost
        grd_DuplicateInterrupt: $i \notin IsDuplicate$
        grd1: $i \mapsto lr \notin notRequired\_ap$
        grdResLostRetry: $i \mapsto lr \notin (RemoveAlloc \cup RemoveBrokenRes)$
        grd2: $pr \in (alloc^{-1}[\{i\}] \cap alloc\_LR^{-1}[\{lr\}])$
        isin_Mobilised: $pr\_allocated\_sm(pr) = Mobilised$
        grd_l0: $l0 = rLocation(pr)$
        grd_l: $l \in LOCATION \setminus \{l0\}$
        grd_dur: $duration(l \mapsto iLocation(i)) < duration(l0 \mapsto iLocation(i))$
    then
        act_loc: $rLocation(pr) := l$
    end
Event ReallocateQuickerRes ⟨ordinary⟩ $\widehat{=}$
extends ReallocateQuickerRes
    any
        $i$
        $lr$
        $pr$
        $pr\_quicker$
    where
        grd_seq_allocation: $i \mapsto lr \in AllocateRes$
        grd0_loop: $i \mapsto lr \notin (ResAttend \cup StopMsgReceived)$
            @grdResLostRetry i ↦ lr ∉ ResLost
        grd_DuplicateInterrupt: $i \notin IsDuplicate$
        grd1: $i \mapsto lr \notin notRequired\_ap$
        grdResLostRetry: $i \mapsto lr \notin (RemoveAlloc \cup RemoveBrokenRes)$
        grd2: $pr \in (alloc^{-1}[\{i\}] \cap alloc\_LR^{-1}[\{lr\}])$
        grd3: $pr\_quicker \notin dom(alloc)$
```

grd4: $ptype(pr\_quicker) = ltype(lr)$

@grd5 ptype(pr) = ltype(lr)

grd5: $pr\_available\_sm(pr\_quicker) = pr\_free$

manually

grd6: $pr\_allocated\_sm(pr) = Mobilised$

manually

grd_quicker: $duration(rLocation(pr\_quicker) \mapsto iLocation(i)) < duration(rLocation(pr) \mapsto iLocation(i))$

then

act1: $alloc := (\{pr\} \lhd alloc) \cup \{pr\_quicker \mapsto i\}$

act2: $alloc\_LR := (\{pr\} \lhd alloc\_LR) \cup \{pr\_quicker \mapsto lr\}$

act3: $pr\_available\_sm := (\{pr, pr\_quicker\} \lhd pr\_available\_sm) \cup \{pr \mapsto pr\_free\} \cup \{pr\_quicker \mapsto pr\_allocated\}$

manually

act4: $pr\_free\_sm := (\{pr, pr\_quicker\} \lhd pr\_free\_sm) \cup \{pr \mapsto Returning\} \cup \{pr\_quicker \mapsto pr\_free\_sm\_NULL\}$

manually

act5: $pr\_allocated\_sm := (\{pr, pr\_quicker\} \lhd pr\_allocated\_sm) \cup \{pr \mapsto pr\_allocated\_sm\_NULL\} \cup \{pr\_quicker \mapsto Mobilised\}$

manually

end

Event StopMsgReceived ⟨ordinary⟩ $\widehat{=}$

extends StopMsgReceived

any

lr

i

pr

where

grd_seq_allocation: $i \mapsto lr \in AllocateRes$

grd_self: $i \mapsto lr \notin StopMsgReceived$

@grdResLostRetry i ↦ lr ∉ ResLost

grd_DuplicateInterrupt: $i \notin IsDuplicate$

grd_xor: $i \mapsto lr \notin ResAttend$

grd1: $i \mapsto lr \in notRequired\_ap$

grdResLostRetry: $i \mapsto lr \notin (RemoveAlloc \cup RemoveBrokenRes)$

grd_seq_: $i \in SetInitialAP$

grd_InputExpression1: $lr \in ap[\{i\}]$

grd2: $pr \in (alloc^{-1}[\{i\}] \cap alloc\_LR^{-1}[\{lr\}])$

then

act: $StopMsgReceived := StopMsgReceived \cup \{i \mapsto lr\}$

end

Event ResAttend ⟨ordinary⟩ $\widehat{=}$

extends ResAttend

any

i

lr

pr

where

grd_InputExpression1: $lr \in ap[\{i\}]$

grd_seq_allocation: $i \mapsto lr \in AllocateRes$

grd_self: $i \mapsto lr \notin ResAttend$

@grdResLostRetry i ↦ lr ∉ ResLost

grd_DuplicateInterrupt: $i \notin IsDuplicate$

grd_xor: $i \mapsto lr \notin StopMsgReceived$

grd_seq_: $i \in SetInitialAP$

grd1: $i \mapsto lr \notin notRequired\_ap$

grdResLostRetry: $i \mapsto lr \notin (RemoveAlloc \cup RemoveBrokenRes)$

grd2: $pr \in (alloc^{-1}[\{i\}] \cap alloc\_LR^{-1}[\{lr\}])$

isin_Mobilised: $pr\_allocated\_sm(pr) = Mobilised$

grd_loc: $rLocation(pr) = iLocation(i)$

then

act: $ResAttend := ResAttend \cup \{i \mapsto lr\}$

enter_InAttendance: $pr\_allocated\_sm(pr) := InAttendance$

end

Event DeallocateRes ⟨ordinary⟩ $\widehat{=}$

extends DeallocateRes

any

lr

i

pr

where

grd_seq_: $i \mapsto lr \in (ResAttend \cup StopMsgReceived)$

grd_self: $i \mapsto lr \notin DeallocateRes$

@grdResLostRetry i ↦ lr ∉ ResLost

grd_DuplicateInterrupt: $i \notin IsDuplicate$

@grd1 lr ∈ ap[{i}] // GRD PO//@grdResLostRetry i ↦ lr ∉ ResLost

grdResLostRetry: $i \mapsto lr \notin (RemoveAlloc \cup RemoveBrokenRes)$

grd1: $pr \in (alloc^{-1}[\{i\}] \cap alloc\_LR^{-1}[\{lr\}])$

manually

isin_pr_allocated: $pr\_available\_sm(pr) = pr\_allocated$

then

act: $DeallocateRes := DeallocateRes \cup \{i \mapsto lr\}$

act1: $alloc := \{pr\} \lhd alloc$
manually
act2: $alloc\_LR := \{pr\} \lhd alloc\_LR$
manually
leave_pr_allocated_sm: $pr\_allocated\_sm(pr) := pr\_allocated\_sm\_NULL$
enter_Returning: $pr\_free\_sm(pr) := Returning$
enter_pr_free: $pr\_available\_sm(pr) := pr\_free$

end

Event IncidentManaged ⟨ordinary⟩ $\widehat{=}$

extends IncidentManaged

any

i

where

grd_self: $i \notin IncidentManaged$
grd_DuplicateInterrupt: $i \notin IsDuplicate$
grd_seq_: $i \in SetInitialAP$
grd_par: $AllocateRes[\{i\}] = ResCompTask[\{i\}]$

then

act: $IncidentManaged := IncidentManaged \cup \{i\}$

end

Event RemoveAlloc ⟨ordinary⟩ $\widehat{=}$

extends RemoveAlloc

any

i
lr

where

grd2: $i \mapsto lr \notin alloc^{-1}; alloc\_LR$
@grd3 (alloc^{-1}[{i}]∩ alloc_LR^{-1}[{lr}]) = ∅
grd_self: $i \mapsto lr \notin RemoveAlloc$
grd_seq: $i \in SetInitialAP$
grd_par: $i \notin IncidentManaged$
grd_xor: $i \mapsto lr \notin RemoveBrokenRes$
grd_DuplicateInterrupt: $i \notin IsDuplicate$
grd_Retry: $i \mapsto lr \notin DeallocateRes$
grd1: $i \mapsto lr \in AllocateRes$

then

act: $RemoveAlloc := RemoveAlloc \cup \{i \mapsto lr\}$
act1: $AllocateRes := AllocateRes \setminus \{i \mapsto lr\}$
act2: $ResAttend := ResAttend \setminus \{i \mapsto lr\}$
act3: $StopMsgReceived := StopMsgReceived \setminus \{i \mapsto lr\}$
act4: $DeallocateRes := DeallocateRes \setminus \{i \mapsto lr\}$

end

Event RemoveBrokenRes ⟨ordinary⟩ $\widehat{=}$

extends RemoveBrokenRes

any

i

lr

pr

where

grd_DuplicateInterrupt: $i \notin IsDuplicate$

grd_self: $i \mapsto lr \notin RemoveBrokenRes$

grd_seq: $i \in SetInitialAP$

grd_par: $i \notin IncidentManaged$

grd_xor: $i \mapsto lr \notin RemoveAlloc$

grd1: $i \mapsto lr \in AllocateRes$

grd2: $i \mapsto lr \notin DeallocateRes$

grd3: $pr \in (alloc^{-1}[\{i\}] \cap alloc\_LR^{-1}[\{lr\}])$

isin_pr_allocated: $pr\_available\_sm(pr) = pr\_allocated$

then

act: $RemoveBrokenRes := RemoveBrokenRes \cup \{i \mapsto lr\}$

act1: $AllocateRes := AllocateRes \setminus \{i \mapsto lr\}$

act2: $ResAttend := ResAttend \setminus \{i \mapsto lr\}$

act3: $StopMsgReceived := StopMsgReceived \setminus \{i \mapsto lr\}$

act4: $DeallocateRes := DeallocateRes \setminus \{i \mapsto lr\}$

act5: $alloc := \{pr\} \lhd alloc$

manually

act6: $alloc\_LR := \{pr\} \lhd alloc\_LR$

manually

leave_pr_allocated_sm: $pr\_allocated\_sm(pr) := pr\_allocated\_sm\_NULL$

leave_pr_available_sm: $pr\_available\_sm(pr) := pr\_available\_sm\_NULL$

leave_pr_free_sm: $pr\_free\_sm(pr) := pr\_free\_sm\_NULL$

enter_Unavailable: $pr\_status(pr) := Unavailable$

end

Event Reset_ResLost ⟨ordinary⟩ $\widehat{=}$

extends Reset_ResLost

any

i

lr

where

grd_seq: $i \mapsto lr \in (RemoveBrokenRes \cup RemoveAlloc)$

then

act1: $RemoveAlloc := RemoveAlloc \setminus \{i \mapsto lr\}$

```
        act2: $RemoveBrokenRes := RemoveBrokenRes \setminus \{i \mapsto lr\}$
    end
Event CloseIncident ⟨ordinary⟩ $\widehat{=}$
extends CloseIncident
    any
        $i$
    where
        grd2: $i \notin ran(alloc)$
        grd_seq_FD: $(i \in SendStopMsg \land i \in IncidentManaged)$
        grd_self: $i \notin CloseIncident$
        grd_DuplicateInterrupt: $i \notin IsDuplicate$
        grd1: $ap[\{i\}] \setminus notRequired\_ap[\{i\}] \subseteq DeallocateRes[\{i\}]$
            to ensure all action plan items are satisfied
    then
        act: $CloseIncident := CloseIncident \cup \{i\}$
    end
Event IsDuplicate ⟨ordinary⟩ $\widehat{=}$
extends IsDuplicate
    any
        $i$
    where
        grd_seq_DuplicateInterrupt: $i \in CreateIncident$
        grd_self: $i \notin IsDuplicate$
        grd1: $i \notin CloseIncident$
    then
        act: $IsDuplicate := IsDuplicate \cup \{i\}$
    end
Event HandleDupAlloc ⟨ordinary⟩ $\widehat{=}$
extends HandleDupAlloc
    any
        $i$
        $lr$
        $pr$
    where
        grd_seq: $i \in IsDuplicate$
        grd_self: $i \mapsto lr \notin HandleDupAlloc$
        grd_exp: $lr \in AllocateRes[\{i\}]$
        grd1: $pr \in (alloc^{-1}[\{i\}] \cap alloc\_LR^{-1}[\{lr\}])$
        isin_pr_allocated: $pr\_available\_sm(pr) = pr\_allocated$
    then
```

act: $HandleDupAlloc := HandleDupAlloc \cup \{i \mapsto lr\}$
act1: $alloc := \{pr\} \lhd alloc$
act26: $alloc\_LR := \{pr\} \lhd alloc\_LR$
leave_pr_allocated_sm: $pr\_allocated\_sm(pr) := pr\_allocated\_sm\_NULL$
enter_Returning: $pr\_free\_sm(pr) := Returning$
enter_pr_free: $pr\_available\_sm(pr) := pr\_free$

end

Event CloseDuplicate ⟨ordinary⟩ $\widehat{=}$
extends CloseDuplicate

any
i
where
grd_seq: $HandleDupAlloc[\{i\}] = AllocateRes[\{i\}]$
grd_self: $i \notin CloseDuplicate$
grd1: $i \in IsDuplicate$
grd2: $i \notin ran(alloc)$
then
act: $CloseDuplicate := CloseDuplicate \cup \{i\}$
end

Event ResCompTask ⟨ordinary⟩ $\widehat{=}$
extends ResCompTask

any
lr
i
where
grd_DuplicateInterrupt: $i \notin IsDuplicate$
grd_seq_: $i \mapsto lr \in DeallocateRes$
grd_self: $i \mapsto lr \notin ResCompTask$
then
act: $ResCompTask := ResCompTask \cup \{i \mapsto lr\}$
end

Event FixRes ⟨ordinary⟩ $\widehat{=}$
extends FixRes

any
pr
where
isin_Unavailable: $pr\_status(pr) = Unavailable$
grd1: $pr \notin dom(alloc)$
then
enter_StationAvailable: $pr\_free\_sm(pr) := StationAvailable$

