Modeling Correct Safety Requirements Using KAOS and Event-B

By Kriangkrai Traichaiyaporn

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Associate Professor Toshiaki Aoki

September, 2013

Modeling Correct Safety Requirements Using KAOS and Event-B

By Kriangkrai Traichaiyaporn (1110203)

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Associate Professor Toshiaki Aoki

and approved by
Associate Professor Toshiaki Aoki
Professor Kokichi Futatsugi
Associate Professor Kazuhiro Ogata

August, 2013 (Submitted)

Abstract

Safety-critical systems are the systems whose failures can cause significant damage to life, property, and environment in which the systems are working on. One major causes of the system failures is the incorrectness of the safety requirements specifications described for developing the systems. Thus, the correctness of the safety requirements specification is crucial. Event-B is a famous formal specification language, which provides a refinement mechanism and a set of proof obligations for modeling and verifying the specifications. Event-B has a good potential for dealing with the correctness. However, Event-B lacks of the semantics of the correctness, and the mechanism to perform requirements analysis and elaboration. The semantics and the mechanism are necessary to ensure the correctness. In addition, there is no guideline for using the refinement mechanism. These shortcomings are hindrances for applying Event-B to the practical development of the safety-critical systems.

This thesis aims to propose an approach to overcome the shortcomings of Event-B. Firstly, the semantics of the properties preserved by the proof obligations are analyzed based on the semantics of the correctness defined in an evolutionary framework. This analysis claims that Event-B can preserve the correctness as defined in the evolutionary framework. Secondly, a new model is proposed to assist structuring and understanding Event-B. The model is named ORDER model. Thirdly, a set of refinement patterns for the ORDER model are created based on the patterns of the KAOS method, which is a goal-oriented requirements engineering method. The KAOS method has the capabilities for requirements analysis and elaboration by the use of goals of systems and the notions of goal refinement. Through the usage of the KAOS-based patterns, the ORDER model can inherit the capabilities of the KAOS method, and the refinement in Event-B can be used in the similar way as the goal refinement. By applying the evolutionary framework and the KAOS method to Event-B, the shortcoming of Event-B can be overcome.

Evaluation of the approach is described through case studies. The case studies shows that the KAOS-based patterns are capable to analyze and elaborate safety requirements. Then, the requirements can be easily modeled in Event-B for verifying the correctness. In summary, through the usage of KAOS and Event-B, a formal model, representing correct safety requirements specification, can be obtained.

Contents

List of Figures

List of Tables

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor, Associate Professor Toshiaki Aoki for his acute guidance, encouragement, and unlimited support throughout the duration of my master research. Without his support, this thesis could not have been completed. In addition, I am indebted to him for giving me this invaluable opportunity to study abroad and providing me the financial support.

I would like to thank Assistant Professor Yuki Chiba for his feedback and wise advices on my research. The quality of this research was significantly improved because of him.

I would like to thank the Hitachi Research Laboratory, Department of Green Mobility Research members Masaharu Nishi, Masahiro Matsubara, and Fumio Narisawa for providing this research with an appropriate industrial case study.

My last acknowledgements go to my family for their support, unconditional love and vital encouragement throughout my study and throughout my life.

Chapter 1

Introduction

Software safety requirements specifications are essential parts in the development of a computer-based safety critical system. This research intends to apply Event-B, a formal specification language, as a central approach for modeling and verifying the correctness of the software safety requirements specifications. However, it is not easy to apply Event-B for such purposes because of its shortcomings. In order to overcome the shortcomings of Event-B, we apply two frameworks proposed by others in Event-B. Those two frameworks are: an evolutionary framework of requirements correctness, and KAOS method.

To understand the broad overview of this research, this chapter starts with the brief explanation of the software safety requirements specifications, Event-B, the evolutionary framework, and the KAOS method. Then, the shortcomings of Event-B, which is to be solved by this research, are described. After describing the shortcomings, our proposed approach for overcoming the shortcomings is shortly explained. Finally, we provide a structure of this thesis.

1.1 Overview

1.1.1 Software safety requirements specifications

Safety-critical systems are the systems whose failures can cause significant damage to life, property, and environment in which the systems are working on. Some examples of safety-critical systems are automotive control systems, medical systems, aerospace control systems, and nuclear reactor systems. This makes the costs of failure of those systems to be potentially high, and not to mention the priceless costs from losing life. The safety-critical systems become increasingly computer-based nowadays because computer can help us manage the complicated behaviour of the systems. Thus, the safety-critical software errors, becoming parts of the system failures, are unacceptable; they should be avoided at all costs.

In software engineering, the earliest phase of software development process is to construct requirements specifications. Then, the other phases of software development are performed according to the specifications. If the requirements specifications are described incorrectly, the latter phases will be eventually affected by the incorrectness leading to the development of an incorrect system. In [Lut93], Lutz et al. have discovered that the safety-related software errors are mostly caused by the incorrectness of the safety requirements. Therefore, the verification for ensuring the correctness of a safety requirements specification is crucial for developing a "safe" safety-critical system. According to international standard for functional safety such as IEC61508 [SS10] and ISO 26262 [ISO11], a safety requirements specification for implementing a safety-critical system is gradually derived from safety goals of the system. Moreover, the standards recommend using formal methods for verifying the safety requirements specification.

1.1.2 Modeling and verifying the requirements specifications in Event-B

A famous formal approach for verifying the correctness of requirements is Event-B. Event-B [Abr10] is a formal specification language for modeling and verifying system requirements. The most important feature provided by Event-B is its refinement mechanism to stepwise transform an abstract specification into a concrete specification. By the refinement mechanism, a requirements specification can be gradually modeled into Event-B until all requirements in the specification are modeled.

Along the refinement, a set of proof obligations are discharged. The purposes of the proof obligations are to verify the consistency of a specification, and to preserve the functionality from its abstract specification. In typical industrial cases, there might be several thousand proof obligations to be discharged. It is difficult to manually generate and prove the proof obligations. Thus, the Rodin platform [ROD13] has been built to provide automated tool to generate and prove the proof obligations automatically or interactively. Besides, The Rodin platform can also support modeling in Event-B.

Event-B, together with the Rodin platform, has been successfully applied to several practical safety-critical systems. Some concrete examples are a train controller system [SAHZ11], hybrid systems [SAZ12], an air traffic information system [REB07], a spacecraft system [SFRB11], and a metro system [Sil12]. Event-B can be regarded as a method for correct-by-construction software development.

1.1.3 An evolutionary framework of requirements correctness

In requirements engineering, the definitions of the correctness of the requirements specifications can be defined from 2 points of views:

- From practical point of view, the correctness means the satisfaction of system goals. This is trivial, since a requirements specification must reflect the intention of a system.

- From formal point of view, the correctness usually means the combination of completeness and consistency of requirements. The completeness can be described as no requirements are missing from a specification. While, the consistency is that there is no internal contradiction among requirements in a specification. Most researches about the formal analysis of requirements specification also focus on checking the completeness and the consistency [HL96, Hei02, YSLS08, SP96, PMPS01].

Zowghi and Gervasi [ZG03] have proposed an evolutionary framework to demonstrate the idea of how requirements are evolved step-by-step and the causal relationship of the completeness and the consistency in the evolution. By the causal relationship, Zowghi and Gervasi inductively prove that if the completeness and the consistency are maintained throughout the evolution of requirements, then the final version of requirements is correct and such correctness is corresponding to the correctness in the practical point of view. This means that the correctness defined in this framework matches formal and practical points of view.

1.1.4 KAOS method

KAOS [VL09] is a goal-oriented methodology for requirements engineering. Goal-oriented requirements engineering (GORE) is a branch of requirements engineering focusing on justifying why a system is needed by specifying its top-level goals. Then, the goals are used for requirements specification process such as elicitation, evaluation, structuring, documentation, analysis, and evolution. The KAOS method supports the main concept of GORE through its UML-like models. The central model in KAOS approach is its goal model.

The goal model of KAOS is in the shape of a tree. The tree is for expressing relationship among goals of a system by showing how higher-level goals are refined into lower-level ones and, conversely, how lower-level goals contribute to higher-level goals. In a goal model, an AND-refinement link relates a parent goal to a set of sub-goals. It means that the parent goals can be satisfied by satisfying all the sub-goals in the refinement. From this, we can identify sub-goals and parent goals of a goal to construct a goal model. To prove that a goal tree is complete and consistent, KAOS supports using linear temporal logic to describe goals and to perform logical proofs. An efficient way to build a goal model is by using KAOS goal refinement patterns because we can reuse proofs and models from patterns. The goal refinement patterns is capable to reduce time and cost for constructing a goal model.

1.2 Shortcomings of Event-B

Even though Event-B is a good formal approach which is successful in applying to several practical case studies, there are some shortcomings of Event-B. These shortcomings potentially obstruct the usage of Event-B to model and verify the safety requirement specifications. The shortcomings focused in this research are as follows:

1. the correctness of requirements ensured by Event-B is based solely on the proof obligations of Event-B. However, as stated above, the correctness in term of requirements can be defined in two points of view. By considering only the proof

obligations of Event-B, it is difficult to relate the correctness preserved by the proof obligations with the correctness in those two points of view. From this, it is difficult to guarantee that a safety requirements specification is truly derived from its corresponding safety goals. To apply Event-B to model safety requirements, it is better to ensure what Event-B can exactly verify.

2. A method for analyzing and elaborating safety requirements specifications is needed to specify essential safety properties of the safety-critical systems. Without sufficiently specifying the safety properties, it is impossible to justify that a system is safe enough to operate. The problem is that Event-B itself does not provide such method. In facts, it is common to perform a preliminary study of a specification before modeling it in Event-B [SAZ12, Abr07]. Thus, just modeling and verifying a specification in Event-B is not adequate for the safety-critical systems.

3. Event-B provides a refinement mechanism to gradually refine a abstract specification into a concrete specification. This mechanism eases the analysis and verification of requirements specifications, especially the complex ones. The idea of refinement is undoubtedly useful. However, there is no guideline for using the refinement mechanism in Event-B effectively. Given that a complicated system is being modeled in Event-B, designers and developers of the system might have no idea how to organize the refinement steps which is a source of difficulty in the usage of refinement [Abr06].

In order to ensure the possibility of using Event-B to verify software safety requirements, we need an approach to cope with those shortcomings.

1.3 Proposed Approach

This research aims to overcome the shortcomings of Event-B stated in the previous section. The purpose of this research is to encourage and assist the usage of Event-B in practical development of software safety requirements specifications. Our proposed approach can be divided into two parts:

1. The first part is to analyse the meaning of the properties preserved by the proof obligations of Event-B. This part is related to the first shortcoming of Event-B, which is about the obscure definition of the correctness in Event-B.

2. The second part is to propose a model supporting the refinement mechanism of Event-B. This model is for elaborating and analysing the safety requirements specifications before modeling them in Event-B. Furthermore, the proposed model should be capable to provide some ideas how to refine an abstract model in Event-B. The second part is to solve the second and the third shortcomings of Event-B.

To analyse the properties preserved by the proof obligations of Event-B, we convert the notions of Event-B specification language into the form of the evolutionary framework proposed by Zowghi and Gervasi. Then, we prove that the properties preserved by the proof obligations conform to the completeness and the consistency of the evolutionary framework. As results of the proofs, we can conclude that the correctness preserved by Event-B refinement is the same with the correctness defined in the evolutionary framework. This explains how Event-B preserve the correctness of the safety requirements specifications.

In our opinion, proposing a model supporting the refinement mechanism of Event-B from scratch is difficult and inefficient. The model should be based on another approach which supports analysis and refinement of the requirements specifications. The goal model of KAOS method is suitable for our needs since it provides the mechanism for requirements analysis and a notion of refinement. However, the direct usage of the goal model in Event-B such as translation from the temporal logic of goals into Event-B specification has some difficulties (discussed in Chapter 4). Thus, we rather propose a new model based on the goal model of the KAOS method. By proposing a new model, we can design it in a way that supports the goals of this research, while we can also use the capabilities of the KAOS method. Our proposed model is named "ORDER model".

The ORDER model is capable to:

- treat the components of Event-B like goals of the KAOS method
- use KAOS's goal refinement patterns
- graphically show Event-B refinement
- support direct transformation to Event-B models

Our approach ensure that the refinement mechanism of Event-B can preserve the correctness of the requirements specifications. Besides, we also propose the ORDER model based on the KAOS method to support analysis and refinement of requirements in Event-B. The means that our proposed approach can reduce the shortcomings of Event-B and encourage applying Event-B in practical. The safety requirements specification modelled through our approach is claimed to be correct. This research is a way to encourages the usage of formal methods in the practical software development.

1.4 Thesis Structure

The remainder of this thesis is organized as follows:

- Chapter 2 provides technical details of Event-B, the evolutionary framework, and the KAOS method.
- Chapter 3 shows our analysis that Event-B can preserve the correctness of requirements specification as defined in the evolutionary framework.
- Chapter 4 thoroughly explains our proposed model: ORDER model.
- Chapter 5 describes the KAOS-based refinement patterns for assist construction of ORDER model.

- Chapter 6 presents the means to evaluate our proposed approach through case studies. The advantages of our approach are also discussed in this chapter.
- Chapter 7 discusses related work.
- Chapter 8 concludes this thesis with future directions.

Chapter 2

Technical Background

This research is based on the three framework: the Event-B formal method, the evolutionary framework, and the KAOS method. This section sufficiently provides technical details of each framework used throughout this thesis.

2.1 Event-B

Event-B [Abr10] is a formal specification language for modeling requirements of systems. The language is based on first-order predicate logic and discrete transition systems. The most important feature provided by Event-B is its refinement mechanism. The mechanism can be used for incrementally constructing a system specification from an abstract one and step-wise refining it into a concrete specification.

2.1.1 Machine and context

An Event-B model may contain a static part called the *context*, and a dynamic part called the *machine*. The machine might access to one or more contexts via a *SEES* relationship. A machine can be *refined* by another one, and a context can be *extended* by another one. Machine and context relationships are illustrated in Figure 2.1.

All *carrier sets*, *constants*, and their definitions declared through *axioms* are described in the context. A carrier set is a term for calling a mathematical set defined in an Event-B model. The machine contains all of the state *variables*. Types and properties of the variables are declared through *invariants*. The values of the variables can be changed by the execution of *events*. Every machine must contain an event called *INITIALISATION* for setting up the values of the variables.

An event can be represented by the following form:

$$evt \mathrel{\hat{=}} \textbf{any} p \textbf{ when } G \textbf{ with } W \textbf{ then } S \textbf{ end}$$

The short form

$$evt \mathrel{\hat{=}} \textbf{begin } S \textbf{ end}$$

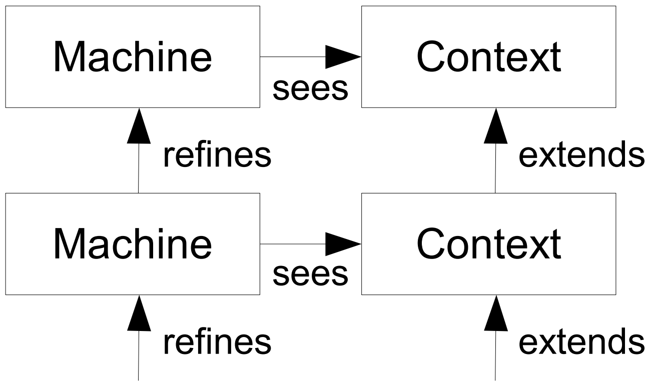


Figure 2.1: Machine and context relationships

where p denotes internal parameters of the event, G is a predicate denoting *guards*, and S denotes the *actions* that update some variables. S can be executed only when G holds. When we refine an abstract event, some variables of that event might be disappeared in its concrete event. W denotes *witnesses* that are additional elements in the concrete event for indicating the disappeared variables and their values. Given that the variables of the machine containing the event are denoted by v, S is composed of several *assignments* of the form:

$$\begin{aligned} x &:= E(v) \\ x &:\in E(v) \\ x &:\mid Q(v, x') \end{aligned}$$

where x are some variables, and x' represents a state of x which their values are changed just after assigning x to an expression $E(v)$ or a predicate $Q(v, x')$. The former form is deterministic, which x are assigned to precise values. The latter form is non-deterministic assigning x to be elements of carrier sets. The effect of each assignments can also be described by a before-after predicate:

$$\begin{aligned} BA(x := E(v)) &\mathrel{\hat{=}} x' = E(v) \\ BA(x :\in E(v)) &\mathrel{\hat{=}} x' \in E(v) \\ BA(x : |Q(v', x')) &\mathrel{\hat{=}} Q(v, x') \end{aligned}$$

A before-after predicate describes the relationship between the state just before an assignment has been triggered and the state just after the assignment has been triggered (represented by x and x' respectively).

The refinement mechanism of Event-B allows us to extend a context, i.e. adding more components into the extending context, whereas we are allowed to refine a machine into a concrete machine by adding new variables, rewriting events description to handle new variables, strengthening the guards, and so on.

We separately define an Event-B abstract model and an Event-B concrete model to clearly distinguish them in our analysis in the next chapter. According to [C$^+$05], we can define an Event-B abstract model (refined model) as follows:

Definition 1 (Event-B abstract model)**.** *An Event-B abstract model is a tuple* (s, c, A, v, I, E)*, where* s *and* c *are the carrier sets and constants respectively;* $A(s, c)$ *is a collection of axioms;* v *are the machine abstract variables;* $I(s, c, v)$ *is the invariants limiting the possible state of* v*;* E *is a set of events. Moreover, each abstract event is defined as a tuple* (G, BA)*, where* $G(s, c, v)$ *is the event guard triggering actions when it holds;* $BA(s, c, v, v')$ *is a before-after predicate defining a relation between the current and next states, and representing event actions; and* v' *is a next state of* v*.*

An Event-B concrete model (refining model) can be defined with respect to an Event-B abstract model as follows:

Definition 2 (Event-B concrete model)**.** *An Event-B concrete model is a tuple* $(s, c, A, v, w, J, E2)$*, where* w *are concrete variables;* $J(s, c, v, w)$ *is the invariants added in the concrete machine;* $E2$ *is a set of concrete events. Moreover, each concrete event is defined as a tuple* $(H, W, BA2)$*, where* $H(s, c, w)$ *is the concrete event guard;* $W(s, c, v, w, v', w')$ *are witnesses for disappearing abstract variables;* $BA2(s, c, w, w')$ *is a before-after predicate defining a relation between the current and next states, and representing event actions; and* w' *is a next state of* w*.*

Note that axioms, invariants, guards, witnesses and actions (before-after predicates) are predicates, while carrier sets, constants, and variables are arguments of those predicates. We sometimes abbreviate the predicates of Event-B by omitting their arguments. For example, witnesses of an event may be denoted by W instead of $W(s, c, v, w, v', w')$. We also use the notion $P[x'/x]$ for a substitution of an argument x in a predicate P by an argument x'.

2.1.2 Proof obligations

When an Event-B model is created or refined, a set of proof obligations must be discharged in order to guarantee certain properties of a model. In this thesis, all the proof obligations are in the form of logical implications:

$$Hypotheses \Rightarrow Goal$$

where *Hypotheses* is a conjunction of hypotheses (predicates) and *Goal* is a goal that can be proved from the hypotheses.

The main purpose of the generated proof obligations is to ensure that invariants are maintained in the model where it belongs to and in all subsequent models. Thus, for an abstract model of Event-B, each event of the model should be checked by the proof obligations that its actions are consistent with the invariants of the model. Furthermore, each event of a concrete model refining the abstract model should be also checked by the proof obligations that it indeed preserves the description of its refined event. This means that the proof obligations can be divided into two types: to ensure internal consistency of a model, and to ensure consistency of a refinement.

Ensuring internal consistency

The proof obligation for internal consistency are separately defined for abstract models and concrete models. Firstly, the proof obligations for abstract model consistency are as follows:

Definition 3 (Proof obligations for abstract model consistency)**.** *For ensuring abstract model consistency, the following proof obligations must be discharged for each event of an Event-B model:*

$$\begin{aligned} FIS &: A \wedge I \wedge G \Rightarrow \exists v' \cdot BA \\ INV &: A \wedge I \wedge G \wedge BA \Rightarrow i \end{aligned}$$

where i are the invariants involving the variables which appears in BA. The purpose of FIS is to ensure that non-deterministic actions are feasible, and INV is to ensure that invariants of a machine is preserved by each event.

The proof obligations for concrete models are as follows:

Definition 4 (Proof obligations for concrete model consistency)**.** *For ensuring concrete model consistency, the following proof obligations must be discharged for an event j of an Event-B model:*

$$\begin{aligned} FIS_{ref} &: A \wedge I \wedge J \wedge H \Rightarrow \exists w' \cdot BA2 \\ INV_{ref} &: A \wedge I \wedge J \wedge H \wedge W \wedge BA2 \Rightarrow j \\ WFIS &: A \wedge I \wedge J \wedge H \wedge BA2 \Rightarrow \exists v' \cdot W \end{aligned}$$

where j are the invariants involving the variables which appears in $BA2$. The purpose of FIS_{ref} is the same as FIS. INV_{ref} is for the preservation of invariants of the concrete model. $WFIS$ is for ensuring that each witness for an abstract variable of a concrete event indeed exists.

Ensuring refinement consistency

Given that there is a concrete model refining an abstract model, a set of proof obligations must be discharged to maintain the invariants from the abstract model. The proof obligations for ensuring the consistency of a refinement are as follows:

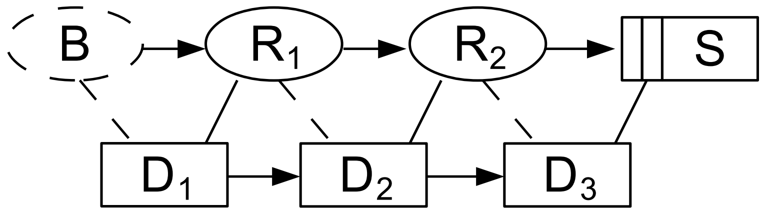


Figure 2.2: The evolutionary framework

Definition 5 (Proof obligations for refinement consistency)**.** *For ensuring the consistency of a refinement, the following proof obligations must be discharged when a concrete event refines an abstract event:*

$$GRD : A \wedge I \wedge J \wedge H \wedge W \Rightarrow G$$
$$SIM : A \wedge I \wedge J \wedge H \wedge W \wedge BA2 \Rightarrow BA$$

The purpose of *GRD* is to make sure that the concrete guards of a concrete event are stronger than the abstract ones of the abstract event. This ensures that when a concrete event is enabled, so is the corresponding abstract one. *SIM* is to ensure that when a concrete event is executed what it does is not contradictory with what the corresponding abstract event does.

There is another proof obligation *EQL* for ensuring refinement consistency. This proof obligation focuses on a set of variables which belongs to both a concrete machine and its abstract machine. If a concrete event of the concrete machine assign values to some variable in the set but the abstract event does not, it must be proved that the variables' values does not change. This is because In order to preserve refinement consistency, any event of a refinement that modifies the state of the machine being refined must itself be a refinement of one or more events of the machine being refined.

2.2 The evolutionary framework

Zowghi and Gervasi in [ZG03] describe that the creation of requirements documents usually starts from the business needs of customers and then they are progressively evolved step-by-step in term of requirements until a specification is created. They believe that there are causal relationships among consistency, completeness, and correctness (3Cs) of requirements at each step. As a consequence, they investigate and conclude the relationships in their evolutionary framework, as in Figure 2.2, based on Jackson's Problem Frames [**?**].

The framework consists of three components: domain, requirements, and specification. Domain (D) is a set of properties (or rules) of the environment in which a software system is going to be implemented on. Requirements (R) are properties which are desired to hold among elements in D. Lastly, Specification (S) is the instruction of a machine implemented on D in order to keep the properties in R hold.

In Figure 2.2, the initial version of requirements is denoted as B instead of R_0 to emphasize that requirements are from the business needs of the customers, and the final version is denoted as S for the specification instead of R_{n+1} for the evolution with $n+1$ steps. Note that D_0 is null because the domain properties can be only described after the study of business needs. Arrows between two consecutive versions of R and D indicate the direction of evolution. Zowghi and Gervasi assume that domain can be evolved by only adding more properties into the next version, whereas requirements can be added, changed, and removed. Such an assumption is called a monotonic domain refinement. The evolution of requirements reflects an increased understanding of the business needs.

In facts, there are two aspects to define the correctness of requirements:

- From a practical point of view, the correctness of requirements can be defined as satisfaction of the business needs of customers.
- From a formal point of view, the correctness is usually meant to be the combination of completeness and consistency.

Zowghi and Gervasi aim to use the evolutionary framework to provide the formal relationship among 3Cs. Hence, the definition of the correctness in the framework follows the latter point of view. The consistency is that there is no conflict in combining requirements and domain properties. The completeness is defined as relative completeness, which means that we determine the completeness of requirements with respect to an external reference that is the previous version of requirements for this framework. At every step of the evolution, we have to show the consistency and the relative completeness of requirements to conclude the correctness. In addition, for the case of the monotonic domain refinement, the domain evolution needs to be shown as well. A lemma can be concluded from the definitions above.

Lemma 1 (Monotonic domain refinement)**.** *Let us assume that we are performing an evolution step, from R_i and D_i to the subsequent versions R_{i+1} and D_{i+1}, and that we are only adding new information about the domain, i.e. $D_{i+1} \models D_i$ (domain evolution). Then, if we can prove $(R_{i+1} \cup D_{i+1}) \not\models \bot$ (consistency) and $(R_{i+1} \cup D_{i+1}) \models R_i$ (relative completeness), then $(R_{i+1} \cup D_{i+1}) \models R_i \cup D_i$ (correctness) holds.*

A relevant result from Lemma 1 is expressed by the following theorem.

Theorem 1 (Inductive correctness)**.** *Let $(R_0, D_0), \ldots, (R_{n+1},$
$D_{n+1})$ be a chain of evolution steps in the development of a specification. If at each step the consistency, the relative completeness, and the domain evolution hold according to Lemma 1, then*

$$\forall i \in [0..n], (R_{i+1} \cup D_{i+1}) \models (R_i \cup D_i)$$

It follows by induction that

$$(R_{n+1} \cup D_{n+1}) \models (R_0 \cup D_0)$$

But R_{n+1} is S, R_0 is B, and D_0 is empty, so we have

$$(S \cup D_{n+1}) \models B$$

Theorem 1 concludes that this process can guarantee that all versions of requirements and domains are correct with respect to the initial one which is the business needs. In other words, we construct a specification, which satisfies the business needs. Thus, the formal definition of the correctness in this framework corresponds to the correctness in the practical point of view.

To prove the entailment operator ($\models$) used in Lemma 1, Zowghi and Gervasi originally represent each component of the framework with a set of predicates, and simply use the inference rules of first-order logic.

2.3 The KAOS method

In software engineering, requirements specifications are documents which are supposed to reflect what stakeholders need from a new software system. In other words, a requirements specification describes what a software system has to perform to accomplish the system's goals.

Goal is a prescriptive statement of intent that the system should satisfy through the cooperation of its agents such as human, devices, or software. Goal-oriented requirements engineering (GORE) refers to the use of goals for requirements elicitation, evaluation, negotiation, elaboration, structuring, documentation, analysis, and evolution. In the context of GORE, a requirement is a goal under the responsibility of a single software agent.

KAOS ('Knowledge Acquisition in autOmated Specification' or 'Keep All Objects Satisfied') [VL09] is a GORE method with a rich set of formal analysis techniques. The KAOS method contains 5 UML-like core models which are goal model, object model, agent model, behaviour model, and operation model for modeling and structuring requirements. This research focuses only on the goal model, which is the central model of the KAOS method. The goal model has a two-level structure: the outer graphical semantic layer and the inner formal layer. The outer layer shows semi-formal relationships among goals. While, the inner layer formally defines goals and their relationships. The formal layer of KAOS is based on linear temporal logic. The details of the linear temporal logic and the goal model of KAOS are described in the following sections.

2.3.1 Linear Temporal Logic

In order to formally describe goals, we may express them in term of state assertions. A *state assertion* is a predicate to express that some descriptive or prescriptive property holds in some arbitrarily chosen current state. However, goals can be refer to not only the current state of a system, but also the future or past states. In the case that the future

Table 2.1: Linear temporal operators

Operator	**Description**	**Formal semantics**
$\diamond P$	eventually	$(H,i) \models \diamond P$ iff for some $j \geq i$: $(H,j) \models P$
$\Box P$	always	$(H,i) \models \Box P$ iff for all $j \geq i$: $(H,j) \models P$
$\circ P$	next	$(H,i) \models \circ P$ iff $(H,i+1) \models P$
$P \Rightarrow Q$	entails	equivalent to: $\Box(P \rightarrow Q)$
$P \Leftrightarrow Q$	congruent	equivalent to: $\Box(P \leftrightarrow Q)$

and the past are involved, state assertions need to be prefixed by temporal operators, they are called *temporal assertions.* The temporal assertions in the context of the KAOS method is based on linear temporal logic.

Linear temporal logic (LTL) is an addition of the usual first order logic, which uses logical connectives ($\wedge \ \vee \ \neg \ \rightarrow \leftarrow$), and quantifiers ($\forall \ \exists$) to express an assertion (for the current state), with a set of temporal operators. The temporal operators are for expressing both the past and the future. However, in the scope of this thesis, we do not consider past LTL.

For this section, we define an entailment operator for linear temporal logic according to [VL09] as follows:

Definition 6 (Temporal entailment)**.** *Let H be history, i be a time position, and P be a temporal assertion, then $(H,i) \models P$ iff P is satisfied by H at the time position i.*

From the definition above, $(H,0) \models P$ means the assertion P is satisfied by the entire history H.

The temporal operators used in the scope of this thesis are described in the Table 2.1. Each operator is described based on the temporal entailment and the logical connectives.

2.3.2 Goal, domain property, and their formal definitions

Goals can be categorized into two types: behavioral goals and soft goals. Behavioural goals prescribe intended system behaviours. While, soft goals prescribe preferences among alternative behaviours. In the scope of this thesis, we focus on the behavioral goals. Behavioural goals can be further specialized into *Achieve* goals and *Maintain* goals [DVLF93].

Aside from goals which are prescriptive statements satisfied by agents, a *domain property* is a descriptive statement about the environment, expected to hold regardless of how the system behaves. A domain property describes a physical law, organization policy, and so on. A domain property is not directly satisfied by any agents of a system.

Formal definitions of goals and domain properties based on LTL can be defined according to the characteristic of each type of goals and domain properties.

Achieve goals

Achieve goals prescribe system behaviours where some target properties must be eventually satisfied in the future. In KAOS, achieve goals are described in the following pattern:

Achieve[TargetCondition]: [**If** CurrentCondition **then**] **eventually** TargetCondition

This generic pattern of achieve goals means that when the current condition is satisfied, then the target condition must be eventually satisfied in some future states. The pattern can be formally described in term of linear temporal logic in two ways:

$$\begin{array}{l} \text{CurrentCondition} \Rightarrow \ \diamond \ \text{TargetCondition} \\ \text{CurrentCondition} \Rightarrow \ \circ \ \text{TargetCondition} \end{array}$$

The previous way does not specific when the target condition is achieved, while the latter way restricts that the target condition must be satisfied immediately in the next state.

Another form of achieve goals is to cease a target condition:

Cease[TargetCondition]: [**If** CurrentCondition **then**] **eventually not** TargetCondition

Its possible formal forms are:

$$\begin{array}{l} \text{CurrentCondition} \Rightarrow \ \diamond \neg \ \text{TargetCondition} \\ \text{CurrentCondition} \Rightarrow \ \circ \neg \ \text{TargetCondition} \end{array}$$

Maintain goals

Maintain goals prescribe behaviours where some target properties must be permanently satisfied in every future state. In KAOS, maintain goals are described in the following pattern:

Maintain[GoodCondition]: [**If** CurrentCondition **then**] **always** GoodCondition

This pattern means all the state in which the current condition is satisfied, this implies that the good condition must also be satisfied. This pattern can be formally described in term of linear temporal logic in the following way:

$$\text{CurrentCondition} \Rightarrow \text{GoodCondition}$$

Another form of maintain goals is to avoid a bad condition:

Avoid[BadCondition]: [**If** CurrentCondition **then**] **always not** BadCondition

Its formal form is:

$$\text{CurrentCondition} \Rightarrow \neg \ \text{BadCondition}$$

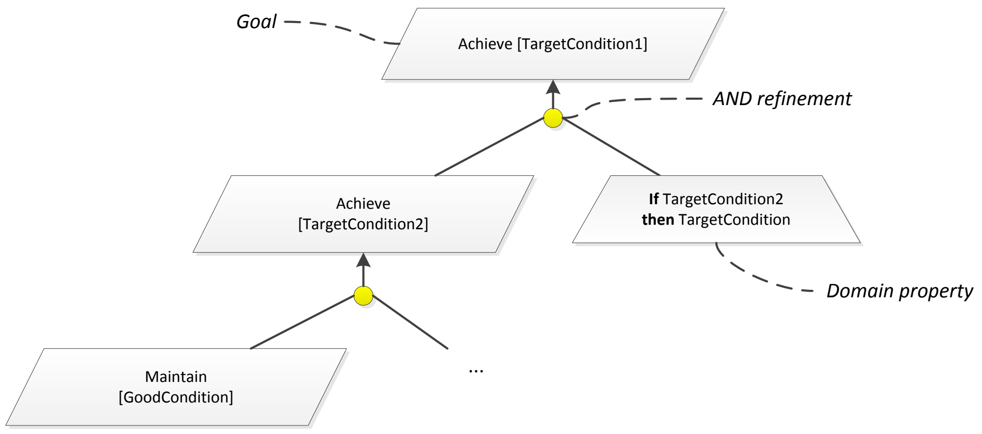

Goal
Achieve [TargetCondition1]
AND refinement
Achieve [TargetCondition2]
If TargetCondition2 **then** TargetCondition
Domain property
Maintain [GoodCondition]
...

Figure 2.3: An example of goal model

Domain properties

Domain properties refer to laws on environment phenomena. This means that a domain property is an invariant property holding in any state. Unlike goals, there is no pattern for describing a domain property in KAOS. However, the formal forms of domain properties can be specified as follows:

$$\text{Condition1} \Rightarrow \text{Condition2}$$
$$\text{Condition1} \Leftrightarrow \text{Condition2}$$

This formal forms show the relations between two conditions which must always hold in the environment of a system.

2.3.3 Goal model

The goal model of KAOS is in the shape of a tree. The tree consists of a refinement graph expressing how higher-level goals are refined into lower-level ones and, conversely, how lower-level goals contribute to higher-level goals. In a refinement graph, a node represents a goal which is either an achieve goal or a maintain goal, and an AND-refinement link relates a parent goal to a set of sub-goals. A parent goal must be satisfied when all of its sub-goals are satisfied. The relationship between a parent goal and the set of its sub-goals is called *goal refinement*. An example of goal model is shown in Figure 2.3. A parallelogram denotes a goal, while a trapezoid denotes a domain property. an AND refinement links a parent goal to its sub-goals.

An AND-refinement of a goal into its sub-goals should ideally be complete and consistent. The formal definitions of a complete and consistent AND-refinement can be defined as follows:

Definition 7 (Complete refinement)**.** *An AND-refinement from a goal G into sub-goals $G_1, G_2, \ldots, G_n$ is complete iff $\{G_1, G_2, \ldots, G_n\} \models G$.*

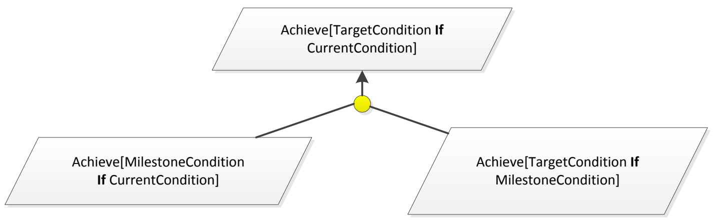

Achieve[TargetCondition If CurrentCondition]
Achieve[MilestoneCondition If CurrentCondition]
Achieve[TargetCondition If MilestoneCondition]

Figure 2.4: Milestone-driven refinement pattern

Definition 8 (Consistent refinement)**.** *An AND-refinement from a goal G into sub-goals $G_1, G_2, \ldots, G_n$ is consistent iff $\{G_1, G_2, \ldots, G_n\} \not\models \bot$.*

Complete refinement means that in any circumstance where all sub-goals are satisfied, their parent goal is always satisfied. In other words, no sub-goals are missing for the parent goal to be satisfied. Consistent refinement means that it is possible for all the sub-goals to be satisfied altogether. In logical entailment, if a set of predicates can entail false ($\bot$), this set of predicates can entail anything, even their parent goal. However, it is more preferable to have a parent goal to be satisfied when all of its sub-goals can be satisfied. Thus, a refinement should also be consistent.

2.3.4 Goal refinement patterns

An effective way to construct a goal model is by reusing goal refinement patterns [DVL96]. This is because "correct goal refinements are often hard to find; goal decompositions made by hand are usually incomplete and sometimes inconsistent" [VLDM95]. The goal refinement patterns are frequently used patterns for refining a goal into sub-goals. Each pattern suggests specific refinements/abstractions for instantiation to the specifics of the modelled system. Parameters are used in each pattern for representing conditions. Ones can instantiate a pattern by replacing each parameter with a corresponding condition from the modelled system. The patterns are proved to be complete and consistent already. Hence, user of the patterns can use the pattern without the necessity of proving their completeness and consistency again. This thesis focuses the usage of two patterns: milestone-driven refinement pattern and decomposition-by-case pattern.

The milestone-driven refinement pattern

This refinement pattern is for establishing an necessary intermediate step (a milestone condition) for reaching a target condition from a current condition. Figure 2.4 shows the generic AND-refinement tree of this pattern.

Let c be a current condition, m be a milestone condition, and t be a target condition. The following formal representation of this pattern shows that this pattern is complete and consistent.

$$\{c \Rightarrow \diamond m, m \Rightarrow \diamond t\} \models c \Rightarrow \diamond t$$

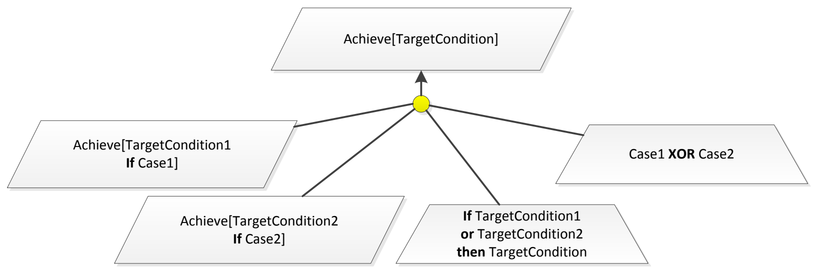

Achieve[TargetCondition]
Achieve[TargetCondition1 If Case1]
Case1 XOR Case2
Achieve[TargetCondition2 If Case2]
If TargetCondition1 or TargetCondition2 then TargetCondition

Figure 2.5: Decomposition-by-case pattern

$$\{c \Rightarrow \diamond m, m \Rightarrow \diamond t\} \not\models \bot$$

Decomposition-by-case pattern

This refinement pattern introduces different cases for reaching a target condition. The cases must be disjoint and cover all possible cases. Figure 2.5 shows the generic AND-refinement tree of this pattern.

Let $t, t1$, and $t2$ be target conditions, $c1$ and $c2$ be cases. The following formal representation of this pattern shows that this pattern is complete and consistent.

$$\{c1 \Rightarrow \diamond t1, c2 \Rightarrow \diamond t2, t1 \vee t2 \Rightarrow t, c1\ xor\ c2\} \models t$$
$$\{c1 \Rightarrow \diamond t1, c2 \Rightarrow \diamond t2, t1 \vee t2 \Rightarrow t, c1\ xor\ c2\} \not\models \bot$$

2.3.5 Generic refinement tree for safety goals

There is also a generic tree which is created especially for refining safety goals as in Figure 2.6. Forbidden states are avoided in this tree by anticipating dangerous states. If a dangerous state is anticipated, an alarm is raised. Then, some response must be provided for the alarm by an appointed guard to clear the potentially dangerous situation. This generic refinement tree is related to the objective of this research.

2.4 Motivated examples

This section discusses about the shortcomings of Event-B, which this research aims to overcome. The shortcomings are presented through the use of example from existing Event-B specifications in a practical context. The approach to deal with the shortcomings based on the evolutionary approach and the KAOS method is also shortly explained.

2.4.1 Ambiguity of the correctness in Event-B

Event-B is a good formal approach for modeling requirements specifications, since its refinement mechanism along with its proof obligations can efficiently help stakeholders

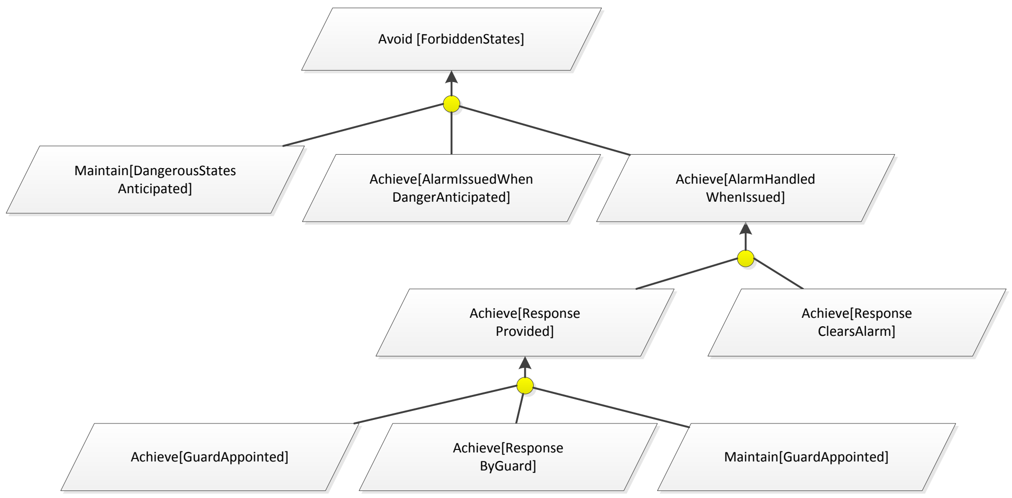

Avoid [ForbiddenStates]
Maintain[DangerousStates Anticipated]
Achieve[AlarmIssuedWhen DangerAnticipated]
Achieve[AlarmHandled WhenIssued]
Achieve[Response Provided]
Achieve[Response ClearsAlarm]
Achieve[GuardAppointed]
Achieve[Response ByGuard]
Maintain[GuardAppointed]

Figure 2.6: An generic refinement tree of safety goals

progressively analyse the specifications, especially the complex ones. Developing a complex specification containing all details of a system at once requires a lot of effort to comprehend and difficult to reason about. One common strategy using Event-B refinement for dealing with the difficulties is to start constructing an initial model by describing only the main purpose of a system. Then, other details can be gradually introduced into subsequent concrete models. This strategy eases the proof of the correctness of requirements, because only a small number of proof obligations are generated at each step.

One successful example of using Event-B refinement to gradually model the specification of an industrial case study is the work of Rezazadeh et al. in [REB07]. They developed the CCF Display and Information System (CDIS), a system providing important airport and flight data for the duties of air traffic controllers, by using Event-B. CDIS is large-scale system,which is difficult to comprehend, seeing that there are 1200 pages of the original specification documents. Even though, to demonstrate their approach using Event-B refinement, they focused only on core specification of CDIS, the specification is still complex. They began with modeling a specification for a generic system which describes the overview of CDIS. Through subsequent refinements, more specific details were introduced. This resulted in six refinement steps which comprehend all details from the core specification. At each step, approximately less than 20 proof obligations are generated, which is relatively small. Because all proof obligations are successfully discharged, they conclude that their models were reasonably correct. Furthermore, Event-B refinement can help them overcome the difficulty of comprehending the original specification.

The main purpose of the generated proof obligations is to ensure that invariants is maintained in the model where it belongs to and in all subsequent models. Hence, even if the correctness of CDIS specification is ensured by Event-B proof obligations along the refinements, the meaning of this correctness is unclear in term of requirements engineering. Practically, a correct requirements specification means that the specification satisfies certain business goals. In this case, there is no direct link between the business goals and what the proof obligations are generated for. On the other hand, the correctness formally means to be the combination of completeness and consistency. The consistency is that there is no contradiction among requirements. The completeness is that there is no information left unstated in a requirements specification with respect to a reference point. It is reasonable to say that the proof obligations is capable to guarantee the consistency, but not the completeness. This is because a relation between the proof obligations and the reference point of the completeness is unclear.

The evolutionary framework has the semantics of the correctness of requirements specification, and describes the process of specifying requirements in a step-wise way. The step-wise specification of the requirements is similar to Event-B. If we regard one step of refinement in Event-B as a step of evolution in the evolutionary framework, it is possible to show whether Event-B can preserve the correctness as defined in the evolutionary framework.

2.4.2 Lacks of the requirements analysis and guideline for the refinement

In [ASZ12], the formalization of hybrid systems in Event-B is described. [ACH$^+$94] explains that "A hybrid system consists of a discrete program with an analog environment. We assume that a run of a hybrid system is a sequence of steps. Within each step the system state evolves continuously according to a dynamical law until a transition occurs. Transitions are instantaneous state changes that separate continuous state evolutions.". The hybrid systems are very important in the development of embedded systems where a piece of software, the controller, is supposed to manage an external situation, the environment. It is usual to find that most safety-critical systems are related to the hybrid systems.

One example of [ASZ12] is about a system controlling trains to provide safe moves of the trains. An preliminary study about the system had been performed before the system was modeled in Event-B. From the preliminary study, some necessary invariants of the system was found, and the information needed for deciding the current acceleration of a train was specified. Without the preliminary study, those necessary information about the system cannot be specified. If the preliminary study is skipped, some necessary information might be missing. The missing information potentially causes the system to be unsafe. The preliminary study is undoubtedly crucial, but no systematical way for the preliminary study has been proposed for Event-B.

Another notice from [ASZ12] is that even though the work focused on the hybrid systems, all of its examples have distinct refinement plans from each other. Here, the refinement plan means the consideration of what models are constructed in each abstraction level of Event-B. The advantage of the refinement mechanism of Event-B is that it provide a lot of (but limited) ways to refine an Event-B model. This is for widely supporting various kinds of systems. Unfortunately, the refinement mechanism is usually poorly used because it is not easy to decide how to organize the construction steps [Abr06]. Non-experts of Event-B often have no idea how to use the refinement mechanism efficiently.

The resulted models from the non-experts might be too rough and the rough refinement does not mitigate the complexity of the Event-B models well.

The preliminary study can be considered as the requirements analysis and elaboration. The analysis and elaboration of requirements are the capabilities of the goal-oriented requirements engineering like the KAOS method. Now that the KAOS method contains the notions of refinement, which is called the goal refinement, for the analysis and elaboration, we plan to apply the goal refinement to fulfil what Event-B lacks, that is, the systematical preliminary study and the guideline for using Event-B refinement.

Chapter 3

Preservation of correctness of safety requirements in Event-B

3.1 Rationale

As we discussed in Chapter 2, the semantics of the correctness of requirements are unclear in Event-B. Thus, we aim to analyse the conformance between the properties preserved by Event-B and the correctness defined in the evolutionary framework. The reason for us to select the evolutionary approach is because we think that the evolutionary framework is compatible with Event-B and the process of specifying safety requirements. The latter parts of this section explains the reason in more details.

In facts, the strategy for gradually constructing specifications in Event-B is similar to the explanation of how a requirements specification are constructed in the evolutionary framework. Both approaches start from capturing the overview (or business needs) of a system in their initial step. Then, more details for the system are gradually included into the subsequents steps. the properties maintained by POs and the properties described in Lemma 1 (monotonic domain refinement) from Section 2.2 are also similar. POs are generated at each step for ensuring that the invariants of the current refinement are not violated by any events, which is similar to showing the consistency. Some POs are generated for verified that the current refinement preserve invariants from the all the former refinement steps, which is similar to the relative completeness. The definition of the relative completeness provides a link between requirements and the corresponding business goals (which can be regarded as a reference point for the formal completeness). Because of these similarities, we think that it is possible to use Event-B to model a requirements specification and preserve the correctness of the requirements conforming to the evolutionary framework.

The step-wise way to evolve requirements of a system as described in the evolutionary framework is similar to how software safety requirements are created. According to an international standard for safety-critical system like ISO26262 [ISO11], the creation of the software safety requirements requirements starts from specifying safety goals of a safety-critical system. Then, a functional safety requirements specification is derived

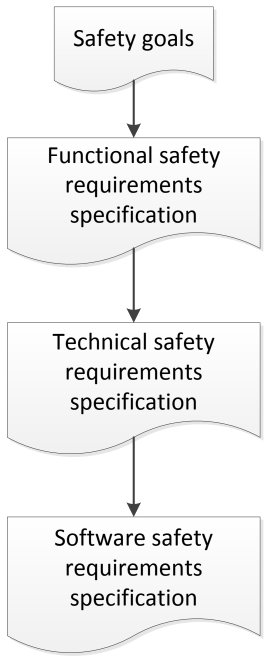

Safety goals
Functional safety requirements specification
Technical safety requirements specification
Software safety requirements specification

Figure 3.1: Hierarchy of safety goals and safety requirements specifications

from the safety goals, following by a technical safety requirements specification derived from the functional specification. Lastly, a software safety requirements specification is derived from the technical specification. Figure 3.1 illustrates the hierarchy of safety goals and the steps of derivation of safety specifications. The development of each level of specification can be iterated to improve the specification. This means that the software safety requirements specifications are created by starting from safety goals and step-wise evolving until completing a software safety requirements specification. Hence, the evolutionary framework is appropriate for explaining how to preserve the correctness of the software safety requirements specification.

As a result, if the correctness preserved by Event-B refinement mechanism conforms to the correctness defined in the evolutionary framework, it means that Event-B can ensure the correctness of safety requirements from both practical and formal points of view (from Theorem 1).

3.2 Converting Event-B into Evolutionary Framework

We aim to analyze whether Event-B has capabilities to preserve the correctness of requirements explained in Lemma 1. In other words, Event-B can preserve the correctness of requirements or not. To perform the analysis, firstly, we need to convert Event-B's mathematical notions into the evolutionary framework's notions in order to prove Lemma 1 by the generated POs of Event-B.

We believe that the conversion is possible because D is similar to Event-B contexts and R is similar to Event-B machines. D can be evolved by including more properties into the next version of D. It works the same way with the context extension in Event-B. Changing requirements is also similar to refining a machine, since we can add, change, and remove requirements by adding variables, rewriting event description, and replacing some parameters respectively. Furthermore, The evolutionary framework follows Lemma 1 to preserve the correctness of requirements at every step of the evolution.

Because of the similarity explained above, we are able to convert Event-B into the evolutionary framework by transforming the context of Event-B to the domain (D), and the machine of Event-B to the requirements (R). In the framework, Zowghi and Gervasi choose to represent each element by a set of predicates based on first-order logic. Thus, just putting the predicates of Event-B into each set is sufficient for the conversion. Starting from the most simplest one, we define the context of Event-B as a set in the following definition:

Definition 9. *A context of Event-B is a set in the form of:*

$$\{A(s, c)\}$$

where s and c are carrier sets and constants respectively; and A are axioms.

Then, we define the abstract model of Event-B once again as a set based on Definition 1 defined in Section 2.1 as follows:

Definition 10. *An Event-B abstract model is a set in the form of:*

$$\{A(s, c)\} \ \cup \{I_{s_i}(s, c, v_i'), I(s, c, v_i) \wedge G_i(s, c, v_i) \wedge BA_i(s, c, v, v_i') \mid i \in [0..n]\}$$

where n is the amount of events; i enumerates event in an abstract model; s_i is a set of indexes of invariants relevant to the before-after predicate BA_i; v_i is a state of abstract variables; I is a conjunction of all invariants; I_{s_i} is a conjunction of invariants whose indexes appearing in s_i; G_i is an guard corresponding to BA_i, and v_i' is the next state of v_i.

The rationale behind the definition above is that invariants must be true at every state of v, so we introduce invariants pairing with every before-after predicate which changes the state of v. We replace v with v_i because each event can function at an arbitrary state of v as long as the guard holds and we want to explicitly separate the symbol denoting the state. The conjunction of invariants, guards and before-after predicate means that, in a state of v allowed by the invariants, when the guard holds, the before-after predicate is triggered. Another note is that this definition is the conversion of $R \cup D$ rather than converting R and D separately. This is more preferable because R always pairs with D in Lemma 1 of the evolutionary framework.

Similarly, we define the concrete model of Event-B as a set based on Definition 2 and Definition 10 from Section 2.1 as follows:

Definition 11. *An Event-B concrete model is a set in the form of:*

$$\{A(s,c)\} \cup \{J_{r_j}(s,c,v'_j,w'_j), I(s,c,v_j) \wedge J(s,c,v_j,w_j) \wedge H_j(s,c,w_j)$$
$$\wedge W_j(s,c,v_j,w_j,v'_j,w'_j) \wedge BA2_j(s,c,w_i,w'_i) \mid j \in [0..m]\}$$

where m is the amount of events; j enumerates event in a concrete model; r_j is a set of indexes of invariants relevant to the before-after predicate $BA2_j$; w_j is a state of concrete variables; J is a conjunction of all concrete invariants; J_{r_j} is a conjunction of concrete whose indexes appearing in r_j; H_j is an event guard corresponding to $BA2_j$, W_j are witnesses for disappearing abstract variables, and w'_j is the next state of w_j.

3.3 Preservation of Correctness in Event-B Refinement

This section aims to demonstrate an attempt to prove the consistency, the relative completeness, and the domain evolution described in Lemma 1 by discharging Event-B proof obligations. If Lemma 1 can be shown by discharging the proof obligations on Event-B models defined by Definition 10 and Definition 11, we can imply the preservation of correctness of requirements through Event-B refinement.

3.3.1 Consistency

The definitions of consistency are defined independently by the evolutionary framework and Event-B. The consistency of the evolutionary framework can be shown by proving $R \cup D \not\models \perp$. This consistency means an internal consistency in the sense that R and D must be at he same step of the evolution. Event-B also generates a set of proof obligations to ensure Event-B's consistency inside a machine description seeing one or more contexts [Abr10]. Proof obligation generation rules for Event-B's consistency are separately defined for abstract models and concrete models (Definition 3 and Definition 4) as mentioned in Section 2.1.

From the proof obligations for abstract model consistency (Definition 3), we conclude the following theorem:

Theorem 2 (Consistent Abstract Model)**.** *Let R and D be sets representing an Event-B abstract machine and a context respectively. By assuming that axioms, invariants and guards are consistent, if proof obligations generated according to Definition 3 can be discharged, then $R \cup D \not\models \perp$ holds.*

Proof. Since each event is free from each other, let the set:

$$\{A, I_i, I \wedge G_i \wedge BA_i\}$$

be a subset of $R \cup D$ representing an event of an Event-B abstract model.
We assume that $A \wedge I \wedge G_i$ holds. If FIS is successfully discharged, then there exists

a state v_i' in which BA_i holds. Consequently, if INV is discharged, then I_i holds. This implies that

$$\{A, I_i, I \wedge G_i \wedge BA_i\}$$

is consistent, which also means every event described in $R \cup D$ is consistent.
Therefore, $R \cup D \not\models \bot$ holds. □

According to the above theorem, we need to assume that axioms, invariants and guards are consistent to conduct the proof. In facts, there is no proof obligation directly ensures that $A \wedge I \wedge G_i$ holds. We have to write axioms, invariants and guards correctly by ourselves in Event-B.

In the case of the concrete model, the following theorem can be concluded from Definition 4 from Section 2.1.

Theorem 3 (Consistent Concrete Model). *Let R and D be sets representing Event-B a concrete machine and a context respectively. By assuming that axioms, invariants and guards are consistent, if proof obligations generated according to Definition 4 can be discharged, then $R \cup D \not\models \bot$ holds.*

Proof. Since each event is free from each other, let the set:

$$\{A, J_j, I \wedge J \wedge H_j \wedge W_j \wedge BA2_j\}$$

be a subset of $R \cup D$ representing an event of an Event-B concrete model. We assume that $A \wedge I \wedge J \wedge H_j$ holds. If FIS_{ref} is successfully discharged, then there exists a state w_j' in which $BA2_j$ holds. Consequently, if $WFIS$ is discharged, then there is a state v_j' in which W_j holds. Then, by discharging INV_{ref}, J_j holds. Finally, this implies that

$$\{A, J_j, I \wedge J \wedge H_j \wedge W_j \wedge BA2_j\}$$

is consistent, which also means every event described in $R \cup D$ is consistent. Therefore, $R \cup D \not\models \bot$ holds. □

This theorem also needs the assumption representing that axioms, invariants, and guards are consistent.

From Theorem 2 and 3, we finally proved $R \cup D \not\models \bot$ in the conversion of Event-B into evolutionary framework.

3.3.2 Relative Completeness

In the Event-B refinement, Event-B generates a set of proof obligations to ensure that a concrete model preserves the original properties and behavior of an abstract model [Abr10]. In other words, invariants of the abstract model must also hold in the concrete model, and when a concrete event is triggered, the abstract event must be triggered as well. Here, we try to prove $R_{i+1} \cup D_{i+1} \models R_i$ that is the relative completeness of evolutionary framework, by discharging the proof obligations of the model refinement. Definition 5 from Section 2.1 defines the proof obligation generation rules for the model refinement

A result from Definition 5 can be expressed in the following theorem:

Theorem 4 (Relative Completeness)**.** *Let $R_i \cup D_i$ and $R_{i+1} \cup D_{i+1}$ be consecutive evolution steps representing an abstract model and a concrete model of Event-B respectively. By assuming that both models are consistent, if proof obligations generated according to Definition 5 can be discharged, then $R_{i+1} \cup D_{i+1} \models R_i$ holds.*

Proof. Let a subset

$$\{I \wedge G_i \wedge BA_i\}$$

be an abstract event of R_i and a subset:

$$\{A, I \wedge J \wedge H_j \wedge W_j \wedge BA2_j\}$$

be a concrete event of $R_{i+1} \cup D_{i+1}$ which refines the abstract event. Assuming that each subset is consistent, this implies that $A \wedge I \wedge J \wedge H_j \wedge W_j \wedge BA2_j$ holds, so I holds trivially. If GRD holds, then G_i holds. If SIM holds, then BA_i holds.
Because I, G_i, and BA_i hold, this implies that the subset

$$\{A, I \wedge J \wedge H_j \wedge W_j \wedge BA2_j\}$$

can conclude the subset

$$\{I \wedge G_i \wedge BA_i\}$$

Therefore, $R_{i+1} \cup D_{i+1} \models R_i$ holds. □

Theorem 4 shows that the relative completeness of the evolutionary framework can be proved in term of the Event-B refinement.

3.3.3 Domain Evolution

The last proof needed to be shown is the domain evolution which can be expressed by the following theorem:

Theorem 5 (Context Evolution)**.** *Let D_i and D_{i+1} be a consecutive evolution steps of domains representing
Event-B contexts. If we assume that axioms of each context are consistent, then $D_{i+1} \models D_i$ holds.*

Since we assume that axioms are consistent and we can only extend the context, the proof of this theorem is straightforward: the extending context can directly infer to the extended context.

3.3.4 Extension and results

Although we already showed the proof of Lemma 1 of the evolutionary framework on Event-B model, we omitted some Event-B components and some proof obligations for simplifying the proofs. In this section, we only show an idea to demonstrate that the omitted parts can be supported by the conversion as well.

Proof obligations in Event-B are always of the following form:

$$Hypos \Rightarrow Goal$$

where $Hypos$ is the set of hypotheses (predicates) and $Goal$ is a goal that can be proved from the hypotheses. In order to support the omitted parts, we change the Event-B model by adding the $Goal$ parts of the proof obligation directly into the model. For example, consider the proof obligation:

$$VAR : A \land I \land G_i \land BA_i \Rightarrow n_i(s, c, v'_i) < n_i(s, c, v_i)$$

where $n_i(s, c, v_i)$ is a variant, which means a mathematical expressions to guarantee that the execution of a corresponding event can be terminated. The termination can be shown by VAR to ensure that it is always decreased by the event. We may express an event related to a variant by a subset:

$$\{A, I \land G_i \land BA_i, n_i(s, c, v'_i) < n_i(s, c, v_i)\}$$

Then, VAR can be discharged in this subset because the goals of these proof obligations are added into it.

We can see from Theorem 2, 3, 4, and 5 that the conversion of Event-B into the evolutionary framework results in the fact that Event-B preserves the correctness of requirements in the evolution by discharging proof obligations under the assumption that axioms, invariants, and guards are consistent.

3.4 Example

This section provides an example to illustrate the conversion from specifications written in Event-B into the form of the evolutionary framework. This example is about an automatic gate, which is for regulating access to the building in which the gate is installed. The gate must be able to allow authorized persons to enter the building, and prevent and non-authorized ones. An authorized person for this system means a person who possesses an authorized ID card. One way to check the possession of the authorized ID card is by asking a person to swipe his/her ID card through a card reader and check whether the ID card is an authorized one. We can extract a business need (B) regarding this system as “the gate is opened only for authorized persons”. In this example, we call each machine and context for each step of refinement as M_i and C_i respectively, where i is the i-th step of the refinement. Then, a set of predicates, which is $R_i \cup D_i$, is formed by converting M_i and C_i into the form of a set of predicates.

First step

Starting from the business need, we ‘refine’ the need into the initial Event-B model M_1 and C_1. M_1 and C_1 is to describe the requirements that if the a person, which is checked by an authority-checking process, is authorized to enter the building, then the gate must be opened. Conversely, the gate must be locked when the process does not find an authorized person. Here, C_1 contains a carrier set G_STAT composing of two constants for the status of the gate, namely, *opened* and *locked*. In M_1, there are two variables: *g_stat* and *auth* represent the current status of the gate and the current result from authority-checking process, respectively. The invariants of M_1 are $g\_stat \in G\_STAT$ and $auth \in BOOL$ to show that the type of *g_stat* is the carrier set G_STAT and the type of *auth* is Boolean. $auth = TRUE$ means that the checking process detects an authorized person. The events of M_1 are as follows:

$$\begin{aligned} Open &\mathrel{\hat{=}} \textbf{when } auth = TRUE \textbf{ then } g\_stat' = opened \textbf{ end} \\ Lock &\mathrel{\hat{=}} \textbf{when } auth = FALSE \textbf{ then } g\_stat' = locked \textbf{ end} \\ Auth &\mathrel{\hat{=}} \textbf{begin } auth' \in BOOL \textbf{ end} \end{aligned}$$

The events *Open* and *Lock* are respectively for opening and locking the gate according to the result from the authority-checking process. The event *Auth* represents the authority-checking process. Since, at this step, we want to abstract the process, so *Auth* contains no guard, and its action is non-deterministic. Then, we convert M_1 and C_1 into a set of predicates named $R_1 \cup D_1$ as follows:

$$\begin{aligned} &\{G\_STAT = \{opened, locked\}, \\ &\ g\_stat'_1 \in G\_STAT, g\_stat_1 \in G\_STAT \wedge auth_1 \in BOOL \\ &\ \wedge auth_1 = TRUE \wedge g\_stat'_1 = opened, \\ &\ g\_stat'_2 \in G\_STAT, g\_stat_2 \in G\_STAT \wedge auth_2 \in BOOL \\ &\ \wedge auth_2 = FALSE \wedge g\_stat'_2 = locked, \\ &\ auth'_3 \in BOOL, g\_stat_3 \in G\_STAT \wedge auth_3 \in BOOL \\ &\ \wedge auth'_3 \in BOOL\} \end{aligned}$$

where the states of variables in $R_1 \cup D_1$ are subscribed with 1, 2, and 3 to denote the states of variables belonging to the events *Open*, *Lock*, and *Auth* respectively. We can systematically conclude that the set of predicates from $R_1 \cup D_1$ is consistent by discharging FIS and INV. In the case of *Open*, INV is discharged as follows:

$$\begin{aligned} INV : {} & G\_STAT = \{opened, locked\} \wedge g\_stat_1 \in G\_STAT \\ & \wedge auth_1 \in BOOL \wedge auth_1 = TRUE \wedge g\_stat'_1 = opened \\ & \Rightarrow g\_stat'_1 \in G\_STAT \end{aligned}$$

which holds, because $g\_stat'_1$ is assigned to *opened* which is a member of G_STAT. We do not need to discharge FIS, since the action of *Open* is deterministic. INV also holds for *Lock* by the same reason. In the case of *Auth*, FIS and INV are discharged as follows:

$$
\begin{aligned}
FIS &: G\_STAT = \{opened, locked\} \land g\_stat_3 \in G\_STAT \\
&\quad \land auth_3 \in BOOL \Rightarrow \exists auth_3' \cdot auth_3' \in BOOL \\
INV &: G\_STAT = \{opened, locked\} \land g\_stat_3 \in G\_STAT \\
&\quad \land auth_3 \in BOOL \land auth_3' \in BOOL \Rightarrow auth_3' \in BOOL
\end{aligned}
$$

FIS holds because $auth_3'$ becomes a member of $BOOL$ which is a non-empty set. While, INV trivially holds. Note that, at this step, we construct M_1 and C_1 from B. Thus, we assume that $R_1 \cup D_1$ is complete with respect to B.

Second step

At the first step, the process to distinguish authorized from non-authorized persons is too conceptual by just returning the result without any specific conditions. Next, for M_2, the process needs to be refined to be more concrete. As described before, we can check the authority of a person through the possession of an ID card. Thus, we introduce a new variable *auth_card* into M_2 as a Boolean variable for representing whether the authority-checking process detects an authorized card. Since the variables *auth* and *auth_card* represent two similar concepts, we replace *auth* with *auth_card* by using an invariant $auth = TRUE \Leftrightarrow auth\_card = TRUE$. The event *Open* and *Lock* is simply refined into M_2 by replacing *auth* with *auth_card*. Then, the event *Auth* is refined into two new events with witnesses for the replacement of *auth* as follows:

$$
\begin{aligned}
NonAuthCard \mathrel{\hat=} &\ \textbf{with}\ auth' = FALSE \\
&\ \textbf{then}\ auth\_card' = FALSE\ \textbf{end} \\
Auth \mathrel{\hat=} &\ \textbf{with}\ auth' = TRUE \\
&\ \textbf{then}\ auth\_card' = TRUE\ \textbf{end}
\end{aligned}
$$

The event *AuthCard* represents the detection of an authorized card, while the event *NonAuthCard* is the opposite. Here, *Open* is similar to *Lock*, and *AuthCard* is similar to *NonAuthCard*. For simplicity, we thus show the conversion of M_2 and C_2 by focusing only on *Open* and *AuthCard*. The result of the conversion focusing on *Open* and *AuthCard* into the subset of the set of predicates $R_2 \cup D_2$ is as follows:

$$
\begin{aligned}
&\{G\_STAT = \{opened, locked\}, \\
&\ g\_stat_1' \in G\_STAT, g\_stat_1 \in G\_STAT \land auth_1 \in BOOL \\
&\ \land auth\_card_1 \in BOOL \land (auth_1 = TRUE \Leftrightarrow auth\_card_1 = TRUE) \\
&\ \land auth\_card_1 = TRUE \land g\_stat_1' = opened, \\
&\ auth\_card_3' \in BOOL, g\_stat_3 \in G\_STAT \land auth_3 \in BOOL \\
&\ \land auth\_card_3 \in BOOL \land (auth_3 = TRUE \Leftrightarrow auth\_card_3 = TRUE) \\
&\ \land auth_3' = TRUE \land auth\_card_3' = TRUE\}
\end{aligned}
$$

where the variables of $R_2 and D_2$ are subscribed with 1 and 3 to denote the variables of the event *Open* and *AuthCard* respectively. Again, by FIS_{ref} and INV_{ref}, we can ensure the consistency of $R_2 \cup D_2$. To ensure the relative completeness, we have to discharge GRD and SIM for each event with respect to its refined event. GRD and SIM are discharged for *Open* with respect to *Open* from $R_1 \cup D_1$ as follows:

$$
\begin{aligned}
GRD :\ & G\_STAT = \{opened, locked\} \land g\_stat_1 \in G\_STAT \\
& \land auth_1 \in BOOL \land auth\_card_1 \in BOOL \\
& \land (auth\_card_1 = TRUE \Leftrightarrow auth_1 = TRUE) \\
& \land auth\_card_1 = TRUE \Rightarrow auth\_card_1 = TRUE \\
SIM :\ & G\_STAT = \{opened, locked\} \land g\_stat_1 \in G\_STAT \\
& \land auth_1 \in BOOL \land auth\_card_1 \in BOOL \\
& \land (auth\_card_1 = TRUE \Leftrightarrow auth_1 = TRUE) \\
& \land auth\_card_1 = TRUE \land auth\_card_1 = TRUE \\
& \land g\_stat'_1 = opened \Rightarrow g\_stat'_1 = opened
\end{aligned}
$$

In this case, GRD holds because the replacement of $auth_1$ to $auth\_card_1$ is valid through the invariant $auth\_card_1 = TRUE \Leftrightarrow auth_1 = TRUE$, and SIM trivially holds. The successfully discharged GRD and SIM mean that *Open* of $R_1 \cup D_1$ can be logically derived from *Open* of $R_2 \cup D_2$. It is also true for *Lock*.

Seeing that the event *AuthCard* contains no guard, we only need to discharge SIM as follows:

$$
\begin{aligned}
SIM :\ & G\_STAT = \{opened, locked\} \land g\_stat_3 \in G\_STAT \\
& \land auth_3 \in BOOL \land auth\_card_3 \in BOOL \\
& \land (auth_3 = TRUE \Leftrightarrow auth\_card_3 = TRUE) \\
& \land auth'_3 = TRUE \land auth\_card'_3 = TRUE \\
& \Rightarrow auth'_3 = TRUE
\end{aligned}
$$

SIM holds in this case because of the existence of the witness $auth'_3 = TRUE$. Consequently, *AuthCard* of $R_2 \cup D_2$ is a valid refinement of *Auth* from $R_1 \cup D_1$. It is the same with $NonAuthCard$. From all the successfully discharged proof obligations, we conclude that $R_2 \cup D_2$ is relatively complete with respect to $R_1 \cup D_1$.

Third step

In this step, we need to more specific about the authority checking process. To check the authority of a person, the system has to ask him/her to swipe his/her card, then the system checks its ID whether the ID is an authorized one. We can introduce such concepts by adding two variables *card_swiped* and *auth_ID* representing the detection of swiping a card and the result from checking whether the swiped card is an authorized one, respectively. These two variables become guards of the event $AuthCard$ and $NonAuthCard$ in the previous model in order to map the new concepts with the events of detecting and not detecting an authorized card. Since, we only add new guards to two existing events, GRD is the only proof obligation to be discharged and it is easy to discharge it. We can conclude that $R_3 \cup D_3$ derived from M_3 and C_3 of this step is consistent and relatively complete with respect to $R_2 \cup D_2$.

The third step captures all the requirements about the automatic gate for the moment. The model may still contain description that are too generic for implementation, but it is sufficient for illustrating the conversion. This conversion shows that the generated proof obligations for Event-B refinement provide systematical way to verify the correctness of requirements as defined in the evolutionary framework.

3.5 Discussion

From the proofs shown in Section 3.3, we can see the compatibility of the properties preserved by the refinement mechanism and the correctness defined in the evolutionary framework. This means that all requirements specified in Event-B in which the generated POs are successfully discharged automatically follow the correctness of the framework with some additional cautions such as the lacks of the verification of axioms. Thus, in principle, Event-B can verify and preserve the correctness of requirements throughout the requirements evolution in both practical and formal aspects (Theorem 1). Besides, since the evolution of requirements in the evolutionary framework is similar to how safety requirements specifications are constructed, Event-B is also capable to preserve correctness of safety requirements specifications.

Our analysis and example also support the idea of beginning a refinement chain in Event-B by constructing a model capturing the overview (or main goals) of a system. After that, all specific details for the system are included later at the subsequent refinement steps. This idea is a development strategy commonly used for constructing Event-B models. The purpose of this strategy is to make the constructed Event-B model becomes more understandable, since stakeholders can quickly grasp the essence of the system by looking at the initial model. Another aspect provided by our analysis is that this strategy is an approach to keep the relative completeness of requirements in consideration. That means, given that the initial model is complete with respect to the goals of the system, all the requirements described in later refinement steps are also complete with respect to the goals.

One may wonder why the concept of the refinement of Event-B is compatible with the requirements evolution of the evolutionary framework since their objectives are different. The refinement is a technique to model a complicated requirements specification by describing it from abstract ones to concrete ones step-by-step. While, the requirements evolution is a common idea in constructing requirements specifications. Abstract requirements of the refinement can represent the broad understanding of the system by stakeholders. Moreover, through the refinement steps, the requirements become more concrete. Evolving the requirements also makes them more suitable to implement the system since the evolution reflects the better understanding of the system from stakeholders. Therefore, the refinement is compatible with the requirement evolution.

The reasons behind the lacks of verifying consistency of axioms, invariants, and guards of each event can be explained as follows: Axioms are designed to be definitions of carrier sets. If the definitions are inconsistent, it is possible to logically prove everything. Invariants are properties that must hold at every state of execution including the state of initialization. We may indirectly detect the inconsistency at such initialization. Lastly, the events with inconsistent guards are events that cannot be triggered at all. Those events cannot cause any changes to the corresponding model, so they are acceptable for Event-B.

3.6 Summary

In this chapter, we analyzed that Event-B refinement mechanism can almost preserve the correctness of requirements in the requirements evolution with respect to the evolutionary framework. This results in the possibility of using Event-B to maintain and verify the correctness of safety requirements specifications. Besides, this analysis also encourages the development strategy to use the overview of a system to construct an initial model in Event-B, and include more details later.

Chapter 4

ORDER model: a KAOS-based graphical approach to Event-B modeling

4.1 Rationale

As stated in Chapter 2, Event-B lacks of mechanism for analyzing and elaborating safety requirements, and also lacks of guidelines for using refinement mechanism. Thus, just modeling and verifying a safety requirements specification through Event-B is not sufficient to guarantee safety of a safety-critical system. Our choice is to propose an approach to use the KAOS method to cope with these shortcomings of Event-B. The rationale behind our choice is that the KAOS method provides a graphical approach for requirements analysis and elaboration through the goal model. The goal model contains the mechanism of the goal refinement for specifying requirement. This mechanism is close to how human think of a system and is easy to understand and reason with by all stakeholders. The goal model in the form of tree is also useful for understanding and reasoning about the goals. Furthermore, there are some similarities in KAOS and Event-B as follows:

- Both KAOS and Event-B have the notions of refinement. Besides, the purpose of both notions of refinement is to gradually realize a set of goals/specifications.
- An achieve goal is a prescriptive statement showing that when a current condition is satisfied, then a target condition will eventually be satisfied. An event in Event-B describes a behavior that when a guard (pre-condition) holds, then a set of action will be executed to make a set of post-conditions holds. Both the achieve goal and the event involve a condition that is satisfied before satisfying the next condition.
- A maintain goal and a domain property in KAOS describes some properties satisfied in all states of a system. An Event-B model also contains invariants, which are properties that must always hold in all states of a system.

Thus, we believe that KAOS is the right choice for overcome the shortcomings of Event-B.

However, direct translation from the KAOS goal model into the Event-B specification is inappropriate and ineffective due to the differences between KAOS and Event-B. Some differences are:

- In KAOS, every goal (except the root goal) must refine from a parent goal. Whereas, in Event-B, an event can be newly introduced into a concrete machine without explicitly refining from an abstract event.
- An achieve goal is usually formalized into the form of a temporal implication, e.g. $C \Rightarrow \diamond T$. While, an event in Event-B is formalized into the form of a conjunction, e.g. $G \wedge BA$.
- The semantics of refinement are different in both approaches. Goal refinement means when all sub-goals are satisfied, then their parent goal is satisfied. Refinement of event means a concrete event preserves the behavior described in its corresponding abstract event.

These differences lead to difficulties to directly translate the KAOS goal model into Event-B specification. Therefore, rather than the direct translation of KAOS to Event-B, our proposed approach is to use KAOS together with Event-B is by introducing a new graphical approach based on the KAOS method to support Event-B modeling. By this approach, we can design the model such that it can support both the characteristic of KAOS goal refinement and Event-B refinement. The model we introduce is named "ORDER model".

4.2 ORDER model

'ORDER' stands for '**O**rganized **R**equirements **D**edicated to **E**vent-B **R**efinement'. Since the main objective we propose the ORDER model is to assist Event-B modeling with the support of the KAOS method, we design the ORDER model to focus on graphical expression of Event-B model in a goal-model-like diagram. Then, KAOS is for assisting the creation of such diagram. Thoroughly, the ORDER model must have capabilities to:

- Show refinements of events in a form of tree
- Clearly separate steps of refinement
- Treat an event similar to an achieve goal of KAOS
- Treat an invariant similar to a maintain goal or a domain property of KAOS
- Apply the characteristic of KAOS goal refinement
- Support transformation to Event-B models

- Aid understandability and justifying Event-B model

To realize all the capabilities, we present two diagrams for expressing Event-B model: refinement tree diagram, and event transition diagram. The detailed explanations of each diagram are provided by the following sections.

4.3 Refinement tree diagram

Refinement tree diagram is a diagram showing refinements of event from a chain of refinements of Event-B machines in the form of tree. This diagram can demonstrate relationship among events and invariants in a machine as well. One level of the tree is regarded as one Event-B machine. We use arrows denoting refinements of events to separate levels of the refinement tree diagram. More details about the arrows can be found in the following subsection.

4.3.1 Components and links

We extend the same set of figures denoting components of KAOS goal model to denote Event-B components in the refinement tree diagram and relationship among them. In the refinement tree diagram, a parallelogram denotes an Event-B event, a trapezoid denotes an Event-B invariant, and an arrow denotes a refinement from an abstract event to a concrete event. The extension parts are a bold line denoting a relationship between two events, and a dashed line denoting a relationship between an event and an invariant.

Each figure representing either an event or an invariant is described with natural language corresponds to the description of the Event-B component. The natural language acts as identifiers for formal descriptions in Event-B specifications. Even if it acts only as the identifiers, the meaning of the natural language should conform to the semantic of the corresponding predicate. Therefore, the natural language that can be used in the diagram is limited to what the first-order predicate logic of Event-B can describe. For examples, if a predicate in Event-B is written as $(P = TRUE \wedge Q = TRUE) => R = FALSE$ where P, Q, R are Boolean variables, one possible identifier of this predicate in the natural language is "If P and Q become true then R becomes false". This is up to what P, Q, R represent in the specification.

The latter parts of this subsection describe these components and links.

Event

An event of Event-B is denoted by a parallelogram with the written description of the event. Figure 4.1 shows the general form denoting event in the refinement tree diagram. The description of an event is written in the same way the event is described when modeling an Event-B model:

$$evt \mathrel{\hat{=}} \textbf{any} p \textbf{ when } G \textbf{ with } W \textbf{ then } S \textbf{ end}$$

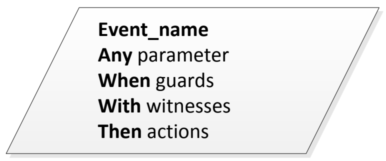


Figure 4.1: An event

This means that the description is composed of name, parameters, guards, witnesses, and actions of the event. The parts, which do not present in the description of an event, can be omitted. We allow using natural language as identifiers for the Event-B description of each event.

Because the refinement tree diagram is in the form of tree, we need to define a root node of the tree. According to semantics of Event-B specification, all events refine from an event named '*skip*'. Thus, we define *skip* as the root node of every tree. Figure 4.2 shows the appearance of a root node.

Figure 4.2: A root node

Invariant

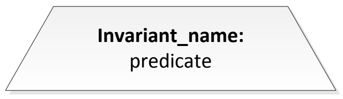


Figure 4.3: An invariant

An invariant in Event-B is denoted by a trapezoid with the predicate form of the invariant. Optionally, one may also include the name of the invariant into the figure. We also allow using natural language instead of formal predicate to describe an invariant. Figure 4.3 shows the general appearance of an invariant.

In all Event-B models, there are invariants which are for variable typing. These kind of invariants can be omitted when drawing a refinement tree diagram.

Refinement of event

A refinement of event is represented by an arrow with a small circle for linking all concrete events refine the same abstract event through lines. Here, we specify that 'refinement of

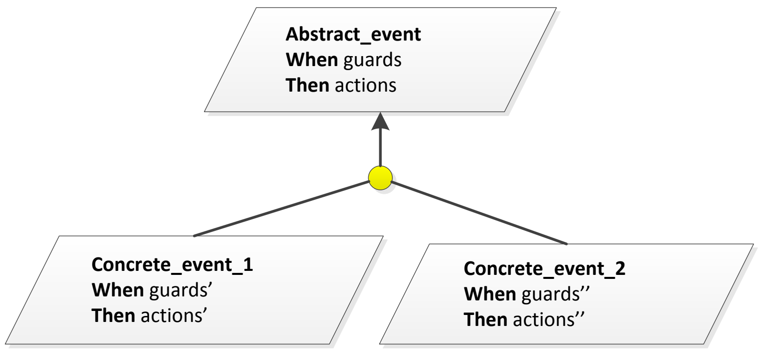


Figure 4.4: Refinements of event

event' means there is some changes in the description of a concrete event comparing to its abstract event. The appearance of refinements of event in a refinement tree diagram is shown in Figure 4.4. Because abstract events belong to an abstract machine and concrete events belong to a concrete machine, this means that the arrow can separate level of the refinement tree diagram. For a refinement, if the proof obligations GRD and FIS as defined in Section 2.1 are successfully discharged, the refinement is valid.

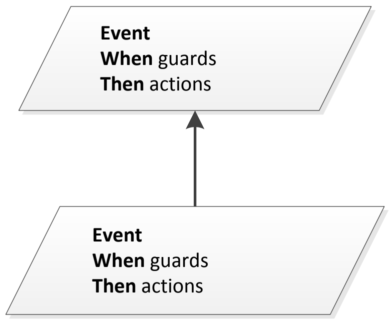


Figure 4.5: Copy of event

If the description of a concrete event is the same with its abstract event, we regard the event as a 'copy' of the abstract event. In this case, we use a plain arrow to show the copy as in Figure 4.5.

Both the refinement and the copy of event can be written into Event-B specification as 'REFINES' relationships between abstract events and concrete events.

Event-event relationship

There might be some relationships among two or more events in the same machine which we want to explicitly describe. These relationships are needed because they can help stakeholders understand how events interact with each other. The relationships can be shown through lines among events. We allow writing type and name of the relationship

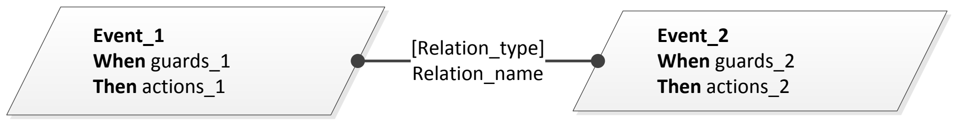


Figure 4.6: Event-event relationship

on the line. Figure 4.6 shows the general appearance of the relationship between events. For the scope of this thesis, we define only two types of relationships: parallel and before. The detailed explanations about these two types of relationships are provided in Section 4.4. Even though these relationships are explicit in our diagram, they are implicit when written in Event-B specification. In facts, Event-B does not provide notations for explicitly denoting these relationships.

Event-invariant relationship

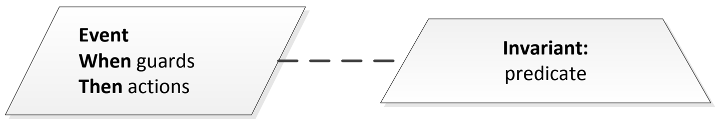


Figure 4.7: Event-invariant relationship

Invariants are needed for restricting possible values of variables. They can show relationship among variables in a system. Thus, invariants also restrict the possible results of events. Conversely, events might provide us some idea about important invariants needed to be include in a specification. These relationships between events and invariants can be shown in a refinement tree diagram through dashed lines between them. Figure 4.7 shows how to link an invariant to an event. We encourage linking an invariant to all events which it is related to. However, this might reduce increase the complexity of a refinement tree diagram. For convention, if an invariant is related to a lot of events of an Event-B machine, we allow omitting all dashed lines for linking them, or showing only some necessary links. Another way to link an invariants to a set of concrete events which refines the same abstract event is to link the invariant to the small circle representing refinement as in Figure 4.8.

4.3.2 Construction rules

A refinement tree diagram can be constructed through the combinations of components and links presented in the previous subsection. Some constraint was already explained for each component and link. This subsection aims to gather and extend those constraint for defining construction rules of the refinement tree diagrams.

The construction rules of the refinement tree diagrams are as follows:

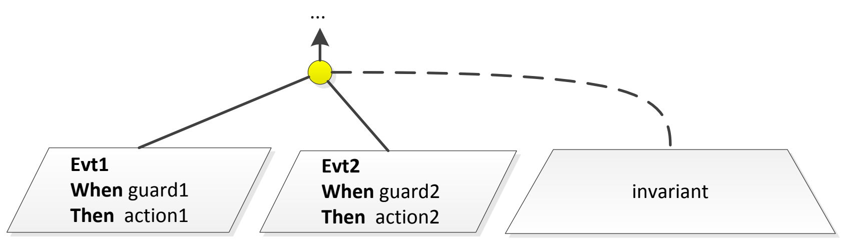


Figure 4.8: Linking an invariant with concrete events refining the same event

1. All arrows, denoting refinements and copies of events, separate two consecutive levels of a refinement tree. If there are arrows from a group of events to another group of events, the former group belongs to upper level and the latter group belongs to the lower (next) level.

2. The root node which is always a node representing *skip* event belongs to the zeroth level of the tree. Even though all events refines from the *skip* event according to the semantics of Event-B specification, there is no need to explicitly copy the *skip* event to all levels of a tree.

3. All events in the next level from the zeroth level (the 'first' level) of a tree refine the *skip* event, except the events whose origins can be explained through relationships with other events.

4. If a level of a tree has its next level, all events in the former level must have at least one copying or refining event in the next level.

5. All bold lines and dashed lines, denoting relationships among events and invariants, must be defined only on events and invariants withing the same level of a tree.

6. It is general to introduce some new event, i.e. they do not refine any event from the upper level, to a level of a refinement tree. In this case, each newly introduced events must link at least with one event refining an event in the upper level. This is for showing the rationale behind the newly introduced events.

Figure 4.9 shows an example of a refinement tree diagram which follows the construction rules. In the example, the zeroth level of the tree contains only the *skip* event, the first level contains two events: $Evt1$ and $Evt2$, and the second level contains four events, one is a copy of $Evt1$, one is a copy of $Evt2$, and $Evt1\_1$ and $Evt1\_2$ refines $Evt1$. The event $Evt2$ of the first level does not explicitly refine the *skip* event, since it has 'before' relationship with $Evt1$. In the second level, there is one invariant links with the event $Evt1\_2$, because guards and actions of $Evt1\_2$ appear in the invariant.

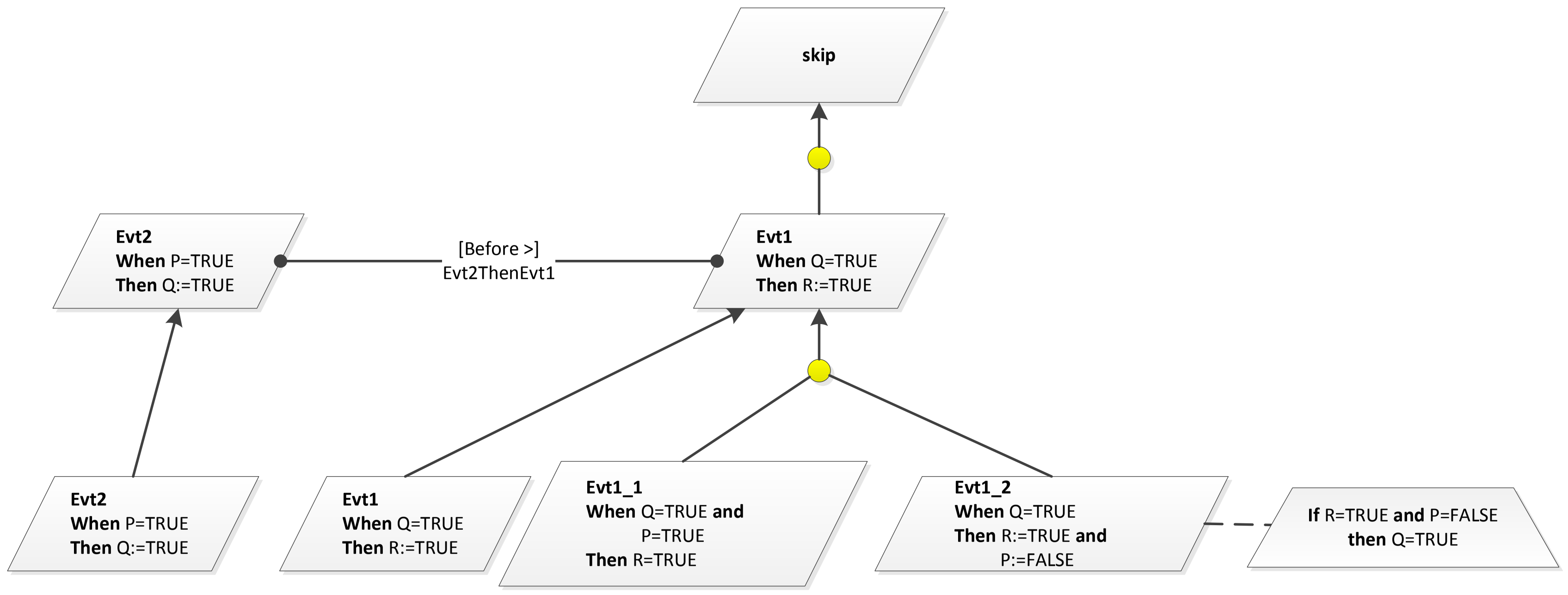


Figure 4.9: An example refinement tree diagram

4.3.3 Transformation to Event-B model

Since we allow using natural language to be identifiers of the Event-B descriptions for events and invariants and the descriptions always contain variables, carrier sets, or constants, at least, we need to know all variables and data structure which can represent data and artifacts of a modelled system. In this research, we assume that all needed variables have been specified before using our approach. Some approaches that can be used for specifying the variables are the class diagram of KAOS [VL09] and the UML-B [SBS09]. Both approaches are based on UML [SH01].

Regardless of how variables and data structures are specified, the transformation from a refinement tree diagram into Event-B specifications can be done through the following principles:

- All events and invariants within the same level of a tree must be written in the description of the same Event-B machine.
- Two consecutive levels of a refinement tree diagram means that the lower level is a concrete machine refines the abstract machine from the upper level. This refinement relationship must be written in the concrete machine as the clause $refines$ followed by the abstract machine's name.
- An event in a refinement tree diagram contains the clauses *any*, *when*, *with*, and *then*. Since these clauses are derived from how an event is described in Event-B. Each clause from the tree can be directly map to the corresponding clause in the Event-B specification.
- Each arrow from an abstract event to a concrete event can be represented in Event-B specification through the clause $refines$ followed by the name of the abstract event. The clause $refines$ must be written in the description of the concrete machine.

Note again that the bold lines and dashed lines are implicit in Event-B specification.

From the refinement tree diagram in Figure 4.9, two Event-b machines can be created. We assume that the variables P, Q, and R are Boolean variables, which their types can be declared through invariants in the form of $P \in BOOL$. The followings are parts of the machines which can be derived from the refinement tree diagram.