```
        enter_pr_free: $pr\_available\_sm(pr) := pr\_free$
        enter_Available: $pr\_status(pr) := Available$
        act1: $rLocation(pr) := rl0(pr)$
    end
Event FreeResBreakdown ⟨ordinary⟩ ≙
extends FreeResBreakdown
    any
        $pr$
    where
        isin_pr_free: $pr\_available\_sm(pr) = pr\_free$
    then
        leave_pr_free_sm: $pr\_free\_sm(pr) := pr\_free\_sm\_NULL$
        leave_pr_available_sm: $pr\_available\_sm(pr) := pr\_available\_sm\_NULL$
        leave_pr_allocated_sm: $pr\_allocated\_sm(pr) := pr\_allocated\_sm\_NULL$
        enter_Unavailable: $pr\_status(pr) := Unavailable$
    end
Event BookResHome ⟨ordinary⟩ ≙
extends BookResHome
    any
        $pr$
    where
        isin_Returning: $pr\_free\_sm(pr) = Returning$
        grd_loc: $rLocation(pr) \in ran(s\_location)$
            can make it rl0
    then
        enter_StationAvailable: $pr\_free\_sm(pr) := StationAvailable$
    end
Event UpdateLoc_RetRes ⟨ordinary⟩ ≙
extends UpdateLoc_RetRes
    any
        $pr$
        l
        l0
    where
        isin_Returning: $pr\_free\_sm(pr) = Returning$
        grd0: $l0 = rLocation(pr)$
        grd1: $l0 \notin (ran(s\_location))$
        grd2: $l \in LOCATION \setminus \{l0\}$
    then
        act1: $rLocation(pr) := l$
    end
END
```

Appendix C

Event-B Model of the Travel Agency Workflow

In this chapter we present the complete version of the travel agency booking system presented in Chapter 7.

C.1 Static Part of the Model: Contexts

C.1.1 Context: C0

```
CONTEXT C0
SETS
        TRIP
END
```

C.1.2 Context: C1

```
CONTEXT C1
EXTENDS C0
SETS
        FLIGHT
        HOTEL
        CAR
END
```

C.2 Model Refinements

C.2.1 Abstract Level: M0

MACHINE M0

SEES C0

VARIABLES

StartTrip

TripItinerary

Quit

INVARIANTS

`inv_StartTrip_type`: $StartTrip \subseteq TRIP$

`inv_FlightItinerary_f_seq`: $TripItinerary \subseteq StartTrip$

`inv_Quit_seq`: $Quit \subseteq StartTrip$

`inv_Interrupt`: $partition((TripItinerary \cup Quit), TripItinerary, Quit)$

EVENTS

Initialisation

begin

`act_StartTrip`: $StartTrip := \varnothing$

`act_TripItinerary`: $TripItinerary := \varnothing$

`act_Quit`: $Quit := \varnothing$

end

Event StartTrip ⟨ordinary⟩ $\widehat{=}$

any

t

where

`grd_self`: $t \notin StartTrip$

then

`act`: $StartTrip := StartTrip \cup \{t\}$

end

Event TripItinerary ⟨ordinary⟩ $\widehat{=}$

any

t

where

`grd_seq_f`: $t \in StartTrip$

`grd_self`: $t \notin TripItinerary$

`grd_interrupt`: $t \notin Quit$

then

`act`: $TripItinerary := TripItinerary \cup \{t\}$

```
    end
Event Quit ⟨ordinary⟩ $\widehat{=}$
    any
        t
    where
        grd_seq: $t \in StartTrip$
        grd_self: $t \notin Quit$
        grd_interrupt: $t \notin TripItinerary$
    then
        act: $Quit := Quit \cup \{t\}$
    end
END
```

C.2.2 First Refinement: M1

```
MACHINE M1
REFINES M0
SEES C0
VARIABLES
      StartTrip
      FlightItinerary
      CarItinerary
      HotelItinerary
      Checkout
      Quit
INVARIANTS
      inv_StartTrip_type: $StartTrip \subseteq TRIP$
      inv_FlightItinerary_f_seq: $FlightItinerary \subseteq StartTrip$
      inv_CarItinerary_f_seq: $CarItinerary \subseteq StartTrip$
      inv_HotelItinerary_f_seq: $HotelItinerary \subseteq StartTrip$
      inv_Checkout_f_seq: $Checkout \subseteq (FlightItinerary \cap CarItinerary \cap HotelItinerary)$
      inv_Quit_seq: $Quit \subseteq StartTrip$
      inv_Checkout_glu: $Checkout = TripItinerary$
      inv_Interrupt: $partition((Checkout \cup Quit), Checkout, Quit)$
EVENTS
Initialisation
    begin
        act_StartTrip: $StartTrip := \varnothing$
```

```
            act_FlightItinerary: $FlightItinerary := \varnothing$
            act_CarItinerary: $CarItinerary := \varnothing$
            act_HotelItinerary: $HotelItinerary := \varnothing$
            act_Checkout: $Checkout := \varnothing$
            act_Quit: $Quit := \varnothing$
      end
Event StartTrip ⟨ordinary⟩ $\widehat{=}$
refines StartTrip
      any
            t
      where
            grd_self: $t \notin StartTrip$
      then
            act: $StartTrip := StartTrip \cup \{t\}$
      end
Event FlightItinerary ⟨ordinary⟩ $\widehat{=}$
      any
            t
      where
            grd_seq_f: $t \in StartTrip$
            grd_self: $t \notin FlightItinerary$
            grd_interrupt: $t \notin Quit$
      then
            act: $FlightItinerary := FlightItinerary \cup \{t\}$
      end
Event CarItinerary ⟨ordinary⟩ $\widehat{=}$
      any
            t
      where
            grd_seq_f: $t \in StartTrip$
            grd_self: $t \notin CarItinerary$
            grd_interrupt: $t \notin Quit$
      then
            act: $CarItinerary := CarItinerary \cup \{t\}$
      end
Event Checkout ⟨ordinary⟩ $\widehat{=}$
refines TripItinerary
      any
            t
      where
```

```
        grd_interrupt:  $t \notin Quit$
        grd_seq_f:  $(t \in FlightItinerary \land t \in CarItinerary \land t \in HotelItinerary)$
        grd_self:  $t \notin Checkout$
    then
        act: $Checkout := Checkout \cup \{t\}$
    end
Event HotelItinerary ⟨ordinary⟩ $\widehat{=}$
    any
        t
    where
        grd_seq_f:  $t \in StartTrip$
        grd_self:  $t \notin HotelItinerary$
        grd_interrupt:  $t \notin Quit$
    then
        act: $HotelItinerary := HotelItinerary \cup \{t\}$
    end
Event Quit ⟨ordinary⟩ $\widehat{=}$
refines Quit
    any
        t
    where
        grd_seq:  $t \in StartTrip$
        grd_self:  $t \notin Quit$
        grd_interrupt:  $t \notin Checkout$
    then
        act: $Quit := Quit \cup \{t\}$
    end
END
```

C.2.3 Second Refinement: M2

```
MACHINE M2
REFINES M1
SEES C1
VARIABLES
      StartTrip
      FlightBooking
      CompFlightItin
      CarBooking
      CompCarItin
```

HotelBooking

CompHotelItin

Checkout

Quit

INVARIANTS

inv_StartTrip_type: $StartTrip \subseteq TRIP$

inv_FlightBooking_type: $FlightBooking \subseteq TRIP \times FLIGHT$

inv_CarBooking_type: $CarBooking \subseteq TRIP \times CAR$

inv_HotelBooking_type: $HotelBooking \subseteq TRIP \times HOTEL$

inv_FlightBooking__seq: $dom(FlightBooking) \subseteq StartTrip$

inv_FlightBooking_rep: $\forall t \cdot t \in TRIP \Rightarrow FlightBooking[\{t\}] \subseteq FLIGHT$

inv_CompFlightItin__seq: $CompFlightItin \subseteq StartTrip$

inv_CompFlightItin_glu: $CompFlightItin = FlightItinerary$

inv_CarBooking__seq: $dom(CarBooking) \subseteq StartTrip$

inv_CarBooking_rep: $\forall t \cdot t \in TRIP \Rightarrow CarBooking[\{t\}] \subseteq CAR$

inv_CompCarItin__seq: $CompCarItin \subseteq StartTrip$

inv_CompCarItin_glu: $CompCarItin = CarItinerary$

inv_HotelBooking__seq: $dom(HotelBooking) \subseteq StartTrip$

inv_HotelBooking_rep: $\forall t \cdot t \in TRIP \Rightarrow HotelBooking[\{t\}] \subseteq HOTEL$

inv_CompHotelItine__seq: $CompHotelItin \subseteq StartTrip$

inv_CompHotelItine_glu: $CompHotelItin = HotelItinerary$

inv_Checkout_f_seq: $Checkout \subseteq (CompFlightItin \cap CompCarItin \cap CompHotelItin)$

inv_Quit_seq: $Quit \subseteq StartTrip$

inv_interrupt: $partition((Checkout \cup Quit), Checkout, Quit)$

EVENTS

Initialisation

begin

act_StartTrip: $StartTrip := \varnothing$

act_FlightBooking: $FlightBooking := \varnothing$

act_CompFlightItin: $CompFlightItin := \varnothing$

act_CarBooking: $CarBooking := \varnothing$

act_CompCarItin: $CompCarItin := \varnothing$

act_HotelBooking: $HotelBooking := \varnothing$

act_CompHotelItine: $CompHotelItin := \varnothing$

act_Checkout: $Checkout := \varnothing$

act_Quit: $Quit := \varnothing$

end

Event FlightBooking ⟨ordinary⟩ $\widehat{=}$

  any

    t

    f

  where

    `grd_seq_`: $t \in StartTrip$

    `grd_self`: $t \mapsto f \notin FlightBooking$

    `grd0_par`: $t \notin CompFlightItin$

    `grd_interrupt`: $t \notin Quit$

  then

    `act`: $FlightBooking := FlightBooking \cup \{t \mapsto f\}$

  end

Event CarBooking ⟨ordinary⟩ $\widehat{=}$

  any

    t

    c

  where

    `grd_seq_`: $t \in StartTrip$

    `grd_self`: $t \mapsto c \notin CarBooking$

    `grd0_par`: $t \notin CompCarItin$

    `grd_interrupt`: $t \notin Quit$

  then

    `act`: $CarBooking := CarBooking \cup \{t \mapsto c\}$

  end

Event HotelBooking ⟨ordinary⟩ $\widehat{=}$

  any

    t

    h

  where

    `grd_seq_`: $t \in StartTrip$

    `grd_self`: $t \mapsto h \notin HotelBooking$

    `grd0_par`: $t \notin CompHotelItin$

    `grd_interrupt`: $t \notin Quit$

  then

    `act`: $HotelBooking := HotelBooking \cup \{t \mapsto h\}$

  end

Event StartTrip ⟨ordinary⟩ $\widehat{=}$

refines StartTrip

  any

    t

  where