Initial model (First level of the tree)

$$Evt2 \mathrel{\hat=} \textbf{when } P = TRUE \textbf{then } Q := TRUE \textbf{ end}$$
$$Evt1 \mathrel{\hat=} \textbf{when } Q = TRUE \textbf{then } R := TRUE \textbf{ end}$$

First refinement (Second level of the tree)

$$\begin{aligned}
Evt2 \mathrel{\hat{=}}\ & \textbf{refines } Evt2 \\
& \textbf{when } P = TRUE \textbf{ then } Q := TRUE \textbf{ end} \\
Evt1 \mathrel{\hat{=}}\ & \textbf{refines } Evt1 \\
& \textbf{when } Q = TRUE \textbf{ then } R := TRUE \textbf{ end} \\
Evt1\_1 \mathrel{\hat{=}}\ & \textbf{refines } Evt1 \\
& \textbf{when } Q = TRUE\ \wedge\ P = TRUE \textbf{ then } R := TRUE \textbf{ end} \\
Evt1\_2 \mathrel{\hat{=}}\ & \textbf{refines } Evt1 \\
& \textbf{when } Q = TRUE \textbf{ then } R := TRUE\ \wedge\ P := FALSE \textbf{ end}
\end{aligned}$$

The following invariant must be included in this machine:

$$R = TRUE\ \wedge\ P = FALSE\ \Rightarrow\ Q = TRUE$$

4.3.4 Correctness of the refinement tree

After the transformation from a refinement tree diagram to an Event-B specification, we can let the Rodin platform generates the proof obligations to verify the specification. Thus, all the refinement relationships appeared in the refinement tree diagram must obey the proof obligations. As we analyzed in the previous chapter, the proof obligations of Event-B can ensure the completeness and consistency of requirements in each step of evolution of the evolutionary framework. This associates the refinement tree diagram with the evolutionary framework through the proof obligations.

We regard a refinement which follows the proof obligations as a valid one. From this, we define a valid level of a refinement tree diagram as follows:

Definition 12 (A valid level of a refinement tree)**.** *a level of a refinement tree is valid if and only if all the generated proof obligations for an Event-B model conforming to the level are successfully discharged.*

By the above definition, we can also define a valid refinement tree as follows:

Definition 13 (A valid refinement tree)**.** *A refinement tree is valid if and only if all levels of the refinement tree are valid.*

From our analysis in Chapter 3 that the proof obligations of Event-B can preserve the correctness according to Lemma 1, we can trivially conclude another lemma from Definition 12 as follows:

Lemma 2 (Correctness of a valid level of a refinement tree)**.** *A valid level of a refinement tree conforms to a correct step of evolution in the evolutionary framework.*

Because of the above lemma, we can finally conclude a theorem:

Theorem 6 (Correctness of a valid tree)**.** *A valid refinement tree represents a correct chain of evolution in the evolutionary framework.*

Theorem 6 means that if we construct a valid refinement tree, it is the same with having a correct chain of evolution of requirements.

4.4 Event transition diagram

While a refinement tree diagram shows how a chain of refinement of Event-B machines, an event transition diagram focuses more on how events in a machine interact with each other. The main purpose of this diagram is to help stakeholders understand how a system behaves through a flow of events. The information appeared in this diagram is from the refinement tree diagram. Hence, the event transition diagram is always used with the refinement tree diagram.

For the event transition diagram, we define that after an event is executed, the actions of the event might trigger some other events. Then, the system make a transition from the former event to the execution of the next event. This causes a chain of execution of events (a flow of events).

The following subsections explained components and links of the event transition diagram and its association with the refinement tree diagram.

4.4.1 Components and links

The transitions of events are similar to transitions of states of a system. There are various diagrams for demonstrating transitions of states such as UML state diagram [SB06], and finite state machine diagram [Gil70]. Usually, a state is denoted by a circle and a transition is denoted by an arrow from one state to another. We also apply this concept to define our event transition diagram.

Event

An event is denoted by a circle with the name of the event is written inside. Figure 4.10 shows the appearance of an event in the event transition diagram.

Figure 4.10: An event

Note that the name written in the circle must conform to the name of the event in a refinement tree diagram. This is for cross-reference between two diagrams.

Since every Event-B machine must contain an initialization event for initiating variables, the notation of the initialization is also defined in our diagram. The initialization event is denoted a black circle as in Figure 4.11.

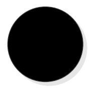


Figure 4.11: An initialization event

Event transition

After an event is executed, the state of variables of a system is changed. This might cause another event to be executed. We represent the transition from one event to another by using an arrow from the former event to the next event. This arrow might be tagged with 'before' relationship to show that one event is executed before another. We can name the relation and write the name after the 'before' relation. If we name the relation, the name must also appear in the refinement tree diagram in the level which corresponds with the event transition diagram. Figure 4.12 shows a one-to-one transition from one event to another.

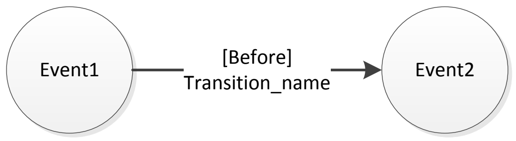


Figure 4.12: One-to-one transition

Some event might be able to trigger more than one event. In this case, two or more arrows which has the same event as their origin are allowed. This can be regarded as one-to-many transitions. Figure 4.13 shows an example of the one-to-many transitions. Note that the one-to-many transitions defined here mean that after the former event is executes, there is only one event to be executed at a time, depending on the results of the former event.

Parallelized event

The one-to-many transitions mean that one event is triggered at a time. In the case of more than one event can be triggered and they can be executed in parallel, the parallelized events are grouped into one rectangle. If there is a transition from an event to a group of parallelized events, an arrow can be pointed directly to the rectangle representing the

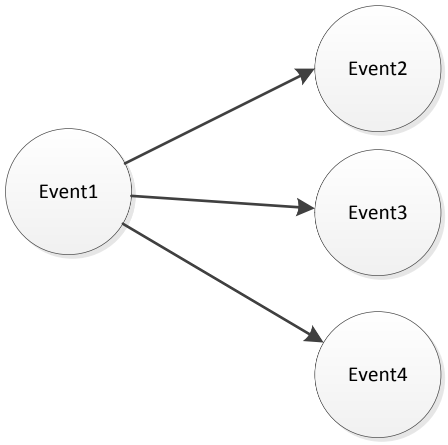


Figure 4.13: One-to-many transition

group of events. The parallelized events might have distinct destination of transitions, so each event can have its own transition arrows to trigger some other events. Figure 4.14 shows an example of a transition from an event to a group of parallelized event and then each parallelized event has a distinct transition to some other events.

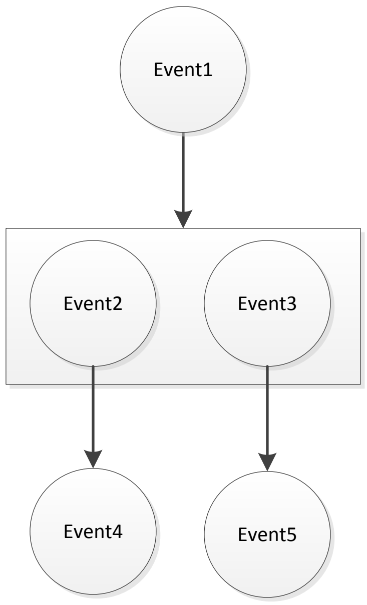


Figure 4.14: Transitions of parallelized event

Note that, according to the semantics of Event-B specification, the execution of parallelized events does not overlap, but it means that the order of execution is arbitrary.

4.4.2 Association with the refinement tree diagram

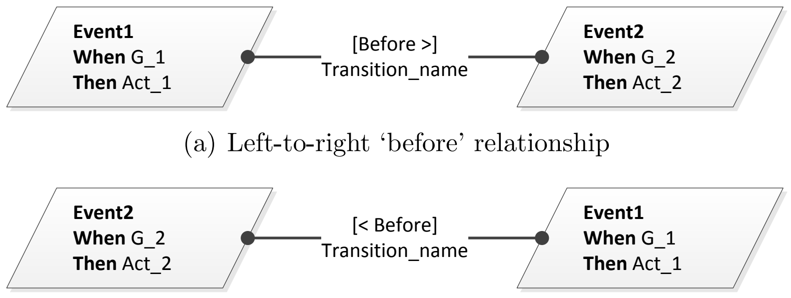
As mentioned before, the information appeared in the event transition diagram must be traceable to the corresponding level of a refinement tree diagram. The associations between elements of the refinement tree diagram and the event transition diagram are thoroughly described in this subsection.

Event

All events appeared in a level of a refinement tree must also be in the corresponding event transition diagram. The name of events appeared in both diagrams must be the same.

'Before' relationship

When there is a 'before' relation from one event to another, the relation must appeared in both the event transition diagram. The direction of a relation in an event transition diagram can be specified through the direction of an arrow. However, the direction of the relation in the corresponding refinement tree diagram is vague, because the the bold line itself has no direction. In facts, the direction can be shown in both left-to-right and right-to-left ways. Figure 4.15 respectively shows the left-to-right and right-to-left ways to represent the 'before' relationship from Figure 4.12 in a refinement tree diagram.



(a) Left-to-right 'before' relationship

(b) Right-to-left 'before' relationship

Figure 4.15: 'Before' relationship in a refinement tree diagram

Parallelized events

Events in the same group of parallelized events can be shown in the refinement tree diagram through the 'parallel' relationship. Figure 4.16 shows how to represent the events *Event*2 and *Event*3 from Figure 4.14 in a refinement tree diagram.

4.4.3 Example

Figure 4.17 shows an example of a event transition diagram. In this example, after the initialization, *Event*1 will be executed. The results of execution of both *Event*1 and *Event*2 can trigger the same group of parallelized events *Event*3 and *Event*4. *Event*4

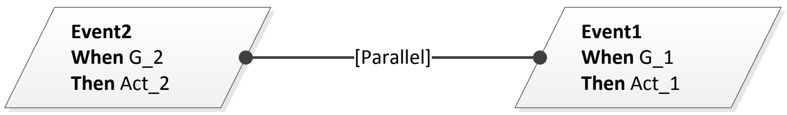


Figure 4.16: 'Parallel' relationship in a refinement tree diagram

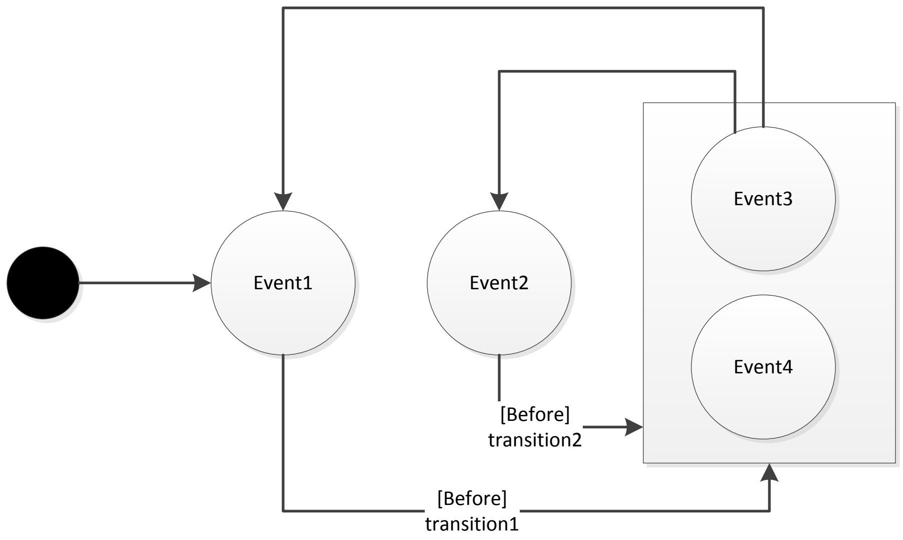


Figure 4.17: An example event transition diagram

does not trigger any event after its execution. Lastly, *Event*3 can trigger both *Event*1 and *Event*2, but just one event at a time.

4.5 Refinement patterns

The refinement tree diagram and the event transition diagram we define in the previous sections can only assist structuring and understanding an Event-B specification. Another objective is to use these diagrams for analysis and elaboration of safety requirements specification in a similar way as the goal model of KAOS. Applying the mechanism of goal refinement directly to refinement of event is difficult, since their purposes and semantics of refinement are different. Our solution is to apply the goal refinement patterns of KAOS to refine Event-B event.

The goal refinement patterns of KAOS as mentioned in Chapter 2 are the frequently used refinement tactics [DVL96]. This entails that the patterns can provide the capabilities of the requirements analysis and elaboration of KAOS. Our intention is to create a set of refinement patterns for Event-B based on the goal refinement patterns. We aims to keep the shape of a sub-tree representing a step of refinement from an event in a refinement tree diagram looking almost the same as the applied goal refinement pattern. By this way, we can analyze and elaborate safety requirements specification through the usage of the goal-based patterns. Furthermore, the refinements of event can inherit the capabilities of KAOS.

Not all the temporal operators and goal refinement mechanism can be supported by Event-B. Some pattern contains the unsupported operators. Some pattern is for refining a maintain goals into sub-goals, while Event-B does not provide mechanism for refining an invariant. Therefore, only some pattern can be applied to Event-B. This research focuses on deriving refinement patterns from two KAOS patterns: the milestone-driven refinement pattern, and the decomposition-by-case pattern. The description of each pattern can be found in Section 2.3.4. Aside from the goal refinement patterns, we also create a few new patterns for supporting more possible ways of refining events.

Considering the generic refinement tree for safety goals in Section 2.3.5, this tree is for describing generic goals of safety-related systems. The tree described that, to avoid a dangerous state, the dangerous state must be anticipated, and when it is detected, an alarm must be issued and finally handled. We can roughly separate the tree into two parts: anticipation part and alarm handling part. We regard the anticipation part as an input monitoring phase, and the alarm handling part as a decision phase of a system. From this, we think that behavior of many safety-critical systems can be divided into phases. Thus, we define the refinement patterns in the phase-based way. The patterns can be regarded as phases-based patterns. A certain number of variables are needed to be included to Event-B specifications for representing phases, in addition from the identified variables before using the ORDER model.

Our phase-based patterns are as follows:

- *Phase-decomposition refinement pattern.* This pattern is for decompose the behavior of a system into phases.

- *Event-forking refinement pattern.* This pattern is for refining an event into one or more parallelized event(s).

- *Case-decomposition refinement pattern.* This pattern is based on the decomposition-by-case refinement pattern of KAOS. Its purpose is for refining an event with different cases.

- *Milestone-driven refinement pattern.* This pattern is from the milestone-driven refinement pattern of KAOS. Its purpose is for decomposing an event into two or more events which will be executed consecutively.

The detailed explanation of each pattern is provided in the next chapter in the form of pattern document. In the pattern document, some constraint for each pattern is reported. The purpose of the constraints is to make the parts of the refinement tree diagram following the patterns can be proved to be correct by the proof obligations of Event-B. This reduces the effort for making the refinement tree diagram valid.

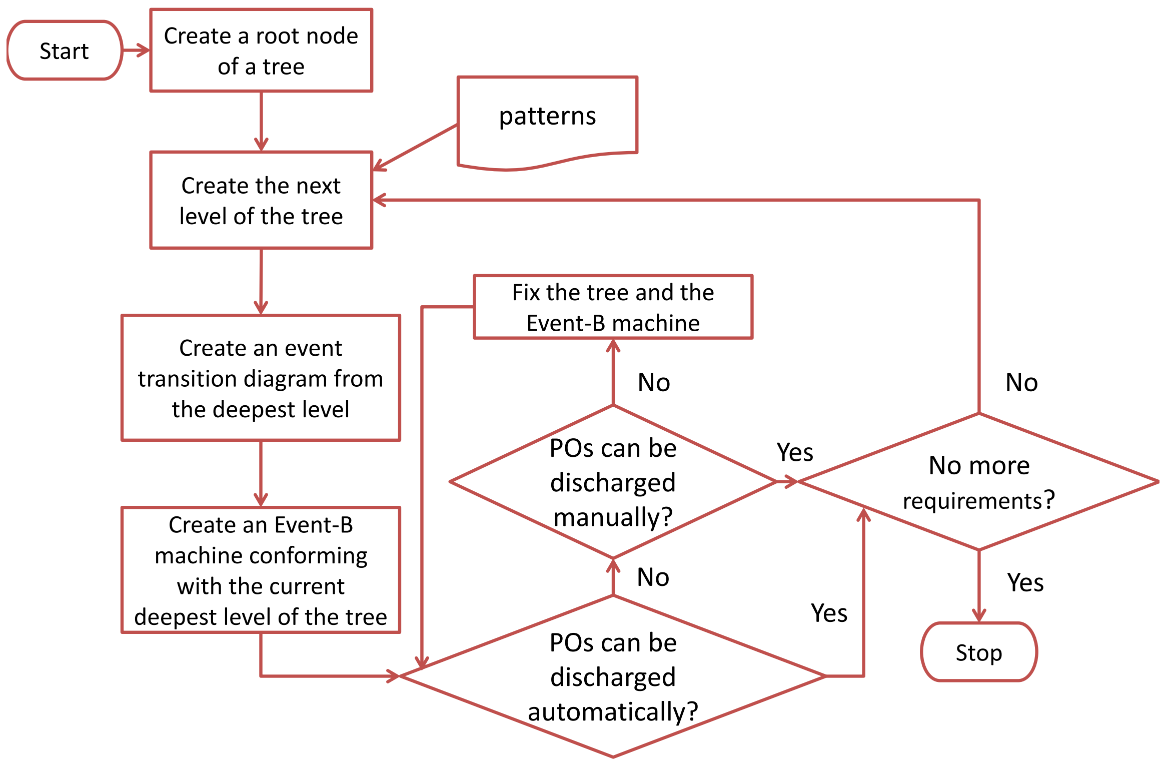

Start
Create a root node of a tree
patterns
Create the next level of the tree
Create an event transition diagram from the deepest level
Create an Event-B machine conforming with the current deepest level of the tree
Fix the tree and the Event-B machine
No
POs can be discharged manually?
Yes
No
No more requirements?
No
Yes
POs can be discharged automatically?
Yes
Yes
Stop

Figure 4.18: Process of using ORDER model

4.6 Guideline for using ORDER model

We propose a guideline for using our ORDER model as in Figure 4.18. The explanation of each step is as follows:

1. Construct a refinement tree diagram by starting with constructing a root node.
2. Refine and copy event from the previous level to construct the next level of the tree. Refinement patterns can be used in this step.
3. Construct an event transition diagram corresponding to the current deepest level of the tree.
4. Transform the current deepest level of the tree to an Event-B machine.
5. Check the result of discharging the proof obligations by the automated prover of the Rodin platform. If they are successfully discharged, go to step 8, otherwise, go to next step.
6. Use the interactive prover of the Rodin platform to try interactively proving. If the interactive prover can discharge the proof obligations, go to step 8, otherwise, go to next step.

7. Fix the tree, the undischarged proof obligations discover some errors in the Event-B machine. Then, go back to step 5.

8. Check that the current Event-B model covers all requirements or not. If all requirements are already modelled, stop the process, otherwise, go back to step 2.

Note that, at each time performing Step 2 of the guideline, only a small number of new variables and new features of the system should be introduced. This is for mitigating the complexity of a specification throughout various levels of the tree.

4.7 Summary

In this chapter, we propose the ORDER model which is composed of two diagrams: the refinement tree diagram and the event transition diagram. The refinement tree diagram is for demonstrating how events are refined from one level into the next level of the tree. Here, one level of the tree corresponds to one Event-B machine. The rationale of some invariant can also be demonstrated through the relationships with some event. While, the event transition diagram is for demonstrating how events in a level of the tree (an Event-B machine) interact to each other. The interaction between events are shown in the form of transitions of events. We define some association between two diagram to make them consistent to each other. These two diagrams can help stakeholders to understand and justify an Event-B model. The transformation from both diagrams into an Event-B model is possible.

Because we define that a refinement tree diagram is valid only if the proof obligations generated after modeling the tree in Event-B can be successfully discharged. As a result, a valid refinement tree diagram conforms to a correct chain of evolution of requirements as analyzed in Chapter 3. The evolutionary framework of requirements correctness is indirectly related to the ORDER model via the proof obligations of Event-B.

Our ultimate goal is to use the goal refinement mechanism from the KAOS method to assist analysis and elaboration of safety requirements specification. It is not appropriate to apply the goal refinement mechanism directly to Event-B because it is different from Event-B refinement. Therefore, we indirectly apply the mechanism through the usage of goal refinement patterns. We create a set of refinement patterns based on the patterns of KAOS for construct the refinement tree diagram. In this way, the Event-B refinement can inherit the nature of goal refinement mechanism.

The next chapter explains our refinement patterns in the form of pattern document.

Chapter 5

ORDER model: refinement patterns

In the previous chapter, we explain our intention to create a set of refinement patterns based on the goal refinement patterns of KAOS, and briefly describe each pattern. This chapter aims to thoroughly present all of our refinement patterns in a form of a pattern document that could be useful for a developer who is interested in using the ORDER model and the patterns.

5.1 Format of pattern document

The format of the pattern document we used in this chapter is adapted from the format of the catalog of goal refinement patterns [VL09] and the Event-B pattern description [Für09] as follows:

- **Description and applicability**: Short description of what the pattern is about and when it should be applied.
- **Illustration**: The general form of the refinement tree diagram and the event transition diagram representing the pattern.
- **Transformation to Event-B model**: Explanation of how to model this pattern in Event-B.
- **Constraint**: The rules to be followed when applying the pattern.
- **Example**: Some examples demonstrating the application of the pattern.
- **Notes**: Additional notes for developer.

5.2 Phase-decomposition refinement pattern

5.2.1 Description and applicability

The phase-decomposition refinement pattern divides abstract behavior of a system into two or more phases. One phase is represented by one event. Only the transition from one phase to another is described in each event. The flow of transitions is in the form of a cycle for iterative behavior of the system. Then, each concrete event of the system in all subsequent levels (refinements) will belong to one of the phases. This pattern is applicable for modeling an initial model of Event-B (the second level of the refinement tree diagram). The possible phases used for dividing behavior of a system are: input phase, decision phase, idle phase, and reset phase. The input phase is for monitoring inputs of the system. The decision phase is for taking a decision based on the inputs. The idle phase is for representing the idle state of the system. The reset phase is for resetting some variable of the system before going the next phase.

5.2.2 Illustration

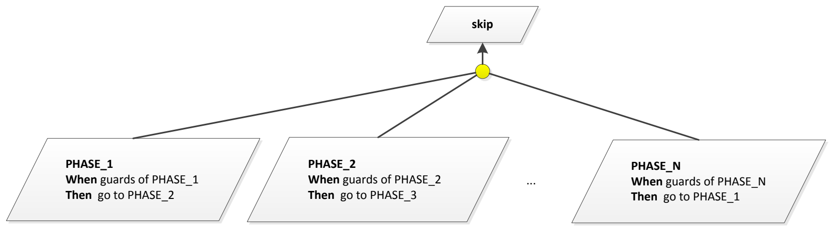


Figure 5.1: General phase-decomposition refinement pattern: the refinement tree diagram

In the phase-decomposition pattern, a finite number of phases can be defined and used for dividing behavior of a system. In Figure 5.1, there are N phases of the system. The system is running in a repeated cycle of these N phases as shown in Figure 5.2.

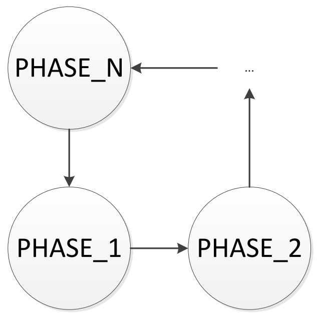


Figure 5.2: General phase-decomposition refinement pattern: the event transition diagram

5.2.3 Transformation to Event-B model

The phases of a system can be represented by Boolean variables. Because each Boolean variable can be either $true$ or $false$, $\lceil log_2 N \rceil$ Boolean variables are needed for representing N phases. One event is for one phase. When guards of an event hold, the system enters the phase representing by the event. After actions of an event change the state of variables, the system transits to the next phase. The name of each event should correspond to the name of the represented phase.

5.2.4 Constraint

To ensure that a system does not obstruct at one of its phases, there are some constraint when using this pattern. Developers need to guarantee that each phase can be passed through and has a unique transition to another phase. Furthermore, to keep the simplicity of abstract behavior of a system, this pattern does not allow branching.

5.2.5 Example

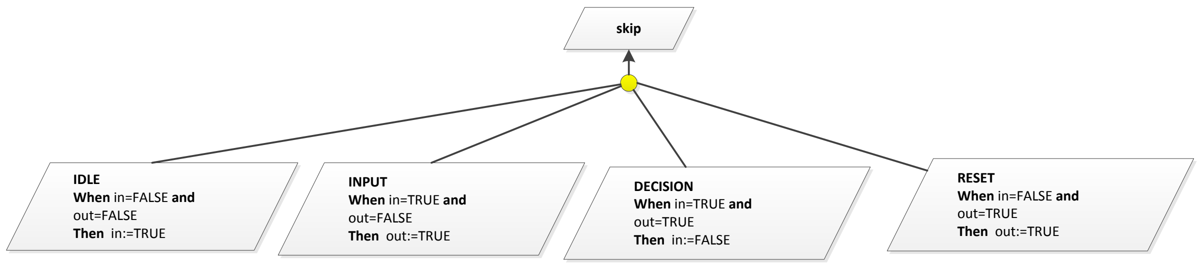


Figure 5.3: An example refinement tree diagram of the phase-decomposition refinement pattern

This example divides a system into 4 phases: idle, input, decision, and reset. Two Boolean variables, *in* and *out*, are declared for representing these 4 phases. Figure 5.3 shows the refinement tree diagram of the example. The flow of transitions is in a repeated cycle as in Figure 5.4.

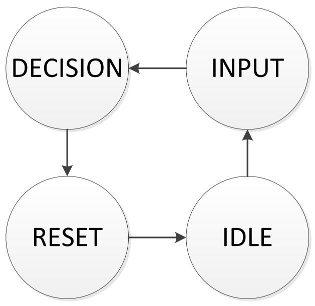


Figure 5.4: An example event transition diagram of the phase-decomposition refinement pattern

This example can be transformed into 4 Event-B events as follows:

$$IDLE \mathrel{\hat{=}} \textbf{when } in = FALSE \wedge out = FALSE \textbf{ then } in := TRUE \textbf{ end}$$
$$INPUT \mathrel{\hat{=}} \textbf{when } in = TRUE \wedge out = FALSE \textbf{ then } out := TRUE \textbf{ end}$$
$$DECISION \mathrel{\hat{=}} \textbf{when } in = TRUE \wedge out = TRUE \textbf{ then } in := FALSE \textbf{ end}$$
$$RESET \mathrel{\hat{=}} \textbf{when } in = FALSE \wedge out = TRUE \textbf{ then } out := FALSE \textbf{ end}$$

5.3 Event-forking refinement pattern

5.3.1 Description and applicability

This pattern is for describing environmental behavior which is usually non-deterministic and can interleave with other environmental behavior. Inputs of a system can be regarded as this kind of behavior. Thus, this pattern is applicable to describe the input phase of the system. A group of parallelized event can be used for representing the interleaving behaviors. The creation of a group of parallelized events is called event forking. One event can be denoting one input and uses a non-deterministic action for representing all possible values of the input. One input can also be represented by two or more events where each event represents different values of the input.

5.3.2 Illustration

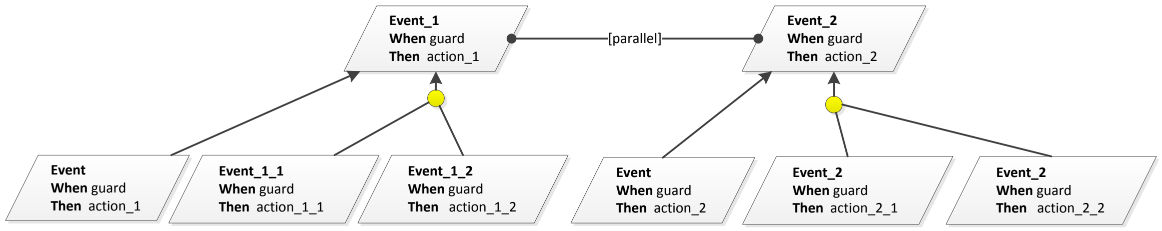
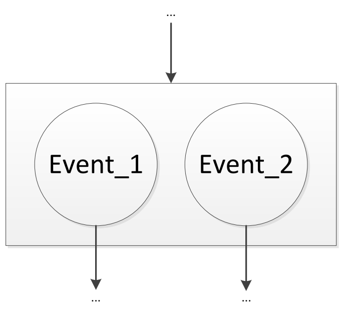
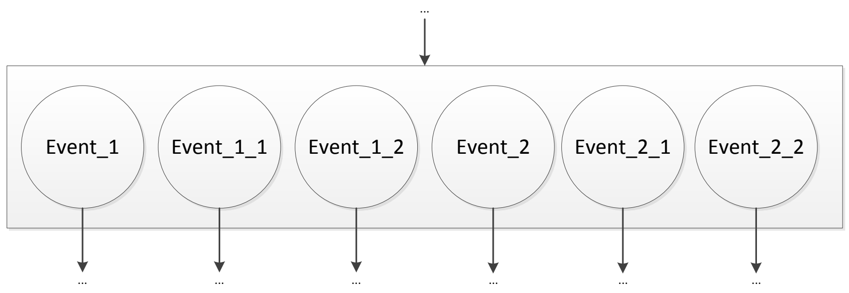


Figure 5.5: General event forking refinement pattern: the refinement tree diagram

The event forking can be performed both in a single level and in consecutive levels. The event forking in a single level is by introducing events with the same set of guards. The event forking in consecutive levels is by refining an abstract event into two or more events with the same set of guards. In Figure 5.5, the event forking is done by introducing two events *Event_1* and *Event_2* with the same set of guard. In the subsequent level, the event forking is done by copying and refining *Event_1* and *Event_2* into 6 concrete events with the same set of guards. If the system has a lot of interleaving behavior, the copy of *Event_1* and *Event_2* can be used for gradually performing the event forking by introducing more parallelized events in later subsequent levels. The corresponding event transition diagrams of the refinement diagram above are shown in Figure 5.6.



(a) Upper level



(b) Next level

Figure 5.6: General event forking refinement pattern: the event transition diagram

5.3.3 Transformation to Event-B model

For the event forking in a single level, one may start from considering a concrete event refining an abstract event as a base, then create a new event with the exactly same guard as the based event. The new event should have distinct actions from the based event in order to differentiate them form each other.

In case of the event forking in the subsequent level, ones may copy and refine an abstract event into two or more events whose actions are differentiated without changing its guards.

Names of the parallelized events are up to developers.

5.3.4 Constraint

For the event forking in a single level, the proof obligation EQL should be considered. If new events modify the same set of variables with the based events, it means that the new events should not be new, but they should refine from the abstract events of the based events.

In case of the event forking in the subsequent level, the modifications of actions should follow the proof obligation SIM, which means that the concrete actions should simulate the abstract actions. Besides, all the modifications should focus on new variables of the subsequent level to avoid violating EQL.

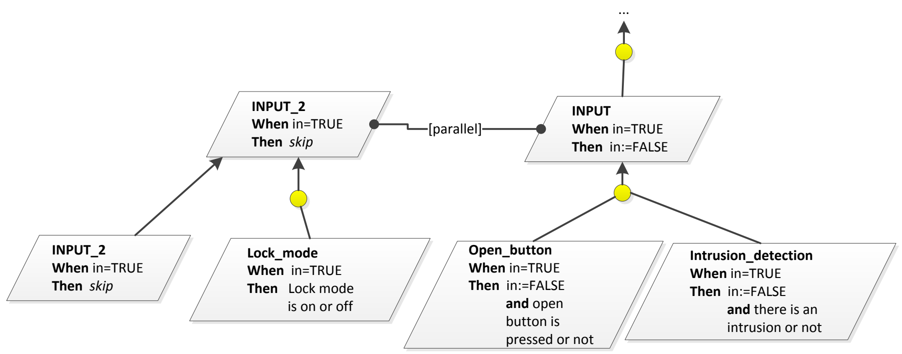


Figure 5.7: An example refinement tree diagram of event forking refinement pattern

5.3.5 Example

This example is about an electrical gate controller, which the inputs of the system are lock mode, open button, and intrusion detector. The lock mode is an input which does not immediately changes the status of the gate. While, if there is the change of status of the open button or the intrusion detector, the system must have some immediate response. The immediate response can be represented by the change of the phase of the system from input phase to another. The changes of the state of each input can be represented by non-deterministic actions. The monitoring of these inputs follows the event-forking refinement pattern by firstly introducing two events, $INPUT$ and $INPUT\_2$. $INPUT$ represents the concept that the phase of the system is changed immediately. On the other hand, $INPUT\_2$ does not change the phase. Both events have the same guard, so they can be executed in parallel.

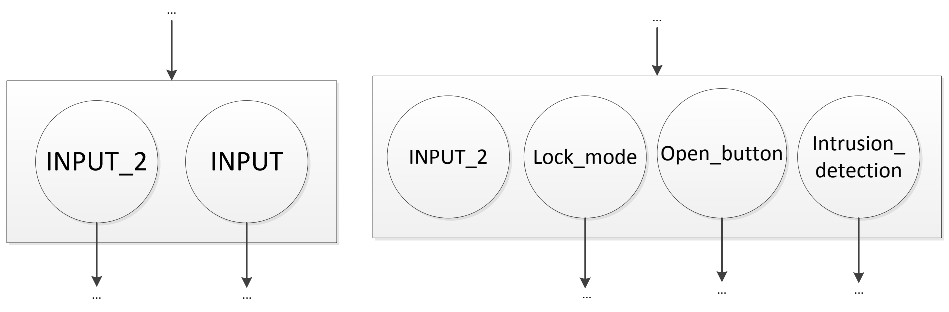
In the subsequent refinement, The event $Lock_mode$ refines $INPUT\_2$, since it changes only the lock mode, not the phase. While, the events $Open_button$ and $intrustion_detection$ refine $INPUT$, because they can change the phase. $INPUT\_2$ is also copied into this refinement for supporting possible new inputs introduced in some further refinement. All of the events have the same guard.

Figure 5.7 and Figure 5.8 show the refinement tree diagram and the event transition diagram of the above modelled system respectively.

The Event-B specification of the above system can be written as follows:

The abstract machine

$$INPUT\_2 \,\hat{=} \mathbf{when}\ in = TRUE\ \mathbf{end}$$
$$INPUT \,\hat{=} \mathbf{when}\ in = TRUE\ \mathbf{then}\ in := FALSE\ \mathbf{end}$$



(a) Upper level (b) Next level

Figure 5.8: General event forking refinement pattern: the event transition diagram

The concrete machine

$$\begin{aligned}
INPUT\_2 \,&\hat{=} \textbf{refines } INPUT\_2 \textbf{ when } in = TRUE \textbf{ end}\\
Lock\_mode \,&\hat{=} \textbf{refines } INPUT\_2 \textbf{ when } in = TRUE\\
&\quad \textbf{then } lock\_mode :\in BOOL \textbf{ end}\\
Open\_button \,&\hat{=} \textbf{refines } INPUT \textbf{ when } in = TRUE\\
&\quad \textbf{then } in := FALSE \ \wedge \ open\_button :\in BOOL \textbf{ end}\\
Intrusion\_detection \,&\hat{=} \textbf{refines } INPUT \textbf{ refines } INPUT \textbf{ when } in = TRUE\\
&\quad \textbf{then } in := FALSE \ \wedge \ intrusion :\in BOOL \textbf{ end}
\end{aligned}$$

5.3.6 Notes

In some case, there might be usage of parameters when performing the event-forking. Most of the time, the possible values of parameters are declared through the guards of events. This might cause the guards to be different in the parallelized events. However, the parameters cannot change the state of variables before executing the parameter-involved events. Thus, the parallelized events can still be executed in arbitrary manners. In conclusion, this pattern allows the guards of parallelized events to be different in the case that the differences come only from the parameters.

5.4 Case-decomposition refinement pattern

5.4.1 Description and applicability

This pattern is for refining an abstract into two or more concrete events for dealing with all possible cases of states of variables. One concrete event is supposed to deal with one case. This is to determine that which actions should be executed for each of the cases. Thus, this pattern is usually used in the decision phase or the output phase of a system. It can also be used in the input phase, if there are some restriction on inputs which needs to be determine case-by-case.

5.4.2 Illustration

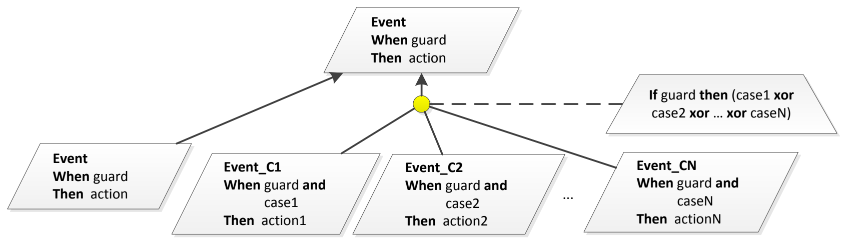


Figure 5.9: General case-decomposition refinement pattern: refinement tree diagram

Figure 5.9 shows the general refinement tree of the case-decomposition refinement pattern. In this tree, an abstract event is copied once and refined by N concrete events. All the concrete events have distinct guards, since each event's guard is in the form of conjunction of the guard from the abstract event and its distinct case. Their actions are also distinct for each case. There is an invariant linking to all the concrete events. This invariant is to ensure that the entire state space of cases are covered and the cases are disjoint. The disjointness is crucial if only deterministic actions are allowed for all cases. The copy of the abstract event is useful when too many cases are introduced in one level. The cases can be gradually introduced in two or more steps of refinements by gradually introducing cases to the copy.

5.4.3 Transformation to Event-B model

The transformation from this pattern into Event-B model is straightforward. The concrete events must explicitly refine the abstract event. The guard from the abstract event should be preserved, while each abstract event must be introduced with new guard representing each case. The actions of each event should also be modified to reflect the responses for each case.

Event-B specification does not have XOR operator. Since $AxorB = (A \vee B) \wedge \not(A \wedge B)$, the latter form can be used for writing the invariant.

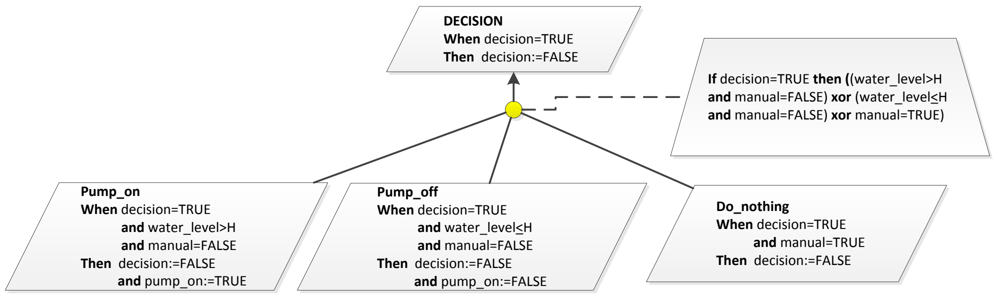
Names of the parallelized events are up to developers.

5.4.4 Constraint

The guard of each concrete should not be contradict with the guard of the abstract event, even though the proof obligation GRD can be discharged due to the inconsistency. The concrete actions must simulate the abstract actions due to the proof obligation SIM.

Because of the proof obligation EQL, ones should carefully apply this pattern, especially when the decomposition is done in more than one step of refinement. The state of variables appearing in the abstract machine must only be changed by the concrete events refining the abstract event which changes the state of the variables. The cases should be well-decomposed to get adequate responses (actions) for modifying the new variables of each refinement step.

5.4.5 Example



DECISION
When decision=TRUE
Then decision:=FALSE
If decision=TRUE then ((water_level>H and manual=FALSE) xor (water_level≤H and manual=FALSE) xor manual=TRUE)
Pump_on
When decision=TRUE and water_level>H and manual=FALSE
Then decision:=FALSE and pump_on:=TRUE
Pump_off
When decision=TRUE and water_level≤H and manual=FALSE
Then decision:=FALSE and pump_on:=FALSE
Do_nothing
When decision=TRUE and manual=TRUE
Then decision:=FALSE

Figure 5.10: An example case-decomposition refinement pattern

Figure 5.10 illustrates the application of the case-decomposition refinement pattern in a water tank system. This system has two inputs: water level and manual mode. If the system is in manual mode, the system will do nothing to let all the decisions are made by its users. If the system is not in the manual mode, when the water level is higher than H (a constant), the system must turn on the pump to decrease the water. Otherwise, the pump must be turned off.

The upper level of the refinement tree only shows abstract behavior that the system is in the decision phase. In the lower level, all cases are modelled by applying case-decomposition pattern. The event $Pump_on$ and $Pump_off$ turn on and turn off the pump respectively according to the water level when the system is not in the manual mode. The event $Do_nothing$ just transit to next phase to reflect that the system do nothing when it is in the manual mode. An invariant is written to ensure that all the cases are considered.

The lower level of the refinement tree can be transform into the following Event-B model:

Events

$Pump\_on$ $\hat{=}$**refines** $DECISION$
when $decision = TRUE \ \wedge \ water\_level > H \wedge \ manual = TRUE$
then $decision := FALSE \ \wedge pump\_on := TRUE$ **end**

$Pump\_off$ $\hat{=}$**refines** $DECISION$
when $decision = TRUE \ \wedge \ water\_level \leq H \wedge \ manual = TRUE$
then $decision := FALSE \ \wedge pump\_on := FALSE$ **end**

$Do\_nothing$ $\hat{=}$**refines** $DECISION$
when $decision = TRUE \ \wedge \ manual = FALSE$
then $decision := FALSE$ **end**

Invariant

$$\begin{aligned} decision = TRUE \Rightarrow (((water\_level > H \wedge manual = FALSE) \vee \\ (water\_level \leq H \wedge manual = FALSE) \vee manual = TRUE) \wedge \\ \neg((water\_level > H \wedge manual = FALSE) \wedge \\ (water\_level \leq H \wedge manual = FALSE) \wedge manual = TRUE)) \end{aligned}$$

5.4.6 Notes

When using more than one refinements to cover all the cases, if it is necessary to refine some concrete events' guard, the invariant as shown in the pattern might not be sufficient for checking the coverage and the disjointness of cases. In this case, the invariant should cover not only where the pattern is applied, but also every modified concrete event.

5.5 Milestone-driven refinement pattern

5.5.1 Description and applicability

This pattern is for decomposing an abstract event into two or more concrete events by introducing intermediate steps (milestones) between the guard and action of the abstract event. More precisely, this pattern transform an abstract event which executes its action directly after its guard holds to a sequence of events. In the sequence of events, after the abstract guard hold, an intermediate action is executed and triggers the next event which its action trigger another event until the abstract action is executed.

5.5.2 Illustration

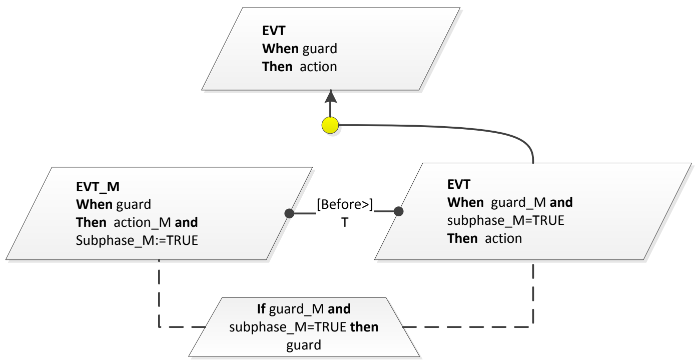
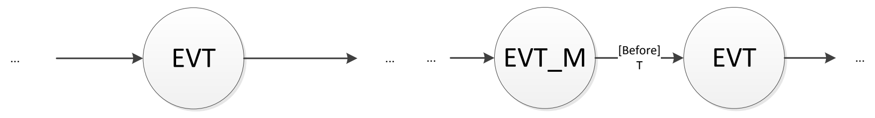


Figure 5.11: Simple milestone-driven refinement pattern: the refinement tree diagram

Figure 5.11 shows a simple form of the milestone-driven refinement pattern. In this simple form, the abstract event is decomposed into two concrete events. In the upper



(a) Upper level (b) Lower level

Figure 5.12: Simple milestone-driven refinement pattern: the event transition diagram

level, after the guard of the abstract event holds, the action is executed. On the other hand, after the same guard holds, an intermediate actions are executed in the event EVT_M of the lower level. Then, the actions trigger the event EVT which executes the same actions with the abstract event. Figure 5.12 illustrates the event transition diagram of the simple form of this pattern. The event EVT refines from the abstract event EVT with the modification of its guards to correspond with the results of the intermediate actions. An invariant is written in the lower level to make the modification of the guard possible in Event-B. The intermediate actions contains actions from the system, and an action representing a transition of sub-phase. This sub-phase is needed for the invariant. Finally, the last-step event must reset the sub-phase.

If there are an arbitrary number n of intermediate steps to be introduced, the general form of this pattern as in Figure 5.13 should be used instead. n sub-phases must be declared, and n invariants must be written. Note that only the event representing the last step refines from the abstract event. The last-step event resets all the sub-phases.

5.5.3 Transformation to Event-B model

For n intermediate steps to be introduced, an abstract machine must be decomposed into $n+1$ concrete events. n events are newly created, and only one event refines the abstract event. The newly created event representing the first step must have the same guard with the abstract event. While, the refining event must have the same action with the abstract event. The n sub-phases can be represented by n Boolean variables. After each sub-phase is passed, the variable of the sub-phase becomes true. The last-step event resets all the sub-phases by assigning false to all variables representing the sub-phases. Lastly, n invariants must be written in the concrete machine.

5.5.4 Constraint

When introducing an intermediate step, it means that new actions and new guards corresponding to the actions are introduced in between a pair of guards and actions. The invariant in the form appearing in this pattern is to ensure the correspondence between the new actions and guards. When using this pattern such correspondence must be taken in account. For examples, if a new action is $x := TRUE$ which assigns the truth value $TRUE$ to a Boolean variable x, the guard corresponding to this action is $x = TRUE$. If a action $y := y+1$ which increments the value of y by 1 is introduced into an event whose guard is $y = 0$, the new guard corresponding to the action is $y = 1$ or $y > 0$.

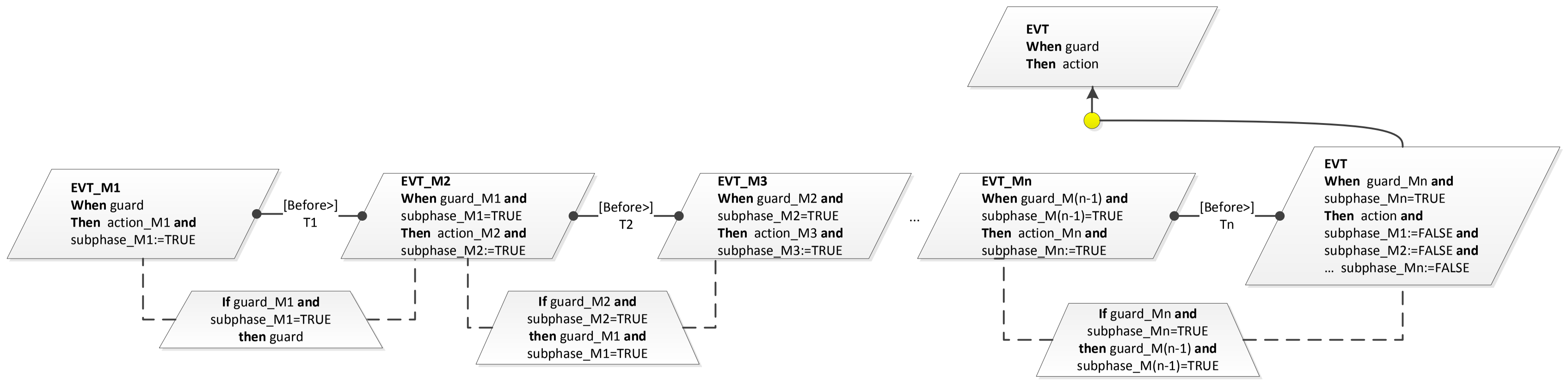


Figure 5.13: General milestone-driven refinement pattern: the refinement tree diagram

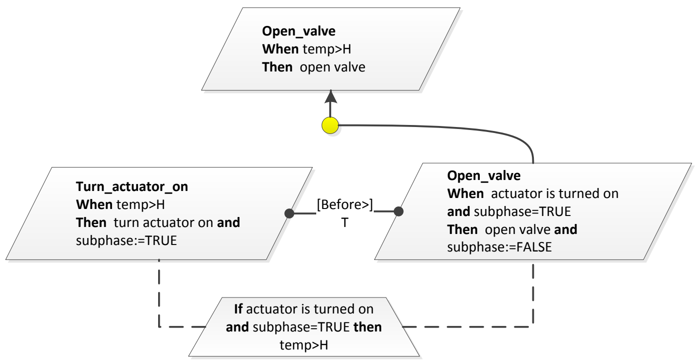


Figure 5.14: An example refinement tree diagram of milestone-driven refinement pattern

5.5.5 Example

Figure 5.14 demonstrates an example refinement tree diagram of this pattern. The example is about a temperature control valve. The valve will open, if the temperature is higher than a limit (H). The only abstract event in the tree represents such behavior. However, it is not possible to open the valve directly through the temperature. The temperature should turn an actuator on, and then, this actuator opens the valve. Thus, the abstract event is decomposed into two concrete events with one intermediate steps about the actuator is introduced. The event transition diagram of this example is shown in Figure 5.15.

This example can be modelled into Event-B as follows:

Abstract machine

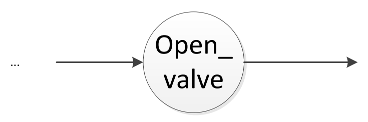
$$\begin{aligned} Open\_valve \triangleq\ & \textbf{when } temp > H \\ & \textbf{then } valve\_open := TRUE \textbf{ end} \end{aligned}$$

Concrete machine

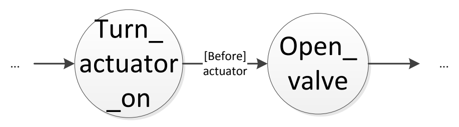
$$\begin{aligned} Pump\_on \triangleq\ & \textbf{when } temp > H \\ & \textbf{then } actuator\_on := TRUE \ \wedge\ subphase := TRUE \textbf{ end} \\ Open\_valve \triangleq\ & \textbf{refines } Open\_valve \\ & \textbf{when } actuator\_on = TRUE \ \wedge\ subphase = TRUE \\ & \textbf{then } valve\_open := TRUE \textbf{ end} \end{aligned}$$

Invariant of the concrete machine

$$actuator\_on = TRUE \wedge subphase = TRUE \Rightarrow temp > H$$



(a) Upper level



(b) Lower level

Figure 5.15: An example event transition diagram of milestone-driven refinement pattern

Chapter 6

Case study and evaluation

In Chapter 1, we stated that Event-B lacks of the requirements analysis and elaboration mechanism, and the guideline for using its refinement mechanism. Thus, we proposed an approach using the ORDER model along with the refinement patterns derived from KAOS for overcome these shortcomings of Event-B. This chapter presents the means to evaluate the proposed approach is capable to reduce the shortcomings and encourage the usage of Event-B in practical development of safety-critical software systems.

Since our objective of this research is about the practical usage of Event-B, the evaluation was done mainly through case studies. We applied our model in action on three examples derived from a real-world context. The examples varied on their size and types of systems in order to increase confidence in the utility of our approach. Then, some fact about our approach were discussed based on the results of the applications.

Our approach were applied on three case studies: a powered sliding door, an automatic gate controller, and Electrical Power Steering (EPS) system.

6.1 Powered sliding door

6.1.1 Overview

The first case study, the powered sliding door, was derived from part 10 of ISO 26262 [ISO11]. This case study was originally described in the standard as an example of decomposition of safety requirements for allocating them to corresponding architectural elements of a safety-critical system. This system is considerably a small example. Even after the decomposition, this system consisted of only 7 functional safety requirements. We applied our approach to this case study in order to ensure that it was feasible to use our approach to analyse, elaborate and model safety requirements.

The powered sliding door is a sliding door of a vehicle which a user can request the door to be opened or closed. The safety goal of the powered sliding door is “not to open the door while the vehicle speed is higher than 15 km/h”. Thus, this system operates based on inputs which are the vehicle speed and the user request. When the user requests the door to be opened, a Power Sliding Door Module (PSDM) drives the power to the door actuator to move the door. PSDM will allow the powering of the actuator only if the vehicle speed is below 15 km/h. Furthermore, there is a switch is on the power line between the PSDM and the door actuator. When the switch is off, the power line can drive the door actuator. However, the switch will be off only if the vehicle speed is below 15 km/h. Both the PSDM and the switch are the mechanisms for preventing the door to be opened while the vehicle speed is higher than 15 km/h.

After applying the ORDER model and the refinement patterns, it resulted in 4 Event-B machines (an initial model and 3 refinement steps). The refinement tree diagrams, the event transition diagrams, and the Event-B specifications of this system from our approach can be found in Appendix A. The explanations for each level of the tree are the followings.

6.1.2 The first level

Firstly, we separated the system into two phases, input phase and decision phase, and used a variable *input* to represent the transition between two phases. The input phase contains two parallelized events: one can transit to the next phase, another cannot.

6.1.3 The second level

At this step, the speed of the vehicle was introduced. The speed must be below 15 km/h, if the door is opened. Otherwise, the speed can be changed freely within the speed limit of the vehicle. Because there are two cases here, the case-decomposition refinement pattern was applied. The only input at this step is the speed, so we assumed that the system arbitrarily transits to the decision phase. For the decision phase, there were two cases of input: the speed is higher than 15 km/h and the speed is below or equalled to 15 km/h. One necessary invariant of this system is that the door can be opened only when the speed is below 15 km/h. If the speed is higher than 15 km/h, the door must be closed immediately. The other case does not decide exactly the door must be opened or closed, because the input is not sufficient for the moment.

6.1.4 The third level

Thirdly, the switch was added. The switch is for realize the process of controlling the door in the decision phase. Both cases of the decision phase were applied by the milestone-driven refinement pattern for the switch. If the speed is higher than 15 km/h, the switch will be off, and then, the door is closed. Otherwise, the switch is on, but the input is still insufficient to determine the status of the door. The change of the speed was more concrete in this step by dividing each case of the change into increasing and decreasing the speed by applying the event-forking refinement pattern.

6.1.5 The fourth level

Finally, the request from the user was added to be another input of the system. This input is capable to trigger the decision phase of the system. The case-decomposition refinement pattern was applied to the decision phase where the switch is on. If there is no request, the door stays in the same phase. Otherwise, the door will be closed, if it is opened. The door will be opened, if it is closed. Note that we assumed that the request is for toggling the status of the door.

6.2 Automatic gate controller

6.2.1 Overview

The automatic gate controller was derived from the complete example of the evolutionary framework [ZG03]. The goal of this system was to allow only authorized persons to enter a building through the automatic gate. The goal was evolved by applying the evolutionary framework. During the evolution, the completeness and the consistency of each step were logically proved. Finally, the final version which was a specification for implementing the system was correctly constructed. We proved in Chapter 3 that, in principle, Event-B refinement can preserve the correctness as defined in the evolutionary framework. We tried to apply the ORDER model to this case study by closely following the evolution steps presented in the example of the evolutionary framework. This was for showing that this proved fact is also valid in application. Furthermore, it showed the coherence of the evolutionary framework and the ORDER model.

This example is about an automatic gate, which is for regulating access to the building in which the gate is installed. The gate must be able to allow authorized persons to enter the building, and prevent and non-authorized ones. An authorized person for this system means a person who possesses an authorized ID card. One way to check the possession of the authorized ID card is by asking a person to swipe his/her ID card through a card reader and check whether the ID card is an authorized one. After the door is opened for 5 seconds, the system sends a “lock command” to the gate. If after 10 more seconds, the gate sensor still reports “gate open”, the system sounds the warning alarm, until the sensor reports “gate closed”.

The resulted refinement tree diagram of this case study had 5 levels, which leaded to 5 Event-B machines (presented in Appendix B). The explanations for each level of the tree are the followings.

6.2.2 The first level

For the first level, we applied the phase-decomposition pattern to this system and divided the system into two phases, input phase and decision phase. The variable *input* was used for representing the input phase. The input phase contains two parallelized events: one can transit to the next phase, another cannot.

6.2.3 The second level

Secondly, the concept of allowing only the persons who have an authorized card to enter the gate was introduced. The inputs were the possession of a card and the id of the card (uid). The possession of the card does not change the phase right away. If a person has a card, the uid will be retrieved from the card, before transiting to the decision phase. On the other hand, if a person does not have a card, the system will transit to the decision phase right away. The event forking pattern can be applied for checking the possession of a card. Then, the case-decomposition pattern was applied based on the two cases, possessing a card or not, before transiting to the next phase.

The decision phase was applied by the case-decomposition pattern to consider each case of the inputs. The first case is that a person has an authorized card. In this case, the system allows the person to enter. The second case is that a person does not have a card, or the person has an unauthorized card. The system does not allow the person in the latter case to enter.

There were inputs that were not introduced yet. Therefore, there are two copies of events for the input phase and the decision phase for supporting further variables and cases.

6.2.4 The third level

Thirdly, the possession of a card can be checked through the use of a card reader. If a person swipes a card through this card reader, it means that the person has the card. The retrieving of the uid can be realized by introducing a buffer for temporarily storing the uid obtained from the card reader. The concepts of allowing and preventing a person from entering the building can be realized by sending command to 'open' and 'lock' a gate respectively. This step just replaced the abstract concepts with the concrete concepts. No pattern was applied.

6.2.5 The fourth level

At this step, two new inputs were added: time and sensor. The time is for representing how long the gate has been opened. The sensor is for checking the current status of the gate. The sensor is needed because after sending the command, there are many possibility that the status of the gate does not correspond to the command. Thus, the sensor was introduced as a new input acting independently.

The decision phase had to be refined for corresponding the new inputs. If the status of the gate is 'opened' and an authorized card is swiped, then the system sends an 'open' command to the gate and starts the timer. If the 'open' command has been sent for 5 seconds and no authorized card is swiped, then the system sends 'lock' command to the gate. If no authorized card is swiped and the 'open' command has been sent for less than 5 seconds, the system just transits back to the input phase. Lastly, if no authorized card is swiped and the gate is already locked, the system just restarts the timer.

6.2.6 The fifth level

In the last step, we focused on the concept of the alarm. Previously, we defined that if the 'open' command has been sent for 5 seconds and no authorized card is swiped, then the system sends 'lock' command to the gate. However, this time, one more case was added: if the command is sent for 15 seconds and no authorized card is swiped, then the system sends 'lock' command to the gate and the alarm must be turned on. The alarm will be turned off, when an authorized card is swiped or the gate is already locked.

6.3 Electrical Power Steering (EPS) system

6.3.1 Overview

This third case study was developed in collaboration with the Department of Green Mobility Research of Hitachi, Ltd., Hitachi Research Laboratory. The EPS system is a safety-critical system controlling the electric steering of cars. The main motivation was to model the safety requirements of the EPS system by using the KAOS goal model. The goal model was truly effective in elaborating the safety requirements, since we discovered that there were a lot of missing requirements and assumptions in the original requirements through the usage of the model. Our idea in applying the ORDER model to this case study was that it might be easy to create the ORDER model following the created goal model. This is potential possible from the facts that the ORDER model was derived from the goal model. The case study offered evidence for the capability to use our approach in the practical development of safety-critical systems. We focused on just a part of the while safety requirements of the EPS system. the focused part contained 48 functional requirements.

The part of the EPS system which was used in this case study is the part regarding the transition to a manual steering mode. This mode is to stops the EPS system when a failure of the system is detected, and then, let the driver manually control the steering of the car. This system is operated through various components, such as a diagnostic function module and a Current Control Unit (CCU). The diagnostic function module monitors the voltage supplied to the CCU. The CCU is composed of a pre-driver and an inverter. If the failure of the voltage applied to the pre-driver or the inverter is detected, it also means that the voltage applied to the CCU is failed. After the detection of the failures, the fact is notified to Fail-Safe Action Function module. Then, the Fail-Safe Action Function module cuts the power supply to the motor. In order to cut the power supply, the pre-driver, the motor relay, and the fail-safe relay of the system must be stopped. This state is the manual steering mode of the system.

Since the EPS system is comparatively large with respect to other case studies, the resulted refinement tree was considerably large. It had 8 levels to be modelled in 8 Event-B machines. The refinement tree diagram and the Event-B specification of this case study is presented in Appendix C. The explanations for each level of the tree are the followings.

6.3.2 The first level

Firstly, we separated the system into two phases, input phase and decision phase, and used a variable *input* to represent the transition between two phases.

6.3.3 The second level

Secondly, the failure of voltage supplied to CCU was added as the input of the system. There are two possible cases from this input: the failure is detected or not. In the decision phase, if the failure is detected, the system will make a transition to the 'Manual Steering' mode. Otherwise, the system will stay in the normal mode. To support these two cases, the case-decomposition refinement pattern was applied to the decision phase.

6.3.4 The third level

Thirdly, this step introduced the concept of the pre-driver and the inverter which are the parts of the CCU. If the failure of the voltage applied to one of the two components is detected, it equals to the detection of the failure of the voltage applied to CCU. If no failure of the voltage applied to both components is detected, then there is no failure at all. Because the detection of both failures can be done in parallel, the event-forking refinement pattern was applied to the model. The decision phase was slightly changed from using the failure of the voltage applied to CCU for making a decision to using the failure of the voltage applied to one of the two components for making a decision.

6.3.5 The fourth level

The behavior of directly making a transition to the 'Manual Steering' mode after detecting a failure was too abstract for the practical system. In facts, after the detection of a failure, a demand for transition to 'Manual Steering' mode must be sent and it must be sent without failure. Then, when the demand is received, the system will make the transition to 'Manual Steering' mode. This means the abstract behavior have to be decomposed into three steps, so the milestone-driven refinement pattern was applied here. On the other hand, if there is no failure, the demand will not be sent. Thus, the demand is not received. This finally leads to the system stays in the normal mode. Here, the milestone-driven refinement pattern was also applied.

6.3.6 The fifth level

At this step, we focused on the concrete behavior of the transition to 'Manual Steering' mode. The 'Manual Steering' mode means that the motors of the EPS system stops working. In order to stop the motor, the power supply which actuates the motor must be cut off. The normal mode is still the negation of the 'Manual Steering' mode. If the power supply actuates the motor, the motor will work, and the EPS system will be in the normal mode. Both of the sequence of behavior can be realized through the use of the milestone-driven refinement pattern. Considering only the transition to 'Manual Steering' mode, after the demand for transition to the 'Manual Steering' mode is received, the system will cut off the power supply. This causes the motor to stop. Finally, the system is in the 'Manual Steering' mode.

6.3.7 The sixth level

To stop the power supply, the system have to stop the pre-driver, or the motor relay, or the fail-safe relay. Stopping just one component can stop the power supply. However, this system is supposed to stop all the three components in order to guarantee that the power supply will stop working. Such ideas can be introduce into Event-B by creating 3 parallelized events, in which each event is for stopping each component after receiving the demand for transition to the 'Manual Steering' mode. Here, all of them can trigger the event representing stopping the power supply. This is a special case where two patterns, the milestone-driven pattern and the event-forking pattern, were applied together.

Again, to let the power supply continue working, all the three components must also continue working. This can be described in Event-B by applying the milestone-driven refinement pattern to create an event for describing that all of the three component work when the demand is not received. Then, this makes the power supply continues working.

6.3.8 The seventh level

This step added the concepts of sending a stop signal to stop the pre-driver, an open circuit demand to stop the motor relay, and an open circuit demand to stop the fail-safe relay. Since these concepts are similar, considering only one concept is sufficient. Considering the pre-driver, after the demand for transition to the 'Manual Steering' mode is received, the stop demand will be sent without failure. Because the stop demand is sent without failure, the demand will be eventually received. Finally, the pre-driver stops working due to the stop demand. The milestone-driven refinement pattern was applied again for describing this concept. The pattern was also applied in a similar way for sending the open circuit demands to stop the motor relay and the fail-safe relay.

For the case of letting the three components continue working, considering only the pre-driver again, the stop demand will not be sent, when the demand for transition to the 'Manual Steering' mode is not received. Consequently, the stop demand will not be received. This causes the pre-driver to continue working. The similar concepts can be applied to the motor relay and the fail-safe relay. However, this time, all of them must continue working to make the power supply to continue working. Therefore, these concepts were described together in a sequence of events from the application of the milestone-driven refinement pattern.

Table 6.1: Number of events according to sources of creation: the powered sliding door

Source of event	Initial model	First refinement	Second refinement	Third refinement	Total
Manual	0	0	0	0	0 (0 %)
Patterns	3	4	8	4	19 (100 %)
Total	3	4	8	4	19

6.3.9 The eighth level

The idea of this step is that the sending of the stop demand and open circuit demand to stop the pre-driver, the motor relay, and the fail-safe relay is not suitable in implementation. Actually, it is more appropriate to send enable signals to the three components for commanding them to work. When the system has to stop the three components, the system stops sending the enable signals. This step is not presented in Appendix C, because the ideas presented in this step are only the replacements of variables.

6.4 Result

Table 6.1 6.3 show the number of events in the Event-B specification of the powered sliding door case study, the automatic gate controller case study, and the EPS system case study respectively. These numbers are categorized according to the source of events and the step of refinements. Actually, there are three sources of event: patterns, copy, and manual. 'Patterns' means that the events were derived from the refinement patterns of the ORDER model. 'Copy' means that the events just copy events from the abstract Event-B model. Lastly, 'manual' means all the other possible ways to come up with events. The tables do not present the copying events due to the irrelevance of them with respect to our discussion in the next section.

Here, we assume that the relative size of the three case studies can be roughly measured by comparing the number of events of those case study. According to the tables, we can infer that the size of the powered sliding door case study and the automatic gate controller case study are almost the same. While, the EPS system case study is a lot larger than the first two case studies.

Another fact worth mentioning here is the number of events derived from the refinement patterns. In the case of the powered sliding door, Table 6.1 indicates that all the created events in all steps of refinement are derived from the refinement patterns. On the other hand, around 70 % of the total number of events are derived from the patterns for the automatic gate controller case study and the EPS system case study.

Table 6.2: Number of events according to sources of creation: the automatic gate controller

Source of event	Initial model	First refinement	Second refinement	Third refinement	Fourth refinement	Total
Manual	0	0	5	1	2	8 (34.8 %)
Patterns	3	5	0	5	2	15 (65.2 %)
Total	3	5	5	5	4	23

Table 6.3: Number of events according to sources of creation: the automatic gate controller

Source of event	Initial model	First refinement	Second refinement	Third refinement	Fourth refinement	Fifth refinement	Sixth refinement	Seventh refinement	Total
Manual	0	0	5	0	0	1	0	12	18 (32.7 %)
Patterns	2	3	0	6	6	6	14	0	37 (67.3 %)
Total	2	3	5	6	6	7	14	12	55

6.5 Discussion

As we presented the results from applying our approach to modelling three case studies. Considering the applications and the results, we can discuss and infer some valuable fact regarding our approach as the following.

6.5.1 Coverage of the patterns

From the three case studies, around 70 % and more of the events in the Event-B specifications were easily derived from our proposed refinement patterns. This can imply that the coverage of the refinement patterns for modelling the safety requirements specifications is high. If we exclude all the events derived from the patterns from the Event-B model of the case studies, most of the manually refining events refines the abstract events by replacing certain abstract variables with concrete variables. The replacements are possible through the usage of the gluing invariants [C$^+$05]. This kind of refinement are regarded as vertical or structural refinement for enriching the structure of a model to bring it closer to an implementation structure [DB09]. The vertical refinement is a part of design issues of a system. The design issues are beyond the scope of this thesis, therefore, we can conclude that the patterns are sufficient for modeling the safety requirements specifications in Event-B.

6.5.2 Scalability of the patterns

Here, scalability means whether the refinement patterns decrease the applicability when the requirements specification is larger. As stated before, the size of the powered sliding door case study and the automatic gate controller case study are almost the same. However, the percentage of the events derived from the patterns decreases a lot for the case of the automatic gate controller. Furthermore, even though the EPS system case study's size is around two times larger than the automatic gate controller case study's, the percentage of the patterns-related events are nearly equalled to each other. As previously discussed, the patterns do not support the vertical refinements. Thus, the applicability of the patterns depends more on the type of the refinement, not the size of the specifications. We conclude that the patterns are scalable.

6.5.3 Preservation of the correctness

We modelled the automatic gate controller in Event-B in a way that followed how the requirements of this system were evolved in the evolutionary framework. One error of the requirements found by the framework was that the lacks of the statement stating that "a person is presumed not to have a card unless he or she swipes it in the card reader" caused the requirements to be incomplete. This error can also be found by the proof obligations of Event-B. This was an encouraging evident aside from our analysis in Chapter 3 that Event-B and its refinement mechanism can preserve the correctness as defined in the evolutionary framework.

6.5.4 Avoidance of Event-B deadlock

Establishing the absence of deadlocks is important in many applications of formal methods [HL11], including the safety-critical systems. One common way to deal with the deadlock in Event-B is by explicitly writing a deadlock-freeness theorem in an Event-B machine. The deadlock-freeness theorem states that one the guards $G_1, G_2, \ldots, G_n$ of the events, exception $INITIALISATION$, is always true [YJ+11]. The theorem can be written in the form of an theorem in Event-B as follows:

$$G_1 \vee G_2 \vee \ldots \vee G_n$$

Our refinement patterns can also avoid the deadlock of Event-B. This is because the constraints and rules of each pattern which are capable to avoid the deadlock are already described. To illustrate, the initial model of the powered sliding door case study contains two phases. These phases can be passed through and they can trigger each other. Therefore, the initial model is deadlock-freeness. In the first refinement, there are two events for dealing with two ranges of the vehicle speed. The invariant generated from the case-decomposition pattern can ensure that the two cases cover all possible inputs. Then, in the second refinement, the milestone-driven refinement pattern was applied twice to add the intermediate steps regarding the switch. Two invariants were generated from the pattern can ensure that the replacements of the guards are valid. This preserve the coverage of the cases from the abstract machine. Thus, we can conclude that the second and the third refinement are also deadlock-freeness.

6.5.5 Benefits of the phase-based approach

One of the benefit of the phase-based patterns is to avoid the deadlock in Event-B as stated before. Another related benefit is that the phase-based approach allows grouping and sequencing the operations of a system, which is similar to the way human thinks of the behavior of the system. Consequently, an Event-B specifications, which are modelled in the phase-based way, tend to be easier to analyse and understand. It is also easy to trace the origin of each event because they are grouped according to their phase. By this way, it is easier to analyse and check the safety of a safety requirements specification in Event-B. To illustrate, in the case study of the automatic gate controller, we can easily understand and trace from the phase-based refinement tree diagram that the inputs of the system are the possession of a card, the uid of the card, time, and sensor. Then, the command sending to the gate, the timer and the alarm are operated based on the combinations of the inputs.

6.5.6 Overcoming the shortcomings of Event-B

The refinement patterns were created for the ORDER model. Certainly, the refinement tree diagram and the event transition diagram of the ORDER model fully support the patterns. In our experience applying the patterns to the case studies especially the EPS system, we found that it is possible and easy to analyse and elaborate safety requirements through the refinement patterns in the KAOS-like way in the form of the refinement tree. The concept of how to structurally refine an Event-B model was also provided by the patterns. Thus, in our point of view, the approach using the ORDER model together with the KAOS-based patterns is pertinent to what this research try to achieve, which is to overcome the shortcomings of Event-B.

6.6 Summary

In this chapter, we presented three safety-related case studies which we applied the ORDER model to analyse and model them. From the results of the case studies, we discussed that our approach to use the ORDER model and the KAOS-based patterns are effective and sufficient for overcoming the shortcomings of Event-B which is the goal of this research. We also discussed that some other property relevant to the safety, such as the deadlock-freeness and the traceability of a model, can be dealt with by our approach. This confirms that the formal model gained from our approach using KAOS and Event-B represents a correct safety requirements specification.

Chapter 7

Related work

To discuss the novelty of our approach, this chapter will discuss and compare the proposed approach with respect to the existing works related to our research. The related works are grouped and discussed according to the relevant aspects of our research.

7.1 Correctness, completeness and consistency of requirements specification

Because specifying requirements is one of the most critical activities in any software development effort. There are many approaches that aim to verifying correctness, completeness and consistency of the requirement specification. One popular formal method for the verification is the model checking method [CGP99]. Another popular formal method is the theorem proving method [Bib87]. Event-B, the core of our approach, is regarded as a theorem proving one. The discussion of pros and cons of each approach can be found in [OL07].

The examples of the works applying the model checking method for the verification of the completeness and consistency are [PMPS01] and [HL96]. The former is the work of Pap et al. to use the model checking for verifying the completeness and consistency of requirements on UML statechart specifications. The latter is the work of Heimdahl et al. for verifying state-based requirements. The definitions of the completeness in both approaches are defined as a response is specified for every possible input sequence. Another work related to the model checking method is the SCR method [HBGL95]. However, the SCR also support applying the theorem proving technique. The SCR presents requirements in a tabular form for specifying outputs based on the inputs of the system. The completeness of SCR is that all combinations of the inputs can be mapped to one of the outputs. The definitions of the consistency from all the three approaches are similar to the consistency of the evolutionary framework. Even though our approach uses the relative kind of completeness, our approach can cope with the completeness defined in these three approaches by using the case-decomposition refinement pattern.

Aside from the model checking method and the theorem proving method, another popular approach for checking the correctness is the knowledge-based approach. The knowledge-based approach checks the correctness by given a set of knowledge about the completeness and consistency of a system to a verifier. Then, users can input a requirements specification to the verifier, and interactively checks the completeness and consistency based on the knowledge of the verifier. [SP96] and [KKK95] are the works applying the knowledge-based to for checking the correctness. The correctness of the knowledge-based approach is defined depending on the knowledge given to the verifier. Thus, the approach can be applied only to the specification compatible to the knowledge. Our approach checks the correctness in a more general way.

Note that none of the approaches discussed in this section takes the evolution and refinement of requirements in account.

7.2 Verification of requirements in requirements evolution

A formal approach supporting verification of the requirements evolution is proposed in [GDPALN$^+$09]. Their approach aims to use analysis-revision cycle, in which the analysis phase verifies the correctness of requirements, and the revision phrase modifies the requirements according to the problems detected by the analysis, in the requirements evolution. The purpose of the cycle is to support evolution of requirements while preserving the main requirements goals and properties, and reasoning about the evolution. Their approach is similar to ours. However, their approach applies a model checking approach for the analysis phase. Besides, their approach is proposed for requirements specification in general, so it might lack of the consideration of safety-related properties, which are considered in our approach.

7.3 Event-B patterns

We defined the refinement patterns of the ORDER model in Chapter 5 for assist the Event-B modeling. In [Für09], Furst et al. proposed that it is possible to create design patterns of Event-B. They proved this fact by defining some pattern for the specifications of a communication protocols. These patterns were proved to be correct already by the proof obligations. Then, they defined the process for matching pattern with a problem, and replacing the patterns' variables with the problem's variables. Since the patterns were already correct, this means that the parts of the problem which are derived from the patterns are also correct. This work shows that it is possible to define patterns for Event-B modeling. Hence, this work supports the creation of our refinement patterns.

7.4 Diagrams supporting Event-B modeling

The proposed ORDER model consists of two diagrams for graphically supporting Event-B modeling: the refinement tree diagram and the event transition diagram. However, our

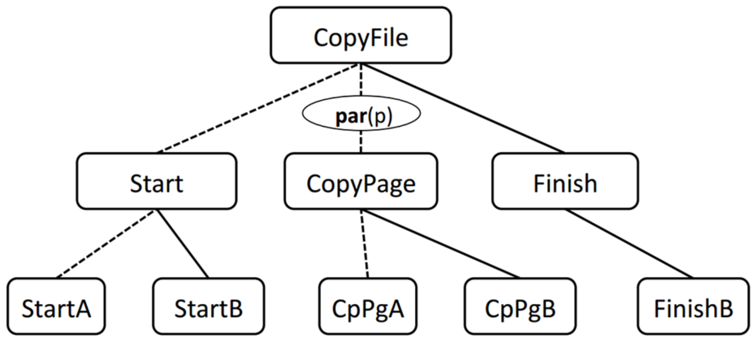


Figure 7.1: Event refinement diagram

approach is not the first graphical approach for Event-B. There is a diagram named 'event refinement diagram' proposed in [But09], which is similar to our refinement tree diagram. This diagram is based on the JSD structure diagram by Jackson [Jon90]. Figure 7.1 illustrates an example of the event refinement diagram. The root of the diagram represents an abstract event. The diagram shows how the root is decomposed into sequential events which are read from left to right. The oval with the keyword **par** represents a quantifier that replicates the tree below it. The solid line indicates the *refines* relationship, and the dashed line indicates that the events are newly introduced. The transformation into Event-B specification is not yet defined for the event refinement diagram. This diagram only roughly shows the way to decompose an abstract event into two or more sequential steps similar to the milestone-driven refinement pattern of the ORDER model.

The UML-B state machine diagram in Figure 7.2 indicates the transition of the state of an Event-B machine similar the event transition diagram of the ORDER model. Nonetheless, nodes and links of the state machine diagram represent the components of Event-B in an opposite way with respect to the event transition diagram. A node in the state machine diagram represents a state of variables in the machine, while an arrow represents an event causing the variables to change their state. But, the state machine diagram lacks of the notation of parallelized events using in the ORDER model, and it is more difficult to define that in the form of the state machine diagram. Therefore, the event transition diagram is more suitable to use with the ORDER model.

7.5 Guideline for using Event-B refinement

In 2012, Kobayashi and Honiden [KH12] proposed an approach to plan what models are constructed in each abstraction level of Event-B. The advantage of this approach is that it can calculate how well a plan can mitigate the complexity of a specification by considering the semantics constraints of Event-B and the relationships between elements in a system. This calculations is useful for selecting a relatively good plan from a set

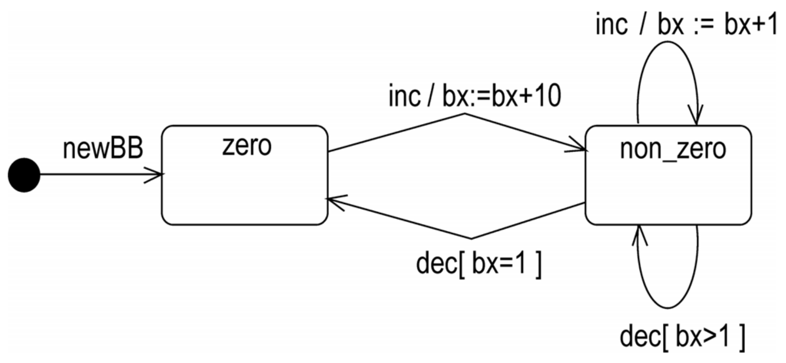


Figure 7.2: UML-B state machine diagram

of plans. One of our concern is that their approach can be used only after all details of a system, such as behavior, are already identified. Although, our approach lacks of such calculation, it integrates the requirements analysis and elaboration which are able to identify the necessary details, and it provides the guideline for Event-B refinement.

7.6 Phase-based approach for Event-B modeling

To the best of our knowledge, there is no work that explicitly propose the phase-based approach for requirements analysis. However, the Event-B specifications, which are written with the assistance of variables representing phases of a system, can be found in [SAHZ11, ASZ12, SAZ12], especially the latter two works which explicitly mentioned in one of their example that a variable is used for representing phases. This infers that the phase-based approach is usual for Event-B.

7.7 KAOS and Event-B

There are many works trying to use the capabilities of requirements analysis and elaboration of the KAOS method before modeling the requirements in Event-B. In [AAB$^+$09, PD11], the goal model of KAOS is used for analysis and elaboration of requirements. From the goal model, an Event-B model is generated only from the leaf goals. This is because [AAB$^+$09] aims to use Event-B to represent the operation model of KAOS, which can be derived only from the leaf goals. While, [PD11] focuses on the modularization of the Event-B specification through the use of agents, which can identified from the leaf goals.

The work which is the closest to our work is [MGL11]. This work defines a way to represent and prove the KAOS goal refinement patterns in Event-B. The difference to our approach is that they directly transform the KAOS goal model into Event-B specifications. To support the linear temporal logic of KAOS, they proposed a set of proof obligations that are necessary to ensure the completeness of the goal model. We discussed in Chapter 4 that, in our opinion, it is not appropriate to directly translate the goal model into Event-B specifications. We rather proposed the ORDER model based on the Event-B and use the KAOS goal refinement patterns for assisting the construction of the ORDER model.

7.8 Semantics of Event-B refinement

Our analysis in Chapter 3 gave the definition of the property preserved by the Event-B refinement, which is the correctness of requirement specification. The analysis can be regarded as giving the axiomatic semantics to the Event-B refinement. In contrast to our axiomatic semantics, Schneider et al. in [**?**] discuss about the behavioural semantics of Event-B through CSP semantics [**?**]. The behavioural semantics explain a relationship among events in a refinement chain. The main difference between their work and ours is that their work focuses on the meaning of the refinement chains of events, while our work focuses on properties that are preserved through the refinement chain of Event-B machines (which contain not only events).

Chapter 8

Conclusions and future work

8.1 Conclusion

The correctness of the safety requirements specification is crucial for the development of safety-critical systems. This is because the incorrectness of the specifications is the major causes of failures in the safety-critical systems. Event-B, a formal specification language, has a high potential in dealing with the correctness due to its well-known refinement mechanism, well-defined proof obligations, and the Rodin platform. However, Event-B lacks the definition of the correctness of requirements specification, mechanism for requirements analysis and elaboration, and the guideline for planning and using the refinement mechanism. These shortcomings of Event-B are hindrances for the usage of Event-B in the practical system development.

This thesis aims to overcome the shortcomings in order to encourage using formal methods like Event-B in the practical development of the safety-critical systems. To achieve this goal, our proposed approach applied the definition of the correctness defined in the evolutionary framework by Zowghi and Gervasi, and the capabilities to requirements analysis and elaboration of the KAOS method to Event-B. We first analysed that, in principle, the refinement mechanism of Event-B can preserve the correctness as defined in the evolutionary framework proposed by Zowghi and Gervasi. This explained the semantics of the properties preserved through the generated proof obligations. Then, we proposed the ORDER model based on the KAOS method to assist structuring and understanding Event-B models. The ORDER model is composed of two diagrams, the refinement tree diagram and the event transition diagram. These diagrams can be transformed into Event-B specifications by following our defined rules. The ORDER model inherits the capabilities of KAOS through the refinement patterns adapted from the KAOS goal refinement patterns. The correctness of the evolutionary can also be explained in the ORDER model through Theorem 6 from Section 4.3.4.

We evaluated our approach through the three case studies from a real-world context. These case studies showed us that the patterns we created can cover around 70 % of the refinement in the case studies, and they are scalable. Some essential property for the safety requirements specifications, such as deadlock freeness and traceability can be dealt with by our approach as well. Seeing that the ORDER model fully supported the patterns and can be dealt with the safety-related properties, we concluded that our approach using the ORDER model along with its patterns is pertinent to the research goal, which is to overcome the shortcomings of Event-B in the field of the development of the safety-critical systems.

In conclusions, we have proposed an approach to use the KAOS method for analysing and elaborating a safety requirement specification, and providing the guideline for Event-B refinement. Then, Event-B is used for modeling and verifying the specification. Through the usage of KAOS and Event-B, we can get a formal model representing correct safety requirements specification.

8.2 Future works

8.2.1 Automated tool for transforming the ORDER model to Event-B

At the current state of this research, after all the diagrams of the ORDER model are drawn, we have to manually transform the diagrams to Event-B. This process is simple in the case of small systems. But, this process becomes more tedious and needs more effort, if the systems become larger. This is trivial that it is better to have an automated tool to transform the diagrams to Event-B.

This idea is possible, since the Rodin platform is constructed on top of Eclipse [Ecl07], an open development platform. The Rodin platform allows creating plug-ins to support the Event-B modeling. UML-B [SBS09] is one of the plug-ins in Event-B supporting graphical approach. One direction might to create a plug-in for the ORDER model in Rodin platform from scratch. Another better direction is by extending the existing plug-ins which are related to graphical modeling.

8.2.2 Extension of the ORDER model

Even though the ORDER model is defined in a way that follows the Event-B specifications, not all the components of the Event-B can be supported by the ORDER model. One example is that Event-B has three kinds of events: *ordinary*, *convergent*, and *anticipated*. The events used throughout this thesis are ordinary ones. The convergent and anticipated events are the events that are needed to take the termination of events in account. This is to ensure the events cannot keep control of a machine forever. To ensure that, the machine must have *variants* which are mathematical expressions for guaranteeing that the events can be terminated through the proof obligation named VAR. These kinds of event and the variants are also crucial for the safety requirements specification. The ORDER model should support these notations of Event-B as well.

As discussed in Chapter 4, we assumed that before using the ORDER model, all the variables, carrier sets, and constants are already identified. The current ORDER model does not support the identification of these components. We suggested that it is possible to use other approaches which contain the notions of class diagram to help the identification. Thus, one possible further work is to extend the ORDER model with the class-diagram-like diagram.

8.2.3 Modularization

If a model becomes too large, it is difficult to manage and comprehend. This is also true for the ORDER model. Usually, most models with the problems should be able to be modularized. For examples, UML [SB06] has the notions of packages to group the classes in a class diagram and communicate with other packages through some interface. Event-B also contains the notions of 'Event-B decomposition' to decompose a single machine into several sub-machines that can be still be seen as one machine. To modularize the ORDER model, the Event-B decomposition is one candidate to be applied in the ORDER model.

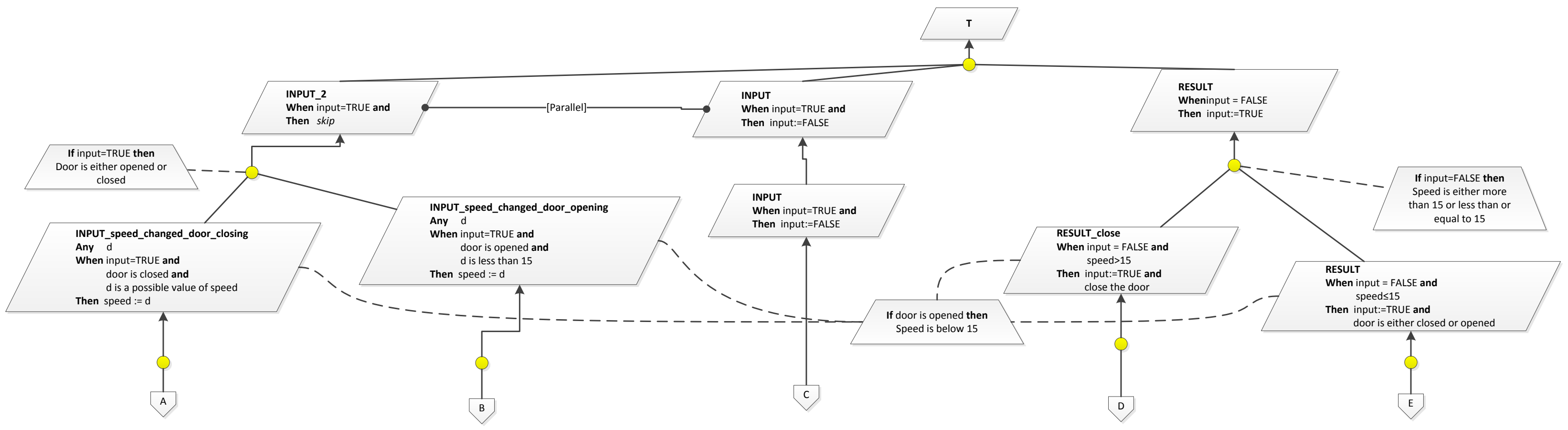
8.2.4 Formalization of the ORDER model

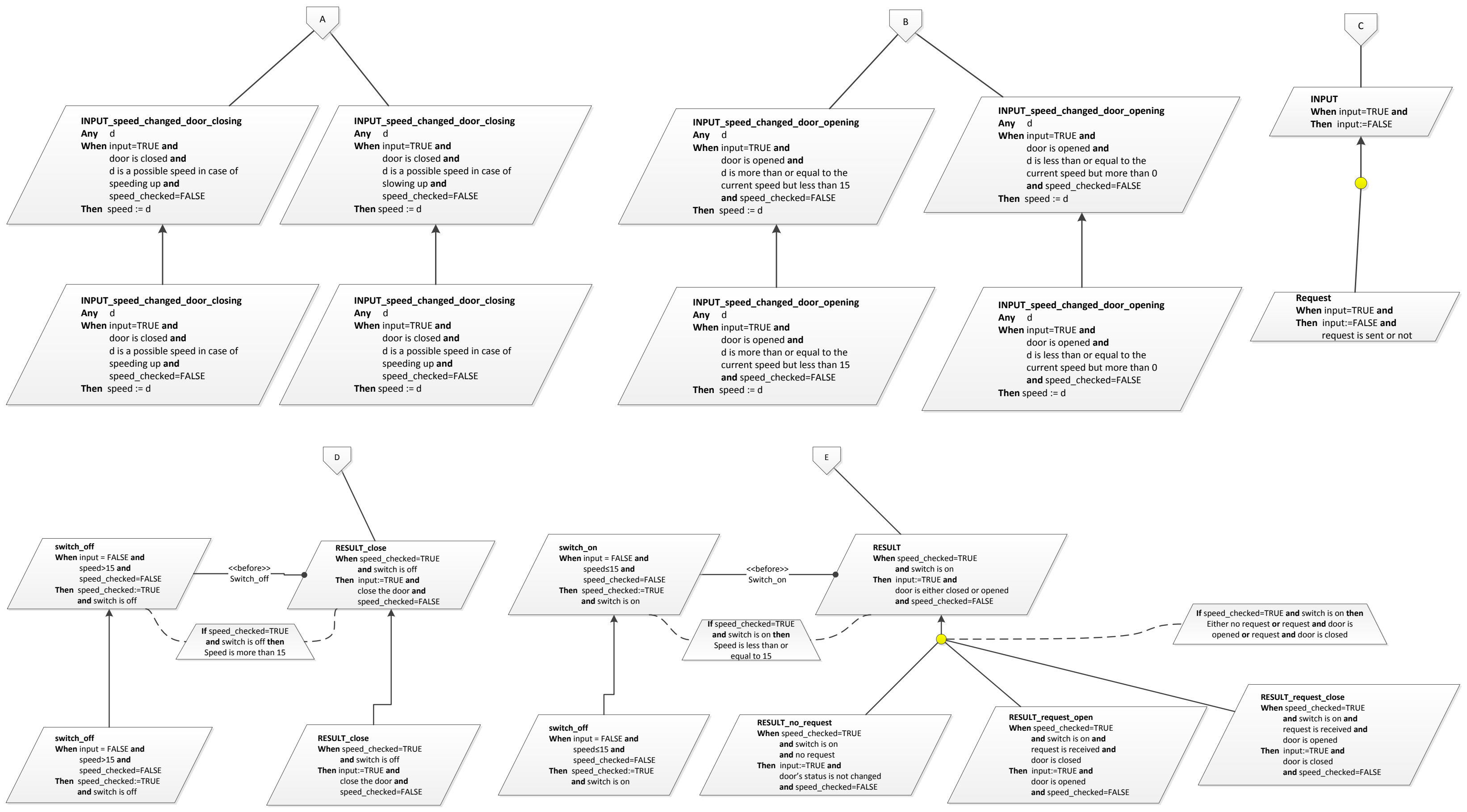
We presented the refinement tree diagram and the event transition diagram along with their construction rules in a semi-formal way. This might be sufficient in the short-term of this research. While, in the long-term, the formal semantics of the ORDER model must be defined. The formal semantics are important for analysis and improving the precision of the model. The precision is needed for avoiding ambiguity and inconsistency of the model. Such concepts can support the implementation of the automated tool and the extension of the model.