```
        grd_self:  $t \notin StartTrip$
    then
        act: $StartTrip := StartTrip \cup \{t\}$
    end
Event CompFlightItin ⟨ordinary⟩ $\widehat{=}$
refines FlightItinerary
    any
        t
    where
        grd_seq_:  $t \in StartTrip$
        grd_self:  $t \notin CompFlightItin$
        grd_interrupt:  $t \notin Quit$
    then
        act: $CompFlightItin := CompFlightItin \cup \{t\}$
    end
Event CompCarItin ⟨ordinary⟩ $\widehat{=}$
refines CarItinerary
    any
        t
    where
        grd_interrupt:  $t \notin Quit$
        grd_seq_:  $t \in StartTrip$
        grd_self:  $t \notin CompCarItin$
    then
        act: $CompCarItin := CompCarItin \cup \{t\}$
    end
Event CompHotelItin ⟨ordinary⟩ $\widehat{=}$
refines HotelItinerary
    any
        t
    where
        grd_seq_:  $t \in StartTrip$
        grd_self:  $t \notin CompHotelItin$
        grd_interrupt:  $t \notin Quit$
    then
        act: $CompHotelItin := CompHotelItin \cup \{t\}$
    end
Event Checkout ⟨ordinary⟩ $\widehat{=}$
refines Checkout
    any
```

```
        t
    where
        grd_seq_f:  $(t \in CompFlightItin \land t \in CompCarItin \land t \in CompHotelItin)$
        grd_self:  $t \notin Checkout$
        grd_interrupt:  $t \notin Quit$
    then
        act: $Checkout := Checkout \cup \{t\}$
    end
Event Quit ⟨ordinary⟩ $\widehat{=}$
refines Quit
    any
        t
    where
        grd_seq:  $t \in StartTrip$
        grd_self:  $t \notin Quit$
        grd_interrupt:  $t \notin Checkout$
    then
        act: $Quit := Quit \cup \{t\}$
    end
END
```

C.2.4 Third Refinement: M3

```
MACHINE M3
REFINES M2
SEES C1
VARIABLES
       StartTrip
       SelectFlight
       BookFlight
       RemoveFlight
       CompFlightItin
       SelectCar
       BookCar
       RemoveCar
       CompCarItin
       SelectHotel
       BookHotel
       RemoveHotel
```

CompHotelItin

Checkout

Quit

flights

cars

hotels manually

INVARIANTS

`inv_StartTrip_type`: $StartTrip \subseteq TRIP$

`inv_SelectFlight_type`: $SelectFlight \subseteq TRIP \times FLIGHT$

`inv_SelectFlight__seq`: $dom(SelectFlight) \subseteq StartTrip$

`inv_SelectFlight_glu`: $SelectFlight = FlightBooking$

`inv_BookFlight__seq`: $BookFlight \subseteq SelectFlight$

`inv_RemoveFlight__seq`: $RemoveFlight \subseteq SelectFlight$

`inv_CompFlightItin__seq`: $CompFlightItin \subseteq StartTrip$

`inv_SelectCar_type`: $SelectCar \subseteq TRIP \times CAR$

`inv_SelectCar__seq`: $dom(SelectCar) \subseteq StartTrip$

`inv_SelectCar_glu`: $SelectCar = CarBooking$

`inv_BookCar__seq`: $BookCar \subseteq SelectCar$

`inv_RemoveCar__seq`: $RemoveCar \subseteq SelectCar$

`inv_CompCarItin__seq`: $CompCarItin \subseteq StartTrip$

`inv_SelectHotel_type`: $SelectHotel \subseteq TRIP \times HOTEL$

`inv_SelectHotel__seq`: $dom(SelectHotel) \subseteq StartTrip$

`inv_SelectHotel_glu`: $SelectHotel = HotelBooking$

`inv_BookHotel__seq`: $BookHotel \subseteq SelectHotel$

`inv_RemoveHotel__seq`: $RemoveHotel \subseteq SelectHotel$

`inv_CompHotelItine__seq`: $CompHotelItin \subseteq StartTrip$

`inv_CompFlightItin_par`: $\forall t \cdot t \in CompFlightItin \Rightarrow SelectFlight[\{t\}] = (BookFlight[\{t\}] \cup RemoveFlight[\{t\}])$

`inv_CompCarItin_par`: $\forall t \cdot t \in CompCarItin \Rightarrow SelectCar[\{t\}] = (BookCar[\{t\}] \cup RemoveCar[\{t\}])$

`inv_CompHotelItin_par`: $\forall t \cdot t \in CompHotelItin \Rightarrow SelectHotel[\{t\}] = (BookHotel[\{t\}] \cup RemoveHotel[\{t\}])$

`inv_Checkout_f_seq`: $Checkout \subseteq (CompFlightItin \cap CompCarItin \cap CompHotelItin)$

`inv1_xorf`: $partition((BookFlight \cup RemoveFlight), BookFlight, RemoveFlight)$

`inv2_xorf`: $partition((BookCar \cup RemoveCar), BookCar, RemoveCar)$

`inv3_xorf`: $partition((BookHotel \cup RemoveHotel), BookHotel, RemoveHotel)$

`inv_Quit_seq`: $Quit \subseteq StartTrip$

inv_interrupt: $partition((Checkout \cup Quit), Checkout, Quit)$

inv_flights: $flights \in TRIP \leftrightarrow FLIGHT$

inv_cars: $cars \in TRIP \leftrightarrow CAR$

inv_hotels: $hotels \in TRIP \leftrightarrow HOTEL$

inv_fligths_select: $flights \subseteq SelectFlight$

inv_hotels_select: $hotels \subseteq SelectHotel$

inv_cars_select: $cars \subseteq SelectCar$

inv_flights_remove: $RemoveFlight \cap flights = \varnothing$

inv_cars_remove: $RemoveCar \cap cars = \varnothing$

inv_hotels_remove: $RemoveHotel \cap hotels = \varnothing$

EVENTS

Initialisation

begin

act_StartTrip: $StartTrip := \varnothing$

act_SelectFlight: $SelectFlight := \varnothing$

act_BookFlight: $BookFlight := \varnothing$

act_RemoveFlight: $RemoveFlight := \varnothing$

act_CompFlightItin: $CompFlightItin := \varnothing$

act_SelectCar: $SelectCar := \varnothing$

act_BookCar: $BookCar := \varnothing$

act_RemoveCar: $RemoveCar := \varnothing$

act_CompCarItin: $CompCarItin := \varnothing$

act_SelectHotel: $SelectHotel := \varnothing$

act_BookHotel: $BookHotel := \varnothing$

act_RemoveHotel: $RemoveHotel := \varnothing$

act_CompHotelItine: $CompHotelItin := \varnothing$

act_Checkout: $Checkout := \varnothing$

act_flights: $flights := \varnothing$

act_hotels: $hotels := \varnothing$

act_cars: $cars := \varnothing$

act_Quit: $Quit := \varnothing$

end

Event StartTrip ⟨ordinary⟩ $\widehat{=}$

refines StartTrip

any

t

where

grd_self: $t \notin StartTrip$

then

act: $StartTrip := StartTrip \cup \{t\}$

```
      end
Event  SelectFlight ⟨ordinary⟩ $\widehat{=}$
refines  FlightBooking
      any
            t
            f
      where
            grd_seq_:  $t \in StartTrip$
            grd_self:  $t \mapsto f \notin SelectFlight$
            grd0_par:  $t \notin CompFlightItin$
            grd_interrupt:  $t \notin Quit$
            grd1:  $f \notin flights[\{t\}]$
      then
            act: $SelectFlight := SelectFlight \cup \{t \mapsto f\}$
            act1: $flights := flights \cup \{t \mapsto f\}$
      end
Event  BookFlight ⟨ordinary⟩ $\widehat{=}$
      any
            t
            f
      where
            grd_seq_:  $t \mapsto f \in SelectFlight$
            grd_self:  $t \mapsto f \notin BookFlight$
            grd1_xor:  $t \mapsto f \notin RemoveFlight$
            grd_interrupt:  $t \notin Quit$
      then
            act: $BookFlight := BookFlight \cup \{t \mapsto f\}$
      end
Event  RemoveFlight ⟨ordinary⟩ $\widehat{=}$
      any
            t
            f
      where
            grd_seq_:  $t \mapsto f \in SelectFlight$
            grd_self:  $t \mapsto f \notin RemoveFlight$
            grd1_xor:  $t \mapsto f \notin BookFlight$
            grd_interrupt:  $t \notin Quit$
      then
            act: $RemoveFlight := RemoveFlight \cup \{t \mapsto f\}$
            act1: $flights := flights \setminus \{t \mapsto f\}$
      end
```

Event CompFlightItin ⟨ordinary⟩ $\widehat{=}$
refines CompFlightItin
  any
    t
  where
    grd_seq_: $t \in StartTrip$
    grd_self: $t \notin CompFlightItin$
    grd_par: $SelectFlight[\{t\}] = BookFlight[\{t\}] \cup RemoveFlight[\{t\}]$
    grd_interrupt: $t \notin Quit$
  then
    act: $CompFlightItin := CompFlightItin \cup \{t\}$
  end
Event SelectCar ⟨ordinary⟩ $\widehat{=}$
refines CarBooking
  any
    t
    c
  where
    grd_seq_: $t \in StartTrip$
    grd_self: $t \mapsto c \notin SelectCar$
    grd0_par: $t \notin CompCarItin$
    grd_interrupt: $t \notin Quit$
    grd1: $c \notin cars[\{t\}]$
  then
    act: $SelectCar := SelectCar \cup \{t \mapsto c\}$
    act1: $cars := cars \cup \{t \mapsto c\}$
  end
Event BookCar ⟨ordinary⟩ $\widehat{=}$
  any
    t
    c
  where
    grd_seq_: $t \mapsto c \in SelectCar$
    grd_self: $t \mapsto c \notin BookCar$
    grd1_xor: $t \mapsto c \notin RemoveCar$
    grd_interrupt: $t \notin Quit$
  then
    act: $BookCar := BookCar \cup \{t \mapsto c\}$
  end
Event RemoveCar ⟨ordinary⟩ $\widehat{=}$