Appendix A

Powered sliding door case study

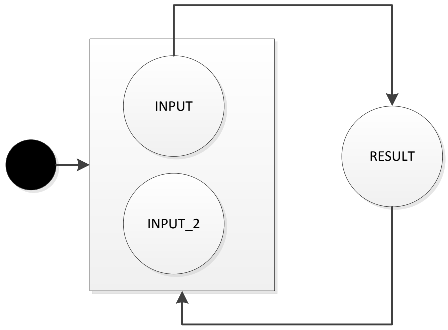
A.1 Refinement tree diagram



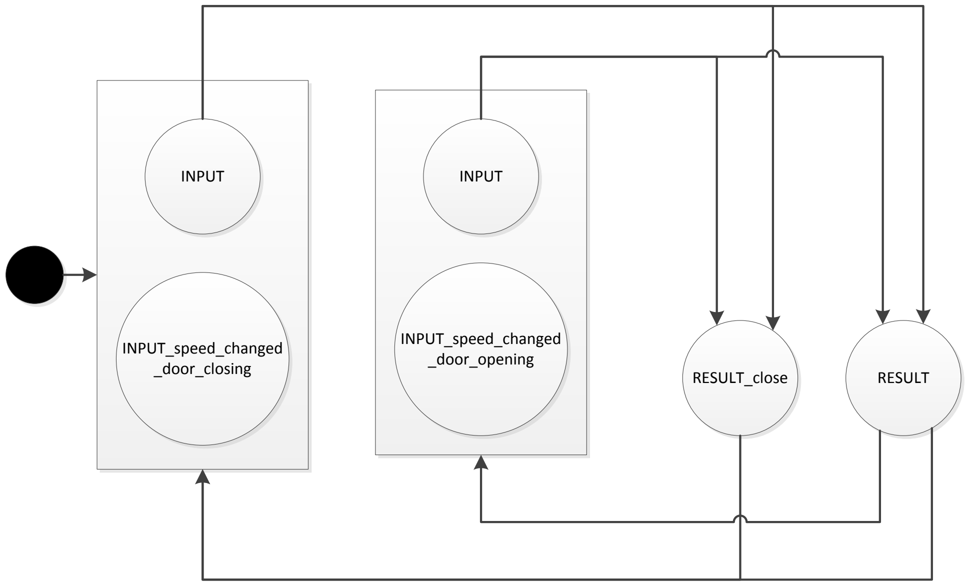

A
INPUT_speed_changed_door_closing Any d When input=TRUE and door is closed and d is a possible speed in case of speeding up and speed_checked=FALSE Then speed := d
INPUT_speed_changed_door_closing Any d When input=TRUE and door is closed and d is a possible speed in case of slowing up and speed_checked=FALSE Then speed := d
INPUT_speed_changed_door_closing Any d When input=TRUE and door is closed and d is a possible speed in case of speeding up and speed_checked=FALSE Then speed := d
INPUT_speed_changed_door_closing Any d When input=TRUE and door is closed and d is a possible speed in case of speeding up and speed_checked=FALSE Then speed := d
B
INPUT_speed_changed_door_opening Any d When input=TRUE and door is opened and d is more than or equal to the current speed but less than 15 and speed_checked=FALSE Then speed := d
INPUT_speed_changed_door_opening Any d When input=TRUE and door is opened and d is less than or equal to the current speed but more than 0 and speed_checked=FALSE Then speed := d
INPUT_speed_changed_door_opening Any d When input=TRUE and door is opened and d is more than or equal to the current speed but less than 15 and speed_checked=FALSE Then speed := d
INPUT_speed_changed_door_opening Any d When input=TRUE and door is opened and d is less than or equal to the current speed but more than 0 and speed_checked=FALSE Then speed := d
C
INPUT When input=TRUE and Then input:=FALSE
Request When input=TRUE and Then input:=FALSE and request is sent or not
D
switch_off When input = FALSE and speed>15 and speed_checked=FALSE Then speed_checked:=TRUE and switch is off
<<before>> Switch_off
RESULT_close When speed_checked=TRUE and switch is off Then input:=TRUE and close the door and speed_checked=FALSE
If speed_checked=TRUE and switch is off then Speed is more than 15
switch_off When input = FALSE and speed>15 and speed_checked=FALSE Then speed_checked:=TRUE and switch is off
RESULT_close When speed_checked=TRUE and switch is off Then input:=TRUE and close the door and speed_checked=FALSE
E
switch_on When input = FALSE and speed≤15 and speed_checked=FALSE Then speed_checked:=TRUE and switch is on
<<before>> Switch_on
RESULT When speed_checked=TRUE and switch is on Then input:=TRUE and door is either closed or opened and speed_checked=FALSE
If speed_checked=TRUE and switch is on then Speed is less than or equal to 15
If speed_checked=TRUE and switch is on then Either no request or request and door is opened or request and door is closed
switch_off When input = FALSE and speed≤15 and speed_checked=FALSE Then speed_checked:=TRUE and switch is on
RESULT_no_request When speed_checked=TRUE and switch is on and no request Then input:=TRUE and door's status is not changed and speed_checked=FALSE
RESULT_request_open When speed_checked=TRUE and switch is on and request is received and door is closed Then input:=TRUE and door is opened and speed_checked=FALSE
RESULT_request_close When speed_checked=TRUE and switch is on and request is received and door is opened Then input:=TRUE and door is closed and speed_checked=FALSE

A.2 Refinement tree diagram

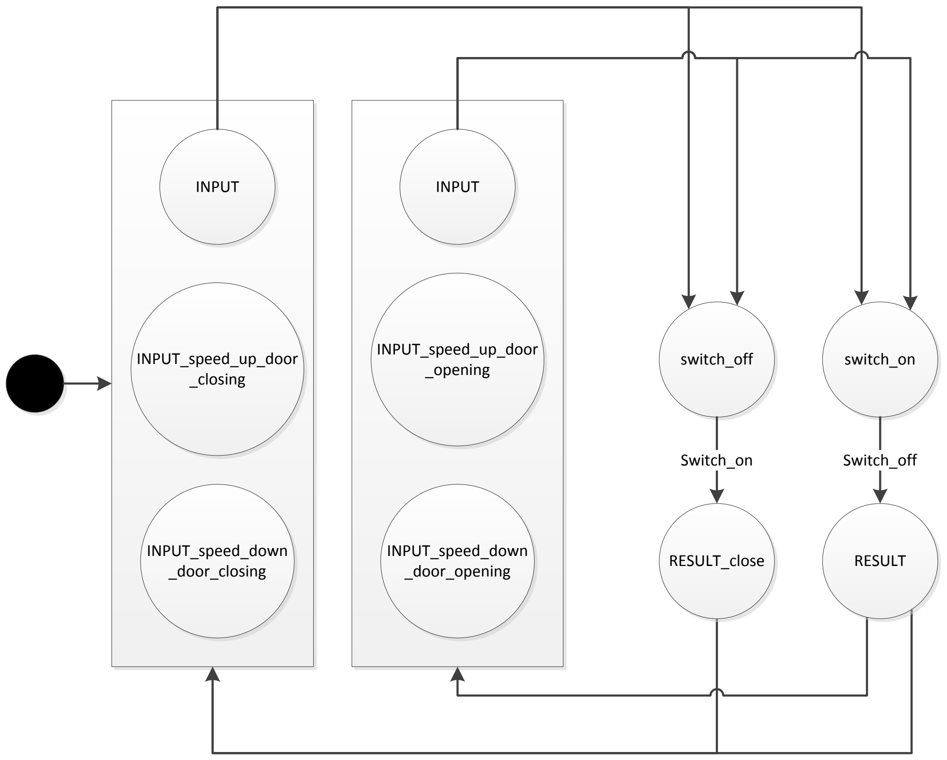
A.2.1 Initial model



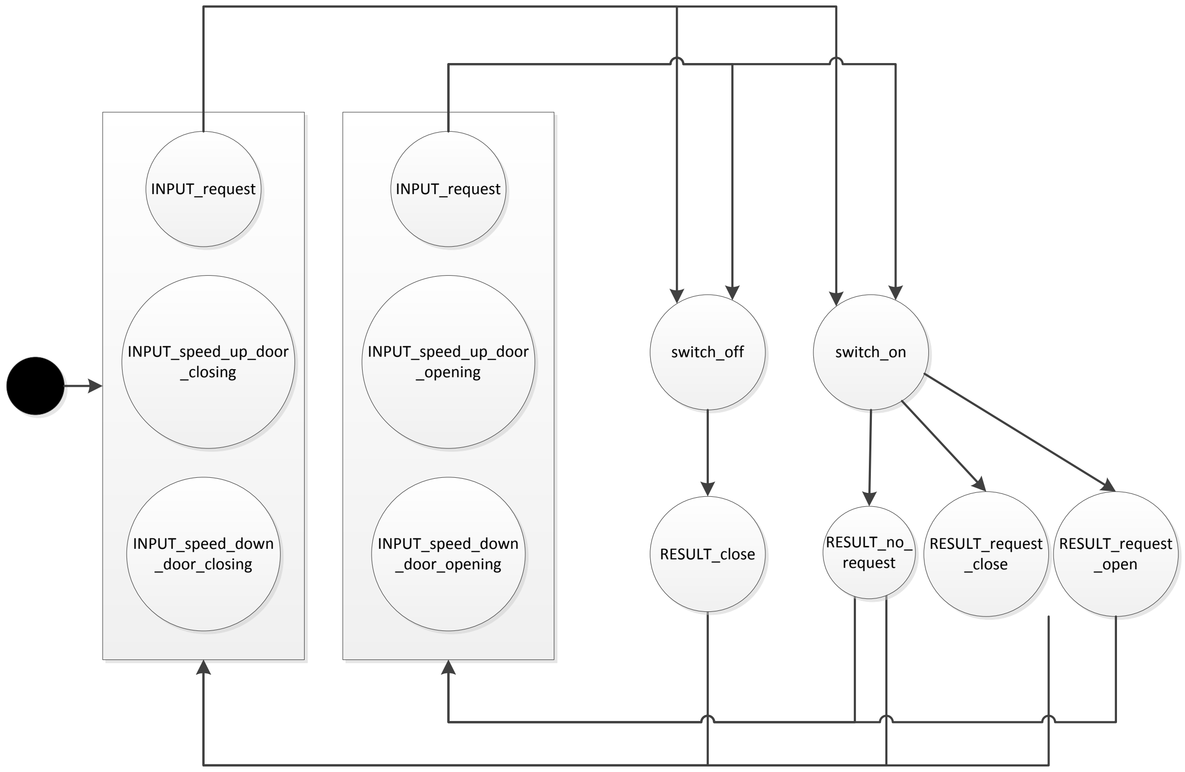
A.2.2 First refinement


INPUT
INPUT_speed_changed_door_closing
INPUT
INPUT_speed_changed_door_opening
RESULT_close
RESULT

A.2.3 Second refinement


INPUT
INPUT_speed_up_door _closing
INPUT_speed_down _door_closing
INPUT
INPUT_speed_up_door _opening
INPUT_speed_down _door_opening
switch_off
switch_on
Switch_on
Switch_off
RESULT_close
RESULT

A.2.4 Third refinement


INPUT_request
INPUT_speed_up_door _closing
INPUT_speed_down _door_closing
INPUT_request
INPUT_speed_up_door _opening
INPUT_speed_down _door_opening
switch_off
switch_on
RESULT_close
RESULT_no_ request
RESULT_request _close
RESULT_request _open

A.3 Description of events

The descriptions of the copying events of each level of the tree are omitted here.

A.3.1 Initial model

Event name	**Description**
INPUT	the input phase which can transit to the decision phase
INPUT_2	the input phase which cannot transit to the decision phase
RESULT	the decision phase

A.3.2 First refinement

Event name	**Description**
INPUT_speed_changed _door_closing	the change of speed when the door is closed
INPUT_speed_changed _door_opening	the change of speed when the door is opened
RESULT_close	the decision to close the door because the speed is higher than 15 km/h
RESULT	the decision when the speed is lower or equalled to 15 km/h

A.3.3 Second refinement

Event name	**Description**
INPUT_speed_up _door_closing	increasing the speed when the door is closed
INPUT_speed_down _door_closing	increasing the speed when the door is opened
INPUT_speed_up _door_opening	decreasing the speed when the door is closed
INPUT_speed_down _door_opening	decreasing the speed when the door is opened
switch_off	turning the switch off when the speed is higher than 15 km/h
RESULT_close	the decision to close the door because the switch is off
switch_on	turning the switch on when the speed is lower or equalled to 15 km/h
RESULT	the decision when the switch is on

A.3.4 Third refinement

Event name	**Description**
Request	the request from users
RESULT_no_request	keeping the same status of the door because there is no request
RESULT_request_open	opening the door when receiving a request
RESULT_request_close	closing the door when receiving a request

A.4 Description of carrier sets, constants, and variables

Component name	**Description**
D_STATUS	the carrier set of the door statuses
open	the ‘open’ status of the door
close	the ‘close’ status of the door
max_speed	the upper bound value of the vehicle speed
max_speed_up	the max acceleration value of the vehicle
max_speed_down	the max deceleration of value the vehicle
speed	the vehicle speed
door_status	the status of the door
switch	the status of the switch
request	the request from users

The variables representing phases:
input
speed_checked

A.5 Event-B specification

A.5.1 Contexts

CONTEXT door
SETS
    D_STATUS
CONSTANTS
    open
    close
AXIOMS
    axm1 : $partition(D\_STATUS, \{open\}, \{close\})$
    axm2 : $\neg open = close$
END

```
CONTEXT speed
CONSTANTS
    max_speed
AXIOMS
    axm1 : max_speed > 15
END

CONTEXT speed2
EXTENDS speed
CONSTANTS
    max_speed_up
    max_speed_down
AXIOMS
    axm1 : max_speed_up > 0
    axm2 : max_speed_down > 0
END
```

A.5.2 Initial model

```
MACHINE Machine_1
VARIABLES
    input
INVARIANTS
    inv1 : input ∈ BOOL
EVENTS
Initialisation
    begin
        act1 : input := TRUE
    end
Event INPUT ≙
    when
        grd1 : input = TRUE
    then
        act1 : input := FALSE
    end
Event RESULT ≙
    when
        grd2 : input = FALSE
    then
```

```
        act2 : input := TRUE
    end
Event   INPUT_2 ≙
    when
        grd1 : input = TRUE
    then
        skip
    end
END
```

A.5.3 First refinement

```
MACHINE   Machine_2
REFINES   Machine_1
SEES   door, speed
VARIABLES
    input
    speed
    door_status
INVARIANTS
    inv1 : speed ≥ 0 ∧ speed ≤ max_speed
    inv2 : door_status ∈ D_STATUS
    inv3 : door_status = open ⇒ speed ≤ 15
    inv4 : input = TRUE⇒((door_status = open∨door_status = close)∧¬(door_status =
          open ∧ door_status = close))
    inv5 : input = FALSE⇒((speed ≤ 15∨speed > 15)∧¬(speed ≤ 15∧speed > 15))
EVENTS
Initialisation
    extended
    begin
        act1 : input := TRUE
        act5 : speed := 0
        act7 : door_status := close
    end
Event   INPUT_speed_changed_door_closing ≙
extends  INPUT_2
    any
        d
    where
```

```
          grd1 : input = TRUE
          grd4 : door_status = close
          grd5 : d ≥ 0 ∧ d ≤ max_speed
     then
          act3 : speed := d
     end
Event   INPUT_speed_changed_door_opening ≙
extends  INPUT_2
     any
          d
     where
          grd1 : input = TRUE
          grd5 : d ≥ 0 ∧ d ≤ 15
          grd6 : door_status = open
     then
          act3 : speed := d
     end
Event   RESULT_close ≙
extends  RESULT
     when
          grd2 : input = FALSE
          grd4 : speed > 15
     then
          act2 : input := TRUE
          act3 : door_status := close
     end
Event   RESULT ≙
extends  RESULT
     when
          grd2 : input = FALSE
          grd4 : speed ≤ 15
     then
          act2 : input := TRUE
          act3 : door_status :∈ D_STATUS
     end
Event   INPUT ≙
extends  INPUT
     when
          grd1 : input = TRUE
     then
          act1 : input := FALSE
```

end

END

A.5.4 Second refinement

MACHINE Machine_3

REFINES Machine_2

SEES door, speed2

VARIABLES

`input`

`speed`

`door_status`

`switch`

`speed_checked`

INVARIANTS

`inv1` : $switch \in BOOL$

`inv2` : $speed\_checked = TRUE \wedge switch = TRUE \Rightarrow speed \leq 15 \wedge input = FALSE$

`inv3` : $speed\_checked = TRUE \wedge switch = FALSE \Rightarrow speed > 15 \wedge input = FALSE$

`inv4` : $speed\_checked \in BOOL$

EVENTS

Initialisation

extended

begin

`act1` : `input` := `TRUE`

`act5` : `speed` := `0`

`act7` : `door_status` := `close`

`act8` : $switch := TRUE$

`act9` : $speed\_checked := FALSE$

end

Event $INPUT\_speed\_up\_door\_closing \mathrel{\widehat{=}}$

refines $INPUT\_speed\_changed\_door\_closing$

any

d

where

`grd1` : $input = TRUE$

`grd4` : $door\_status = close$

`grd5` : $d \geq 0 \wedge d \leq max\_speed$

`grd2` : $d \geq speed \wedge d \leq speed + max\_speed\_up \wedge d \leq max\_speed$

`grd6` : $speed\_checked = FALSE$

then

```
            act3 : $speed := d$
      end
Event   $INPUT\_speed\_down\_door\_closing \mathrel{\widehat{=}}$
refines $INPUT\_speed\_changed\_door\_closing$
      any
            $d$
      where
            grd1 : $input = TRUE$
            grd4 : $door\_status = close$
            grd5 : $d \geq speed - max\_speed\_down \land d \leq speed \land d \geq 0$
            grd6 : $speed\_checked = FALSE$
      then
            act3 : $speed := d$
      end
Event   $INPUT\_speed\_up\_door\_opening \mathrel{\widehat{=}}$
refines $INPUT\_speed\_changed\_door\_opening$
      any
            $d$
      where
            grd1 : $input = TRUE$
            grd5 : $d \geq speed \land d \leq speed + max\_speed\_up \land d \leq 15$
            grd6 : $door\_status = open$
            grd7 : $speed\_checked = FALSE$
      then
            act3 : $speed := d$
      end
Event   $INPUT\_speed\_down\_door\_opening \mathrel{\widehat{=}}$
refines $INPUT\_speed\_changed\_door\_opening$
      any
            $d$
      where
            grd1 : $input = TRUE$
            grd5 : $d \geq speed - max\_speed\_down \land d \leq speed \land d \geq 0$
            grd6 : $door\_status = open$
            grd7 : $speed\_checked = FALSE$
      then
            act3 : $speed := d$
      end
Event   $switch\_off \mathrel{\widehat{=}}$
      when
            grd2 : $input = FALSE$
            grd3 : $speed > 15$
            grd4 : $speed\_checked = FALSE$
        then
            act1 : $switch := FALSE$
            act2 : $speed\_checked := TRUE$
        end
Event   RESULT_close ≙
refines RESULT_close
        when
            grd1 : $switch = FALSE$
            grd2 : $speed\_checked = TRUE$
        then
            act2 : $input := TRUE$
            act3 : $door\_status := close$
            act4 : $speed\_checked := FALSE$
        end
Event   switch_on ≙
        when
            grd2 : $input = FALSE$
            grd3 : $speed \leq 15$
            grd4 : $speed\_checked = FALSE$
        then
            act1 : $switch := TRUE$
            act2 : $speed\_checked := TRUE$
        end
Event   RESULT ≙
refines RESULT
        when
            grd1 : $switch = TRUE$
            grd2 : $speed\_checked = TRUE$
        then
            act2 : $input := TRUE$
            act3 : $door\_status :\in D\_STATUS$
            act4 : $speed\_checked := FALSE$
        end
Event   INPUT ≙
extends INPUT
        when
            grd1 : input = TRUE
        then
            act1 : input := FALSE
```

  end

END

A.5.5 Third refinement

MACHINE Machine_4

REFINES Machine_3

SEES door, speed2

VARIABLES

  input

  speed

  door_status

  switch

  speed_checked

  request

INVARIANTS

  inv1 : $request \in BOOL$

  inv3 : $switch = TRUE \wedge speed\_checked = TRUE \Rightarrow (request = FALSE \vee (request = TRUE \wedge door\_status = open) \vee (request = TRUE \wedge door\_status = close))$

EVENTS

Initialisation

  extended

  begin

    act1 : input := TRUE

    act5 : speed := 0

    act7 : door_status := close

    act8 : switch := TRUE

    act9 : speed_checked := FALSE

    act10 : *request* := *FALSE*

  end

Event *INPUT_speed_up_door_closing* $\widehat{=}$

extends *INPUT_speed_up_door_closing*

  any

    d

  where

    grd1 : input = TRUE

    grd4 : door_status = close

    grd5 : $d \geq 0 \wedge d \leq max\_speed$

    grd2 : $d \geq speed \wedge d \leq speed + max\_speed\_up \wedge d \leq max\_speed$

    grd6 : speed_checked = FALSE

```
    then
        act3 : speed := d
    end
Event  INPUT_speed_down_door_closing ≙
extends  INPUT_speed_down_door_closing
    any
        d
    where
        grd1 : input = TRUE
        grd4 : door_status = close
        grd5 : d ≥ speed − max_speed_down ∧ d ≤ speed ∧ d ≥ 0
        grd6 : speed_checked = FALSE
    then
        act3 : speed := d
    end
Event  INPUT_speed_up_door_opening ≙
extends  INPUT_speed_up_door_opening
    any
        d
    where
        grd1 : input = TRUE
        grd5 : d ≥ speed ∧ d ≤ speed + max_speed_up ∧ d ≤ 15
        grd6 : door_status = open
        grd7 : speed_checked = FALSE
    then
        act3 : speed := d
    end
Event  INPUT_speed_down_door_opening ≙
extends  INPUT_speed_down_door_opening
    any
        d
    where
        grd1 : input = TRUE
        grd5 : d ≥ speed − max_speed_down ∧ d ≤ speed ∧ d ≥ 0
        grd6 : door_status = open
        grd7 : speed_checked = FALSE
    then
        act3 : speed := d
    end
Event  switch_off ≙
extends  switch_off
```

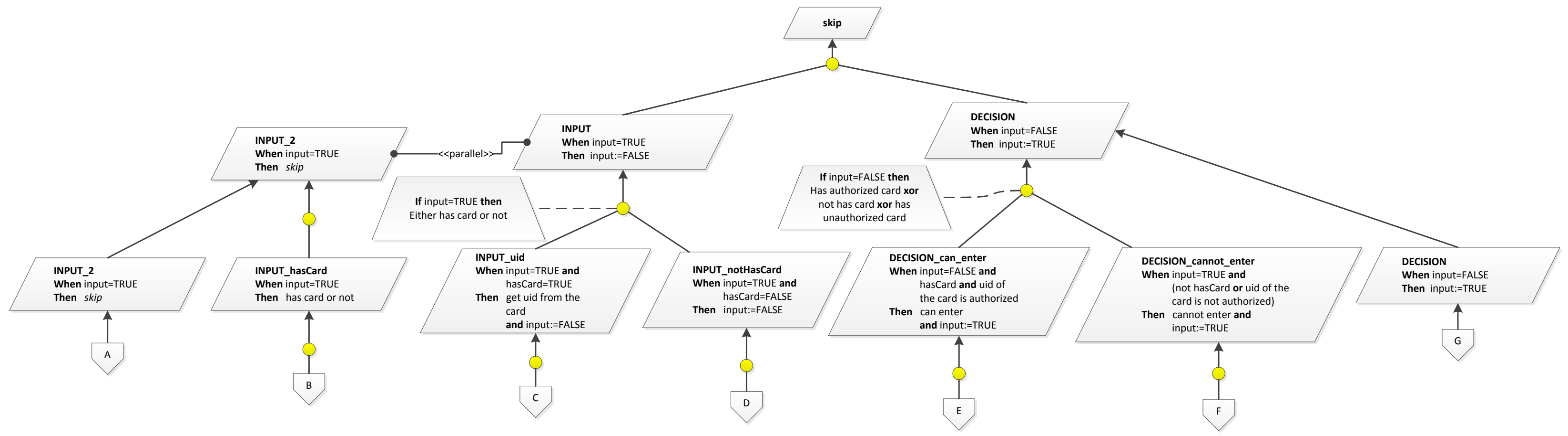
```
    when
        grd2 : input = FALSE
        grd3 : speed > 15
        grd4 : speed_checked = FALSE
    then
        act1 : switch := FALSE
        act2 : speed_checked := TRUE
    end
Event   RESULT_close ≙
extends RESULT_close
    when
        grd1 : switch = FALSE
        grd2 : speed_checked = TRUE
    then
        act2 : input := TRUE
        act3 : door_status := close
        act4 : speed_checked := FALSE
    end
Event   switch_on ≙
extends switch_on
    when
        grd2 : input = FALSE
        grd3 : speed ≤ 15
        grd4 : speed_checked = FALSE
    then
        act1 : switch := TRUE
        act2 : speed_checked := TRUE
    end
Event   RESULT_no_request ≙
refines RESULT
    when
        grd1 : switch = TRUE
        grd2 : speed_checked = TRUE
        grd3 : request = FALSE
    then
        act2 : input := TRUE
        act3 : door_status := door_status
        act4 : speed_checked := FALSE
    end
Event   RESULT_request_open ≙
refines RESULT
```

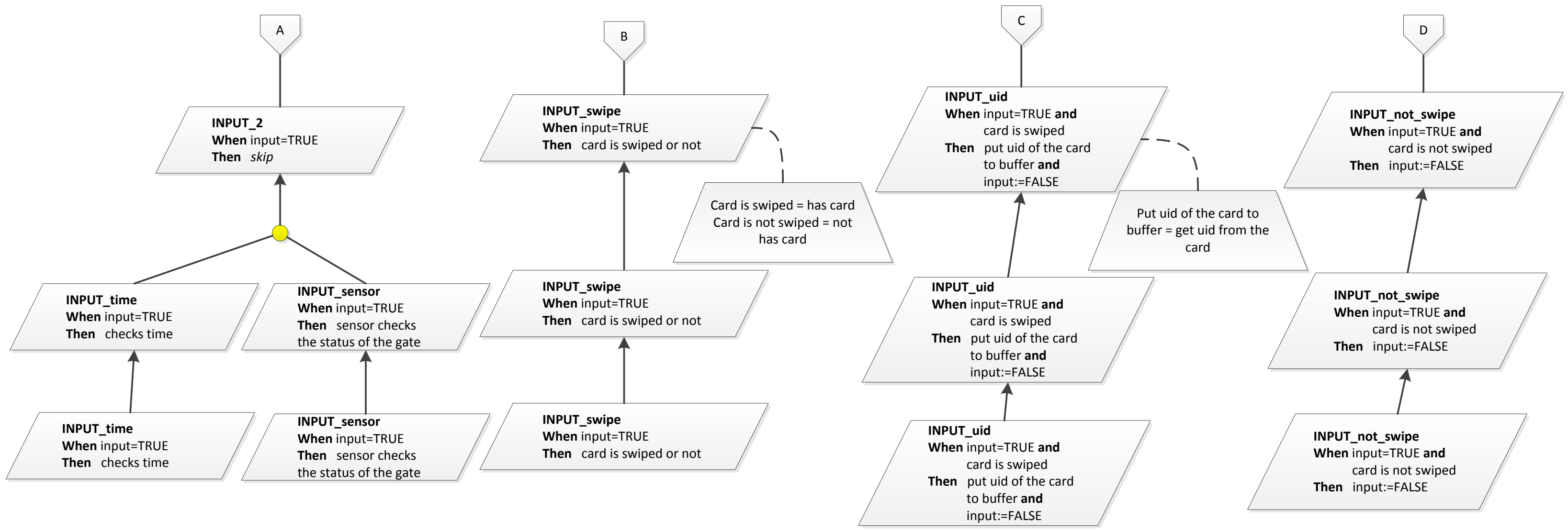
```
    when
        grd1 : $switch = TRUE$
        grd2 : $speed\_checked = TRUE$
        grd3 : $request = TRUE$
        grd4 : $door\_status = close$
    then
        act2 : $input := TRUE$
        act3 : $door\_status := open$
        act4 : $speed\_checked := FALSE$
    end
Event  $RESULT\_request\_close \; \widehat{=}$
refines  $RESULT$
    when
        grd1 : $switch = TRUE$
        grd2 : $speed\_checked = TRUE$
        grd3 : $request = TRUE$
        grd4 : $door\_status = open$
    then
        act2 : $input := TRUE$
        act3 : $door\_status := close$
        act4 : $speed\_checked := FALSE$
    end
Event  $Request \; \widehat{=}$
extends  $INPUT$
    when
        grd1 : input = TRUE
    then
        act1 : input := FALSE
        act2 : $request :\in BOOL$
    end
END
```

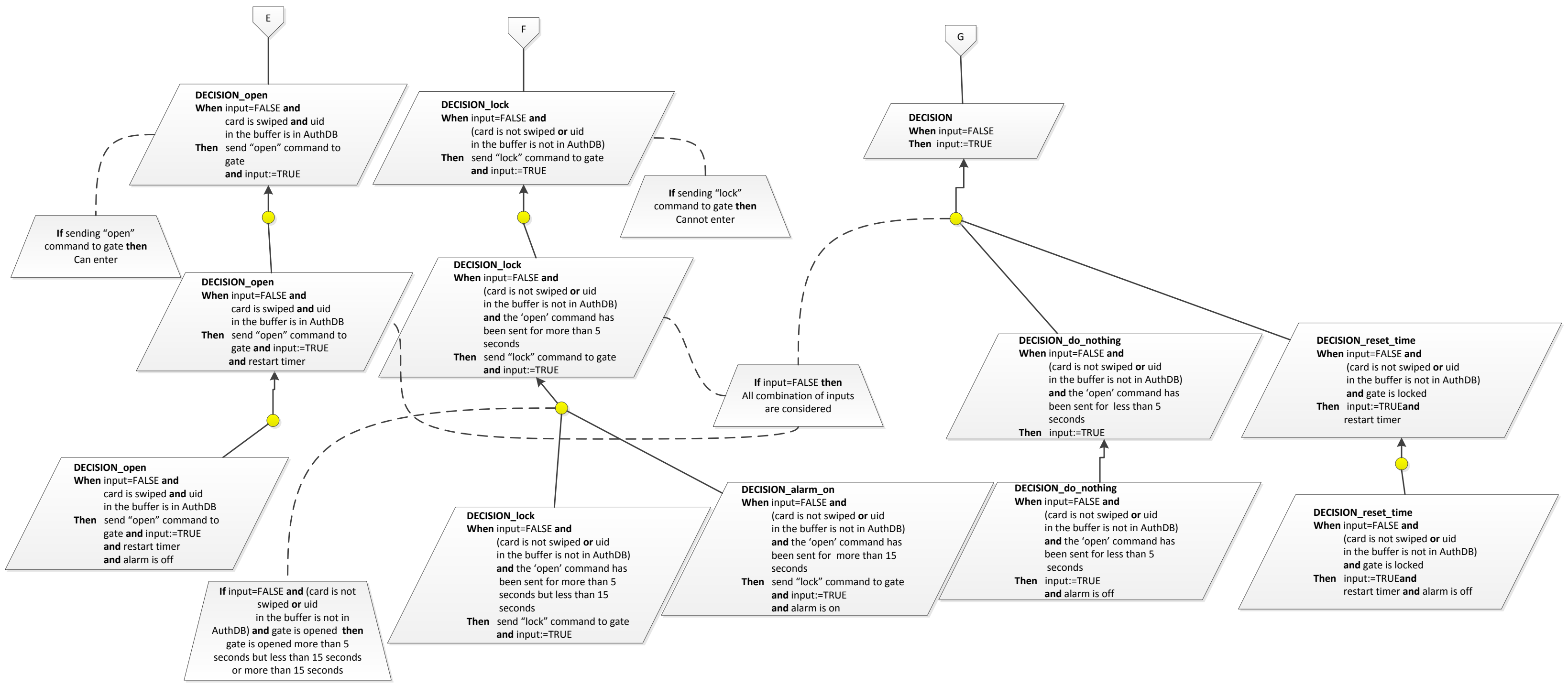
Appendix B

Automatic gate controller case study

B.1 Refinement tree diagram

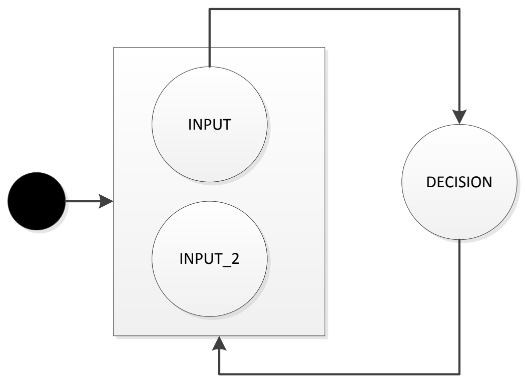

skip
INPUT_2 When input=TRUE Then skip
<<parallel>>
INPUT When input=TRUE Then input:=FALSE
DECISION When input=FALSE Then input:=TRUE
If input=TRUE then Either has card or not
If input=FALSE then Has authorized card xor not has card xor has unauthorized card
INPUT_2 When input=TRUE Then skip
INPUT_hasCard When input=TRUE Then has card or not
INPUT_uid When input=TRUE and hasCard=TRUE Then get uid from the card and input:=FALSE
INPUT_notHasCard When input=TRUE and hasCard=FALSE Then input:=FALSE
DECISION_can_enter When input=FALSE and hasCard and uid of the card is authorized Then can enter and input:=TRUE
DECISION_cannot_enter When input=TRUE and (not hasCard or uid of the card is not authorized) Then cannot enter and input:=TRUE
DECISION When input=FALSE Then input:=TRUE
A
B
C
D
E
F
G


A
INPUT_2
When input=TRUE
Then skip
INPUT_time
When input=TRUE
Then checks time
INPUT_sensor
When input=TRUE
Then sensor checks the status of the gate
INPUT_time
When input=TRUE
Then checks time
INPUT_sensor
When input=TRUE
Then sensor checks the status of the gate
B
INPUT_swipe
When input=TRUE
Then card is swiped or not
Card is swiped = has card
Card is not swiped = not has card
INPUT_swipe
When input=TRUE
Then card is swiped or not
INPUT_swipe
When input=TRUE
Then card is swiped or not
C
INPUT_uid
When input=TRUE and card is swiped
Then put uid of the card to buffer and input:=FALSE
Put uid of the card to buffer = get uid from the card
INPUT_uid
When input=TRUE and card is swiped
Then put uid of the card to buffer and input:=FALSE
INPUT_uid
When input=TRUE and card is swiped
Then put uid of the card to buffer and input:=FALSE
D
INPUT_not_swipe
When input=TRUE and card is not swiped
Then input:=FALSE
INPUT_not_swipe
When input=TRUE and card is not swiped
Then input:=FALSE
INPUT_not_swipe
When input=TRUE and card is not swiped
Then input:=FALSE

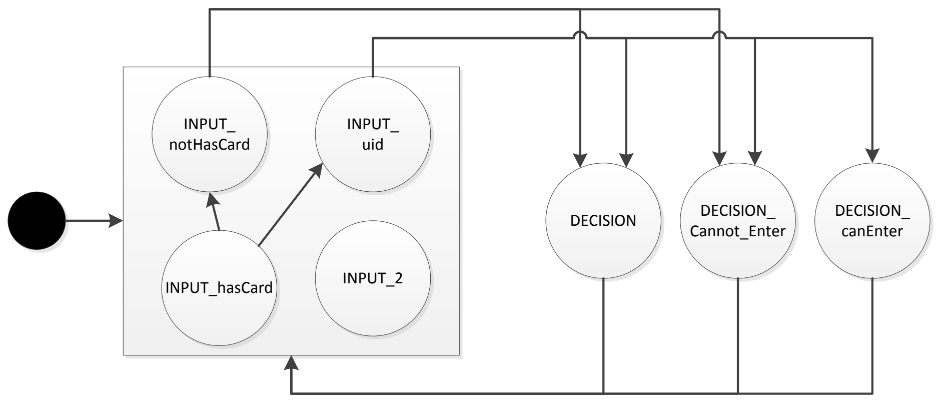

E
F
G
DECISION_open When input=FALSE and card is swiped and uid in the buffer is in AuthDB Then send "open" command to gate and input:=TRUE
DECISION_lock When input=FALSE and (card is not swiped or uid in the buffer is not in AuthDB) Then send "lock" command to gate and input:=TRUE
DECISION When input=FALSE Then input:=TRUE
If sending "lock" command to gate then Cannot enter
If sending "open" command to gate then Can enter
DECISION_open When input=FALSE and card is swiped and uid in the buffer is in AuthDB Then send "open" command to gate and input:=TRUE and restart timer
DECISION_lock When input=FALSE and (card is not swiped or uid in the buffer is not in AuthDB) and the 'open' command has been sent for more than 5 seconds Then send "lock" command to gate and input:=TRUE
If input=FALSE then All combination of inputs are considered
DECISION_do_nothing When input=FALSE and (card is not swiped or uid in the buffer is not in AuthDB) and the 'open' command has been sent for less than 5 seconds Then input:=TRUE
DECISION_reset_time When input=FALSE and (card is not swiped or uid in the buffer is not in AuthDB) and gate is locked Then input:=TRUEand restart timer
DECISION_open When input=FALSE and card is swiped and uid in the buffer is in AuthDB Then send "open" command to gate and input:=TRUE and restart timer and alarm is off
DECISION_alarm_on When input=FALSE and (card is not swiped or uid in the buffer is not in AuthDB) and the 'open' command has been sent for more than 15 seconds Then send "lock" command to gate and input:=TRUE and alarm is on
DECISION_do_nothing When input=FALSE and (card is not swiped or uid in the buffer is not in AuthDB) and the 'open' command has been sent for less than 5 seconds Then input:=TRUE and alarm is off
DECISION_lock When input=FALSE and (card is not swiped or uid in the buffer is not in AuthDB) and the 'open' command has been sent for more than 5 seconds but less than 15 seconds Then send "lock" command to gate and input:=TRUE
DECISION_reset_time When input=FALSE and (card is not swiped or uid in the buffer is not in AuthDB) and gate is locked Then input:=TRUEand restart timer and alarm is off
If input=FALSE and (card is not swiped or uid in the buffer is not in AuthDB) and gate is opened then gate is opened more than 5 seconds but less than 15 seconds or more than 15 seconds

B.2 Event transition diagram

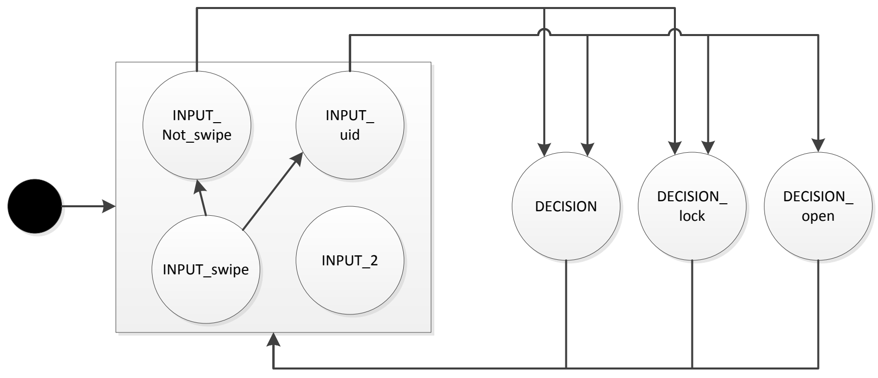
B.2.1 Initial model



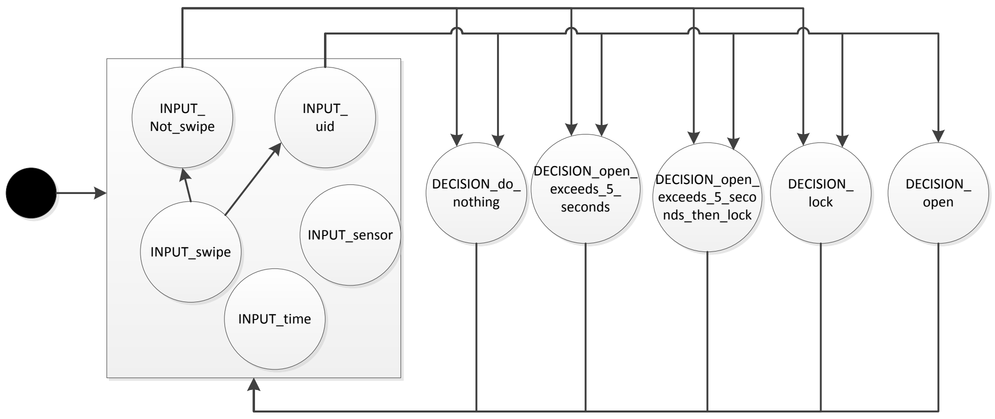
B.2.2 First refinement



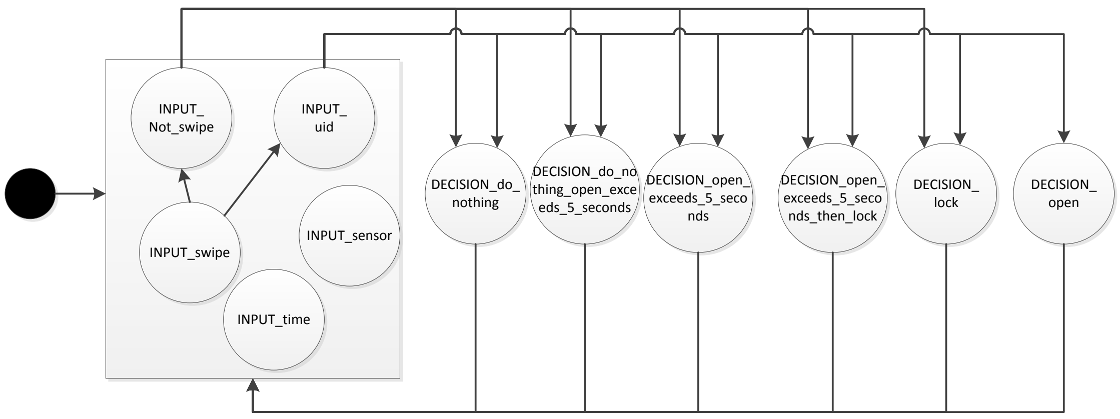
B.2.3 Second refinement



B.2.4 Third refinement



B.2.5 Fourth refinement


INPUT_ Not_swipe
INPUT_ uid
INPUT_swipe
INPUT_sensor
INPUT_time
DECISION_do_ nothing
DECISION_do_no thing_open_exce eds_5_seconds
DECISION_open_ exceeds_5_seco nds
DECISION_open_ exceeds_5_seco nds_then_lock
DECISION_ lock
DECISION_ open

B.3 Description of events

The descriptions of the copying events of each level of the tree are omitted here.