```
    any
        t
        c
    where
        grd_seq_:   $t \mapsto c \in SelectCar$
        grd_self:   $t \mapsto c \notin RemoveCar$
        grd1_xor:   $t \mapsto c \notin BookCar$
        grd_interrupt:   $t \notin Quit$
    then
        act: $RemoveCar := RemoveCar \cup \{t \mapsto c\}$
        act1: $cars := cars \setminus \{t \mapsto c\}$
    end
Event CompCarItin ⟨ordinary⟩ $\widehat{=}$
refines CompCarItin
    any
        t
    where
        grd_seq_:   $t \in StartTrip$
        grd_self:   $t \notin CompCarItin$
        grd_par:   $SelectCar[\{t\}] = BookCar[\{t\}] \cup RemoveCar[\{t\}]$
        grd_interrupt:   $t \notin Quit$
    then
        act: $CompCarItin := CompCarItin \cup \{t\}$
    end
Event SelectHotel ⟨ordinary⟩ $\widehat{=}$
refines HotelBooking
    any
        t
        h
    where
        grd_seq_:   $t \in StartTrip$
        grd_self:   $t \mapsto h \notin SelectHotel$
        grd0_par:   $t \notin CompHotelItin$
        grd_interrupt:   $t \notin Quit$
        grd1:   $h \notin hotels[\{t\}]$
    then
        act: $SelectHotel := SelectHotel \cup \{t \mapsto h\}$
        act1: $hotels := hotels \cup \{t \mapsto h\}$
    end
Event BookHotel ⟨ordinary⟩ $\widehat{=}$
    any
```

t
h
where
grd_seq_: $t \mapsto h \in SelectHotel$
grd_self: $t \mapsto h \notin BookHotel$
grd1_xor: $t \mapsto h \notin RemoveHotel$
grd_interrupt: $t \notin Quit$
then
act: $BookHotel := BookHotel \cup \{t \mapsto h\}$
end

Event RemoveHotel ⟨ordinary⟩ $\widehat{=}$
any
t
h
where
grd_seq_: $t \mapsto h \in SelectHotel$
grd_self: $t \mapsto h \notin RemoveHotel$
grd1_xor: $t \mapsto h \notin BookHotel$
grd_interrupt: $t \notin Quit$
then
act: $RemoveHotel := RemoveHotel \cup \{t \mapsto h\}$
act1: $hotels := hotels \setminus \{t \mapsto h\}$
end

Event CompHotelItin ⟨ordinary⟩ $\widehat{=}$
refines CompHotelItin
any
t
where
grd_seq_: $t \in StartTrip$
grd_self: $t \notin CompHotelItin$
grd_par: $SelectHotel[\{t\}] = BookHotel[\{t\}] \cup RemoveHotel[\{t\}]$
grd_interrupt: $t \notin Quit$
then
act: $CompHotelItin := CompHotelItin \cup \{t\}$
end

Event Checkout ⟨ordinary⟩ $\widehat{=}$
refines Checkout
any
t
where
grd_seq_f: $(t \in CompFlightItin \wedge t \in CompCarItin \wedge t \in CompHotelItin)$

grd_self: $t \notin Checkout$

grd_interrupt: $t \notin Quit$

then

act: $Checkout := Checkout \cup \{t\}$

end

Event Quit ⟨ordinary⟩ $\widehat{=}$

refines Quit

any

t

where

grd_seq: $t \in StartTrip$

grd_self: $t \notin Quit$

grd_interrupt: $t \notin Checkout$

then

act: $Quit := Quit \cup \{t\}$

act1: $flights := \{t\} \lhd flights$

act2: $hotels := \{t\} \lhd hotels$

act3: $cars := \{t\} \lhd cars$

end

END

C.2.5 Fourth Refinement: M4

MACHINE M4

REFINES M3

SEES C1

VARIABLES

StartTrip

SelectFlight

BookFlight

RemoveFlight

CompFlightItin

SelectCar

BookCar

RemoveCar

CompCarItin

SelectHotel

BookHotel

RemoveHotel

CompHotelItin

Checkout

Quit

flights

cars

hotels manually

time manually

deadline manually

tFlight

tCar

tHotel time selected

INVARIANTS

`inv_time`: $time \in \mathbb{N}$

`inv_deadline`: $deadline \in \mathbb{N}$

`inv_tFlight`: $tFlight \in flights \rightarrow \mathbb{N}$

`inv_tCar`: $tCar \in cars \rightarrow \mathbb{N}$

`inv_tHotel`: $tHotel \in hotels \rightarrow \mathbb{N}$

EVENTS

Initialisation ⟨extended⟩

begin

`act_StartTrip`: $StartTrip := \varnothing$

`act_SelectFlight`: $SelectFlight := \varnothing$

`act_BookFlight`: $BookFlight := \varnothing$

`act_RemoveFlight`: $RemoveFlight := \varnothing$

`act_CompFlightItin`: $CompFlightItin := \varnothing$

`act_SelectCar`: $SelectCar := \varnothing$

`act_BookCar`: $BookCar := \varnothing$

`act_RemoveCar`: $RemoveCar := \varnothing$

`act_CompCarItin`: $CompCarItin := \varnothing$

`act_SelectHotel`: $SelectHotel := \varnothing$

`act_BookHotel`: $BookHotel := \varnothing$

`act_RemoveHotel`: $RemoveHotel := \varnothing$

`act_CompHotelItine`: $CompHotelItin := \varnothing$

`act_Checkout`: $Checkout := \varnothing$

`act_flights`: $flights := \varnothing$

`act_hotels`: $hotels := \varnothing$

`act_cars`: $cars := \varnothing$

`act_Quit`: $Quit := \varnothing$

`act_time`: $time := 0$

```
        act_deadline: $deadline :\in \mathbb{N}_1$
        act_tFlight: $tFlight := \varnothing$
        act_tcars: $tCar := \varnothing$
        act_thotels: $tHotel := \varnothing$
    end
Event StartTrip ⟨ordinary⟩ $\widehat{=}$
extends StartTrip
    any
        $t$
    where
        grd_self:  $t \notin StartTrip$
    then
        act: $StartTrip := StartTrip \cup \{t\}$
    end
Event SelectFlight ⟨ordinary⟩ $\widehat{=}$
extends SelectFlight
    any
        $t$
        $f$
    where
        grd_seq_:  $t \in StartTrip$
        grd_self:  $t \mapsto f \notin SelectFlight$
        grd0_par:  $t \notin CompFlightItin$
        grd_interrupt:  $t \notin Quit$
        grd1:  $f \notin flights[\{t\}]$
    then
        act: $SelectFlight := SelectFlight \cup \{t \mapsto f\}$
        act1: $flights := flights \cup \{t \mapsto f\}$
        act2: $tFlight(t \mapsto f) := time$
    end
Event BookFlight ⟨ordinary⟩ $\widehat{=}$
extends BookFlight
    any
        $t$
        $f$
    where
        grd_seq_:  $t \mapsto f \in SelectFlight$
        grd_self:  $t \mapsto f \notin BookFlight$
        grd1_xor:  $t \mapsto f \notin RemoveFlight$
        grd_interrupt:  $t \notin Quit$
        grd1:  $(t \mapsto f) \in flights$
        grd2:  $time \leq (tFlight(t \mapsto f) + deadline)$
    then
        act: $BookFlight := BookFlight \cup \{t \mapsto f\}$
    end
Event RemoveFlight ⟨ordinary⟩ $\widehat{=}$
extends RemoveFlight
    any
        $t$
        $f$
    where
        grd_seq_:  $t \mapsto f \in SelectFlight$
        grd_self:  $t \mapsto f \notin RemoveFlight$
        grd1_xor:  $t \mapsto f \notin BookFlight$
        grd_interrupt:  $t \notin Quit$
        grd2:  $(t \mapsto f) \in flights$
    then
        act: $RemoveFlight := RemoveFlight \cup \{t \mapsto f\}$
        act1: $flights := flights \setminus \{t \mapsto f\}$
        act2: $tFlight := \{t \mapsto f\} \lhd tFlight$
    end
Event CompFlightItin ⟨ordinary⟩ $\widehat{=}$
extends CompFlightItin
    any
        $t$
    where
        grd_seq_:  $t \in StartTrip$
        grd_self:  $t \notin CompFlightItin$
        grd_par:  $SelectFlight[\{t\}] = BookFlight[\{t\}] \cup RemoveFlight[\{t\}]$
        grd_interrupt:  $t \notin Quit$
    then
        act: $CompFlightItin := CompFlightItin \cup \{t\}$
    end
Event SelectCar ⟨ordinary⟩ $\widehat{=}$
extends SelectCar
    any
        $t$
        $c$
    where
        grd_seq_:  $t \in StartTrip$
```

```
            grd_self:  $t \mapsto c \notin SelectCar$
            grd0_par:  $t \notin CompCarItin$
            grd_interrupt:  $t \notin Quit$
            grd1:  $c \notin cars[\{t\}]$
      then
            act: $SelectCar := SelectCar \cup \{t \mapsto c\}$
            act1: $cars := cars \cup \{t \mapsto c\}$
            act2: $tCar(t \mapsto c) := time$
      end
Event BookCar ⟨ordinary⟩ $\widehat{=}$
extends BookCar
      any
            $t$
            $c$
      where
            grd_seq_:  $t \mapsto c \in SelectCar$
            grd_self:  $t \mapsto c \notin BookCar$
            grd1_xor:  $t \mapsto c \notin RemoveCar$
            grd_interrupt:  $t \notin Quit$
            grd1:  $(t \mapsto c) \in cars$
            grd2:  $time \leq (tCar(t \mapsto c) + deadline)$
      then
            act: $BookCar := BookCar \cup \{t \mapsto c\}$
      end
Event RemoveCar ⟨ordinary⟩ $\widehat{=}$
extends RemoveCar
      any
            $t$
            $c$
      where
            grd_seq_:  $t \mapsto c \in SelectCar$
            grd_self:  $t \mapsto c \notin RemoveCar$
            grd1_xor:  $t \mapsto c \notin BookCar$
            grd_interrupt:  $t \notin Quit$
            grd2:  $(t \mapsto c) \in cars$
      then
            act: $RemoveCar := RemoveCar \cup \{t \mapsto c\}$
            act1: $cars := cars \setminus \{t \mapsto c\}$
            act2: $tCar := \{t \mapsto c\} \lhd tCar$
      end
Event CompCarItin ⟨ordinary⟩ $\widehat{=}$
```

```
extends CompCarItin
    any
        $t$
    where
        grd_seq_:  $t \in StartTrip$
        grd_self:  $t \notin CompCarItin$
        grd_par:  $SelectCar[\{t\}] = BookCar[\{t\}] \cup RemoveCar[\{t\}]$
        grd_interrupt:  $t \notin Quit$
    then
        act: $CompCarItin := CompCarItin \cup \{t\}$
    end
Event SelectHotel ⟨ordinary⟩ $\widehat{=}$
extends SelectHotel
    any
        $t$
        $h$
    where
        grd_seq_:  $t \in StartTrip$
        grd_self:  $t \mapsto h \notin SelectHotel$
        grd0_par:  $t \notin CompHotelItin$
        grd_interrupt:  $t \notin Quit$
        grd1:  $h \notin hotels[\{t\}]$
    then
        act: $SelectHotel := SelectHotel \cup \{t \mapsto h\}$
        act1: $hotels := hotels \cup \{t \mapsto h\}$
        act2: $tHotel(t \mapsto h) := time$
    end
Event BookHotel ⟨ordinary⟩ $\widehat{=}$
extends BookHotel
    any
        $t$
        $h$
    where
        grd_seq_:  $t \mapsto h \in SelectHotel$
        grd_self:  $t \mapsto h \notin BookHotel$
        grd1_xor:  $t \mapsto h \notin RemoveHotel$
        grd_interrupt:  $t \notin Quit$
        grd1:  $(t \mapsto h) \in hotels$
        grd2:  $time \leq (tHotel(t \mapsto h) + deadline)$
    then
        act: $BookHotel := BookHotel \cup \{t \mapsto h\}$
    end
Event RemoveHotel ⟨ordinary⟩ $\widehat{=}$
extends RemoveHotel
    any
        $t$
        $h$
    where
        grd_seq_:  $t \mapsto h \in SelectHotel$
        grd_self:  $t \mapsto h \notin RemoveHotel$
        grd1_xor:  $t \mapsto h \notin BookHotel$
        grd_interrupt:  $t \notin Quit$
        grd2:  $(t \mapsto h) \in hotels$
    then
        act: $RemoveHotel := RemoveHotel \cup \{t \mapsto h\}$
        act1: $hotels := hotels \setminus \{t \mapsto h\}$
        act2: $tHotel := \{t \mapsto h\} \lhd tHotel$
    end
Event CompHotelItin ⟨ordinary⟩ $\widehat{=}$
extends CompHotelItin
    any
        $t$
    where
        grd_seq_:  $t \in StartTrip$
        grd_self:  $t \notin CompHotelItin$
        grd_par:  $SelectHotel[\{t\}] = BookHotel[\{t\}] \cup RemoveHotel[\{t\}]$
        grd_interrupt:  $t \notin Quit$
    then
        act: $CompHotelItin := CompHotelItin \cup \{t\}$
    end
Event Checkout ⟨ordinary⟩ $\widehat{=}$
extends Checkout
    any
        $t$
    where
        grd_seq_f:  $(t \in CompFlightItin \land t \in CompCarItin \land t \in CompHotelItin)$
        grd_self:  $t \notin Checkout$
        grd_interrupt:  $t \notin Quit$
    then
        act: $Checkout := Checkout \cup \{t\}$
    end
```