B.3.1 Initial model

Event name	**Description**
INPUT	the input phase which can transit to the decision phase
INPUT_2	the input phase which cannot transit to the decision phase
DECISION	the decision phase

B.3.2 First refinement

Event name	**Description**
INPUT_hasCard	checking the possession of a card
INPUT_uid	retrieving the uid of a card when the possession is detected
INPUT_notHasCard	directly transiting to the next phase when detecting no card
DECISION_can_enter	allowing a person to enter when the person has an authorized card
DECISION_cannot_enter	preventing a person from entering when the person does not have an authorized card

B.3.3 Second refinement

Event name	**Description**
INPUT_swipe	checking whether a card is swiped
INPUT_uid	retrieving the uid from a card and storing it in a buffer when the card is swiped
INPUT_notSwipe	directly transiting to the next phase when no card is swiped
DECISION_open	sending an ‘open’ command to the gate when an authorized card is swiped
DECISION_lock	sending an ‘lock’ command to the gate when no authorized card is swiped

B.3.4 Third refinement

Event name	**Description**
INPUT_time	checking the current time
INPUT_sensor	checking the current status of the gate
DECISION_open	sending an ‘open’ command to the gate and restating timer when an authorized card is swiped
DECISION_lock	sending an ‘lock’ command to the gate when no authorized card is swiped and the ‘open’ command is sent for more than 5 seconds
DECISION_do_nothing	doing nothing when no authorized card is swiped and the ‘open’ command is sent for lower than 5 seconds
DECISION_reset_time	restarting the timer when no authorized card is swiped and the gate is locked

B.3.5 Fourth refinement

Event name	**Description**
DECISION_open	sending an 'open' command to the gate and restating timer and turning alarm off when an authorized card is swiped
DECISION_lock	sending an 'lock' command to the gate when no authorized card is swiped and the 'open' command is sent for more than 5 seconds but less than 15 seconds
DECISION_alarm_on	sending an 'lock' command to the gate and turning alarm on when no authorized card is swiped and the 'open' command is sent for more than 15 seconds
DECISION_reset_time	restarting the timer and turning alarm off when no authorized card is swiped and the gate is locked

B.4 Descriptions of carrier sets, constants, and variables

Component name	**Description**
UID	a set of uid of cards
GATE_COMMAND	a set of command to change the status of the gate
AuthID	a set of authorized uid
opened	the 'open' command to open the gate
locked	the 'lock' command to lock the gate
AuthDB	a database of authorized uid
out_of_AuthDB	a database of unauthorized uid
canEnter	whether the system allows a person to enter that gate
hasCard	whether the possession of a card is detected
card_uid	the uid of a detected card
gate_command	the command from the system to the gate
swipe	whether a card is swiped
read_uid_buffer	the buffer for storing the uid of the swiped card
gate_sensor	the sensor for sensing the current status of the gate
last_lock_time	the latest time when the gate is locked
time	current time
alarm_on	whether the alarm is turned on

The variables representing phases:
input

B.5 Event-B specification

B.5.1 Contexts

CONTEXT Context
SETS
    UID
CONSTANTS
    AuthID
AXIOMS
    axm1 : $AuthID \in UID \to BOOL$
END

CONTEXT Context1
EXTENDS Context
SETS
    GATE_COMMAND
CONSTANTS
    opened
    locked
    AuthDB
    out_of_AuthDB
AXIOMS
    axm1 : $partition(GATE\_COMMAND, \{opened\}, \{locked\})$
    axm2 : $AuthDB = dom(AuthID \rhd \{TRUE\})$
    axm3 : $out\_of\_AuthDB = dom(AuthID \rhd \{FALSE\})$
END

B.5.2 Initial model

MACHINE Machine_1
VARIABLES
    input
INVARIANTS
    inv1 : $input \in BOOL$
EVENTS
Initialisation
    begin
        act1 : $input := TRUE$

```
        end
Event   INPUT ≙
        when
            grd1 : $input = TRUE$
        then
            act1 : $input := FALSE$
        end
Event   INPUT_2 ≙
        when
            grd1 : $input = TRUE$
        then
            skip
        end
Event   DECISION ≙
        when
            grd1 : $input = FALSE$
        then
            act1 : $input := TRUE$
        end
END
```

B.5.3 First refinement

```
MACHINE   Machine_2
REFINES   Machine_1
SEES   Context
VARIABLES
        input
        canEnter
        hasCard
        card_uid
INVARIANTS
        inv2 : $canEnter \in BOOL$
        inv4 : $hasCard \in BOOL$
        inv5 : $card\_uid \in UID$
        inv6 : $input = FALSE \Rightarrow (((hasCard = TRUE \land AuthID(card\_uid) = TRUE) \lor$
            $hasCard = FALSE \lor AuthID(card\_uid) = FALSE) \land \neg((hasCard = TRUE \land$
            $AuthID(card\_uid) = TRUE) \land hasCard = FALSE \land AuthID(card\_uid) =$
            $FALSE))$
```

inv7 : $input = TRUE \Rightarrow ((hasCard = TRUE \lor hasCard = FALSE) \land \neg(hasCard = TRUE \land hasCard = FALSE))$

EVENTS

Initialisation

extended

begin

act1 : $\texttt{input} := \texttt{TRUE}$

act3 : $canEnter := FALSE$

act4 : $hasCard := FALSE$

act5 : $card\_uid :\in UID$

end

Event $INPUT\_hasCard \mathrel{\widehat{=}}$

extends $INPUT\_2$

when

grd1 : $\texttt{input} = \texttt{TRUE}$

then

act2 : $hasCard :\in BOOL$

end

Event $INPUT\_uid \mathrel{\widehat{=}}$

extends $INPUT$

when

grd1 : $\texttt{input} = \texttt{TRUE}$

grd2 : $hasCard = TRUE$

then

act1 : $\texttt{input} := \texttt{FALSE}$

act2 : $card\_uid :\in UID$

end

Event $INPUT\_notHasCard \mathrel{\widehat{=}}$

extends $INPUT$

when

grd1 : $\texttt{input} = \texttt{TRUE}$

grd2 : $hasCard = FALSE$

then

act1 : $\texttt{input} := \texttt{FALSE}$

end

Event $DECISION\_can\_enter \mathrel{\widehat{=}}$

extends $DECISION$

when

grd1 : $\texttt{input} = \texttt{FALSE}$

grd2 : $hasCard = TRUE \land AuthID(card\_uid) = TRUE$

```
      then
           act1 : input := TRUE
           act2 : canEnter := TRUE
      end
Event   DECISION_cannot_enter ≙
extends  DECISION
      when
           grd1 : input = FALSE
           grd2 : hasCard = FALSE ∨ AuthID(card_uid) = FALSE
      then
           act1 : input := TRUE
           act2 : canEnter := FALSE
      end
Event   INPUT_2 ≙
extends  INPUT_2
      when
           grd1 : input = TRUE
      then
           skip
      end
Event   DECISION ≙
extends  DECISION
      when
           grd1 : input = FALSE
      then
           act1 : input := TRUE
      end
END
```

B.5.4 Second refinement

```
MACHINE   Machine_3
REFINES   Machine_2
SEES   Context1
VARIABLES
      input
      gate_command
      swipe
      read_uid_buffer
INVARIANTS
```

```
    inv1 : $gate\_command \in GATE\_COMMAND$
    inv2 : $swipe \in BOOL$
    inv4 : $swipe = hasCard$
    inv6 : $read\_uid\_buffer = card\_uid$
    inv7 : $gate\_command = opened \Rightarrow canEnter = TRUE$
    inv8 : $gate\_command = locked \Rightarrow canEnter = FALSE$
EVENTS
Initialisation
    begin
        with
            card_uid' : card_uid' = read_uid_buffer'
        act1 : $input := TRUE$
        act6 : $gate\_command := locked$
        act7 : $swipe := FALSE$
        act8 : $read\_uid\_buffer :\in UID$
    end
Event   $INPUT\_swipe \widehat{=}$
refines $INPUT\_hasCard$
    when
        grd1 : $input = TRUE$
    with
        hasCard' : hasCard' = swipe'
    then
        act2 : $swipe :\in BOOL$
    end
Event   $INPUT\_uid \widehat{=}$
refines $INPUT\_uid$
    when
        grd1 : $input = TRUE$
        grd2 : $swipe = TRUE$
    with
        card_uid' : card_uid' = read_uid_buffer'
    then
        act1 : $input := FALSE$
        act2 : $read\_uid\_buffer :\in UID$
    end
Event   $INPUT\_not\_swipe \widehat{=}$
refines $INPUT\_notHasCard$
    when
        grd1 : $input = TRUE$
```

```
            grd2 : $swipe = FALSE$
        then
            act1 : $input := FALSE$
        end
Event   $DECISION\_open \;\widehat{=}$
refines  $DECISION\_can\_enter$
        when
            grd1 : $input = FALSE$
            grd2 : $swipe = TRUE \wedge read\_uid\_buffer \in AuthDB$
        then
            act1 : $input := TRUE$
            act2 : $gate\_command := opened$
        end
Event   $DECISION\_lock \;\widehat{=}$
refines  $DECISION\_cannot\_enter$
        when
            grd1 : $input = FALSE$
            grd2 : $swipe = FALSE \vee read\_uid\_buffer \in out\_of\_AuthDB$
        then
            act1 : $input := TRUE$
            act2 : $gate\_command := locked$
        end
Event   $INPUT\_2 \;\widehat{=}$
extends  $INPUT\_2$
        when
            grd1 : input = TRUE
        then
            skip
        end
Event   $DECISION \;\widehat{=}$
extends  $DECISION$
        when
            grd1 : input = FALSE
        then
            act1 : input := TRUE
        end
END
```

B.5.5 Third refinement

MACHINE Machine_4

REFINES Machine_3

SEES Context1

VARIABLES

```
    input
    gate_command
    swipe
    read_uid_buffer
    gate_sensor
    last_lock_time
    time
```

INVARIANTS

inv1 : $gate\_sensor \in GATE\_COMMAND$

inv2 : $last\_lock\_time \in 0 \,..\, time$

inv3 : $time \in \mathbb{N}$

inv4 : $input = FALSE \Rightarrow (((swipe = TRUE \land read\_uid\_buffer \in AuthDB) \lor ((swipe = FALSE \lor read\_uid\_buffer \in out\_of\_AuthDB) \land time - last\_lock\_time \geq 5 \land gate\_sensor = locked) \lor ((swipe = FALSE \lor read\_uid\_buffer \in out\_of\_AuthDB) \land time - last\_lock\_time < 5 \land gate\_sensor = locked) \lor ((swipe = FALSE \lor read\_uid\_buffer \in out\_of\_AuthDB) \land gate\_sensor = opened)) \land \neg((swipe = TRUE \land read\_uid\_buffer \in AuthDB) \land ((swipe = FALSE \lor read\_uid\_buffer \in out\_of\_AuthDB) \land time - last\_lock\_time \geq 5 \land gate\_sensor = locked) \land ((swipe = FALSE \lor read\_uid\_buffer \in out\_of\_AuthDB) \land time - last\_lock\_time < 5 \land gate\_sensor = locked) \land ((swipe = FALSE \lor read\_uid\_buffer \in out\_of\_AuthDB) \land gate\_sensor = opened)))$

inv5 : $\forall uid \cdot (uid \in UID \land (uid \in AuthDB \Leftrightarrow uid \notin out\_of\_AuthDB))$

EVENTS

Initialisation

extended

begin

act1 : input := TRUE

act6 : gate_command := locked

act7 : swipe := FALSE

act8 : read_uid_buffer :∈ UID

act9 : $gate\_sensor := locked$

act10 : $last\_lock\_time := 0$

act11 : $time := 0$

end

Event $INPUT\_swipe \widehat{=}$
extends $INPUT\_swipe$
  when
    grd1 : `input` $=$ `TRUE`
  then
    act2 : `swipe` $:\in$ `BOOL`
  end
Event $INPUT\_uid \widehat{=}$
extends $INPUT\_uid$
  when
    grd1 : `input` $=$ `TRUE`
    grd2 : `swipe` $=$ `TRUE`
  then
    act1 : `input` $:=$ `FALSE`
    act2 : `read_uid_buffer` $:\in$ `UID`
  end
Event $INPUT\_time \widehat{=}$
extends $INPUT\_2$
  any
    n
  where
    grd1 : `input` $=$ `TRUE`
    grd2 : $n > time$
  then
    act1 : $time :\in time \mathrel{..} n$
  end
Event $INPUT\_sensor \widehat{=}$
extends $INPUT\_2$
  when
    grd1 : `input` $=$ `TRUE`
  then
    act1 : $gate\_sensor :\in GATE\_COMMAND$
  end
Event $INPUT\_not\_swipe \widehat{=}$
extends $INPUT\_not\_swipe$
  when
    grd1 : `input` $=$ `TRUE`
    grd2 : `swipe` $=$ `FALSE`
  then
    act1 : `input` $:=$ `FALSE`

```
    end
Event  DECISION_open ≙
extends  DECISION_open
    when
        grd1 : input = FALSE
        grd2 : swipe = TRUE ∧ read_uid_buffer ∈ AuthDB
    then
        act1 : input := TRUE
        act2 : gate_command := opened
        act3 : last_lock_time := time
    end
Event  DECISION_lock ≙
extends  DECISION_lock
    when
        grd1 : input = FALSE
        grd2 : swipe = FALSE ∨ read_uid_buffer ∈ out_of_AuthDB
        grd3 : gate_sensor = opened
        grd4 : time − last_lock_time ≥ 5
    then
        act1 : input := TRUE
        act2 : gate_command := locked
    end
Event  DECISION_do_nothing ≙
extends  DECISION
    when
        grd1 : input = FALSE
        grd2 : swipe = FALSE ∨ read_uid_buffer ∈ out_of_AuthDB
        grd4 : gate_sensor = opened
        grd3 : time − last_lock_time < 5
    then
        act1 : input := TRUE
    end
Event  DECISION_reset_time ≙
extends  DECISION
    when
        grd1 : input = FALSE
        grd2 : swipe = FALSE ∨ read_uid_buffer ∈ out_of_AuthDB
        grd3 : gate_sensor = locked
    then
        act1 : input := TRUE
        act2 : last_lock_time := time
```

end

END

B.5.6 Fourth refinement

MACHINE Machine_5

REFINES Machine_4

SEES Context1

VARIABLES

```
    input
    gate_command
    swipe
    read_uid_buffer
    gate_sensor
    last_lock_time
    time
    alarm_on
```

INVARIANTS

inv1 : $alarm\_on \in BOOL$

inv2 : $(input = FALSE \wedge (swipe = FALSE \vee read\_uid\_buffer \in out\_of\_AuthDB) \wedge gate\_sensor = opened) \Rightarrow ((time - last\_lock\_time \geq 5 \wedge time - last\_lock\_time < 15) \vee time - last\_lock\_time \geq 15)$

EVENTS

Initialisation

extended

begin

```
        act1 : input := TRUE
        act6 : gate_command := locked
        act7 : swipe := FALSE
        act8 : read_uid_buffer :∈ UID
        act9 : gate_sensor := locked
        act10 : last_lock_time := 0
        act11 : time := 0
```

act12 : $alarm\_on := FALSE$

end

Event $INPUT\_swipe \mathrel{\widehat{=}}$

extends $INPUT\_swipe$

when

```
        grd1 : input = TRUE
```

then

```
        act2 : swipe :∈ BOOL
    end
Event   INPUT_uid ≙
extends INPUT_uid
    when
        grd1 : input = TRUE
        grd2 : swipe = TRUE
    then
        act1 : input := FALSE
        act2 : read_uid_buffer :∈ UID
    end
Event   INPUT_time ≙
extends INPUT_time
    any
        n
    where
        grd1 : input = TRUE
        grd2 : n > time
    then
        act1 : time :∈ time .. n
    end
Event   INPUT_sensor ≙
extends INPUT_sensor
    when
        grd1 : input = TRUE
    then
        act1 : gate_sensor :∈ GATE_COMMAND
    end
Event   INPUT_not_swipe ≙
extends INPUT_not_swipe
    when
        grd1 : input = TRUE
        grd2 : swipe = FALSE
    then
        act1 : input := FALSE
    end
Event   DECISION_open ≙
extends DECISION_open
    when
        grd1 : input = FALSE
        grd2 : swipe = TRUE ∧ read_uid_buffer ∈ AuthDB
    then
        act1 : input := TRUE
        act2 : gate_command := opened
        act3 : last_lock_time := time
        act4 : alarm_on := FALSE
    end
Event   DECISION_lock ≙
refines DECISION_lock
    when
        grd1 : input = FALSE
        grd2 : swipe = FALSE ∨ read_uid_buffer ∈ out_of_AuthDB
        grd3 : gate_sensor = opened
        grd4 : time − last_lock_time ≥ 5
        grd5 : time − last_lock_time < 15
    then
        act1 : input := TRUE
        act2 : gate_command := locked
    end
Event   DECISION_alarm_on ≙
refines DECISION_lock
    when
        grd1 : input = FALSE
        grd2 : swipe = FALSE ∨ read_uid_buffer ∈ out_of_AuthDB
        grd3 : gate_sensor = opened
        grd4 : time − last_lock_time ≥ 15
    then
        act1 : input := TRUE
        act2 : gate_command := locked
        act3 : alarm_on := TRUE
    end
Event   DECISION_do_nothing ≙
extends DECISION_do_nothing
    when
        grd1 : input = FALSE
        grd2 : swipe = FALSE ∨ read_uid_buffer ∈ out_of_AuthDB
        grd4 : gate_sensor = opened
        grd3 : time − last_lock_time < 5
    then
        act1 : input := TRUE
    end
```

Event *DECISION_reset_time* $\widehat{=}$
extends *DECISION_reset_time*

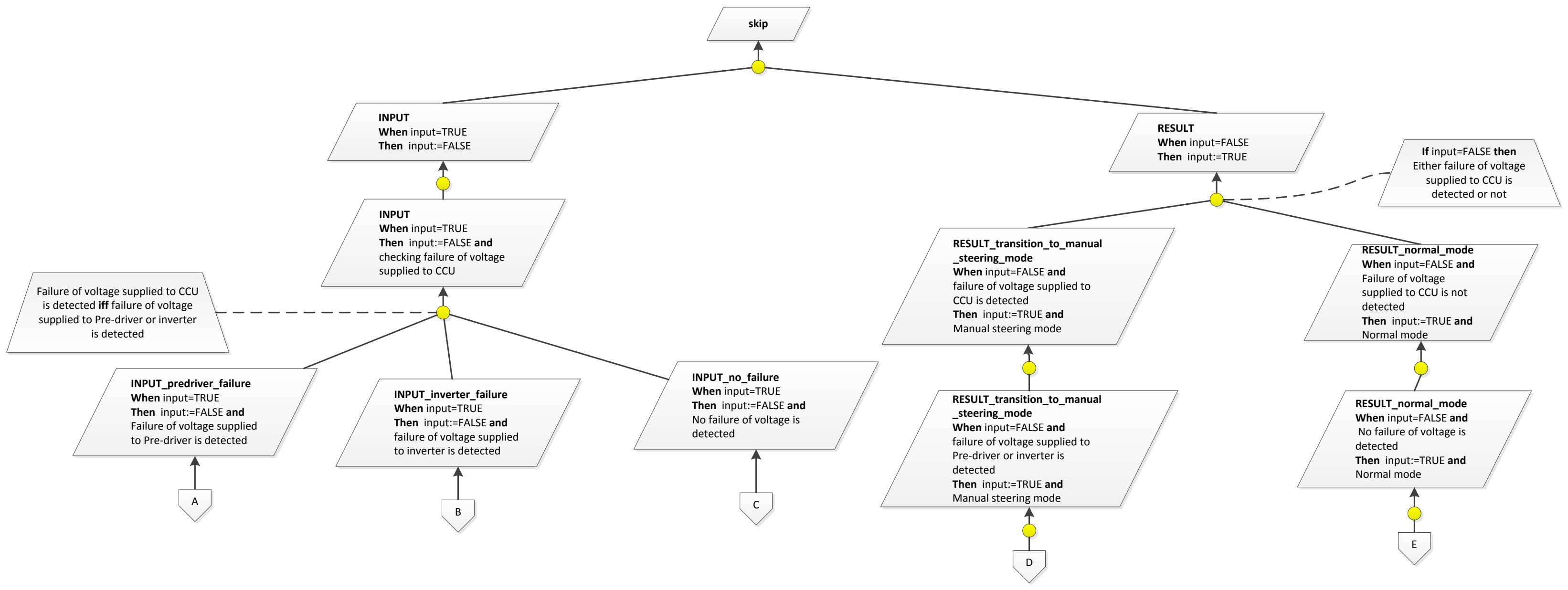
```
    when
        grd1 : input = FALSE
        grd2 : swipe = FALSE ∨ read_uid_buffer ∈ out_of_AuthDB
        grd3 : gate_sensor = locked
    then
        act1 : input := TRUE
        act2 : last_lock_time := time
        act3 : alarm_on := FALSE
    end
```

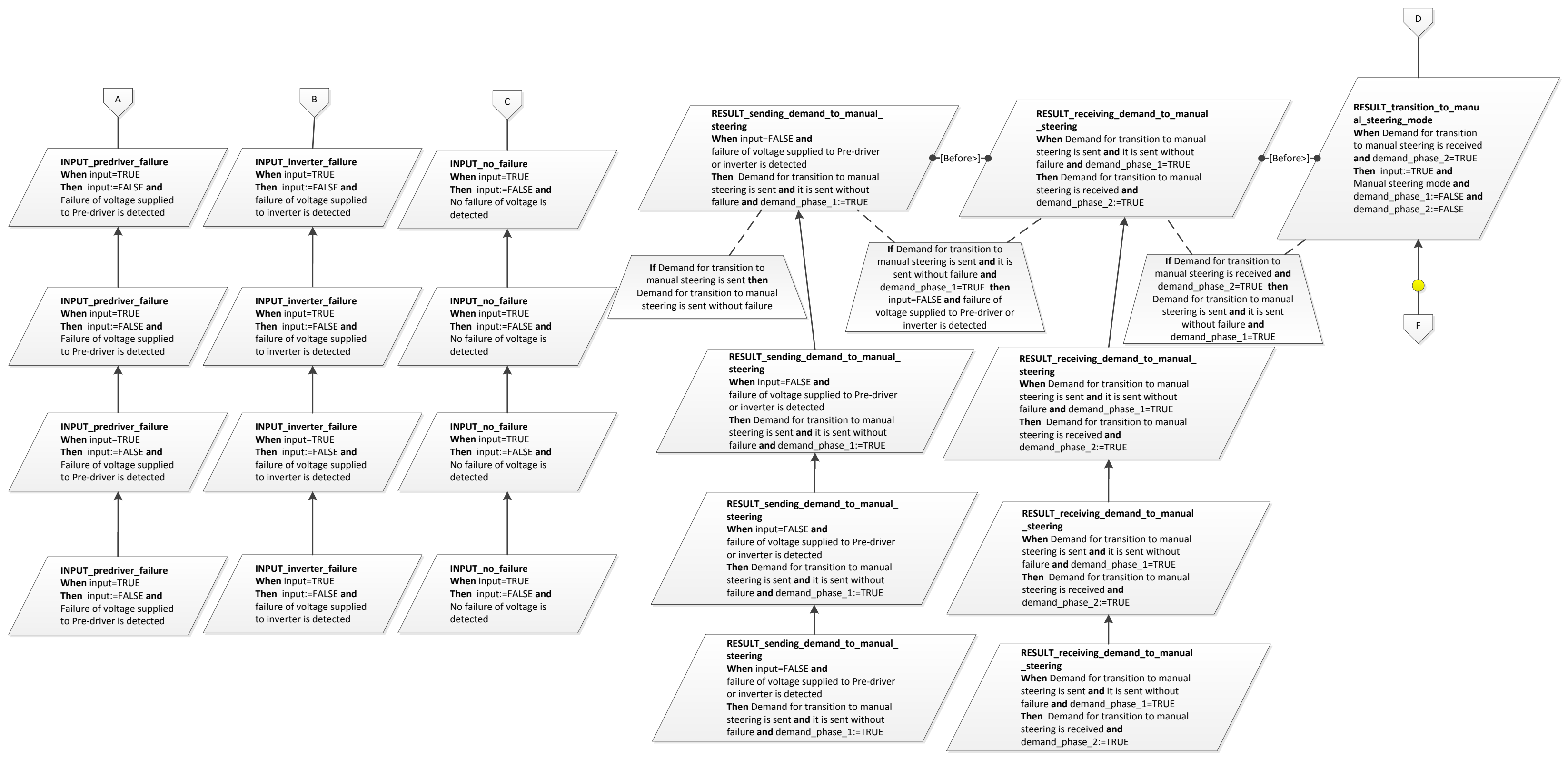
END

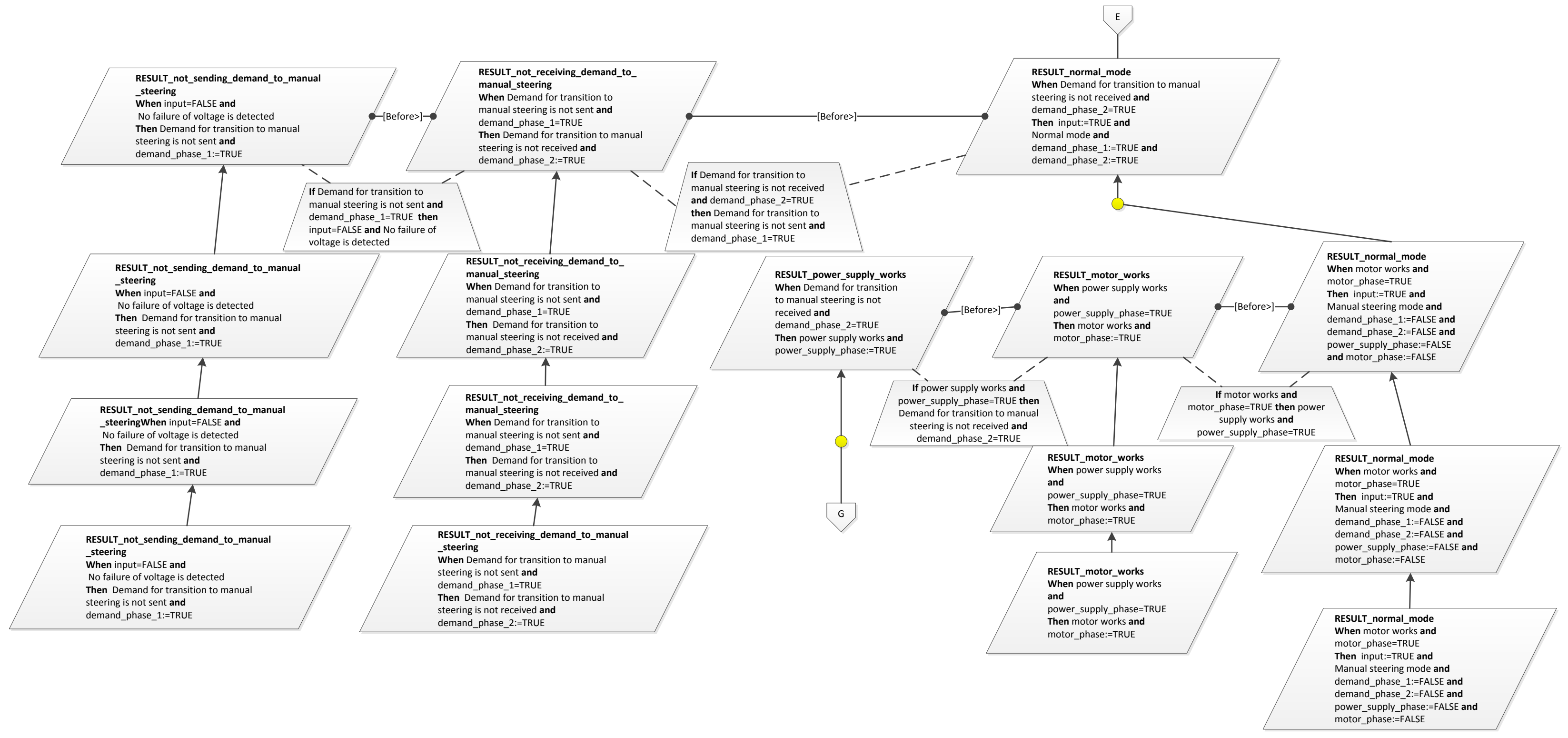
Appendix C

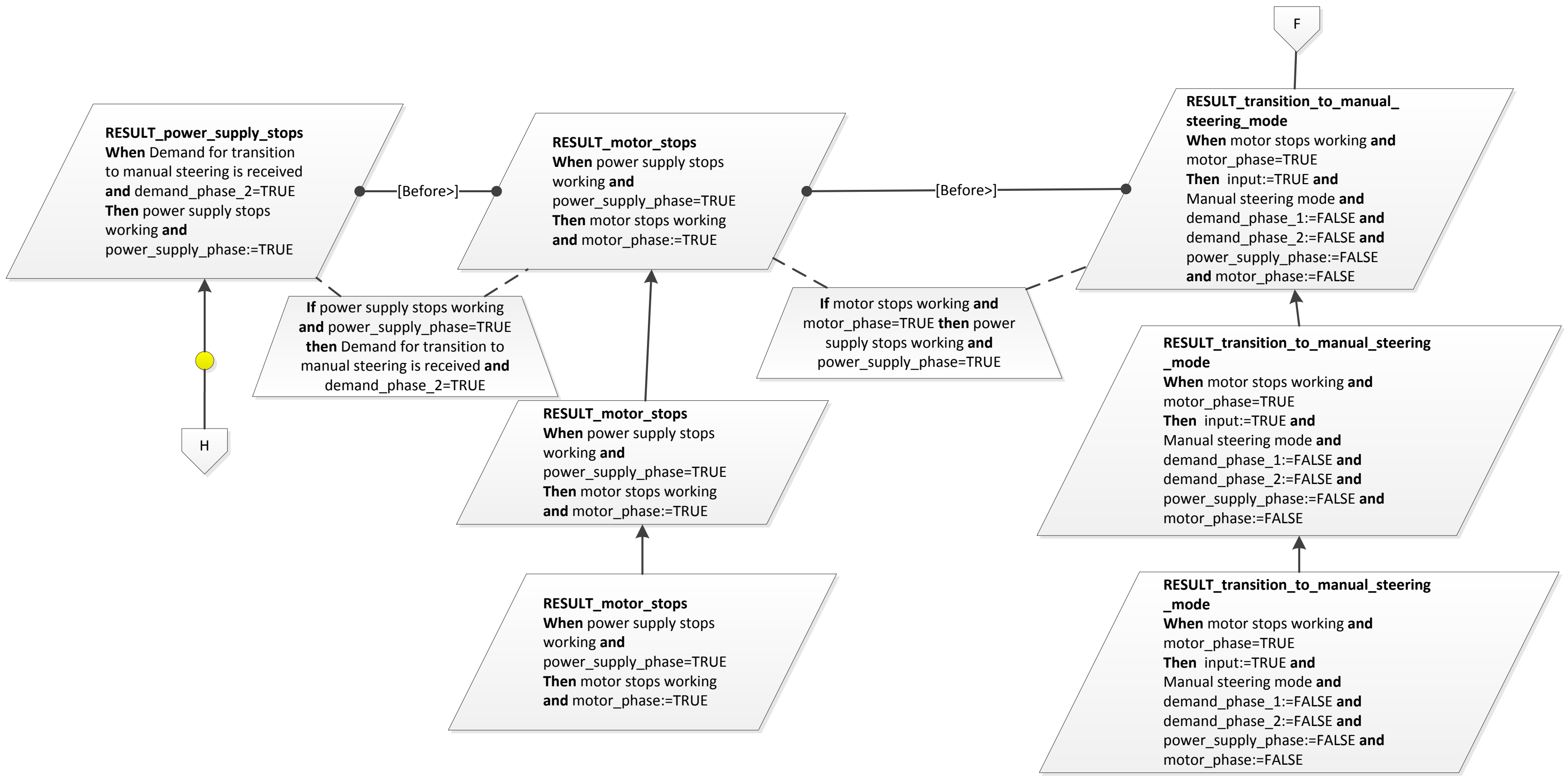
Electrical Power Steering (EPS) system case study

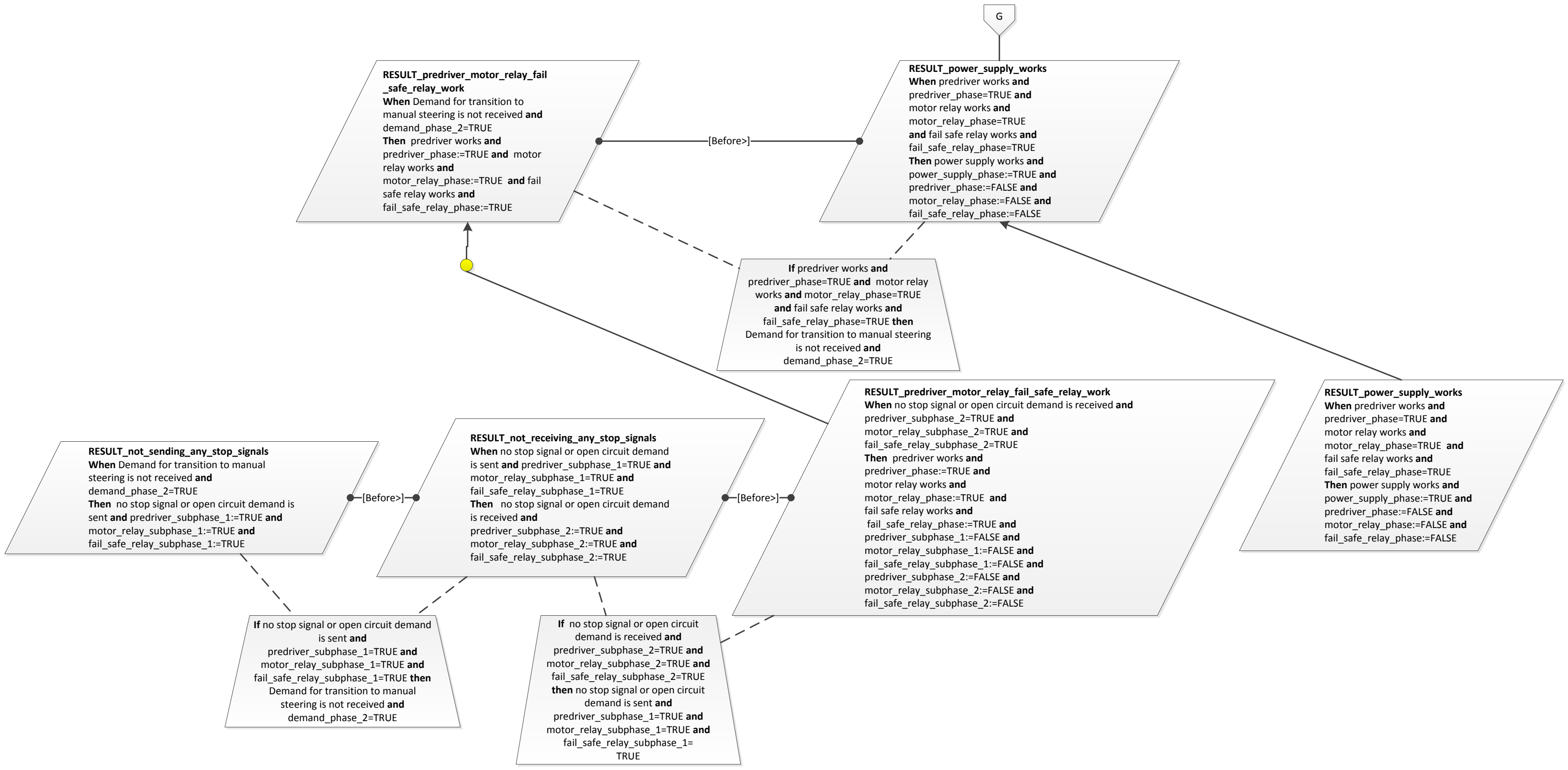
C.1 Refinement tree diagram

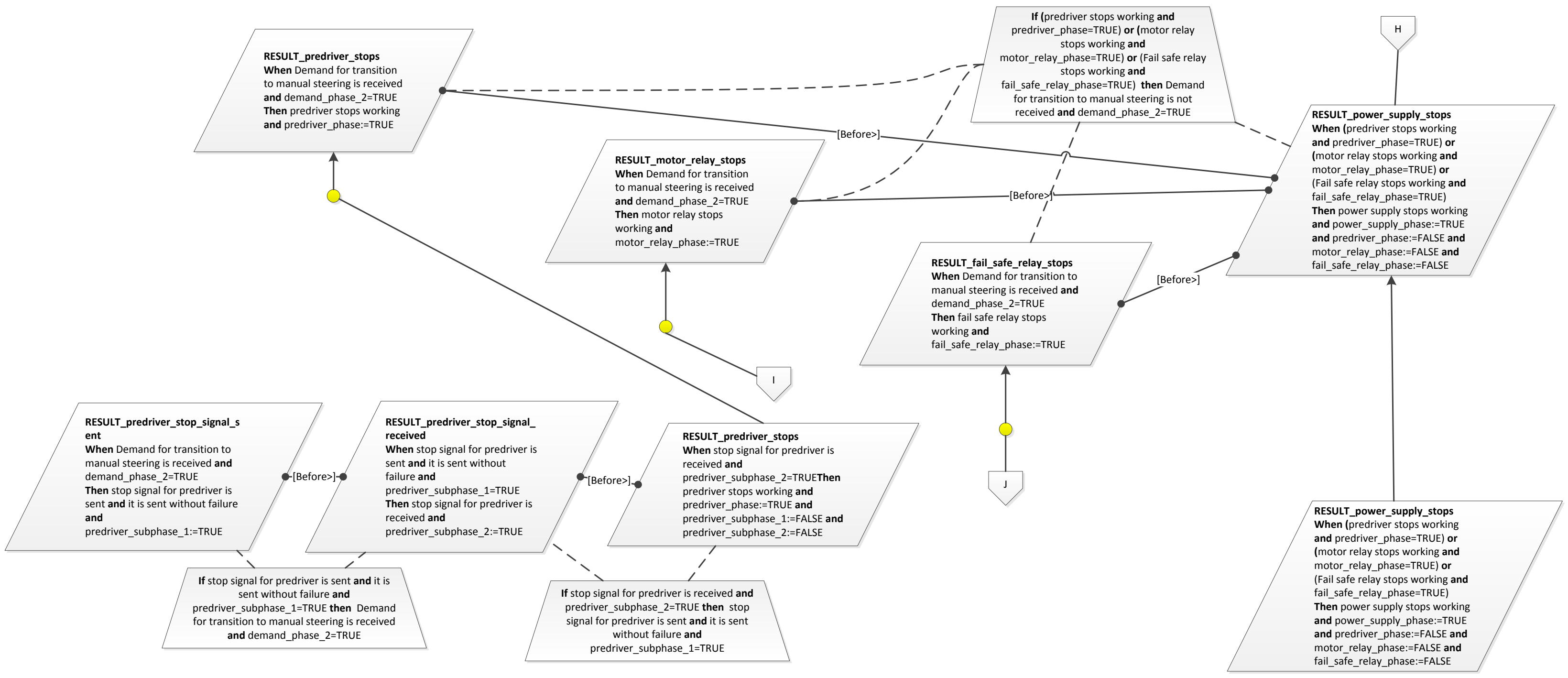

skip
INPUT When input=TRUE Then input:=FALSE
RESULT When input=FALSE Then input:=TRUE
If input=FALSE then Either failure of voltage supplied to CCU is detected or not
INPUT When input=TRUE Then input:=FALSE and checking failure of voltage supplied to CCU
Failure of voltage supplied to CCU is detected iff failure of voltage supplied to Pre-driver or inverter is detected
RESULT_transition_to_manual_steering_mode When input=FALSE and failure of voltage supplied to CCU is detected Then input:=TRUE and Manual steering mode
RESULT_normal_mode When input=FALSE and Failure of voltage supplied to CCU is not detected Then input:=TRUE and Normal mode
INPUT_predriver_failure When input=TRUE Then input:=FALSE and Failure of voltage supplied to Pre-driver is detected
INPUT_inverter_failure When input=TRUE Then input:=FALSE and failure of voltage supplied to inverter is detected
INPUT_no_failure When input=TRUE Then input:=FALSE and No failure of voltage is detected
RESULT_transition_to_manual_steering_mode When input=FALSE and failure of voltage supplied to Pre-driver or inverter is detected Then input:=TRUE and Manual steering mode
RESULT_normal_mode When input=FALSE and No failure of voltage is detected Then input:=TRUE and Normal mode
A
B
C
D
E

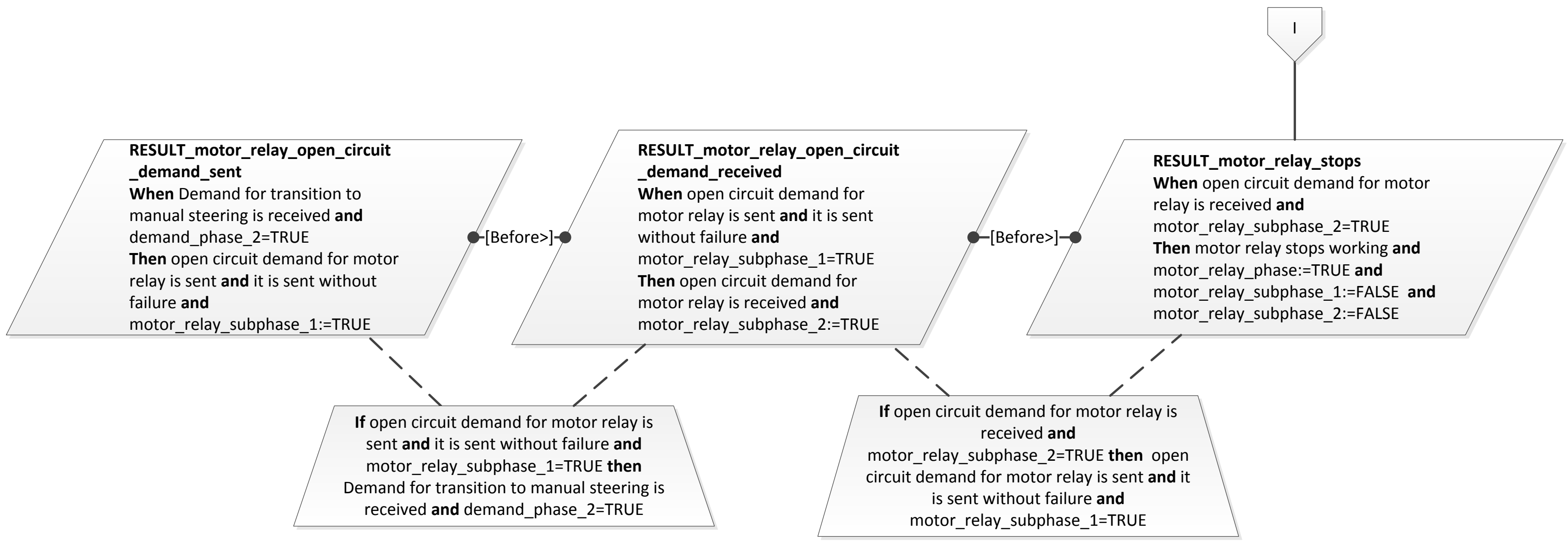

A
B
C
D
F
INPUT_predriver_failure When input=TRUE Then input:=FALSE and Failure of voltage supplied to Pre-driver is detected
INPUT_predriver_failure When input=TRUE Then input:=FALSE and Failure of voltage supplied to Pre-driver is detected
INPUT_predriver_failure When input=TRUE Then input:=FALSE and Failure of voltage supplied to Pre-driver is detected
INPUT_predriver_failure When input=TRUE Then input:=FALSE and Failure of voltage supplied to Pre-driver is detected
INPUT_inverter_failure When input=TRUE Then input:=FALSE and failure of voltage supplied to inverter is detected
INPUT_inverter_failure When input=TRUE Then input:=FALSE and failure of voltage supplied to inverter is detected
INPUT_inverter_failure When input=TRUE Then input:=FALSE and failure of voltage supplied to inverter is detected
INPUT_inverter_failure When input=TRUE Then input:=FALSE and failure of voltage supplied to inverter is detected
INPUT_no_failure When input=TRUE Then input:=FALSE and No failure of voltage is detected
INPUT_no_failure When input=TRUE Then input:=FALSE and No failure of voltage is detected
INPUT_no_failure When input=TRUE Then input:=FALSE and No failure of voltage is detected
INPUT_no_failure When input=TRUE Then input:=FALSE and No failure of voltage is detected
RESULT_sending_demand_to_manual_ steering When input=FALSE and failure of voltage supplied to Pre-driver or inverter is detected Then Demand for transition to manual steering is sent and it is sent without failure and demand_phase_1:=TRUE
[Before>]
RESULT_receiving_demand_to_manual _steering When Demand for transition to manual steering is sent and it is sent without failure and demand_phase_1=TRUE Then Demand for transition to manual steering is received and demand_phase_2:=TRUE
[Before>]
RESULT_transition_to_manu al_steering_mode When Demand for transition to manual steering is received and demand_phase_2=TRUE Then input:=TRUE and Manual steering mode and demand_phase_1:=FALSE and demand_phase_2:=FALSE
If Demand for transition to manual steering is sent then Demand for transition to manual steering is sent without failure
If Demand for transition to manual steering is sent and it is sent without failure and demand_phase_1=TRUE then input=FALSE and failure of voltage supplied to Pre-driver or inverter is detected
If Demand for transition to manual steering is received and demand_phase_2=TRUE then Demand for transition to manual steering is sent and it is sent without failure and demand_phase_1=TRUE
RESULT_sending_demand_to_manual_ steering When input=FALSE and failure of voltage supplied to Pre-driver or inverter is detected Then Demand for transition to manual steering is sent and it is sent without failure and demand_phase_1:=TRUE
RESULT_sending_demand_to_manual_ steering When input=FALSE and failure of voltage supplied to Pre-driver or inverter is detected Then Demand for transition to manual steering is sent and it is sent without failure and demand_phase_1:=TRUE
RESULT_sending_demand_to_manual_ steering When input=FALSE and failure of voltage supplied to Pre-driver or inverter is detected Then Demand for transition to manual steering is sent and it is sent without failure and demand_phase_1:=TRUE
RESULT_receiving_demand_to_manual_ steering When Demand for transition to manual steering is sent and it is sent without failure and demand_phase_1=TRUE Then Demand for transition to manual steering is received and demand_phase_2:=TRUE
RESULT_receiving_demand_to_manual _steering When Demand for transition to manual steering is sent and it is sent without failure and demand_phase_1=TRUE Then Demand for transition to manual steering is received and demand_phase_2:=TRUE
RESULT_receiving_demand_to_manual _steering When Demand for transition to manual steering is sent and it is sent without failure and demand_phase_1=TRUE Then Demand for transition to manual steering is received and demand_phase_2:=TRUE


E
RESULT_not_sending_demand_to_manual_steering When input=FALSE and No failure of voltage is detected Then Demand for transition to manual steering is not sent and demand_phase_1:=TRUE
[Before>]
RESULT_not_receiving_demand_to_manual_steering When Demand for transition to manual steering is not sent and demand_phase_1=TRUE Then Demand for transition to manual steering is not received and demand_phase_2:=TRUE
[Before>]
RESULT_normal_mode When Demand for transition to manual steering is not received and demand_phase_2=TRUE Then input:=TRUE and Normal mode and demand_phase_1:=TRUE and demand_phase_2:=TRUE
If Demand for transition to manual steering is not sent and demand_phase_1=TRUE then input=FALSE and No failure of voltage is detected
If Demand for transition to manual steering is not received and demand_phase_2=TRUE then Demand for transition to manual steering is not sent and demand_phase_1=TRUE
RESULT_not_sending_demand_to_manual_steering When input=FALSE and No failure of voltage is detected Then Demand for transition to manual steering is not sent and demand_phase_1:=TRUE
RESULT_not_receiving_demand_to_manual_steering When Demand for transition to manual steering is not sent and demand_phase_1=TRUE Then Demand for transition to manual steering is not received and demand_phase_2:=TRUE
RESULT_power_supply_works When Demand for transition to manual steering is not received and demand_phase_2=TRUE Then power supply works and power_supply_phase:=TRUE
[Before>]
RESULT_motor_works When power supply works and power_supply_phase=TRUE Then motor works and motor_phase:=TRUE
[Before>]
RESULT_normal_mode When motor works and motor_phase=TRUE Then input:=TRUE and Manual steering mode and demand_phase_1:=FALSE and demand_phase_2:=FALSE and power_supply_phase:=FALSE and motor_phase:=FALSE
If power supply works and power_supply_phase=TRUE then Demand for transition to manual steering is not received and demand_phase_2=TRUE
If motor works and motor_phase=TRUE then power supply works and power_supply_phase=TRUE
RESULT_not_sending_demand_to_manual_steeringWhen input=FALSE and No failure of voltage is detected Then Demand for transition to manual steering is not sent and demand_phase_1:=TRUE
RESULT_not_receiving_demand_to_manual_steering When Demand for transition to manual steering is not sent and demand_phase_1=TRUE Then Demand for transition to manual steering is not received and demand_phase_2:=TRUE
G
RESULT_motor_works When power supply works and power_supply_phase=TRUE Then motor works and motor_phase:=TRUE
RESULT_normal_mode When motor works and motor_phase=TRUE Then input:=TRUE and Manual steering mode and demand_phase_1:=FALSE and demand_phase_2:=FALSE and power_supply_phase:=FALSE and motor_phase:=FALSE
RESULT_not_sending_demand_to_manual_steering When input=FALSE and No failure of voltage is detected Then Demand for transition to manual steering is not sent and demand_phase_1:=TRUE
RESULT_not_receiving_demand_to_manual_steering When Demand for transition to manual steering is not sent and demand_phase_1=TRUE Then Demand for transition to manual steering is not received and demand_phase_2:=TRUE
RESULT_motor_works When power supply works and power_supply_phase=TRUE Then motor works and motor_phase:=TRUE
RESULT_normal_mode When motor works and motor_phase=TRUE Then input:=TRUE and Manual steering mode and demand_phase_1:=FALSE and demand_phase_2:=FALSE and power_supply_phase:=FALSE and motor_phase:=FALSE