```
Event Quit ⟨ordinary⟩ ≙
      refines Quit
extends Quit
      any
            t
            th
            tf
            tc
      where
            grd_seq:  $t \in StartTrip$
            grd_self:  $t \notin Quit$
            grd_interrupt:  $t \notin Checkout$
            grd1:  $th = \{t\} \lhd hotels$
            grd2:  $tf = \{t\} \lhd flights$
            grd3:  $tc = \{t\} \lhd cars$
      then
            act: $Quit := Quit \cup \{t\}$
            act1: $flights := \{t\} \unlhd flights$
            act2: $hotels := \{t\} \unlhd hotels$
            act3: $cars := \{t\} \unlhd cars$
            act4: $tFlight := tf \unlhd tFlight$
            act5: $tHotel := th \unlhd tHotel$
            act6: $tCar := tc \unlhd tCar$
      end
Event clock ⟨ordinary⟩ ≙
      begin
            act: $time := time + 1$
      end
END
```

Appendix D

Functions Used in Formalising the ERS Translation Rules

In this chapter we present the auxiliary functions reused from Salehi Fathabadi (2012) and show how they are updated to include the newly added ERS extensions.

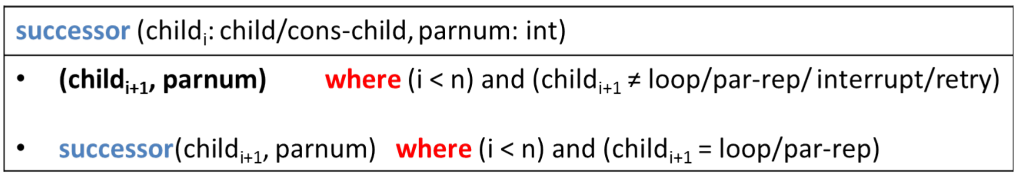
D.1 Predecessor and Successor Functions

In Figure D.1, the *predecessor* function is updated to include the *par-replicator*, which is treated similar to a *loop* by finding its *predecessor* due to the zero execution case in both combinators. The abstract flow is distinguished from a refined flow by having no solid lines, and in the case there is *no predecessor*, sequencing is not defined however the type of the leaf will be defined.

predecessor (child$_i$: child/cons-child, *par: list of parameter(s), sw: boolean) =	
• (child$_{i-1}$, *par) =	**where** (i > 1) and (child$_{i-1}$ ≠ (loop/par-rep)
• **predecessor** (child$_{i-1}$, *par, sw)	**where** (i > 1) and (child$_{i-1}$ = (loop/par-rep)
• **“no predecessor”**	**where** (i = 1) and (sw = 0)
• **“no predecessor”**	**where** (i = 1) and (sw = 1) and (parentFlow(child$_i$) is an abstract flow)
• **predecessor** (parent(child$_i$), *par, sw)	**where** (i = 1) and (sw = 1) and (parentFlow(child$_i$) is not (all/some/one) child)
• **predecessor** (parent(child$_i$), *par / p, sw)	**where** (i = 1) and (sw = 1) and (parentFlow(child$_i$) is a (all/some/one) child

Figure D.1: Updated *predecessor* Function

In Figure D.2 the *successor* function is updated to include the *par-rep*, which is treated similar to a *loop*, whenever encountered find its *successor*. The *successor* function is used in the *par-replicator* and the *loop* translation rules, the current definition will work since both **cannot** be the last events in a flow, and since we introduced the restriction that the follow-on event of a *loop* and a *par-replicator* **cannot** be an exception handler (*interrupt*, *retry*).



successor (child$_i$: child/cons-child, parnum: int)

- **(child$_{i+1}$, parnum)** where (i < n) and (child$_{i+1}$ ≠ loop/par-rep/ interrupt/retry)
- successor(child$_{i+1}$, parnum) where (i < n) and (child$_{i+1}$ = loop/par-rep)

Figure D.2: Updated *successor* Function

D.2 Traversing Functions

The traversing functions are functions that help in traversing down the tree to get to the last refinement level. In all of these functions we do not consider the case of the *loop*, because the *loop* does not have a variable that records its execution, hence the restriction the *loop* cannot be the first child of a flow (Salehi Fathabadi, 2012). All these functions are updated to include the *par-replicator* and the exception handling combinators. In the case of the *par-replicator* the function is applied to the *par-replicator* child and its *successor* child, while in the case of an exception handling combinator, the function will be applied to both the *normal-child* and the *interrupting-child*.

The traversing functions include three functions, the *conjunction_of_leaves* function, which is presented in Figure D.3, the output of this function is the conjunction of the guards. The *disjunction_of_leaves* as shown in Figure D.4 will result in the disjunction of the guards. Finally the *union_of_leaves* function which will result in the union of the control variables of the leaf events, as presented in Figure D.5.

D.3 Sequencing Functions

The sequencing of a leaf event depends on its predecessor. In Figure D.6, we have updated the *build_seq_grd* function to include the exception handling combinators, where in the case of *retry* we only care about the *normal-child* whereas an *interrupt-combinator* takes into account both the *normal-child* and the *interrupting-child*. The *build_seq_grd* depends on the function *seq_grd* (Figures D.7 and D.8) which is still the same as described in Salehi Fathabadi (2012), as this function only checks the common parameters and the *all-replicator* parameter. The *build_seq_inv* function is also updated in a similar way to

conjunction_of_leaves(ch: child/cons-child, parnum: int)

Output Operation:

conjunction_of_leaves(leaf (name), parnum) = **name = FALSE** **where** parnum = 0
conjunction_of_leaves(leaf (name), parnum) = **name = ∅** **where** parnum > 0

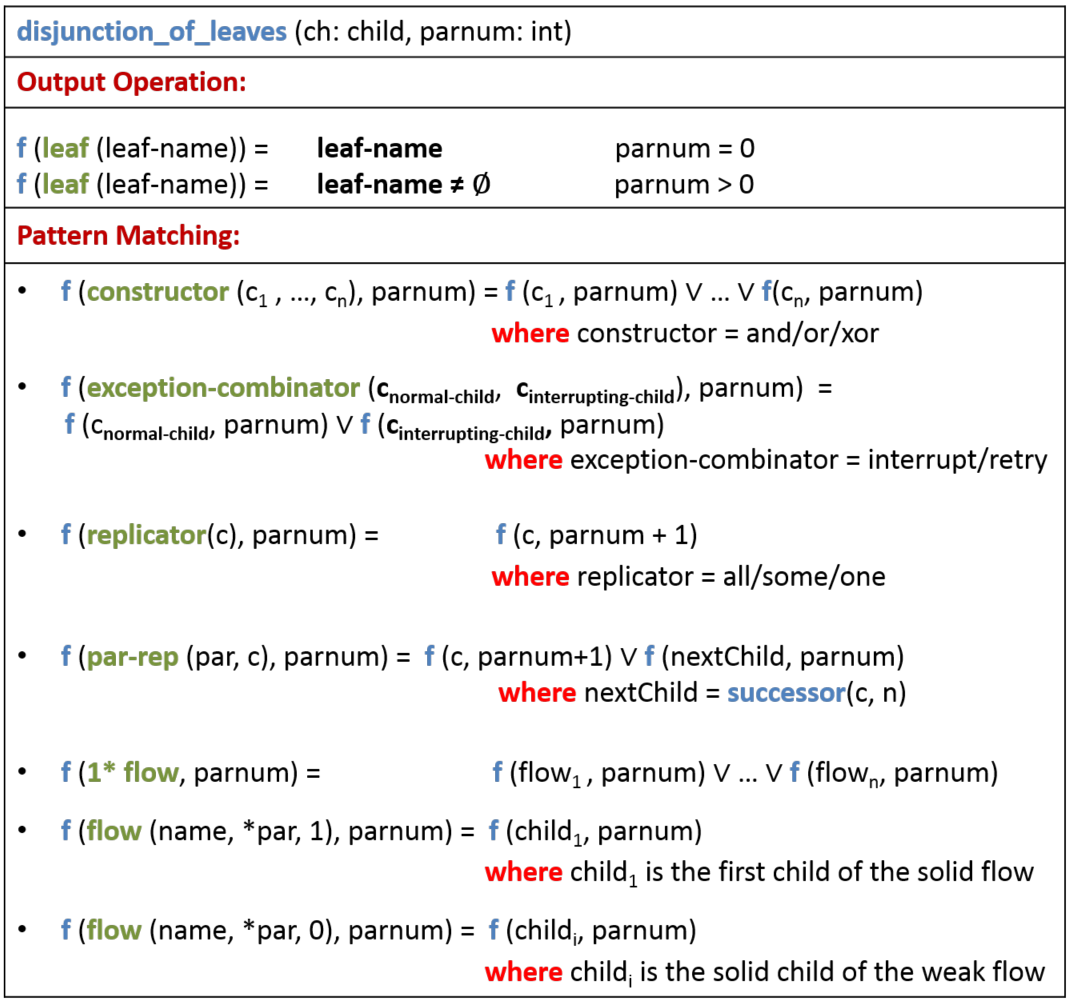
Traversing Steps:

- f (constructor (c_1 , ..., c_n), parnum) = f (c_1, parnum) ∧... ∧ f (c_n, parnum)
  where constructor = and/or/xor
- f (exception-combinator ($c_{normal\text{-}child}$, $c_{interrupting\text{-}child}$), parnum) =
  f ($c_{normal\text{-}child}$, parnum) ∧ f ($c_{interrupting\text{-}child}$, parnum)
  where exception-combinator = interrupt/retry
- f (replicator (par, c), parnum) = f (c, parnum+1)
  where replicator = all/some/one
- f (par-rep (par, c), parnum) = f (c, parnum+1) ∧ f (nextChild, parnum)
  where nextChild = successor(c, n)
- f (1* flow, parnum) = f ($flow_1$, parnum) ∧ ...∧ f ($flow_n$, parnum)
- f (flow (name, *par, 1), parnum) = f ($child_1$, parnum)
  where $child_1$ is the first child of the strong flow
- f (flow (name, *par, 0), parnum) = f ($child_1$, parnum)
  where $child_i$ is the solid child of the weak flow

Figure D.3: Updated *conjunction_of_leaves* Function

the *build_seq_grd* function, but it is not shown here because it is not used in our newly introduced translation rules presented in Chapter 8.

In Figure D.8, the *seq_grd* function (Figure D.8) needs to determine the possible common parent parameters ($p_1 \cdots p_i$), which are the first in the parameter list, since any added new parameters will be added to the end of the parameter list. We also need the *all-replicator* parameter which affects the sequencing guards and invariants. In Figure D.7 from (Salehi Fathabadi, 2012) presents the definitions of X, Y, W and K which are used in defining the sequencing between two leaves.



disjunction_of_leaves (ch: child, parnum: int)

Output Operation:

f (**leaf** (leaf-name)) =	**leaf-name**	parnum = 0
f (**leaf** (leaf-name)) =	**leaf-name ≠ ∅**	parnum > 0

Pattern Matching:

- f (**constructor** (c_1 , ..., c_n), parnum) = f (c_1 , parnum) ∨ ... ∨ f(c_n, parnum)
  where constructor = and/or/xor
- f (**exception-combinator** ($\mathbf{c}_{\textbf{normal-child}}$, $\mathbf{c}_{\textbf{interrupting-child}}$), parnum) =
  f ($c_{\textbf{normal-child}}$, parnum) ∨ f ($\mathbf{c}_{\textbf{interrupting-child}}$, parnum)
  where exception-combinator = interrupt/retry
- f (**replicator**(c), parnum) = f (c, parnum + 1)
  where replicator = all/some/one
- f (**par-rep** (par, c), parnum) = f (c, parnum+1) ∨ f (nextChild, parnum)
  where nextChild = **successor**(c, n)
- f (**1* flow**, parnum) = f (flow_1 , parnum) ∨ ... ∨ f (flow_n, parnum)
- f (**flow** (name, *par, 1), parnum) = f (child_1, parnum)
  where child_1 is the first child of the solid flow
- f (**flow** (name, *par, 0), parnum) = f (child_i, parnum)
  where child_i is the solid child of the weak flow

Figure D.4: Updated *disjunction_of_leaves* Function

union_of_leaves(ch: child/cons-child, n: int)

Output Operation:

union_of_leaves(leaf (name), n) = **dom$_1$ (…dom$_n$ (name) …)**

Traversing Steps:

- **f** (**constructor** (c_1 , …, c_n), n) = **f** (c_1, n) ∪… ∪ **f** (c_n, n)
  where constructor = and/or/xor

- **f** (**exception-combinator** (**c**$_{\text{normal-child}}$, **c**$_{\text{interrupting-child}}$), n) =
  f ($c_{\text{normal-child}}$, n) ∪ **f** (**c**$_{\text{interrupting-child}}$, n)
  where exception-combinator = interrupt/retry

- **f** (**replicator** (par, c), n) = **f** (c, n+1)
  where replicator = all/some/one

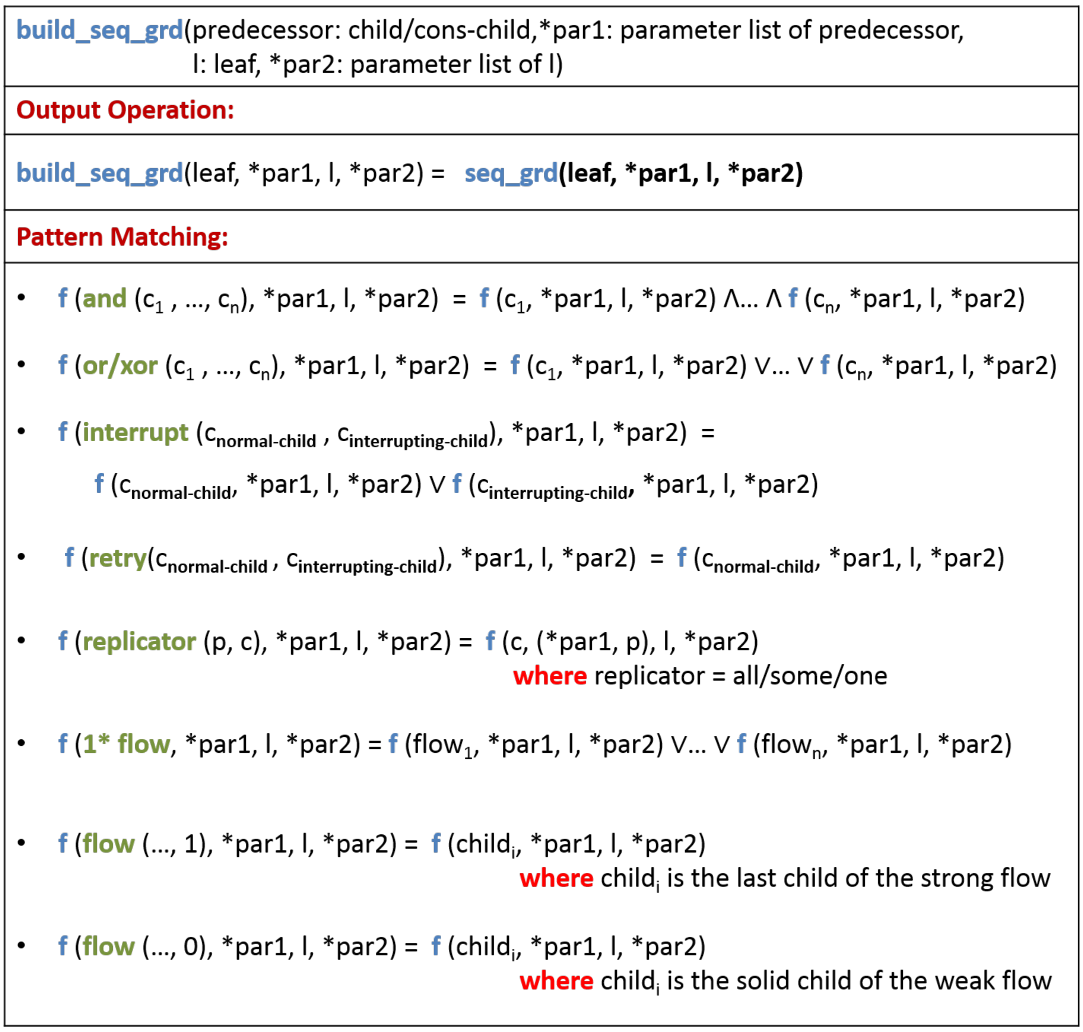
- **f** (**par-rep**(par, c), n) = **f** (c, n+1) ∪ **f** (nextChild, n)
  where nextChild = **successor**(c, n)

- **f** (**1* flow**, n) = **f** (flow_1, n) ∪… ∪ **f** (flow_n, n)

- **f** (**flow** (name, *par, 1), n) = **f** (child_1, n)
  where child_1 is the first child of the strong flow

- **f** (**flow** (name, *par, 0), n) = **f** (child_1, n)
  where child_i is the solid child of the weak flow

Figure D.5: Updated *union_of_leaves* Function



build_seq_grd(predecessor: child/cons-child,*par1: parameter list of predecessor,
l: leaf, *par2: parameter list of l)

Output Operation:

build_seq_grd(leaf, *par1, l, *par2) = **seq_grd(leaf, *par1, l, *par2)**

Pattern Matching:

- **f** (**and** (c_1 , …, c_n), *par1, l, *par2) = **f** (c_1, *par1, l, *par2) $\wedge$… $\wedge$ **f** (c_n, *par1, l, *par2)
- **f** (**or/xor** (c_1 , …, c_n), *par1, l, *par2) = **f** (c_1, *par1, l, *par2) $\vee$… $\vee$ **f** (c_n, *par1, l, *par2)
- **f** (**interrupt** ($c_{\text{normal-child}}$, $c_{\text{interrupting-child}}$), *par1, l, *par2) =
  f ($c_{\text{normal-child}}$, *par1, l, *par2) $\vee$ **f** ($c_{\text{interrupting-child}}$, *par1, l, *par2)
- **f** (**retry**($c_{\text{normal-child}}$, $c_{\text{interrupting-child}}$), *par1, l, *par2) = **f** ($c_{\text{normal-child}}$, *par1, l, *par2)
- **f** (**replicator** (p, c), *par1, l, *par2) = **f** (c, (*par1, p), l, *par2)
  where replicator = all/some/one
- **f** (**1* flow**, *par1, l, *par2) = **f** (flow_1, *par1, l, *par2) $\vee$… $\vee$ **f** (flow_n, *par1, l, *par2)
- **f** (**flow** (…, 1), *par1, l, *par2) = **f** (child_i, *par1, l, *par2)
  where child_i is the last child of the strong flow
- **f** (**flow** (…, 0), *par1, l, *par2) = **f** (child_i, *par1, l, *par2)
  where child_i is the solid child of the weak flow

Figure D.6: Updated *build_seq_grd* Function

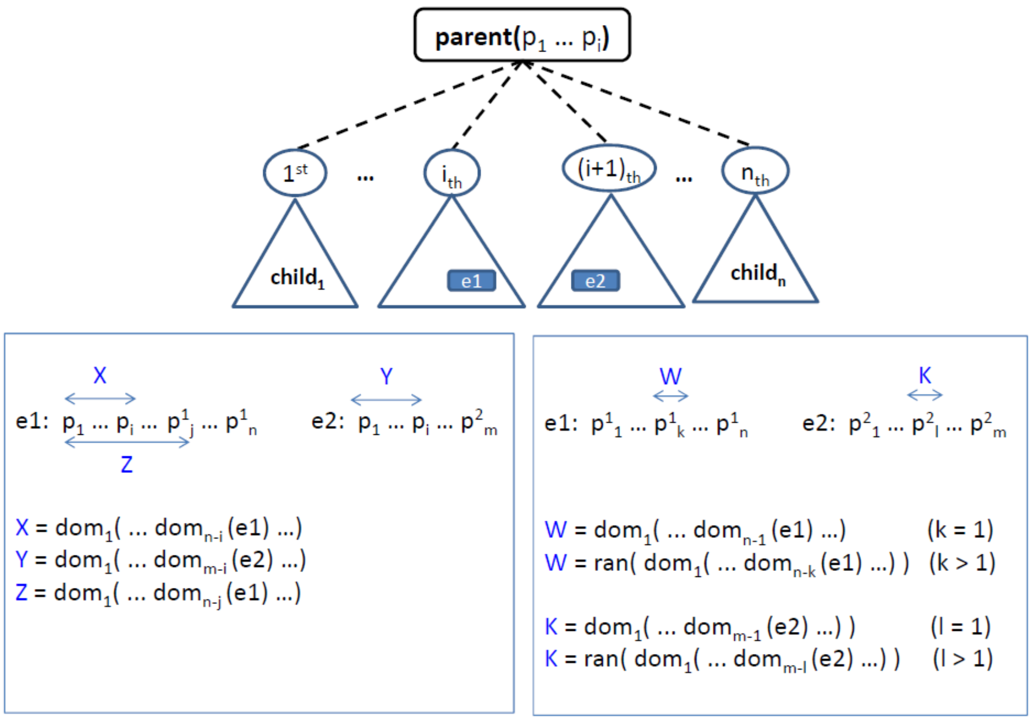
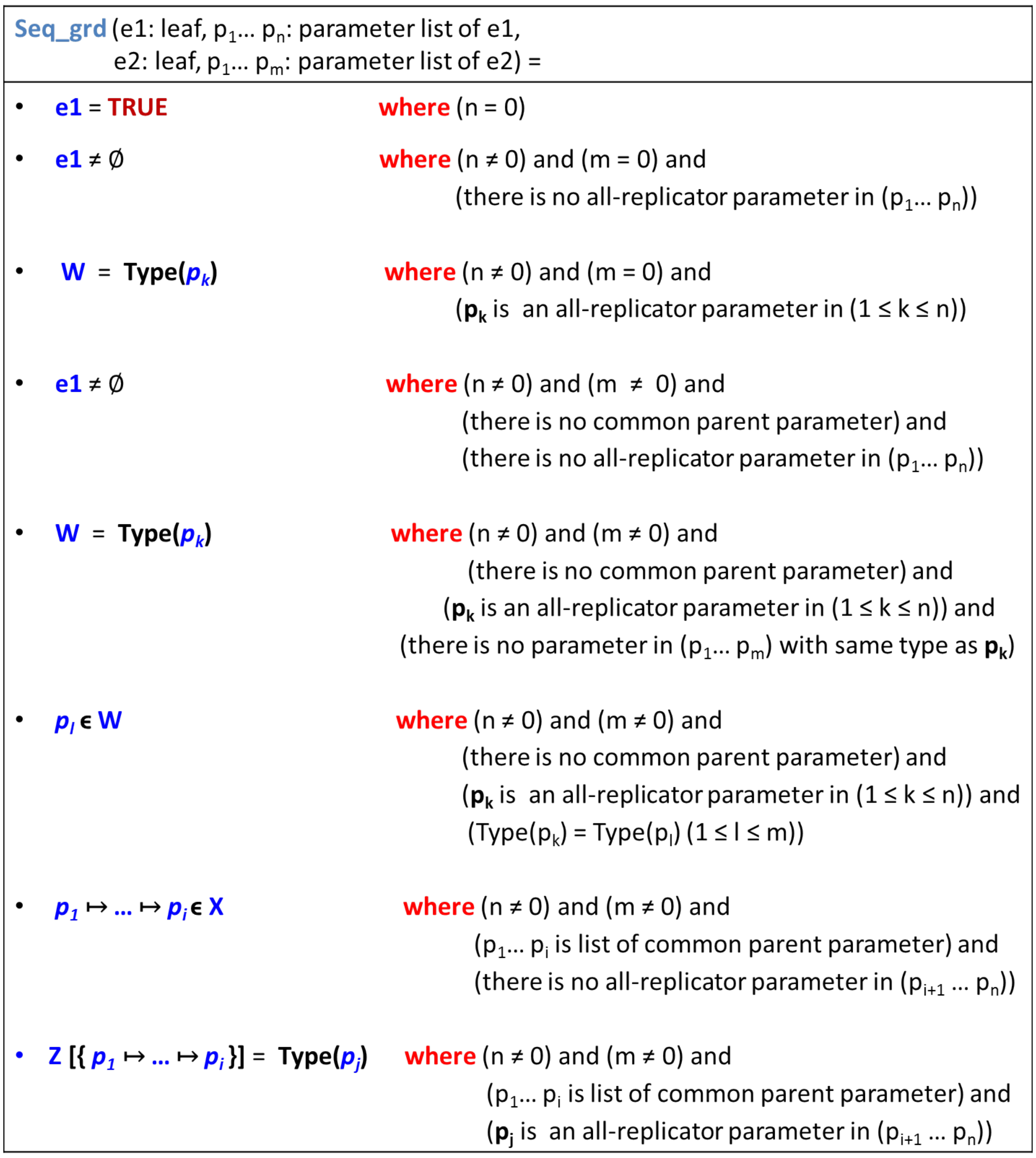

parent(p_1 … p_i)
1st
…
i_{th}
$(i+1)_{th}$
…
n_{th}
$child_1$
e1
e2
$child_n$
X
Y
e1: p_1 … p_i … p^1_j … p^1_n
e2: p_1 … p_i … p^2_m
Z
X = dom_1(… dom_{n-i} (e1) …)
Y = dom_1(… dom_{m-i} (e2) …)
Z = dom_1(… dom_{n-j} (e1) …)
W
K
e1: p^1_1 … p^1_k … p^1_n
e2: p^2_1 … p^2_l … p^2_m
W = dom_1(… dom_{n-1} (e1) …) (k = 1)
W = ran(dom_1(… dom_{n-k} (e1) …)) (k > 1)
K = dom_1(… dom_{m-1} (e2) …)) (l = 1)
K = ran(dom_1(… dom_{m-l} (e2) …)) (l > 1)

Figure D.7: Sequencing between Two Leaf Events (Salehi Fathabadi, 2012)



Seq_grd (e1: leaf, $p_1 \ldots p_n$: parameter list of e1,
e2: leaf, $p_1 \ldots p_m$: parameter list of e2) =

• **e1** = **TRUE**	**where** $(n = 0)$
• **e1** $\neq \emptyset$	**where** $(n \neq 0)$ and $(m = 0)$ and (there is no all-replicator parameter in $(p_1 \ldots p_n)$)
• **W** = **Type(p_k)**	**where** $(n \neq 0)$ and $(m = 0)$ and ($\mathbf{p_k}$ is an all-replicator parameter in $(1 \leq k \leq n)$)
• **e1** $\neq \emptyset$	**where** $(n \neq 0)$ and $(m \neq 0)$ and (there is no common parent parameter) and (there is no all-replicator parameter in $(p_1 \ldots p_n)$)
• **W** = **Type(p_k)**	**where** $(n \neq 0)$ and $(m \neq 0)$ and (there is no common parent parameter) and ($\mathbf{p_k}$ is an all-replicator parameter in $(1 \leq k \leq n)$) and (there is no parameter in $(p_1 \ldots p_m)$ with same type as $\mathbf{p_k}$)
• $p_l \in$ **W**	**where** $(n \neq 0)$ and $(m \neq 0)$ and (there is no common parent parameter) and ($\mathbf{p_k}$ is an all-replicator parameter in $(1 \leq k \leq n)$) and $(\mathrm{Type}(p_k) = \mathrm{Type}(p_l)\ (1 \leq l \leq m))$
• $p_1 \mapsto \ldots \mapsto p_i \in$ **X**	**where** $(n \neq 0)$ and $(m \neq 0)$ and ($p_1 \ldots p_i$ is list of common parent parameter) and (there is no all-replicator parameter in $(p_{i+1} \ldots p_n)$)
• **Z** $[\{ p_1 \mapsto \ldots \mapsto p_i \}]$ = **Type(p_j)**	**where** $(n \neq 0)$ and $(m \neq 0)$ and ($p_1 \ldots p_i$ is list of common parent parameter) and ($\mathbf{p_j}$ is an all-replicator parameter in $(p_{i+1} \ldots p_n)$)

Figure D.8: Original *seq_grd* Function (Salehi Fathabadi, 2012)

References

Abrial, J.-R. (1996). *The B-Book: Assigning Programs to Meanings*. Cambridge University Press.

Abrial, J.-R. (2007). Formal methods: Theory becoming practice. *Journal of Universal Computer Science*, 13(5):619–628.

Abrial, J.-R. (2009). Event model decomposition. Technical report, ETH Zurich.

Abrial, J.-R. (2010). *Modeling in Event-B: System and Software Engineering*. Cambridge University Press.

Abrial, J.-R., Butler, M., Hallerstede, S., Hoang, T., Mehta, F., and Voisin, L. (2010). Rodin: an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer*, 12:447–466.

Abrial, J.-R., Butler, M., Hallerstede, S., and Voisin, L. (2006). An open extensible tool environment for Event-B. In Liu, Z. and He, J., editors, *ICFEM 2006*, volume Lectur. Springer. Event Dates: November 2006.

Abrial, J.-R. and Hallerstede, S. (2007). Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundam. Inf.*, 77(1-2):1–28.

Abrial, J.-R., Hallerstede, S., Mehta, F., Metayer, C., and Voisin, L. (2005). Specification of basic tools and platform. rodin project deliverable 3.3 (d10). Available online at http://rodin.cs.ncl.ac.uk/deliverables/rodinD10.pdf.

Amálio, N., Polack, F., and Stepney, S. (2005). An object-oriented structuring for Z based on views. In *ZB 2005: Formal Specification and Development in Z and B*, pages 262–278. Springer.

Amálio, N., Polack, F., and Stepney, S. (2007). Frameworks based on templates for rigorous model-driven development. *Electronic Notes in Theoretical Computer Science*, 191:3–23.

Amálio, N., Polack, F., and Stepney, S. (2010). UML+ Z: Augmenting UML with Z. *Software Specification Methods*, page 81.

Back, R.-J. (1990). Refinement calculus, part ii: Parallel and reactive programs. In *Stepwise Refinement of Distributed Systems Models, Formalisms, Correctness*, pages 67–93. Springer.

Ball, E. and Butler, M. (2007). Event-B Patterns for Specifying Fault-Tolerance in Multi-Agent Interaction. In *Methods, Models and Tools for Fault Tolerance.* Event Dates: 3rd July 2007.

Ben Younes, A. and Ben Ayed, L. (2008). From UML activity diagrams to event B for the specification and the verification of workflow applications. In *2008 IEEE 32nd International Computer Software and Applications Conference (COMPSAC)*, pages 643–8. IEEE.

Ben Younes, A., Jemni Ben Ayed, L., and Hlaoui, Y. (2012). UML AD Refinement Patterns for Modeling Workflow Applications. In Xiaoying Bai, Belli, F., Bertino, E., Chang, C., Elci, A., Seceleanu, C., Haihua Xie, and Zulkernine, M., editors, *Proceedings of the 2012 IEEE 36th IEEE Annual Computer Software and Applications Conference Workshops (COMPSACW)*, pages 236–41. IEEE Computer Society.

Bodeveix, J. P., Filali, M., and Muñoz, C. A. (1999). A Formalization of the B-Method in Coq and PVS. *LNCS*, 1709:33–49.

Börger, E. (2003). The ASM Ground Model Method as a Foundation of Requirements Engineering. In Dershowitz, N., editor, *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 145–160. Springer Berlin Heidelberg.

Börger, E. (2005). The ASM Method for System Design and Analysis. A Tutorial Introduction. In Gramlich, B., editor, *Frontiers of Combining Systems*, volume 3717 of *Lecture Notes in Computer Science*, pages 264–283. Springer Berlin Heidelberg.

Börger, E. (2007a). A critical analysis of workflow patterns. *ASM*, 7:12–15.

Börger, E. (2007b). Modeling workflow patterns from first principles. In *Conceptual Modeling-ER 2007*, pages 1–20. Springer.

Börger, E. (2012). Approaches to modeling business processes: a critical analysis of bpmn, workflow patterns and yawl. *Software & Systems Modeling*, 11(3):305–318.

Börger, E., Cavarra, A., and Riccobene, E. (2000). *An ASM Semantics for UML Activity Diagrams*, pages 293–308. Springer Berlin Heidelberg, Berlin, Heidelberg.

Börger, E. and Sörensen, O. (2011). BPMN Core Modeling Concepts: Inheritance-Based Execution Semantics. In Embley, D. W. and Thalheim, B., editors, *Handbook of Conceptual Modeling*, pages 287–332. Springer Berlin Heidelberg.

Bowen, J. P. (2003). *Formal Specification and Documentation using Z: A Case Study Approach.* International Thomson Computer Press London.