F
RESULT_transition_to_manual_steering_mode When motor stops working and motor_phase=TRUE Then input:=TRUE and Manual steering mode and demand_phase_1:=FALSE and demand_phase_2:=FALSE and power_supply_phase:=FALSE and motor_phase:=FALSE
RESULT_power_supply_stops When Demand for transition to manual steering is received and demand_phase_2=TRUE Then power supply stops working and power_supply_phase:=TRUE
[Before>]
RESULT_motor_stops When power supply stops working and power_supply_phase=TRUE Then motor stops working and motor_phase:=TRUE
[Before>]
If power supply stops working and power_supply_phase=TRUE then Demand for transition to manual steering is received and demand_phase_2=TRUE
If motor stops working and motor_phase=TRUE then power supply stops working and power_supply_phase=TRUE
RESULT_transition_to_manual_steering_mode When motor stops working and motor_phase=TRUE Then input:=TRUE and Manual steering mode and demand_phase_1:=FALSE and demand_phase_2:=FALSE and power_supply_phase:=FALSE and motor_phase:=FALSE
RESULT_motor_stops When power supply stops working and power_supply_phase=TRUE Then motor stops working and motor_phase:=TRUE
H
RESULT_motor_stops When power supply stops working and power_supply_phase=TRUE Then motor stops working and motor_phase:=TRUE
RESULT_transition_to_manual_steering_mode When motor stops working and motor_phase=TRUE Then input:=TRUE and Manual steering mode and demand_phase_1:=FALSE and demand_phase_2:=FALSE and power_supply_phase:=FALSE and motor_phase:=FALSE


G
RESULT_predriver_motor_relay_fail _safe_relay_work
When Demand for transition to manual steering is not received and demand_phase_2=TRUE
Then predriver works and predriver_phase:=TRUE and motor relay works and motor_relay_phase:=TRUE and fail safe relay works and fail_safe_relay_phase:=TRUE
[Before>]
RESULT_power_supply_works
When predriver works and predriver_phase=TRUE and motor relay works and motor_relay_phase=TRUE and fail safe relay works and fail_safe_relay_phase=TRUE
Then power supply works and power_supply_phase:=TRUE and predriver_phase:=FALSE and motor_relay_phase:=FALSE and fail_safe_relay_phase:=FALSE
If predriver works and predriver_phase=TRUE and motor relay works and motor_relay_phase=TRUE and fail safe relay works and fail_safe_relay_phase=TRUE then Demand for transition to manual steering is not received and demand_phase_2=TRUE
RESULT_not_sending_any_stop_signals
When Demand for transition to manual steering is not received and demand_phase_2=TRUE
Then no stop signal or open circuit demand is sent and predriver_subphase_1:=TRUE and motor_relay_subphase_1:=TRUE and fail_safe_relay_subphase_1:=TRUE
[Before>]
RESULT_not_receiving_any_stop_signals
When no stop signal or open circuit demand is sent and predriver_subphase_1=TRUE and motor_relay_subphase_1=TRUE and fail_safe_relay_subphase_1=TRUE
Then no stop signal or open circuit demand is received and predriver_subphase_2:=TRUE and motor_relay_subphase_2:=TRUE and fail_safe_relay_subphase_2:=TRUE
[Before>]
RESULT_predriver_motor_relay_fail_safe_relay_work
When no stop signal or open circuit demand is received and predriver_subphase_2=TRUE and motor_relay_subphase_2=TRUE and fail_safe_relay_subphase_2=TRUE
Then predriver works and predriver_phase:=TRUE and motor relay works and motor_relay_phase:=TRUE and fail safe relay works and fail_safe_relay_phase:=TRUE and predriver_subphase_1:=FALSE and motor_relay_subphase_1:=FALSE and fail_safe_relay_subphase_1:=FALSE and predriver_subphase_2:=FALSE and motor_relay_subphase_2:=FALSE and fail_safe_relay_subphase_2:=FALSE
RESULT_power_supply_works
When predriver works and predriver_phase=TRUE and motor relay works and motor_relay_phase=TRUE and fail safe relay works and fail_safe_relay_phase=TRUE
Then power supply works and power_supply_phase:=TRUE and predriver_phase:=FALSE and motor_relay_phase:=FALSE and fail_safe_relay_phase:=FALSE
If no stop signal or open circuit demand is sent and predriver_subphase_1=TRUE and motor_relay_subphase_1=TRUE and fail_safe_relay_subphase_1=TRUE then Demand for transition to manual steering is not received and demand_phase_2=TRUE
If no stop signal or open circuit demand is received and predriver_subphase_2=TRUE and motor_relay_subphase_2=TRUE and fail_safe_relay_subphase_2=TRUE then no stop signal or open circuit demand is sent and predriver_subphase_1=TRUE and motor_relay_subphase_1=TRUE and fail_safe_relay_subphase_1= TRUE


RESULT_predriver_stops
When Demand for transition to manual steering is received and demand_phase_2=TRUE
Then predriver stops working and predriver_phase:=TRUE
If (predriver stops working and predriver_phase=TRUE) or (motor relay stops working and motor_relay_phase=TRUE) or (Fail safe relay stops working and fail_safe_relay_phase=TRUE) then Demand for transition to manual steering is not received and demand_phase_2=TRUE
H
RESULT_power_supply_stops
When (predriver stops working and predriver_phase=TRUE) or (motor relay stops working and motor_relay_phase=TRUE) or (Fail safe relay stops working and fail_safe_relay_phase=TRUE)
Then power supply stops working and power_supply_phase:=TRUE and predriver_phase:=FALSE and motor_relay_phase:=FALSE and fail_safe_relay_phase:=FALSE
[Before>]
RESULT_motor_relay_stops
When Demand for transition to manual steering is received and demand_phase_2=TRUE
Then motor relay stops working and motor_relay_phase:=TRUE
[Before>]
RESULT_fail_safe_relay_stops
When Demand for transition to manual steering is received and demand_phase_2=TRUE
Then fail safe relay stops working and fail_safe_relay_phase:=TRUE
[Before>]
I
J
RESULT_predriver_stop_signal_sent
When Demand for transition to manual steering is received and demand_phase_2=TRUE
Then stop signal for predriver is sent and it is sent without failure and predriver_subphase_1:=TRUE
[Before>]
RESULT_predriver_stop_signal_received
When stop signal for predriver is sent and it is sent without failure and predriver_subphase_1=TRUE
Then stop signal for predriver is received and predriver_subphase_2:=TRUE
[Before>]
RESULT_predriver_stops
When stop signal for predriver is received and predriver_subphase_2=TRUEThen predriver stops working and predriver_phase:=TRUE and predriver_subphase_1:=FALSE and predriver_subphase_2:=FALSE
If stop signal for predriver is sent and it is sent without failure and predriver_subphase_1=TRUE then Demand for transition to manual steering is received and demand_phase_2=TRUE
If stop signal for predriver is received and predriver_subphase_2=TRUE then stop signal for predriver is sent and it is sent without failure and predriver_subphase_1=TRUE
RESULT_power_supply_stops
When (predriver stops working and predriver_phase=TRUE) or (motor relay stops working and motor_relay_phase=TRUE) or (Fail safe relay stops working and fail_safe_relay_phase=TRUE)
Then power supply stops working and power_supply_phase:=TRUE and predriver_phase:=FALSE and motor_relay_phase:=FALSE and fail_safe_relay_phase:=FALSE



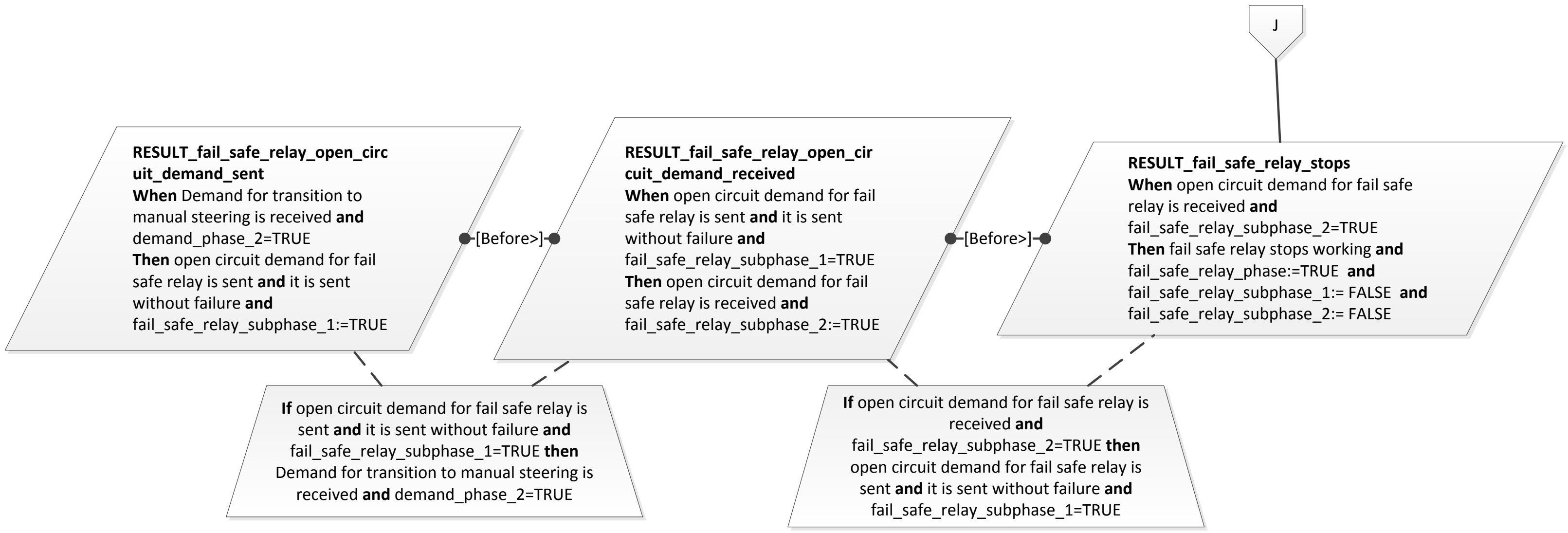
RESULT_motor_relay_open_circuit _demand_sent
When Demand for transition to manual steering is received **and** demand_phase_2=TRUE
Then open circuit demand for motor relay is sent **and** it is sent without failure **and** motor_relay_subphase_1:=TRUE

RESULT_motor_relay_open_circuit _demand_received
When open circuit demand for motor relay is sent **and** it is sent without failure **and** motor_relay_subphase_1=TRUE
Then open circuit demand for motor relay is received **and** motor_relay_subphase_2:=TRUE

RESULT_motor_relay_stops
When open circuit demand for motor relay is received **and** motor_relay_subphase_2=TRUE
Then motor relay stops working **and** motor_relay_phase:=TRUE **and** motor_relay_subphase_1:=FALSE **and** motor_relay_subphase_2:=FALSE

If open circuit demand for motor relay is sent **and** it is sent without failure **and** motor_relay_subphase_1=TRUE **then** Demand for transition to manual steering is received **and** demand_phase_2=TRUE

If open circuit demand for motor relay is received **and** motor_relay_subphase_2=TRUE **then** open circuit demand for motor relay is sent **and** it is sent without failure **and** motor_relay_subphase_1=TRUE



RESULT_fail_safe_relay_open_circ uit_demand_sent
When Demand for transition to manual steering is received **and** demand_phase_2=TRUE
Then open circuit demand for fail safe relay is sent **and** it is sent without failure **and** fail_safe_relay_subphase_1:=TRUE

RESULT_fail_safe_relay_open_cir cuit_demand_received
When open circuit demand for fail safe relay is sent **and** it is sent without failure **and** fail_safe_relay_subphase_1=TRUE
Then open circuit demand for fail safe relay is received **and** fail_safe_relay_subphase_2:=TRUE

RESULT_fail_safe_relay_stops
When open circuit demand for fail safe relay is received **and** fail_safe_relay_subphase_2=TRUE
Then fail safe relay stops working **and** fail_safe_relay_phase:=TRUE **and** fail_safe_relay_subphase_1:= FALSE **and** fail_safe_relay_subphase_2:= FALSE

If open circuit demand for fail safe relay is sent **and** it is sent without failure **and** fail_safe_relay_subphase_1=TRUE **then** Demand for transition to manual steering is received **and** demand_phase_2=TRUE

If open circuit demand for fail safe relay is received **and** fail_safe_relay_subphase_2=TRUE **then** open circuit demand for fail safe relay is sent **and** it is sent without failure **and** fail_safe_relay_subphase_1=TRUE

C.2 Descriptions of events

The descriptions of the copying events of each level of the tree are omitted here.

C.2.1 Initial model

Event name	**Description**
INPUT	the input phase
RESULT	the decision phase

C.2.2 First refinement

Event name	**Description**
INPUT	checking the failure of the voltage supplied to CCU
RESULT_transition_to_manual_steering_mode	transition to the 'Manual Steering' mode when the failure is detected
RESULT_normal_mode	keeping in the normal mode when no failure is detected

C.2.3 Second refinement

Event name	**Description**
INPUT_inverter_failure	detecting the failure of the voltage supplied to the inverter
INPUT_predriver_failure	checking the failure of the voltage supplied to the predriver
INPUT_no_failure	detecting no failure
RESULT_transition_to_manual_steering_mode	transition to the 'Manual Steering' mode when one of the failure is detected
RESULT_normal_mode	keeping in the normal mode when no failure is detected

C.2.4 Third refinement

Event name	Description
RESULT_sending_demand_to_manual_steering	sending demand for transition to the ‘Manual Steering’ mode without failure when one of the failure is detected
RESULT_receiving_demand_to_manual_steering	receiving the demand when it is sent without failure
RESULT_transition_to_manual_steering_mode	transition to the ‘Manual Steering’ mode when the demand is received
RESULT_not_sending_demand_to_manual_steering	not sending the demand for transition to the ‘Manual Steering’ mode when no failure is detected
RESULT_not_receiving_demand_to_manual_steering	not receiving the demand for transition to the ‘Manual Steering’ mode when it is not sent
RESULT_normal_mode	keeping in the normal mode when no demand for transition to the ‘Manual Steering’ mode is received

C.2.5 Fourth refinement

Event name	Description
RESULT_power_supply_stops	stopping the power supply when the demand for transition to the ‘Manual Steering’ mode is received
RESULT_motor_stops	stopping the motor when the power supply stops
RESULT_transition_to_manual_steering_mode	transition to the ‘Manual Steering’ mode when the motor stops
RESULT_power_supply_works	turning the power supply on when the demand for transition to the ‘Manual Steering’ mode is not received
RESULT_motor_works	actuating the motor when the power supply works
RESULT_normal_mode	keeping in the normal mode when the motor works

C.2.6 Fifth refinement

Event name	**Description**
RESULT_predriver_stops	stopping the pre-driver when the demand for transition to the 'Manual Steering' mode is received
RESULT_motor_relay_stops	stopping the motor relay when the demand for transition to the 'Manual Steering' mode is received
RESULT_fail_safe_relay _stops	stopping the fail-safe relay when the demand for transition to the 'Manual Steering' mode is received
RESULT_power_supply _stops	stopping the power supply when one of the components stops
RESULT_predriver_motor _relay_fail_safe _relay_work	keeping all the three components working when the demand for transition to the 'Manual Steering' mode is not received
RESULT_power_supply _works	turning the power supply on all the three components work

C.2.7 Sixth refinement

Event name	Description
RESULT_predriver_stop _signal_sent	sending the stop signal to the pre-driver without failure when the demand for transition to the 'Manual Steering' mode is received
RESULT_predriver_stop _signal_received	receiving the stop demand when it is sent without failure
RESULT_predriver_stops	stopping the pre-driver when the stop demand is received
RESULT_motor_relay _open_circuit_demand _sent	sending the open circuit demand to the motor relay without failure when the demand for transition to the 'Manual Steering' mode is received
RESULT_motor_relay _open_circuit_demand _received	receiving the open cicuit demand when the demand is sent without failure
RESULT_motor_relay _stops	stopping the motor relay when open circuit demand is received
RESULT_fail_safe_relay _open_circuit_demand _sent	sending the open circuit demand to the fail-safe relay without failure when the demand for transition to the 'Manual Steering' mode is received
RESULT_fail_safe_relay _open_circuit_demand _received	receiving the open circuit demand when the demand is sent without failure
RESULT_fail_safe_relay _stops	stopping the fail-safe relay when the open circuit demand is received
RESULT_not_sending_any _stop_signals	not sending any stop demand and open circuit demand when the demand for transition to the 'Manual Steering' mode is not received
RESULT_not_receiving _any_stop_signals	not receiving any stop demand and open circuit demand when none of them is sent
RESULT_predriver_motor _relay_fail_safe _relay_work	keeping all the three components working when no stop signal and open circuit demand is received

C.2.8 Seventh refinement

Event name	Description
RESULT_predriver_enable _signal_not_sent	not sending the enable signal to the pre-driver when the demand for transition to the 'Manual Steering' mode is received
RESULT_predriver_enable _signal_not_received	not receiving the enable signal when it is not sent
RESULT_predriver_stops	stopping the pre-driver when the enable signal is not received
RESULT_motor_relay _enable_signal_not_sent	not sending the enable signal to the motor relay when the demand for transition to the 'Manual Steering' mode is received
RESULT_motor_relay _enable_signal_not_received	not receiving the enable when the demand is not sent
RESULT_motor_relay _stops	stopping the motor relay when the enable signal is not received
RESULT_fail_safe_relay _enable_signal_not_sent	not sending the enable to the fail-safe relay when the demand for transition to the 'Manual Steering' mode is received
RESULT_fail_safe_relay _enable_signal_not_received	not receiving the enable signal when the demand is not sent
RESULT_fail_safe_relay _stops	stopping the fail-safe relay when the enable signal is not received
RESULT_sending_all _enable_signals	sending all enable signals to all the three components without failure when the demand for transition to the 'Manual Steering' mode is not received
RESULT_receiving_all _enable_signals	receiving all enable signals when all of them are sent without failure
RESULT_predriver_motor _relay_fail_safe _relay_work	keeping all the three components working when all enable signals are received

C.3 Descriptions of variables

Component name	**Description**
CCU_failure	whether the failure of the voltage applied to CCU is detected
manual_steering_mode	whether the system is in the 'Manual Steering' mode
inverter_failure	whether the failure of the voltage applied to the inverter is detected
predriver_failure	whether the failure of the voltage applied to pre-driver is detected
demand_to_manual_steering_sent	whether the demand for transition to the 'Manual Steering' mode is sent
demand_to_manual_steering_sent_without_failure	whether the demand for transition to the 'Manual Steering' mode is sent without failure
demand_to_manual_steering_received	whether the demand for transition to the 'Manual Steering' mode is received
power_supply_working	whether the power supply is working
motor_working	whether the motor is working
predriver_working	whether the pre-driver is working
motor_relay_working	whether the motor relay is working
fail_safe_relay_working	whether the fail-safe relay is working
predriver_stop_signal_sent	whether the stop signal for pre-driver is sent
motor_relay_open_circuit_demand_sent	whether the open circuit demand for the motor relay is sent
fail_safe_relay_open_circuit_demand_sent	whether the open circuit demand for the fail-safe relay is sent
predriver_stop_signal_sent_without_failure	whether the stop signal for the pre-driver is sent without failure
motor_relay_open_circuit_demand_sent_without_failure	whether the open circuit demand for the motor relay is sent without failure
fail_safe_relay_open_circuit_demand_sent_without_failure	whether the open circuit demand for the fail-safe relay is sent without failure
predriver_stop_signal_received	whether the stop signal for pre-driver is received
motor_relay_open_circuit_demand_received	whether the open circuit demand for the motor relay is received
fail_safe_relay_open_circuit_demand_received	whether the open circuit demand for the fail-safe relay is received
predriver_enable_signal_sent	whether the enable signal for the pre-driver is sent

Component name	**Description**
predriver_enable_signal _received	whether the enable signal for the pre-driver relay is received
predriver_enable_signal_sent _without_failure	whether the enable signal for the pre-driver is sent without failure
motor_relay_enable _signal_sent	whether the enable signal for the motor relay is sent
motor_relay_enable _signal_sent _without_failure	whether the enable signal for the motor relay is sent without failure
motor_relay_enable _signal_received	whether the enable signal for the motor relay is received
fail_safe_relay_enable _signal_sent	whether the enable signal for the fail-safe relay is sent
fail_safe_relay_enable _signal_sent _without_failure	whether the enable signal for the fail-safe relay is sent without failure
fail_safe_relay_enable _signal_received	whether the enable signal for the fail-safe relay is received

The variables representing phases:
input
demand_phase_1
demand_phase_2
power_supply_phase
motor_phase
predriver_phase
motor_relay_phase
fail_safe_relay_phase
predriver_subphase_1
predriver_subphase_2
motor_relay_subphase_1
motor_relay_subphase_2
fail_safe_relay_subphase_1
fail_safe_relay_subphase_2

C.4 Event-B specification

C.4.1 Initial model

MACHINE Machine_1
VARIABLES

```
    input
```

INVARIANTS

    inv1 : $input \in BOOL$

EVENTS

Initialisation

    begin

        act1 : $input := TRUE$

    end

Event $INPUT \ \widehat{=}$

    when

        grd1 : $input = TRUE$

    then

        act1 : $input := FALSE$

    end

Event $RESULT \ \widehat{=}$

    when

        grd1 : $input = FALSE$

    then

        act1 : $input := TRUE$

    end

END

C.4.2 First refinement

MACHINE Machine_2

REFINES Machine_1

VARIABLES

    `input`

    `CCU_failure`

    `manual_steering_mode`

INVARIANTS

    inv1 : $CCU\_failure \in BOOL$

    inv2 : $manual\_steering\_mode \in BOOL$

    inv3 : $input = FALSE \Rightarrow ((CCU\_failure = TRUE \lor CCU\_failure = FALSE) \land \neg(CCU\_failure = TRUE \land CCU\_failure = FALSE))$

EVENTS

Initialisation

    extended

    begin

        act1 : `input` := `TRUE`

```
        act2 : CCU_failure := FALSE
        act3 : manual_steering_mode := FALSE
    end
Event  INPUT_failure ≙
extends  INPUT
    when
        grd1 : input = TRUE
    then
        act1 : input := FALSE
        act2 : CCU_failure :∈ BOOL
    end
Event  RESULT_transition_to_manual_steering_mode ≙
extends  RESULT
    when
        grd1 : input = FALSE
        grd2 : CCU_failure = TRUE
    then
        act1 : input := TRUE
        act2 : manual_steering_mode := TRUE
    end
Event  RESULT_normal_mode ≙
extends  RESULT
    when
        grd1 : input = FALSE
        grd2 : CCU_failure = FALSE
    then
        act1 : input := TRUE
        act2 : manual_steering_mode := FALSE
    end
END
```

C.4.3 Second refinement

```
MACHINE  Machine_3
REFINES  Machine_2
VARIABLES
    input
    manual_steering_mode
    inverter_failure
    predriver_failure
```

INVARIANTS

inv1 : $inverter\_failure \in BOOL$

inv2 : $predriver\_failure \in BOOL$

inv3 : $(inverter\_failure = TRUE \lor predriver\_failure = TRUE) \Leftrightarrow CCU\_failure = TRUE$

EVENTS

Initialisation

begin

act1 : $input := TRUE$

act3 : $manual\_steering\_mode := FALSE$

act4 : $inverter\_failure := FALSE$

act5 : $predriver\_failure := FALSE$

end

Event $INPUT\_inverter\_failure \mathrel{\widehat{=}}$

refines $INPUT\_failure$

when

grd1 : $input = TRUE$

with

CCU_failure′ : CCU_failure′ = TRUE

then

act1 : $input := FALSE$

act2 : $inverter\_failure := TRUE$

end

Event $INPUT\_predriver\_failure \mathrel{\widehat{=}}$

refines $INPUT\_failure$

when

grd1 : $input = TRUE$

with

CCU_failure′ : CCU_failure′ = TRUE

then

act1 : $input := FALSE$

act2 : $predriver\_failure := TRUE$

end

Event $INPUT\_no\_failure \mathrel{\widehat{=}}$

refines $INPUT\_failure$

when

grd1 : $input = TRUE$

with

CCU_failure′ : CCU_failure′ = FALSE

then

act1 : $input := FALSE$
act2 : $inverter\_failure := FALSE$
act3 : $predriver\_failure := FALSE$
end

Event $RESULT\_transition\_to\_manual\_steering\_mode \mathrel{\widehat{=}}$
refines $RESULT\_transition\_to\_manual\_steering\_mode$
when
grd1 : $input = FALSE$
grd2 : $inverter\_failure = TRUE \lor predriver\_failure = TRUE$
then
act1 : $input := TRUE$
act2 : $manual\_steering\_mode := TRUE$
end

Event $RESULT\_normal\_mode \mathrel{\widehat{=}}$
refines $RESULT\_normal\_mode$
when
grd1 : $input = FALSE$
grd2 : $inverter\_failure = FALSE \land predriver\_failure = FALSE$
then
act1 : $input := TRUE$
act2 : $manual\_steering\_mode := FALSE$
end

END

C.4.4 Third refinement

MACHINE Machine_4
REFINES Machine_3
VARIABLES

```
input
manual_steering_mode
inverter_failure
predriver_failure
demand_to_manual_steering_sent
demand_to_manual_steering_sent_without_failure
demand_to_manual_steering_received
demand_phase_1
demand_phase_2
```

INVARIANTS

inv1 : $demand\_to\_manual\_steering\_sent \in BOOL$

inv2 : $demand\_to\_manual\_steering\_sent\_without\_failure \in BOOL$

inv3 : $demand\_to\_manual\_steering\_received \in BOOL$

inv4 : $demand\_to\_manual\_steering\_sent = TRUE \Rightarrow demand\_to\_manual\_steering\_sent\_without\_failure = TRUE$

inv5 : $demand\_phase\_1 \in BOOL$

inv6 : $demand\_phase\_2 \in BOOL$

inv8 : $demand\_phase\_2 = TRUE \wedge demand\_to\_manual\_steering\_received = TRUE \Rightarrow demand\_phase\_1 = TRUE \wedge demand\_to\_manual\_steering\_sent = TRUE \wedge demand\_phase\_1 = TRUE$

inv9 : $demand\_phase\_1 = TRUE \wedge demand\_to\_manual\_steering\_sent = TRUE \wedge demand\_to\_manual\_steering\_sent\_without\_failure = TRUE \Rightarrow input = FALSE \wedge (inverter\_failure = TRUE \vee predriver\_failure = TRUE)$

inv10 : $demand\_phase\_2 = TRUE \wedge demand\_to\_manual\_steering\_received = FALSE \Rightarrow demand\_phase\_1 = TRUE \wedge demand\_to\_manual\_steering\_sent = FALSE$

inv11 : $demand\_phase\_1 = TRUE \wedge demand\_to\_manual\_steering\_sent = FALSE \Rightarrow input = FALSE \wedge inverter\_failure = FALSE \wedge predriver\_failure = FALSE$

EVENTS

Initialisation

extended

begin

act1 : `input := TRUE`

act3 : `manual_steering_mode := FALSE`

act4 : `inverter_failure := FALSE`

act5 : `predriver_failure := FALSE`

act6 : $demand\_to\_manual\_steering\_sent := FALSE$

act7 : $demand\_to\_manual\_steering\_sent\_without\_failure := FALSE$

act8 : $demand\_to\_manual\_steering\_received := FALSE$

act9 : $demand\_phase\_1 := FALSE$

act10 : $demand\_phase\_2 := FALSE$

end

Event $INPUT\_inverter\_failure \,\widehat{=}$

extends $INPUT\_inverter\_failure$

when

grd1 : `input = TRUE`

then

act1 : `input := FALSE`

act2 : `inverter_failure := TRUE`

end

Event $INPUT\_predriver\_failure \,\widehat{=}$

extends $INPUT\_predriver\_failure$

```
    when
        grd1 : input = TRUE
    then
        act1 : input := FALSE
        act2 : predriver_failure := TRUE
    end
Event  INPUT_no_failure ≙
extends  INPUT_no_failure
    when
        grd1 : input = TRUE
    then
        act1 : input := FALSE
        act2 : inverter_failure := FALSE
        act3 : predriver_failure := FALSE
    end
Event  RESULT_sending_demand_to_manual_steering ≙
    when
        grd1 : input = FALSE
        grd2 : inverter_failure = TRUE ∨ predriver_failure = TRUE
    then
        act1 : demand_to_manual_steering_sent := TRUE
        act2 : demand_to_manual_steering_sent_without_failure := TRUE
        act3 : demand_phase_1 := TRUE
    end
Event  RESULT_receiving_demand_to_manual_steering ≙
    when
        grd3 : demand_to_manual_steering_sent = TRUE
        grd4 : demand_to_manual_steering_sent_without_failure = TRUE
        grd5 : demand_phase_1 = TRUE
    then
        act1 : demand_to_manual_steering_received := TRUE
        act2 : demand_phase_2 := TRUE
    end
Event  RESULT_transition_to_manual_steering_mode ≙
refines  RESULT_transition_to_manual_steering_mode
    when
        grd4 : demand_phase_2 = TRUE
        grd3 : demand_to_manual_steering_received = TRUE
    then
        act1 : input := TRUE
        act2 : manual_steering_mode := TRUE
        act3 : $demand\_phase\_1 := FALSE$
        act4 : $demand\_phase\_2 := FALSE$
    end
Event  $RESULT\_not\_sending\_demand\_to\_manual\_steering \widehat{=}$
    when
        grd1 : $input = FALSE$
        grd2 : $inverter\_failure = FALSE \land predriver\_failure = FALSE$
    then
        act1 : $demand\_to\_manual\_steering\_sent := FALSE$
        act2 : $demand\_phase\_1 := TRUE$
    end
Event  $RESULT\_not\_receiving\_demand\_to\_manual\_steering \widehat{=}$
    when
        grd2 : $demand\_phase\_1 = TRUE$
        grd3 : $demand\_to\_manual\_steering\_sent = FALSE$
    then
        act1 : $demand\_to\_manual\_steering\_received := FALSE$
        act2 : $demand\_phase\_2 := TRUE$
    end
Event  $RESULT\_normal\_mode \widehat{=}$
refines  $RESULT\_normal\_mode$
    when
        grd3 : $demand\_to\_manual\_steering\_received = FALSE$
        grd4 : $demand\_phase\_2 = TRUE$
    then
        act1 : $input := TRUE$
        act2 : $manual\_steering\_mode := FALSE$
        act3 : $demand\_phase\_1 := FALSE$
        act4 : $demand\_phase\_2 := FALSE$
    end
END
```

C.4.5 Fourth refinement

```
MACHINE  Machine_5
REFINES  Machine_4
VARIABLES
    input
    manual_steering_mode
    inverter_failure
    predriver_failure
```

demand_to_manual_steering_sent
demand_to_manual_steering_sent_without_failure
demand_to_manual_steering_received
demand_phase_1
demand_phase_2
power_supply_working
motor_working
power_supply_phase
motor_phase

INVARIANTS

inv1 : $power\_supply\_working \in BOOL$
inv2 : $motor\_working \in BOOL$
inv3 : $power\_supply\_phase \in BOOL$
inv4 : $motor\_phase \in BOOL$
inv5 : $motor\_phase = TRUE \land motor\_working = FALSE \Rightarrow power\_supply\_phase = TRUE \land power\_supply\_working = FALSE$
inv6 : $power\_supply\_phase = TRUE \land power\_supply\_working = FALSE \Rightarrow demand\_phase\_2 = TRUE \land demand\_to\_manual\_steering\_received = TRUE$
inv7 : $motor\_phase = TRUE \land motor\_working = TRUE \Rightarrow power\_supply\_phase = TRUE \land power\_supply\_working = TRUE$
inv8 : $power\_supply\_phase = TRUE \land power\_supply\_working = TRUE \Rightarrow demand\_phase\_2 = TRUE \land demand\_to\_manual\_steering\_received = FALSE$

EVENTS

Initialisation

extended

begin

act1 : input := TRUE
act3 : manual_steering_mode := FALSE
act4 : inverter_failure := FALSE
act5 : predriver_failure := FALSE
act6 : demand_to_manual_steering_sent := FALSE
act7 : demand_to_manual_steering_sent_without_failure := FALSE
act8 : demand_to_manual_steering_received := FALSE
act9 : demand_phase_1 := FALSE
act10 : demand_phase_2 := FALSE
act11 : $power\_supply\_working := TRUE$
act12 : $motor\_working := TRUE$
act13 : $power\_supply\_phase := FALSE$
act14 : $motor\_phase := FALSE$

end

Event *INPUT_inverter_failure* $\widehat{=}$
extends *INPUT_inverter_failure*
 when
  grd1 : input = TRUE
 then
  act1 : input := FALSE
  act2 : inverter_failure := TRUE
 end
Event *INPUT_predriver_failure* $\widehat{=}$
extends *INPUT_predriver_failure*
 when
  grd1 : input = TRUE
 then
  act1 : input := FALSE
  act2 : predriver_failure := TRUE
 end
Event *INPUT_no_failure* $\widehat{=}$
extends *INPUT_no_failure*
 when
  grd1 : input = TRUE
 then
  act1 : input := FALSE
  act2 : inverter_failure := FALSE
  act3 : predriver_failure := FALSE
 end
Event *RESULT_sending_demand_to_manual_steering* $\widehat{=}$
extends *RESULT_sending_demand_to_manual_steering*
 when
  grd1 : input = FALSE
  grd2 : inverter_failure = TRUE $\vee$ predriver_failure = TRUE
 then
  act1 : demand_to_manual_steering_sent := TRUE
  act2 : demand_to_manual_steering_sent_without_failure := TRUE
  act3 : demand_phase_1 := TRUE
 end
Event *RESULT_receiving_demand_to_manual_steering* $\widehat{=}$
extends *RESULT_receiving_demand_to_manual_steering*
 when
  grd3 : demand_to_manual_steering_sent = TRUE
  grd4 : demand_to_manual_steering_sent_without_failure = TRUE

```
        grd5 : demand_phase_1 = TRUE
    then
        act1 : demand_to_manual_steering_received := TRUE
        act2 : demand_phase_2 := TRUE
    end
Event  RESULT_power_supply_stops ≙
    when
        grd1 : demand_phase_2 = TRUE
        grd2 : demand_to_manual_steering_received = TRUE
    then
        act1 : power_supply_phase := TRUE
        act2 : power_supply_working := FALSE
    end
Event  RESULT_motor_stops ≙
    when
        grd1 : power_supply_phase = TRUE
        grd2 : power_supply_working = FALSE
    then
        act1 : motor_phase := TRUE
        act2 : motor_working := FALSE
    end
Event  RESULT_transition_to_manual_steering_mode ≙
refines  RESULT_transition_to_manual_steering_mode
    when
        grd4 : motor_phase = TRUE
        grd3 : motor_working = FALSE
    then
        act1 : input := TRUE
        act2 : manual_steering_mode := TRUE
        act3 : demand_phase_1 := FALSE
        act4 : demand_phase_2 := FALSE
        act5 : motor_phase := FALSE
        act6 : power_supply_phase := FALSE
    end
Event  RESULT_not_sending_demand_to_manual_steering ≙
extends  RESULT_not_sending_demand_to_manual_steering
    when
        grd1 : input = FALSE
        grd2 : inverter_failure = FALSE ∧ predriver_failure = FALSE
    then
        act1 : demand_to_manual_steering_sent := FALSE
        act2 : demand_phase_1 := TRUE
    end
Event   RESULT_not_receiving_demand_to_manual_steering ≙
extends RESULT_not_receiving_demand_to_manual_steering
    when
        grd2 : demand_phase_1 = TRUE
        grd3 : demand_to_manual_steering_sent = FALSE
    then
        act1 : demand_to_manual_steering_received := FALSE
        act2 : demand_phase_2 := TRUE
    end
Event   RESULT_power_supply_works ≙
    when
        grd1 : demand_phase_2 = TRUE
        grd2 : demand_to_manual_steering_received = FALSE
    then
        act1 : power_supply_phase := TRUE
        act2 : power_supply_working := TRUE
    end
Event   RESULT_motor_works ≙
    when
        grd1 : power_supply_phase = TRUE
        grd2 : power_supply_working = TRUE
    then
        act1 : motor_phase := TRUE
        act2 : motor_working := TRUE
    end
Event   RESULT_normal_mode ≙
refines RESULT_normal_mode
    when
        grd3 : motor_phase = TRUE
        grd4 : motor_working = TRUE
    then
        act1 : input := TRUE
        act2 : manual_steering_mode := FALSE
        act3 : demand_phase_1 := FALSE
        act4 : demand_phase_2 := FALSE
        act5 : power_supply_phase := FALSE
        act6 : motor_phase := FALSE
    end
END
```

C.4.6 Fifth refinement

MACHINE Machine_6

REFINES Machine_5

VARIABLES

```
input
manual_steering_mode
inverter_failure
predriver_failure
demand_to_manual_steering_sent
demand_to_manual_steering_sent_without_failure
demand_to_manual_steering_received
demand_phase_1
demand_phase_2
power_supply_working
motor_working
power_supply_phase
motor_phase
predriver_working
motor_relay_working
fail_safe_relay_working
predriver_phase
motor_relay_phase
fail_safe_relay_phase
```

INVARIANTS

inv1 : $predriver\_working \in BOOL$

inv2 : $motor\_relay\_working \in BOOL$

inv3 : $fail\_safe\_relay\_working \in BOOL$

inv4 : $((predriver\_phase = TRUE \wedge predriver\_working = FALSE) \vee (motor\_relay\_phase = TRUE \wedge motor\_relay\_working = FALSE) \vee (fail\_safe\_relay\_phase = TRUE \wedge fail\_safe\_relay\_working = FALSE)) \Rightarrow (demand\_phase\_2 = TRUE \wedge demand\_to\_manual\_steering\_received = TRUE)$

inv5 : $predriver\_phase \in BOOL$

inv6 : $motor\_relay\_phase \in BOOL$

inv7 : $fail\_safe\_relay\_phase \in BOOL$

inv8 : $motor\_relay\_working = TRUE \wedge motor\_relay\_phase = TRUE \wedge predriver\_phase = TRUE \wedge predriver\_working = TRUE \wedge fail\_safe\_relay\_phase = TRUE \wedge fail\_safe\_relay\_working = TRUE \Rightarrow demand\_phase\_2 = TRUE \wedge demand\_to\_manual\_steering\_received = FALSE$

```
EVENTS
Initialisation
          extended
          begin
                act1 : input := TRUE
                act3 : manual_steering_mode := FALSE
                act4 : inverter_failure := FALSE
                act5 : predriver_failure := FALSE
                act6 : demand_to_manual_steering_sent := FALSE
                act7 : demand_to_manual_steering_sent_without_failure := FALSE
                act8 : demand_to_manual_steering_received := FALSE
                act9 : demand_phase_1 := FALSE
                act10 : demand_phase_2 := FALSE
                act11 : power_supply_working := TRUE
                act12 : motor_working := TRUE
                act13 : power_supply_phase := FALSE
                act14 : motor_phase := FALSE
                act15 : predriver_working := TRUE
                act16 : motor_relay_working := TRUE
                act17 : fail_safe_relay_working := TRUE
                act18 : predriver_phase := FALSE
                act19 : motor_relay_phase := FALSE
                act20 : fail_safe_relay_phase := FALSE
          end
Event   INPUT_inverter_failure ≙
extends INPUT_inverter_failure
          when
                grd1 : input = TRUE
          then
                act1 : input := FALSE
                act2 : inverter_failure := TRUE
          end
Event   INPUT_predriver_failure ≙
extends INPUT_predriver_failure
          when
                grd1 : input = TRUE
          then
                act1 : input := FALSE
                act2 : predriver_failure := TRUE
          end
Event   INPUT_no_failure ≙
```

```
extends INPUT_no_failure
      when
            grd1 : input = TRUE
      then
            act1 : input := FALSE
            act2 : inverter_failure := FALSE
            act3 : predriver_failure := FALSE
      end
Event   RESULT_sending_demand_to_manual_steering ≙
extends RESULT_sending_demand_to_manual_steering
      when
            grd1 : input = FALSE
            grd2 : inverter_failure = TRUE ∨ predriver_failure = TRUE
      then
            act1 : demand_to_manual_steering_sent := TRUE
            act2 : demand_to_manual_steering_sent_without_failure := TRUE
            act3 : demand_phase_1 := TRUE
      end
Event   RESULT_receiving_demand_to_manual_steering ≙
extends RESULT_receiving_demand_to_manual_steering
      when
            grd3 : demand_to_manual_steering_sent = TRUE
            grd4 : demand_to_manual_steering_sent_without_failure = TRUE
            grd5 : demand_phase_1 = TRUE
      then
            act1 : demand_to_manual_steering_received := TRUE
            act2 : demand_phase_2 := TRUE
      end
Event   RESULT_predriver_stops ≙
      when
            grd1 : demand_phase_2 = TRUE
            grd2 : demand_to_manual_steering_received = TRUE
      then
            act1 : predriver_working := FALSE
            act2 : predriver_phase := TRUE
      end
Event   RESULT_motor_relay_stops ≙
      when
            grd1 : demand_phase_2 = TRUE
            grd2 : demand_to_manual_steering_received = TRUE
      then
```

```
        act1 : $motor\_relay\_working := FALSE$
        act2 : $motor\_relay\_phase := TRUE$
    end
Event   $RESULT\_fail\_safe\_relay\_stops \widehat{=}$
    when
        grd1 : $demand\_phase\_2 = TRUE$
        grd2 : $demand\_to\_manual\_steering\_received = TRUE$
    then
        act1 : $fail\_safe\_relay\_working := FALSE$
        act2 : $fail\_safe\_relay\_phase := TRUE$
    end
Event   $RESULT\_power\_supply\_stops \widehat{=}$
refines $RESULT\_power\_supply\_stops$
    when
        grd1 : $(predriver\_phase = TRUE \land predriver\_working = FALSE) \lor (motor\_relay\_phase =$
               $TRUE \land motor\_relay\_working = FALSE) \lor (fail\_safe\_relay\_phase = TRUE \land$
               $fail\_safe\_relay\_working = FALSE)$
    then
        act1 : $power\_supply\_phase := TRUE$
        act2 : $power\_supply\_working := FALSE$
        act3 : $predriver\_phase := FALSE$
        act4 : $motor\_relay\_phase := FALSE$
        act5 : $fail\_safe\_relay\_phase := FALSE$
    end
Event   $RESULT\_motor\_stops \widehat{=}$
extends $RESULT\_motor\_stops$
    when
        grd1 : power_supply_phase = TRUE
        grd2 : power_supply_working = FALSE
    then
        act1 : motor_phase := TRUE
        act2 : motor_working := FALSE
    end
Event   $RESULT\_transition\_to\_manual\_steering\_mode \widehat{=}$
extends $RESULT\_transition\_to\_manual\_steering\_mode$
    when
        grd4 : motor_phase = TRUE
        grd3 : motor_working = FALSE
    then
        act1 : input := TRUE
        act2 : manual_steering_mode := TRUE
```

```
          act3 : demand_phase_1 := FALSE
          act4 : demand_phase_2 := FALSE
          act5 : motor_phase := FALSE
          act6 : power_supply_phase := FALSE
     end
Event   RESULT_not_sending_demand_to_manual_steering ≙
extends  RESULT_not_sending_demand_to_manual_steering
     when
          grd1 : input = FALSE
          grd2 : inverter_failure = FALSE ∧ predriver_failure = FALSE
     then
          act1 : demand_to_manual_steering_sent := FALSE
          act2 : demand_phase_1 := TRUE
     end
Event   RESULT_not_receiving_demand_to_manual_steering ≙
extends  RESULT_not_receiving_demand_to_manual_steering
     when
          grd2 : demand_phase_1 = TRUE
          grd3 : demand_to_manual_steering_sent = FALSE
     then
          act1 : demand_to_manual_steering_received := FALSE
          act2 : demand_phase_2 := TRUE
     end
Event   RESULT_predriver_motor_relay_fail_safe_relay_work ≙
     when
          grd1 : demand_phase_2 = TRUE
          grd2 : demand_to_manual_steering_received = FALSE
     then
          act1 : motor_relay_working := TRUE
          act2 : predriver_working := TRUE
          act3 : fail_safe_relay_working := TRUE
          act4 : motor_relay_phase := TRUE
          act5 : predriver_phase := TRUE
          act6 : fail_safe_relay_phase := TRUE
     end
Event   RESULT_power_supply_works ≙
refines  RESULT_power_supply_works
     when
          grd1 : motor_relay_working = TRUE
          grd2 : motor_relay_phase = TRUE
          grd3 : predriver_working = TRUE
```

```
        grd4 : predriver_phase = TRUE
        grd5 : fail_safe_relay_working = TRUE
        grd6 : fail_safe_relay_phase = TRUE
    then
        act1 : power_supply_phase := TRUE
        act2 : power_supply_working := TRUE
        act3 : predriver_phase := FALSE
        act4 : motor_relay_phase := FALSE
        act5 : fail_safe_relay_phase := FALSE
    end
Event  RESULT_motor_works ≙
extends RESULT_motor_works
    when
        grd1 : power_supply_phase = TRUE
        grd2 : power_supply_working = TRUE
    then
        act1 : motor_phase := TRUE
        act2 : motor_working := TRUE
    end
Event  RESULT_normal_mode ≙
extends RESULT_normal_mode
    when
        grd3 : motor_phase = TRUE
        grd4 : motor_working = TRUE
    then
        act1 : input := TRUE
        act2 : manual_steering_mode := FALSE
        act3 : demand_phase_1 := FALSE
        act4 : demand_phase_2 := FALSE
        act5 : power_supply_phase := FALSE
        act6 : motor_phase := FALSE
    end
END
```