Bowen, J. P. and Hinchey, M. G. (1995). Seven more myths of formal methods. *Software, IEEE*, 12(4):34–41.

Bryans, J. and Wei, W. (2010). Formal Analysis of BPMN Models Using Event-B. In Kowalewski, S. and Roveri, M., editors, *Formal Methods for Industrial Critical Systems*, volume 6371 of *Lecture Notes in Computer Science*, pages 33–49. Springer Berlin Heidelberg.

Butler, M. (2006). Synchronisation-based Decomposition for Event B. In Jones, C., editor, *Rodin Deliverable D19: Intermediate report on methodology*, pages 47–57. Available online at http://rodin.cs.ncl.ac.uk/deliverables/D19.pdf.

Butler, M. (2009a). Decomposition Structures for Event-B. In Leuschel, M. and Wehrheim, H., editors, *Integrated Formal Methods*, volume 5423 of *Lecture Notes in Computer Science*, pages 20–38. Springer Berlin Heidelberg.

Butler, M. (2009b). Incremental Design of Distributed Systems with Event-B. Chapter: 4.

Butler, M. (2013). Mastering System Analysis and Design through Abstraction and Refinement. Lectures from Marktoberdorf Summer School 2012.

Butler, M. and Ferreira, C. (2000). A Process Compensation Language. In Grieskamp, W., Santen, T., and Stoddart, B., editors, *Integrated Formal Methods IFM2000*, volume 1945, page 61. Springer.

Butler, M. and Ferreira, C. (2003). Using B Refinement to Analyse Compensating Business Processes. In *ZB 2003: Third International Conference of B and Z Users*, volume LNCS 2. Springer. Event Dates: June.

Butler, M., Ferreira, C., and Ng, M. Y. (2005a). Precise Modelling of Compensating Business Transactions and its Application to BPEL. *Journal of Universal Computer Science*, 11(5):712–743.

Butler, M. and Hallerstede, S. (2007). The Rodin Formal Modelling Tool. BCS-FACS Christmas 2007 Meeting - Formal Methods In Industry, London.

Butler, M., Hoare, C., and Ferreira, C. (2005b). *A trace semantics for long-running transactions*, volume Lectur, pages 133–150. Springer. Event Dates: July 2004.

Butler, M., Leuschel, M., and Snook, C. (2005c). Tools for system validation with B abstract machines. In *ASM 2005: 12th International Workshop on Abstract State Machines*. Event Dates: March 2005.

Butler, M. and Yadav, D. (2008). An incremental development of the mondex system in event-B. *Formal Aspects of Computing*, 20(1):61–77.

Butler, M. J. (2000). csp2B: A Practical Approach to Combining CSP and B. *Formal Aspects of Computing*, 12:182–196.

Cansell, D. and Mery, D. (2003). Foundations of the B method. *COMPUTING AND INFORMATICS*, 22(3-4):221–256.

Casati, F., Ceri, S., Paraboschi, S., and Pozzi, G. (1999). Specification and Implementation of Exceptions in Workflow Management Systems. *ACM Transactions on Database Systems*, 24(3):405 – 451.

Chessell, M., Griffin, C., Vines, D., Butler, M., Ferreira, C., and Henderson, P. (2002). Extending the concept of transaction compensation. *IBM Systems Journal*, 41(4):743–758.

Chinosi, M. and Trombetta, A. (2012). BPMN: An introduction to the standard. *COMPUTER STANDARDS & INTERFACES*, 34(1):124 – 134.

Damchoom, K. and Butler, M. (2009). Applying Event and Machine Decomposition to a Flash-Based Filestore in Event-B. In *SBMF 2009*, volume 5902, pages 134–152. Springer LNCS. Springer LNCS 5902 Event Dates: 19-21 August.

Damchoom, K., Butler, M., and Abrial, J.-R. (2008). Modelling and proof of a Tree-structured File System in Event-B and Rodin. In *ICFEM 2008*, volume LNCS 5, pages 25–44. Springer. Springer LNCS 5256.

Darimont, R. and van Lamsweerde, A. (1996). Formal refinement patterns for goal-driven requirements elaboration. In *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, SIGSOFT '96, pages 179–190, New York, NY, USA. ACM.

de Sousa, T., Muniz Silva, P., and Snook, C. (2012). A practical event-B Refinement Method Based On a UML-driven development process. In Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., and Riccobene, E., editors, *Abstract State Machines, Alloy, B, VDM, and Z. Proceedings Third International Conference, ABZ 2012*, pages 357–60, Berlin, Germany. Springer-Verlag. Abstract State Machines, Alloy, B, VDM, and Z. Third International Conference, ABZ 2012, 18-21 June 2012, Pisa, Italy.

de Sousa, T., Snook, C., and Muniz Silva, P. (2011). A proposal for extending UML-B to support a conceptual model. *Innovations and Systems and Software Engineering*, 7:293–301.

Dghaym, D., Butler, M., and Fathabadi, A. S. (2013). Evaluation of graphical control flow management approaches for Event-B modelling. In *AVoCS 2013: 13th International Workshop on Automated Verification of Critical Systems*, volume 66.

Dghaym, D., Trindade, M. G., Butler, M., and Fathabadi, A. S. (2016). *A Graphical Tool for Event Refinement Structures in Event-B*, pages 269–274. Springer International Publishing, Cham.

Eder, J. and Liebhart, W. (1996). Workflow recovery. *Proceedings First IFCIS International Conference on Cooperative Information Systems*, page 124.

Fathabadi, A. S., Butler, M., and Rezazadeh, A. (2014). Language and tool support for event refinement structures in Event-B. *Formal Aspects of Computing*.

Group, L. P. W. (2006). Little-JIL 1.5 Language Report. Technical report, University of Massachusetts Amherst. http://www.umass.edu/eei/EEI%20Website%20Articles/Little-JIL%201.5%20Language%20Report.pdf.

Hall, A. (1990). Seven myths of formal methods. *Software, IEEE*, 7(5):11–19.

Hallerstede, S. and Leuschel, M. (2011). Finding Deadlocks of Event-B Models by Constraint Solving. *Electronic Notes in Theoretical Computer Science*, 280.

Hallerstede, S., Leuschel, M., and Plagge, D. (2010). Refinement-animation for Event-B - towards a method of validation. In *Abstract State Machines, Alloy, B and Z*, pages 287–301. Springer.

Hallerstede, S. and Snook, C. (2011). Refining nodes and edges of state machines. In *ICFEM 2011: 13th International Conference on Formal Engineering Methods*. Event Dates: 26th–28th October 2011.

Hantsfire.gov.uk (2000). The control room. Available online at http://www.hantsfire.gov.uk/theservice/operations/firecontrol/thecontrolroom.htm [Accessed: 6 Mar 2013].

Hoare, C. A. R. et al. (1985). *Communicating sequential processes*, volume 178. Prentice-hall Englewood Cliffs.

Idani, A., Ledru, Y., and Bert, D. (2006). A Reverse-Engineering Approach to Understanding B Specifications with UML Diagrams. In *Software Engineering Workshop, 2006. SEW '06. 30th Annual IEEE/NASA*, pages 97–106.

Jackson, M. A. (1983). *System Development*. Englewood Cliffs, N.J. : Prentice-Hall.

Jensen, K. (1997). A brief introduction to coloured Petri Nets. In Brinksma, E., editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1217 of *Lecture Notes in Computer Science*, pages 203–208. Springer Berlin Heidelberg.

Jones, C. B. (1990). *Systematic software development using VDM (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Kolovos, D., Rose, L., García-Domínguez, A., and Paige, R. (2014). *The epsilon Book*. Eclipse.org. http://www.eclipse.org/epsilon/doc/book/.

Laleau, R., Semmak, F., Matoussi, A., Petit, D., Hammad, A., and Tatibouet, B. (2010). A first attempt to combine sysml requirements diagrams and b. *Innovations in Systems and Software Engineering*, 6(1-2):47–54.

Lausdahl, K., Lintrup, H., and Larsen, P. (2009). Connecting UML and VDM++ with Open Tool Support. In Cavalcanti, A. and Dams, D., editors, *FM 2009: Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 563–578. Springer Berlin Heidelberg.

Lerner, B. S., Christov, S., Osterweil, L. J., Bendraou, R., Kannengiesser, U., and Wise, A. (2010). Exception handling patterns for process modeling. *IEEE Transactions on Software Engineering*, 36(2):162–183.

Leuschel, M. and Butler, M. (2003). ProB: A model checker for B. In Araki, K., Gnesi, S., and Mandrioli, D., editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag.

Lindsay, P. A., Jones, C. B., Jones, K. D., and Moore, R. D. (1991). *Mural: A Formal Development Support System*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Matoussi, A., Gervais, F., and Laleau, R. (2011). A Goal-Based Approach to Guide the Design of an Abstract Event-B Specification. In *Engineering of Complex Computer Systems (ICECCS), 2011 16th IEEE International Conference on*, pages 139–148.