C.4.7 Sixth refinement

```
MACHINE  Machine_7
REFINES  Machine_6
VARIABLES
    input
    manual_steering_mode
```

`inverter_failure`
`predriver_failure`
`demand_to_manual_steering_sent`
`demand_to_manual_steering_sent_without_failure`
`demand_to_manual_steering_received`
`demand_phase_1`
`demand_phase_2`
`power_supply_working`
`motor_working`
`power_supply_phase`
`motor_phase`
`predriver_working`
`motor_relay_working`
`fail_safe_relay_working`
`predriver_phase`
`motor_relay_phase`
`fail_safe_relay_phase`
`predriver_stop_signal_sent`
`motor_relay_open_circuit_demand_sent`
`fail_safe_relay_open_circuit_demand_sent`
`predriver_subphase_1`
`predriver_subphase_2`
`motor_relay_subphase_1`
`motor_relay_subphase_2`
`fail_safe_relay_subphase_1`
`fail_safe_relay_subphase_2`
`predriver_stop_signal_sent_without_failure`
`motor_relay_open_circuit_demand_sent_without_failure`
`fail_safe_relay_open_circuit_demand_sent_without_failure`
`predriver_stop_signal_received`
`motor_relay_open_circuit_demand_received`
`fail_safe_relay_open_circuit_demand_received`

INVARIANTS

inv1 : $predriver\_stop\_signal\_sent \in BOOL$
inv2 : $motor\_relay\_open\_circuit\_demand\_sent \in BOOL$
inv3 : $fail\_safe\_relay\_open\_circuit\_demand\_sent \in BOOL$
inv4 : $predriver\_subphase\_1 \in BOOL$
inv5 : $predriver\_subphase\_2 \in BOOL$

inv6 : $motor\_relay\_subphase\_1 \in BOOL$

inv7 : $motor\_relay\_subphase\_2 \in BOOL$

inv8 : $fail\_safe\_relay\_subphase\_1 \in BOOL$

inv9 : $fail\_safe\_relay\_subphase\_2 \in BOOL$

inv10 : $predriver\_stop\_signal\_sent\_without\_failure \in BOOL$

inv11 : $motor\_relay\_open\_circuit\_demand\_sent\_without\_failure \in BOOL$

inv12 : $fail\_safe\_relay\_open\_circuit\_demand\_sent\_without\_failure \in BOOL$

inv13 : $predriver\_stop\_signal\_received \in BOOL$

inv14 : $motor\_relay\_open\_circuit\_demand\_received \in BOOL$

inv15 : $fail\_safe\_relay\_open\_circuit\_demand\_received \in BOOL$

inv16 : $predriver\_subphase\_2 = TRUE \land predriver\_stop\_signal\_received = TRUE \Rightarrow predriver\_subphase\_1 = TRUE \land predriver\_stop\_signal\_sent = TRUE \land predriver\_stop\_signal\_sent\_without\_failure = TRUE$

inv17 : $predriver\_stop\_signal\_sent = TRUE \land predriver\_stop\_signal\_sent\_without\_failure = TRUE \land predriver\_subphase\_1 = TRUE \Rightarrow demand\_phase\_2 = TRUE \land demand\_to\_manual\_steering\_received = TRUE$

inv18 : $predriver\_stop\_signal\_sent = TRUE \Rightarrow predriver\_stop\_signal\_sent\_without\_failure = TRUE$

inv19 : $motor\_relay\_open\_circuit\_demand\_received = TRUE \land motor\_relay\_subphase\_2 = TRUE \Rightarrow motor\_relay\_open\_circuit\_demand\_sent = TRUE \land motor\_relay\_open\_circuit\_demand\_sent\_without\_failure = TRUE \land motor\_relay\_subphase\_1 = TRUE$

inv20 : $motor\_relay\_open\_circuit\_demand\_sent = TRUE \land motor\_relay\_open\_circuit\_demand\_sent\_without\_failure = TRUE \land motor\_relay\_subphase\_1 = TRUE \Rightarrow demand\_phase\_2 = TRUE \land demand\_to\_manual\_steering\_received = TRUE$

inv21 : $motor\_relay\_open\_circuit\_demand\_sent = TRUE \Rightarrow motor\_relay\_open\_circuit\_demand\_sent\_without\_failure = TRUE$

inv22 : $fail\_safe\_relay\_subphase\_2 = TRUE \land fail\_safe\_relay\_open\_circuit\_demand\_received = TRUE \Rightarrow fail\_safe\_relay\_open\_circuit\_demand\_sent = TRUE \land fail\_safe\_relay\_open\_circuit\_demand\_sent\_without\_failure = TRUE \land fail\_safe\_relay\_subphase\_1 = TRUE$

inv23 : $fail\_safe\_relay\_open\_circuit\_demand\_sent = TRUE \land fail\_safe\_relay\_open\_circuit\_demand\_sent\_without\_failure = TRUE \land fail\_safe\_relay\_subphase\_1 = TRUE \Rightarrow demand\_phase\_2 = TRUE \land demand\_to\_manual\_steering\_received = TRUE$

inv24 : $fail\_safe\_relay\_open\_circuit\_demand\_sent = TRUE \Rightarrow demand\_to\_manual\_steering\_sent\_without\_failure = TRUE$

inv25 : $motor\_relay\_open\_circuit\_demand\_received = FALSE \land predriver\_stop\_signal\_received = FALSE \land fail\_safe\_relay\_open\_circuit\_demand\_received = FALSE \land motor\_relay\_subphase\_2 = TRUE \land predriver\_subphase\_2 = TRUE \land$

$fail\_safe\_relay\_subphase\_2 = TRUE \Rightarrow motor\_relay\_open\_circuit\_demand\_sent = FALSE \wedge predriver\_stop\_signal\_sent = FALSE \wedge fail\_safe\_relay\_open\_circuit\_demand\_sent = FALSE \wedge motor\_relay\_subphase\_1 = TRUE \wedge predriver\_subphase\_1 = TRUE \wedge fail\_safe\_relay\_subphase\_1 = TRUE$

inv26 : $motor\_relay\_open\_circuit\_demand\_sent = FALSE \wedge predriver\_stop\_signal\_sent = FALSE \wedge fail\_safe\_relay\_open\_circuit\_demand\_sent = FALSE \wedge motor\_relay\_subphase\_1 = TRUE \wedge predriver\_subphase\_1 = TRUE \wedge fail\_safe\_relay\_subphase\_1 = TRUE \Rightarrow demand\_phase\_2 = TRUE \wedge demand\_to\_manual\_steering\_received = FALSE$

EVENTS

Initialisation

extended

begin

act1 : input := TRUE
act3 : manual_steering_mode := FALSE
act4 : inverter_failure := FALSE
act5 : predriver_failure := FALSE
act6 : demand_to_manual_steering_sent := FALSE
act7 : demand_to_manual_steering_sent_without_failure := FALSE
act8 : demand_to_manual_steering_received := FALSE
act9 : demand_phase_1 := FALSE
act10 : demand_phase_2 := FALSE
act11 : power_supply_working := TRUE
act12 : motor_working := TRUE
act13 : power_supply_phase := FALSE
act14 : motor_phase := FALSE
act15 : predriver_working := TRUE
act16 : motor_relay_working := TRUE
act17 : fail_safe_relay_working := TRUE
act18 : predriver_phase := FALSE
act19 : motor_relay_phase := FALSE
act20 : fail_safe_relay_phase := FALSE
act21 : $predriver\_stop\_signal\_sent := FALSE$
act22 : $motor\_relay\_open\_circuit\_demand\_sent := FALSE$
act23 : $fail\_safe\_relay\_open\_circuit\_demand\_sent := FALSE$
act24 : $predriver\_subphase\_1 := FALSE$
act25 : $predriver\_subphase\_2 := FALSE$
act26 : $motor\_relay\_subphase\_1 := FALSE$
act27 : $motor\_relay\_subphase\_2 := FALSE$
act28 : $fail\_safe\_relay\_subphase\_1 := FALSE$
act29 : $fail\_safe\_relay\_subphase\_2 := FALSE$
act30 : $predriver\_stop\_signal\_sent\_without\_failure := TRUE$
act31 : $motor\_relay\_open\_circuit\_demand\_sent\_without\_failure := TRUE$
act32 : $fail\_safe\_relay\_open\_circuit\_demand\_sent\_without\_failure := TRUE$

```
            act33 : predriver_stop_signal_received := FALSE
            act34 : motor_relay_open_circuit_demand_received := FALSE
            act35 : fail_safe_relay_open_circuit_demand_received := FALSE
        end
Event   INPUT_inverter_failure ≙
extends INPUT_inverter_failure
        when
            grd1 : input = TRUE
        then
            act1 : input := FALSE
            act2 : inverter_failure := TRUE
        end
Event   INPUT_predriver_failure ≙
extends INPUT_predriver_failure
        when
            grd1 : input = TRUE
        then
            act1 : input := FALSE
            act2 : predriver_failure := TRUE
        end
Event   INPUT_no_failure ≙
extends INPUT_no_failure
        when
            grd1 : input = TRUE
        then
            act1 : input := FALSE
            act2 : inverter_failure := FALSE
            act3 : predriver_failure := FALSE
        end
Event   RESULT_sending_demand_to_manual_steering ≙
extends RESULT_sending_demand_to_manual_steering
        when
            grd1 : input = FALSE
            grd2 : inverter_failure = TRUE ∨ predriver_failure = TRUE
        then
            act1 : demand_to_manual_steering_sent := TRUE
            act2 : demand_to_manual_steering_sent_without_failure := TRUE
            act3 : demand_phase_1 := TRUE
        end
Event   RESULT_receiving_demand_to_manual_steering ≙
```

extends *RESULT_receiving_demand_to_manual_steering*

```
      when
            grd3 : demand_to_manual_steering_sent = TRUE
            grd4 : demand_to_manual_steering_sent_without_failure = TRUE
            grd5 : demand_phase_1 = TRUE
      then
            act1 : demand_to_manual_steering_received := TRUE
            act2 : demand_phase_2 := TRUE
      end
Event  RESULT_predriver_stop_signal_sent ≙
      when
            grd1 : demand_phase_2 = TRUE
            grd2 : demand_to_manual_steering_received = TRUE
      then
            act1 : predriver_stop_signal_sent := TRUE
            act2 : predriver_stop_signal_sent_without_failure := TRUE
            act3 : predriver_subphase_1 := TRUE
      end
Event  RESULT_predriver_stop_signal_received ≙
      when
            grd1 : predriver_stop_signal_sent = TRUE
            grd2 : predriver_stop_signal_sent_without_failure = TRUE
            grd3 : predriver_subphase_1 = TRUE
      then
            act1 : predriver_stop_signal_received := TRUE
            act2 : predriver_subphase_2 := TRUE
      end
Event  RESULT_predriver_stops ≙
refines  RESULT_predriver_stops
      when
            grd1 : predriver_subphase_2 = TRUE
            grd2 : predriver_stop_signal_received = TRUE
      then
            act1 : predriver_working := FALSE
            act2 : predriver_phase := TRUE
            act3 : predriver_subphase_1 := FALSE
            act4 : predriver_subphase_2 := FALSE
      end
Event  RESULT_motor_relay_open_circuit_demand_sent ≙
      when
            grd1 : demand_phase_2 = TRUE
            grd2 : demand_to_manual_steering_received = TRUE
        then
            act1 : motor_relay_open_circuit_demand_sent := TRUE
            act2 : motor_relay_open_circuit_demand_sent_without_failure := TRUE
            act3 : motor_relay_subphase_1 := TRUE
        end
Event   RESULT_motor_relay_open_circuit_demand_received ≙
        when
            grd1 : motor_relay_open_circuit_demand_sent = TRUE
            grd2 : motor_relay_open_circuit_demand_sent_without_failure = TRUE
            grd3 : motor_relay_subphase_1 = TRUE
        then
            act1 : motor_relay_open_circuit_demand_received := TRUE
            act2 : motor_relay_subphase_2 := TRUE
        end
Event   RESULT_motor_relay_stops ≙
refines RESULT_motor_relay_stops
        when
            grd1 : motor_relay_subphase_2 = TRUE
            grd2 : motor_relay_open_circuit_demand_received = TRUE
        then
            act1 : motor_relay_working := FALSE
            act2 : motor_relay_phase := TRUE
            act3 : motor_relay_subphase_1 := FALSE
            act4 : motor_relay_subphase_2 := FALSE
        end
Event   RESULT_fail_safe_relay_open_circuit_demand_sent ≙
        when
            grd1 : demand_phase_2 = TRUE
            grd2 : demand_to_manual_steering_received = TRUE
        then
            act1 : fail_safe_relay_open_circuit_demand_sent := TRUE
            act2 : fail_safe_relay_open_circuit_demand_sent_without_failure := TRUE
            act3 : fail_safe_relay_subphase_1 := TRUE
        end
Event   RESULT_fail_safe_relay_open_circuit_demand_received ≙
        when
            grd1 : fail_safe_relay_open_circuit_demand_sent = TRUE
            grd2 : fail_safe_relay_open_circuit_demand_sent_without_failure = TRUE
            grd3 : fail_safe_relay_subphase_1 = TRUE
        then
```

```
            act1 : fail_safe_relay_open_circuit_demand_received := TRUE
            act2 : fail_safe_relay_subphase_2 := TRUE
        end
Event   RESULT_fail_safe_relay_stops ≙
refines RESULT_fail_safe_relay_stops
        when
            grd1 : fail_safe_relay_subphase_2 = TRUE
            grd2 : fail_safe_relay_open_circuit_demand_received = TRUE
        then
            act1 : fail_safe_relay_working := FALSE
            act2 : fail_safe_relay_phase := TRUE
            act3 : fail_safe_relay_subphase_2 := FALSE
            act4 : fail_safe_relay_subphase_1 := FALSE
        end
Event   RESULT_power_supply_stops ≙
extends RESULT_power_supply_stops
        when
            grd1 : (predriver_phase = TRUE ∧ predriver_working = FALSE) ∨
                (motor_relay_phase = TRUE ∧ motor_relay_working = FALSE) ∨
                (fail_safe_relay_phase = TRUE ∧ fail_safe_relay_working = FALSE)
        then
            act1 : power_supply_phase := TRUE
            act2 : power_supply_working := FALSE
            act3 : predriver_phase := FALSE
            act4 : motor_relay_phase := FALSE
            act5 : fail_safe_relay_phase := FALSE
        end
Event   RESULT_motor_stops ≙
extends RESULT_motor_stops
        when
            grd1 : power_supply_phase = TRUE
            grd2 : power_supply_working = FALSE
        then
            act1 : motor_phase := TRUE
            act2 : motor_working := FALSE
        end
Event   RESULT_transition_to_manual_steering_mode ≙
extends RESULT_transition_to_manual_steering_mode
        when
            grd4 : motor_phase = TRUE
            grd3 : motor_working = FALSE
```

```
    then
        act1 : input := TRUE
        act2 : manual_steering_mode := TRUE
        act3 : demand_phase_1 := FALSE
        act4 : demand_phase_2 := FALSE
        act5 : motor_phase := FALSE
        act6 : power_supply_phase := FALSE
    end
Event   RESULT_not_sending_demand_to_manual_steering ≙
extends RESULT_not_sending_demand_to_manual_steering
    when
        grd1 : input = FALSE
        grd2 : inverter_failure = FALSE ∧ predriver_failure = FALSE
    then
        act1 : demand_to_manual_steering_sent := FALSE
        act2 : demand_phase_1 := TRUE
    end
Event   RESULT_not_receiving_demand_to_manual_steering ≙
extends RESULT_not_receiving_demand_to_manual_steering
    when
        grd2 : demand_phase_1 = TRUE
        grd3 : demand_to_manual_steering_sent = FALSE
    then
        act1 : demand_to_manual_steering_received := FALSE
        act2 : demand_phase_2 := TRUE
    end
Event   RESULT_not_sending_any_stop_signals ≙
    when
        grd1 : demand_phase_2 = TRUE
        grd2 : demand_to_manual_steering_received = FALSE
    then
        act1 : motor_relay_open_circuit_demand_sent := FALSE
        act2 : predriver_stop_signal_sent := FALSE
        act3 : fail_safe_relay_open_circuit_demand_sent := FALSE
        act4 : motor_relay_subphase_1 := TRUE
        act5 : predriver_subphase_1 := TRUE
        act6 : fail_safe_relay_subphase_1 := TRUE
    end
Event   RESULT_not_receiving_any_stop_signals ≙
    when
        grd1 : motor_relay_open_circuit_demand_sent = FALSE
        grd2 : predriver_stop_signal_sent = FALSE
        grd3 : fail_safe_relay_open_circuit_demand_sent = FALSE
        grd4 : motor_relay_subphase_1 = TRUE
        grd5 : predriver_subphase_1 = TRUE
        grd6 : fail_safe_relay_subphase_1 = TRUE
    then
        act1 : motor_relay_open_circuit_demand_received := FALSE
        act2 : predriver_stop_signal_received := FALSE
        act3 : fail_safe_relay_open_circuit_demand_received := FALSE
        act4 : motor_relay_subphase_2 := TRUE
        act5 : predriver_subphase_2 := TRUE
        act6 : fail_safe_relay_subphase_2 := TRUE
    end
Event  RESULT_predriver_motor_relay_fail_safe_relay_work ≙
refines  RESULT_predriver_motor_relay_fail_safe_relay_work
    when
        grd1 : motor_relay_subphase_2 = TRUE
        grd2 : motor_relay_open_circuit_demand_received = FALSE
        grd3 : predriver_subphase_2 = TRUE
        grd4 : predriver_stop_signal_received = FALSE
        grd5 : fail_safe_relay_open_circuit_demand_received = FALSE
        grd6 : fail_safe_relay_subphase_2 = TRUE
    then
        act1 : motor_relay_working := TRUE
        act2 : predriver_working := TRUE
        act3 : fail_safe_relay_working := TRUE
        act4 : motor_relay_phase := TRUE
        act5 : predriver_phase := TRUE
        act6 : fail_safe_relay_phase := TRUE
        act7 : predriver_subphase_1 := FALSE
        act8 : predriver_subphase_2 := FALSE
        act9 : motor_relay_subphase_1 := FALSE
        act10 : motor_relay_subphase_2 := FALSE
        act11 : fail_safe_relay_subphase_1 := FALSE
        act12 : fail_safe_relay_subphase_2 := FALSE
    end
Event  RESULT_power_supply_works ≙
extends  RESULT_power_supply_works
    when
        grd1 : motor_relay_working = TRUE
        grd2 : motor_relay_phase = TRUE
        grd3 : predriver_working = TRUE
        grd4 : predriver_phase = TRUE
        grd5 : fail_safe_relay_working = TRUE
        grd6 : fail_safe_relay_phase = TRUE
    then
        act1 : power_supply_phase := TRUE
        act2 : power_supply_working := TRUE
        act3 : predriver_phase := FALSE
        act4 : motor_relay_phase := FALSE
        act5 : fail_safe_relay_phase := FALSE
    end
Event  RESULT_motor_works ≙
extends RESULT_motor_works
    when
        grd1 : power_supply_phase = TRUE
        grd2 : power_supply_working = TRUE
    then
        act1 : motor_phase := TRUE
        act2 : motor_working := TRUE
    end
Event  RESULT_normal_mode ≙
extends RESULT_normal_mode
    when
        grd3 : motor_phase = TRUE
        grd4 : motor_working = TRUE
    then
        act1 : input := TRUE
        act2 : manual_steering_mode := FALSE
        act3 : demand_phase_1 := FALSE
        act4 : demand_phase_2 := FALSE
        act5 : power_supply_phase := FALSE
        act6 : motor_phase := FALSE
    end
END
```

C.4.8 Seventh refinement

```
MACHINE  Machine_8
REFINES  Machine_7
VARIABLES
    input
    manual_steering_mode
```

```
inverter_failure
predriver_failure
demand_to_manual_steering_sent
demand_to_manual_steering_sent_without_failure
demand_to_manual_steering_received
demand_phase_1
demand_phase_2
power_supply_working
motor_working
power_supply_phase
motor_phase
predriver_working
motor_relay_working
fail_safe_relay_working
predriver_phase
motor_relay_phase
fail_safe_relay_phase
predriver_subphase_1
predriver_subphase_2
motor_relay_subphase_1
motor_relay_subphase_2
fail_safe_relay_subphase_1
fail_safe_relay_subphase_2
predriver_enable_signal_sent
predriver_enable_signal_received
predriver_enable_signal_sent_without_failure
motor_relay_enable_signal_sent
motor_relay_enable_signal_sent_without_failure
motor_relay_enable_signal_received
fail_safe_relay_enable_signal_sent
fail_safe_relay_enable_signal_sent_without_failure
fail_safe_relay_enable_signal_received
```

INVARIANTS

inv1 : $predriver\_enable\_signal\_sent \in BOOL$

inv2 : $predriver\_enable\_signal\_received \in BOOL$

inv3 : $predriver\_enable\_signal\_sent\_without\_failure \in BOOL$

inv4 : $motor\_relay\_enable\_signal\_sent \in BOOL$

inv5 : $motor\_relay\_enable\_signal\_sent\_without\_failure \in BOOL$

inv6 : $motor\_relay\_enable\_signal\_received \in BOOL$

inv7 : $fail\_safe\_relay\_enable\_signal\_sent \in BOOL$

inv8 : $fail\_safe\_relay\_enable\_signal\_sent\_without\_failure \in BOOL$

inv9 : $fail\_safe\_relay\_enable\_signal\_received \in BOOL$

inv10 : $predriver\_enable\_signal\_sent = TRUE \land$
$predriver\_enable\_signal\_sent\_without\_failure = TRUE \Rightarrow$
$predriver\_stop\_signal\_sent = FALSE$

inv11 : $predriver\_enable\_signal\_sent = FALSE \Rightarrow predriver\_stop\_signal\_sent = TRUE \land$
$predriver\_stop\_signal\_sent\_without\_failure = TRUE$

inv12 : $predriver\_enable\_signal\_received = TRUE \Leftrightarrow predriver\_stop\_signal\_received =$
$FALSE$

inv13 : $motor\_relay\_enable\_signal\_sent = TRUE \land$
$motor\_relay\_enable\_signal\_sent\_without\_failure = TRUE \Rightarrow$
$motor\_relay\_open\_circuit\_demand\_sent = FALSE$

inv14 : $motor\_relay\_enable\_signal\_sent = FALSE \Rightarrow motor\_relay\_open\_circuit\_demand\_sent =$
$TRUE \land motor\_relay\_open\_circuit\_demand\_sent\_without\_failure = TRUE$

inv15 : $motor\_relay\_enable\_signal\_received = TRUE \Leftrightarrow$
$motor\_relay\_open\_circuit\_demand\_received = FALSE$

inv16 : $fail\_safe\_relay\_enable\_signal\_sent = TRUE \land$
$fail\_safe\_relay\_enable\_signal\_sent\_without\_failure = TRUE \Rightarrow$
$fail\_safe\_relay\_open\_circuit\_demand\_sent = FALSE$

inv17 : $fail\_safe\_relay\_enable\_signal\_sent = FALSE \Rightarrow$
$fail\_safe\_relay\_open\_circuit\_demand\_sent = TRUE \land$
$fail\_safe\_relay\_open\_circuit\_demand\_sent\_without\_failure = TRUE$

inv18 : $fail\_safe\_relay\_enable\_signal\_received = TRUE \Leftrightarrow$
$fail\_safe\_relay\_open\_circuit\_demand\_received = FALSE$

EVENTS

Initialisation

begin

act1 : $input := TRUE$
act3 : $manual\_steering\_mode := FALSE$
act4 : $inverter\_failure := FALSE$
act5 : $predriver\_failure := FALSE$
act6 : $demand\_to\_manual\_steering\_sent := FALSE$
act7 : $demand\_to\_manual\_steering\_sent\_without\_failure := FALSE$
act8 : $demand\_to\_manual\_steering\_received := FALSE$
act9 : $demand\_phase\_1 := FALSE$
act10 : $demand\_phase\_2 := FALSE$
act11 : $power\_supply\_working := TRUE$
act12 : $motor\_working := TRUE$
act13 : $power\_supply\_phase := FALSE$

```
        act14 : motor_phase := FALSE
        act15 : predriver_working := TRUE
        act16 : motor_relay_working := TRUE
        act17 : fail_safe_relay_working := TRUE
        act18 : predriver_phase := FALSE
        act19 : motor_relay_phase := FALSE
        act20 : fail_safe_relay_phase := FALSE
        act24 : predriver_subphase_1 := FALSE
        act25 : predriver_subphase_2 := FALSE
        act26 : motor_relay_subphase_1 := FALSE
        act27 : motor_relay_subphase_2 := FALSE
        act28 : fail_safe_relay_subphase_1 := FALSE
        act29 : fail_safe_relay_subphase_2 := FALSE
        act36 : predriver_enable_signal_sent := TRUE
        act37 : predriver_enable_signal_received := TRUE
        act38 : predriver_enable_signal_sent_without_failure := TRUE
        act39 : motor_relay_enable_signal_sent := TRUE
        act40 : motor_relay_enable_signal_sent_without_failure := TRUE
        act41 : motor_relay_enable_signal_received := TRUE
        act42 : fail_safe_relay_enable_signal_sent := TRUE
        act43 : fail_safe_relay_enable_signal_sent_without_failure := TRUE
        act44 : fail_safe_relay_enable_signal_received := TRUE
    end
Event  INPUT_inverter_failure ≙
extends  INPUT_inverter_failure
    when
        grd1 : input = TRUE
    then
        act1 : input := FALSE
        act2 : inverter_failure := TRUE
    end
Event  INPUT_predriver_failure ≙
extends  INPUT_predriver_failure
    when
        grd1 : input = TRUE
    then
        act1 : input := FALSE
        act2 : predriver_failure := TRUE
    end
Event  INPUT_no_failure ≙
extends  INPUT_no_failure
    when
```

```
        grd1 : input = TRUE
    then
        act1 : input := FALSE
        act2 : inverter_failure := FALSE
        act3 : predriver_failure := FALSE
    end
Event  RESULT_sending_demand_to_manual_steering ≙
extends  RESULT_sending_demand_to_manual_steering
    when
        grd1 : input = FALSE
        grd2 : inverter_failure = TRUE ∨ predriver_failure = TRUE
    then
        act1 : demand_to_manual_steering_sent := TRUE
        act2 : demand_to_manual_steering_sent_without_failure := TRUE
        act3 : demand_phase_1 := TRUE
    end
Event  RESULT_receiving_demand_to_manual_steering ≙
extends  RESULT_receiving_demand_to_manual_steering
    when
        grd3 : demand_to_manual_steering_sent = TRUE
        grd4 : demand_to_manual_steering_sent_without_failure = TRUE
        grd5 : demand_phase_1 = TRUE
    then
        act1 : demand_to_manual_steering_received := TRUE
        act2 : demand_phase_2 := TRUE
    end
Event  RESULT_predriver_enable_signal_not_sent ≙
refines  RESULT_predriver_stop_signal_sent
    when
        grd1 : demand_phase_2 = TRUE
        grd2 : demand_to_manual_steering_received = TRUE
    then
        act1 : predriver_enable_signal_sent := FALSE
        act3 : predriver_subphase_1 := TRUE
    end
Event  RESULT_predriver_enable_signal_not_received ≙
refines  RESULT_predriver_stop_signal_received
    when
        grd1 : predriver_enable_signal_sent = FALSE
        grd3 : predriver_subphase_1 = TRUE
    then
```

```
          act1 :  predriver_enable_signal_received := FALSE
          act2 :  predriver_subphase_2 := TRUE
      end
Event   RESULT_predriver_stops ≙
refines  RESULT_predriver_stops
      when
          grd1 :  predriver_subphase_2 = TRUE
          grd2 :  predriver_enable_signal_received = FALSE
      then
          act1 :  predriver_working := FALSE
          act2 :  predriver_phase := TRUE
          act3 :  predriver_subphase_1 := FALSE
          act4 :  predriver_subphase_2 := FALSE
      end
Event   RESULT_motor_relay_enable_signal_not_sent ≙
refines  RESULT_motor_relay_open_circuit_demand_sent
      when
          grd1 :  demand_phase_2 = TRUE
          grd2 :  demand_to_manual_steering_received = TRUE
      then
          act1 :  motor_relay_enable_signal_sent := FALSE
          act3 :  motor_relay_subphase_1 := TRUE
      end
Event   RESULT_motor_relay_enable_signal_not_received ≙
refines  RESULT_motor_relay_open_circuit_demand_received
      when
          grd1 :  motor_relay_enable_signal_sent = FALSE
          grd3 :  motor_relay_subphase_1 = TRUE
      then
          act1 :  motor_relay_enable_signal_received := FALSE
          act2 :  motor_relay_subphase_2 := TRUE
      end
Event   RESULT_motor_relay_stops ≙
refines  RESULT_motor_relay_stops
      when
          grd1 :  motor_relay_subphase_2 = TRUE
          grd2 :  motor_relay_enable_signal_received = FALSE
      then
          act1 :  motor_relay_working := FALSE
          act2 :  motor_relay_phase := TRUE
          act3 :  motor_relay_subphase_1 := FALSE
```

```
            act4 : motor_relay_subphase_2 := FALSE
        end
    Event   RESULT_fail_safe_relay_enable_signal_not_sent ≙
    refines  RESULT_fail_safe_relay_open_circuit_demand_sent
        when
            grd1 : demand_phase_2 = TRUE
            grd2 : demand_to_manual_steering_received = TRUE
        then
            act1 : fail_safe_relay_enable_signal_sent := FALSE
            act3 : fail_safe_relay_subphase_1 := TRUE
        end
    Event   RESULT_fail_safe_relay_enable_signal_not_received ≙
    refines  RESULT_fail_safe_relay_open_circuit_demand_received
        when
            grd1 : fail_safe_relay_enable_signal_sent = FALSE
            grd3 : fail_safe_relay_subphase_1 = TRUE
        then
            act1 : fail_safe_relay_enable_signal_received := FALSE
            act2 : fail_safe_relay_subphase_2 := TRUE
        end
    Event   RESULT_fail_safe_relay_stops ≙
    refines  RESULT_fail_safe_relay_stops
        when
            grd1 : fail_safe_relay_subphase_2 = TRUE
            grd2 : fail_safe_relay_enable_signal_received = FALSE
        then
            act1 : fail_safe_relay_working := FALSE
            act2 : fail_safe_relay_phase := TRUE
            act3 : fail_safe_relay_subphase_1 := FALSE
            act4 : fail_safe_relay_subphase_2 := FALSE
        end
    Event   RESULT_power_supply_stops ≙
    extends  RESULT_power_supply_stops
        when
            grd1 : (predriver_phase = TRUE∧predriver_working = FALSE)∨(motor_relay_phase =
                TRUE∧motor_relay_working = FALSE)∨(fail_safe_relay_phase = TRUE∧
                fail_safe_relay_working = FALSE)
        then
            act1 : power_supply_phase := TRUE
            act2 : power_supply_working := FALSE
            act3 : predriver_phase := FALSE
            act4 : motor_relay_phase := FALSE
            act5 : fail_safe_relay_phase := FALSE
        end
    Event  RESULT_motor_stops ≙
    extends RESULT_motor_stops
        when
            grd1 : power_supply_phase = TRUE
            grd2 : power_supply_working = FALSE
        then
            act1 : motor_phase := TRUE
            act2 : motor_working := FALSE
        end
    Event  RESULT_transition_to_manual_steering_mode ≙
    extends RESULT_transition_to_manual_steering_mode
        when
            grd4 : motor_phase = TRUE
            grd3 : motor_working = FALSE
        then
            act1 : input := TRUE
            act2 : manual_steering_mode := TRUE
            act3 : demand_phase_1 := FALSE
            act4 : demand_phase_2 := FALSE
            act5 : motor_phase := FALSE
            act6 : power_supply_phase := FALSE
        end
    Event  RESULT_not_sending_demand_to_manual_steering ≙
    extends RESULT_not_sending_demand_to_manual_steering
        when
            grd1 : input = FALSE
            grd2 : inverter_failure = FALSE ∧ predriver_failure = FALSE
        then
            act1 : demand_to_manual_steering_sent := FALSE
            act2 : demand_phase_1 := TRUE
        end
    Event  RESULT_not_receiving_demand_to_manual_steering ≙
    extends RESULT_not_receiving_demand_to_manual_steering
        when
            grd2 : demand_phase_1 = TRUE
            grd3 : demand_to_manual_steering_sent = FALSE
        then
            act1 : demand_to_manual_steering_received := FALSE
            act2 : demand_phase_2 := TRUE
        end
    Event   RESULT_sending_all_enable_signals ≙
    refines RESULT_not_sending_any_stop_signals
        when
            grd1 : demand_phase_2 = TRUE
            grd2 : demand_to_manual_steering_received = FALSE
        then
            act1 : motor_relay_enable_signal_sent := TRUE
            act2 : predriver_enable_signal_sent := TRUE
            act3 : fail_safe_relay_enable_signal_sent := TRUE
            act4 : motor_relay_subphase_1 := TRUE
            act5 : predriver_subphase_1 := TRUE
            act6 : fail_safe_relay_subphase_1 := TRUE
            act7 : predriver_enable_signal_sent_without_failure := TRUE
            act8 : motor_relay_enable_signal_sent_without_failure := TRUE
            act9 : fail_safe_relay_enable_signal_sent_without_failure := TRUE
        end
    Event   RESULT_receiving_all_enable_signals ≙
    refines RESULT_not_receiving_any_stop_signals
        when
            grd1 : motor_relay_enable_signal_sent = TRUE
            grd2 : predriver_enable_signal_sent = TRUE
            grd3 : fail_safe_relay_enable_signal_sent = TRUE
            grd4 : motor_relay_subphase_1 = TRUE
            grd5 : predriver_subphase_1 = TRUE
            grd6 : fail_safe_relay_subphase_1 = TRUE
            grd7 : predriver_enable_signal_sent_without_failure = TRUE
            grd8 : motor_relay_enable_signal_sent_without_failure = TRUE
            grd9 : fail_safe_relay_enable_signal_sent_without_failure = TRUE
        then
            act1 : motor_relay_enable_signal_received := TRUE
            act2 : predriver_enable_signal_received := TRUE
            act3 : fail_safe_relay_enable_signal_received := TRUE
            act4 : motor_relay_subphase_2 := TRUE
            act5 : predriver_subphase_2 := TRUE
            act6 : fail_safe_relay_subphase_2 := TRUE
        end
    Event   RESULT_predriver_motor_relay_fail_safe_relay_work ≙
    refines RESULT_predriver_motor_relay_fail_safe_relay_work
        when
            grd1 : motor_relay_subphase_2 = TRUE
            grd2 : motor_relay_enable_signal_received = TRUE
            grd3 : predriver_subphase_2 = TRUE
            grd4 : predriver_enable_signal_received = TRUE
            grd5 : fail_safe_relay_enable_signal_received = TRUE
            grd6 : fail_safe_relay_subphase_2 = TRUE
        then
            act1 : motor_relay_working := TRUE
            act2 : predriver_working := TRUE
            act3 : fail_safe_relay_working := TRUE
            act4 : motor_relay_phase := TRUE
            act5 : predriver_phase := TRUE
            act6 : fail_safe_relay_phase := TRUE
            act7 : predriver_subphase_1 := FALSE
            act8 : predriver_subphase_2 := FALSE
            act9 : motor_relay_subphase_1 := FALSE
            act10 : motor_relay_subphase_2 := FALSE
            act11 : fail_safe_relay_subphase_1 := FALSE
            act12 : fail_safe_relay_subphase_2 := FALSE
        end
    Event  RESULT_power_supply_works ≙
    extends RESULT_power_supply_works
        when
            grd1 : motor_relay_working = TRUE
            grd2 : motor_relay_phase = TRUE
            grd3 : predriver_working = TRUE
            grd4 : predriver_phase = TRUE
            grd5 : fail_safe_relay_working = TRUE
            grd6 : fail_safe_relay_phase = TRUE
        then
            act1 : power_supply_phase := TRUE
            act2 : power_supply_working := TRUE
            act3 : predriver_phase := FALSE
            act4 : motor_relay_phase := FALSE
            act5 : fail_safe_relay_phase := FALSE
        end
    Event  RESULT_motor_works ≙
    extends RESULT_motor_works
        when
            grd1 : power_supply_phase = TRUE
            grd2 : power_supply_working = TRUE
        then
            act1 : motor_phase := TRUE
            act2 : motor_working := TRUE
        end
    Event   RESULT_normal_mode ≙
    extends  RESULT_normal_mode
        when
            grd3 : motor_phase = TRUE
            grd4 : motor_working = TRUE
        then
            act1 : input := TRUE
            act2 : manual_steering_mode := FALSE
            act3 : demand_phase_1 := FALSE
            act4 : demand_phase_2 := FALSE
            act5 : power_supply_phase := FALSE
            act6 : motor_phase := FALSE
        end
    END
```

Bibliography

[AAB$^+$09] Benjamin Aziz, Alvaro Arenas, Juan Bicarregui, Christophe Ponsard, and Philippe Massonet. From goal-oriented requirements to event-b specifications. 2009.

[Abr06] Jean-Raymond Abrial. Formal methods in industry: achievements, problems, future. In *Proceedings of the 28th international conference on Software engineering*, pages 761–768. ACM, 2006.

[Abr07] J-R Abrial. A system development process with event-b and the rodin platform. In *Formal Methods and Software Engineering*, pages 1–3. Springer, 2007.

[Abr10] J.R. Abrial. *Modeling in Event-B: system and software engineering.* Cambridge University Press, 2010.

[ACH$^+$94] Rajeev Alur, Costas Courcoubetis, T Henzinger, P Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. In *11th International Conference on Analysis and Optimization of Systems Discrete Event Systems*, pages 329–351. Springer, 1994.

[ASZ12] Jean-Raymond Abrial, Wen Su, and Huibiao Zhu. Formalizing hybrid systems with event-b. In *Abstract State Machines, Alloy, B, VDM, and Z*, pages 178–193. Springer, 2012.

[Bib87] Wolfgang Bibel. Automated theorem proving. 1987.

[But09] Michael Butler. Decomposition structures for event-b. In *Integrated Formal Methods*, pages 20–38. Springer, 2009.

[C$^+$05] RODIN Consortium et al. Rodin deliverable D7-Event B language. Technical report, Available at http://rodin.cs.ncl.ac.uk/deliverables/rodinD7.pdf, 2005.

[CGP99] Edmund M Clarke, Orna Grumberg, and Doron A Peled. *Model checking.* MIT press, 1999.

[DB09] Kriangsak Damchoom and Michael Butler. Applying event and machine decomposition to a flash-based filestore in event-b. In *Formal Methods: Foundations and Applications*, pages 134–152. Springer, 2009.

[DVL96] Robert Darimont and Axel Van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. *ACM SIGSOFT Software Engineering Notes*, 21(6):179–190, 1996.

[DVLF93] Anne Dardenne, Axel Van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of computer programming*, 20(1):3–50, 1993.

[Ecl07] IDE Eclipse. The eclipse foundation, 2007.

[Für09] Andreas Fürst. Design patterns in event-b and their tool support. Master's thesis, Department of Computer Science, ETH Zurich, 2009.

[GDPALN$^+$09] Jorge García-Duque, José J Pazos-Arias, Martín López-Nores, Yolanda Blanco-Fernández, Ana Fernández-Vilas, Rebeca P Díaz-Redondo, Manuel Ramos-Cabrer, and Alberto Gil-Solla. Methodologies to evolve formal specifications through refinement and retrenchment in an analysis–revision cycle. *Requirements engineering*, 14(3):129–153, 2009.

[Gil70] Arthur Gill. Finite-state machines. *IEEE TRANSACTIONS ON COMPUTERS*, 19(11), 1970.

[HBGL95] Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw. Scr: A toolset for specifying and analyzing requirements. In *Computer Assurance, 1995. COMPASS'95. Systems Integrity, Software Safety and Process Security. Proceedings of the Tenth Annual Conference on*, pages 109–122. IEEE, 1995.

[Hei02] Constance L Heitmeyer. *Software cost reduction.* Wiley Online Library, 2002.

[HL96] Mats Per Erik Heimdahl and Nancy G Leveson. Completeness and consistency in hierarchical state-based requirements. *Software Engineering, IEEE Transactions on*, 22(6):363–377, 1996.

[HL11] Stefan Hallerstede and Michael Leuschel. Finding deadlocks of event-b models by constraint solving. *Electronic Notes in Theoretical Computer Science*, 280, 2011.

[ISO11] CD ISO. 26262, road vehicles–functional safety, 2011.

[Jon90] Cliff B Jones. *Systematic software development using VDM*, volume 2. Prentice Hall Englewood Cliffs, 1990.

[KH12] T. Kobayashi and S. Honiden. Towards refinement strategy planning for Event-B. *arXiv preprint arXiv:1210.7036*, 2012.

[KKK95] Eun Mi Kim, Shinji Kusumoto, and Tohru Kikuno. An approach to safety and correctness verification of software design specification. In *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, pages 78–83. IEEE, 1995.

[Lut93] R.R. Lutz. Analyzing software requirements errors in safety-critical, embedded systems. In *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on*, pages 126–133. IEEE, 1993.

[MGL11] Abderrahman Matoussi, Frédéríc Gervais, and Régine Laleau. 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. IEEE, 2011.

[OL07] Martin Ouimet and Kristina Lundqvist. Formal software verification: Model checking and theorem proving. Technical report, Embeded Systems Laboratory, Massachusetts Institute of Technology, 2007.

[PD11] Christophe Ponsard and Xavier Devroey. Generating high-level event-b system models from kaos requirements models. 2011.

[PMPS01] Zs Pap, I Majzik, A Pataricza, and A Szegi. Completeness and consistency analysis of uml statechart specifications. In *Proc. of IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop*, pages 83–90, 2001.

[REB07] A. Rezazadeh, N. Evans, and M. Butler. Redevelopment of an industrial case study using Event-B and Rodin. *BCS-FACS Christmas 2007 Meeting - Formal Methods In Industry, London*, 2007.

[ROD13] RODIN - Rigorous Open Development Environment for Complex Systems, 2013.

[SAHZ11] W. Su, J.R. Abrial, R. Huang, and H. Zhu. From requirements to development: methodology and example. *Formal Methods and Software Engineering*, pages 437–455, 2011.

[SAZ12] W. Su, J.R. Abrial, and H. Zhu. Complementary methodologies for developing hybrid systems with Event-B. *Formal Methods and Software Engineering*, pages 230–248, 2012.

[SB06] Colin Snook and Michael Butler. Uml-b: Formal modeling and design aided by uml. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(1):92–122, 2006.

[SBS09] Mar Yah Said, Michael Butler, and Colin Snook. Language and tool support for class and state machine refinement in uml-b. In *FM 2009: Formal Methods*, pages 579–595. Springer, 2009.

[SFRB11] A. Salehi Fathabadi, A. Rezazadeh, and M. Butler. Applying atomicity and model decomposition to a space craft system in Event-B. *NASA Formal Methods*, pages 328–342, 2011.

[SH01] Keng Siau and Terence Aidan Halpin. *Unified modeling language.* IGI Global, 2001.

[Sil12] R. Silva. Lessons learned/sharing the experience of developing a metro system case study. *arXiv preprint arXiv:1210.7030*, 2012.

[SP96] Atish P Sinha and Doug Popken. Completeness and consistency checking of system requirements: An expert agent approach. *Expert Systems with Applications*, 11(3):263–276, 1996.

[SS10] D.J. Smith and KG Simpson. Safety critical systems handbook, 2010.

[VL09] A. Van Lamsweerde. *Requirements engineering: from system goals to UML models to software specifications*, volume 3. Wiley, 2009.

[VLDM95] Axel Van Lamsweerde, Robert Darimont, and Philippe Massonet. Goal-directed elaboration of requirements for a meeting scheduler: problems and lessons learnt. In *Requirements Engineering, 1995., Proceedings of the Second IEEE International Symposium on*, pages 194–203. IEEE, 1995.

[YJ$^+$11] Faqing Yang, Jean-Pierre Jacquot, et al. An event-b plug-in for creating deadlock-freeness theorems. In *14th Brazilian Symposium on Formal Methods*, 2011.

[YSLS08] Lian Yu, Shuang Su, Shan Luo, and Yu Su. Completeness and consistency analysis on requirements of distributed event-driven systems. In *Theoretical Aspects of Software Engineering, 2008. TASE'08. 2nd IFIP/IEEE International Symposium on*, pages 241–244. IEEE, 2008.

[ZG03] D. Zowghi and V. Gervasi. On the interplay between consistency, completeness, and correctness in requirements evolution. *Information and Software Technology*, 45(14):993–1009, 2003.