Mayer, R. E., Bove, W., Bryman, A., Mars, R., and Tapangco, L. (1996). When less is more: Meaningful learning from visual and verbal summaries of science textbook lessons. *Journal of educational psychology*, 88(1):64.

Metayer, C., Abrial, J.-R., and Voisin, L. (2005). Event-B language. RODIN Project Deliverable 3.2. Available online at http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf.

Murali, R., Ireland, A., and Grov, G. (2015). A Rigorous Approach to Combining Use Case Modelling and Accident Scenarios. In Havelund, K., Holzmann, G., and Joshi, R., editors, *NASA Formal Methods*, volume 9058 of *Lecture Notes in Computer Science*, pages 263–278. Springer International Publishing.

Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580.

OASIS (2007). Web Services Business Process Execution Language Version 2.0. Available online at http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

Object Management Group (2010). BPMN 2.0 by Example. http://www.omg.org/cgi-bin/doc?dtc/10-06-02 [Accessed: 07 Jul 2014].

Object Management Group (2011). Business Process Model and Notation (BPMN) Version 2.0. http://www.omg.org/spec/BPMN/2.0/ [Accessed: 04 Jul 2014].

Object Management Group (OMG) (2015). OMG Unified Modeling Language (version 2.5). Technical report. http://www.omg.org/spec/UML/2.5/PDF/.

OMG ([Last Updated: 14 Feb 2013]). Introduction to OMG's Unified Modeling Language (UML). Available online at http://www.omg.org/gettingstarted/what_is_uml.htm [Accessed: 20 June 2013].

omg.org (2012). UML 2.5. Available online at http://www.omg.org/spec/UML/2.5/Beta1/PDF [Accessed: 17 Mar 2013].

Ouyang, C., Dumas, M., Aalst, W. M. P. V. D., Hofstede, A. H. M. T., and Mendling, J. (2009). From Business Process Models to Process-oriented Software Systems. *ACM Trans. Softw. Eng. Methodol.*, 19(1).

Overell, P. and Crocker, D. (2008). Augmented BNF for syntax specifications: ABNF.

Owre, S., Rushby, J. M., , and Shankar, N. (1992). PVS: A prototype verification system. In Kapur, D., editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY. Springer-Verlag.

Razali, R., Snook, C. F., and Poppleton, M. R. (2007). Comprehensibility of UML-based Formal Model - A Series of Controlled Experiments. In *1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies (WEASELTech) 2007*, pages 25–30.

Ripon, S. H. and Butler, M. J. (2009). Pvs embedding of ccsp semantic models and their relationship. *Electronic Notes in Theoretical Computer Science*, 250(2):103 – 118.

Rumbaugh, J., Jacobson, I., and Booch, G. (1998). *The Unified Modelling Language Reference Manual*. Addison-Wesley.

Russell, N., Ter Hofstede, A. H., and Mulyar, N. (2006a). Workflow controlflow patterns: A revised view.

Russell, N., van der Aalst, W., and ter Hofstede, A. (2006b). Workflow exception patterns. *ADVANCED INFORMATION SYSTEMS ENGINEERING, PROCEEDINGS*, 4001:288 – 302.

Russell, N., van der Aalst, W. M., and Ter Hofstede, A. H. (2006c). Exception handling patterns in process-aware information systems. *BPM Center Report BPM-06-04, BPMcenter. org*, 208.

Said, M. Y., Butler, M., and Snook, C. (2009). Language and tool support for class and state machine refinement in UML-B. In Cavalcanti, A. and Dams, D., editors, *FM 2009: Formal Methods*, LNCS 5850, pages 579–595. Springer. This work has been presented at the IM FMT 2009 workshop of the IFM2009 conference, Dusseldorf, Germany on 16 February 2009.

Said, M. Y., Butler, M., and Snook, C. (2013). A Method of Refinement in UML-B. *Software and Systems Modeling*.

Salehi Fathabadi, A. (2012). *An approach to atomicity decomposition in the Event-B formal method*. PhD thesis, University of Southampton. Available online at http://eprints.soton.ac.uk/340357/.

Salehi Fathabadi, A. and Butler, M. (2010). Applying Event-B Atomicity Decomposition to a Multi Media Protocol. In Boer, F., Bonsangue, M., Hallerstede, S., and Leuschel, M., editors, *Formal Methods for Components and Objects*, volume 6286 of *Lecture Notes in Computer Science*, pages 89–104. Springer Berlin Heidelberg.

Salehi Fathabadi, A., Butler, M., and Rezazadeh, A. (2012). A Systematic Approach to Atomicity Decomposition in Event-B. In Eleftherakis, G., Hinchey, M., and Holcombe, M., editors, *Software Engineering and Formal Methods*, volume 7504 of *Lecture Notes in Computer Science*, pages 78–93. Springer Berlin Heidelberg.

Salehi Fathabadi, A., Rezazadeh, A., and Butler, M. (2011). *Applying Atomicity and Model Decomposition to a Space Craft System in Event-B.*

Sarshogh, M. R. and Butler, M. (2011). Specification and refinement of discrete timing properties in Event-B. In *AVoCS 2011*. Event Dates: September 2011.

Savicks, V. and Snook, C. (2012). A Framework for Diagrammatic Modelling Extensions in Rodin. Technical report. In: Rodin Workshop, Fontainbleau.

Silva, R. and Butler, M. (2010). Shared Event Composition/Decomposition in Event-B. In *FMCO Formal Methods for Components and Objects*. Event Dates: 29 November - 1 December 2010.

Silva, R., Pascal, C., Hoang, T. S., and Butler, M. (2010). Decomposition Tool for Event-B. In *Workshop on Tool Building in Formal Methods - ABZ Conference*. Event Dates: 22nd February 2010.

Snook, C. (2014). iUML-B Statemachines: New Features and Usage Examples. In Butler, M. and Hallerstede, S., editors, *Proceedings of the 5th Rodin User and Developer Workshop, 2014*. University of Southampton.

Snook, C. and Butler, M. (2006). UML-B: Formal modeling and design aided by UML. *ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122.

Snook, C. and Butler, M. (2008). UML-B and Event-B: an integration of languages and tools. In *The IASTED International Conference on Software Engineering - SE2008*.

Spivey, J. M. (1992). The z notation: a reference manual. international series in computer science.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2008). *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison-Wesley Professional.

Storrle, H. (2004). Structured nodes in UML 2.0 activities. *Nordic Journal of Computing*, 11:279–302.

Syriani, E. and Ergin, H. (2012). Operational Semantics of UML Activity Diagram: An Application in Project Management.

van der Aalst, W. (1998). Three Good Reasons for Using a Petri-Net-Based Workflow Management System. In Wakayama, T., Kannapan, S., Khoong, C., Navathe, S., and Yates, J., editors, *Information and Process Integration in Enterprises*, volume 428 of *The Springer International Series in Engineering and Computer Science*, pages 161–182. Springer US.

van der Aalst, W. and ter Hofstede, A. (2005). YAWL: yet another workflow language. *Information Systems*, 30(4):245 – 275.

van der Aalst, W., ter Hofstede, A., Kiepuszewski, B., and Barros, A. (2003). Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51.

van Lamsweerde, A. (2008). Requirements engineering: from craft to discipline. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 238–249, New York, NY, USA. ACM.

Wang, C. W., Davies, J., and Welch, J. (2010). A Guarded Workflow Language and Its Formal Semantics. In *Theoretical Aspects of Software Engineering (TASE), 2010 4th IEEE International Symposium on*, pages 25–34.

Wiki.event-b.org (2012a). Atomicity Decomposition Plug-in User Guide - Event-B. Available online at `http://wiki.event-b.org/index.php/Atomicity_Decomposition_Plug-in_User_Guide` [Accessed: 2 Jul 2013].

Wiki.event-b.org (2012b). Media: Atomicity Decomposition. Available online at `http://wiki.event-b.org/images/Atomicity_Decomposition.pdf` [Accessed: 7 Jul 2013].

Wiki.event-b.org (2014). Event-B Statemachines. Available online at `http://wiki.event-b.org/index.php/Event-B_Statemachines` [Accessed: 06 Aug 2014].

Wiki.event-b.org (2015). Generic Event-B EMF extensions. In: `http://wiki.event-b.org/index.php/Generic_Event-B_EMF_extensions`.

Wing, J. (1990). A specifier's introduction to formal methods. *Computer*, 23(9):8–22.

Wise, A., Cass, A. G., Lerner, B. S., McCall, E. K., Osterweil, L. J., and Sutton, S. M. (2000). Using Little-JIL to coordinate agents in software engineering. In *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on*, pages 155–163.

Wong, P. Y. H. and Gibbons, J. (2011). Formalisations and Applications of BPMN. *Sci. Comput. Program.*, 76(8):633–650.

Woodcock, J. and Cavalcanti, A. (2002). *The Semantics of Circus*, pages 184–203. Springer Berlin Heidelberg, Berlin, Heidelberg.