D.5.3 - ADVANCE PROCESS INTEGRATION III

ADVANCE

Grant Agreement:	287563
Date:	30/11/2014
Pages:	56
Status:	Final
Authors:	John Colley, Michael Butler University of Southampton
Reference:	D5.3
Issue:	1

Partners / Clients:

FP7 Framework Programme	European Union

Consortium Members:

University of Southampton	Critical Software Technologies	Alstom Transport	Systerel	Heinrich Heine Universität	Selex ES

Project ADVANCE
Grant Agreement 287563
"Advanced Design and Verification Environment for Cyber-physical System Engineering"

ADVANCE Deliverable D5.3

ADVANCE Process Integration III

Public Document

November 25, 2014

http://www.advance-ict.eu

Contributors:

Michael Butler
John Colley

Reviewers:

Laurent Voisin
Luis-Fernando Mejia

Contents

Chapter 1

Preface

This deliverable reports on the final work on developing the ADVANCE process flow, combining formal modelling with requirements analysis and safety analysis, a process which has now been applied to both industrial case studies. We present the complete ADVANCE flow, from conception to concrete realisation, verification and certification. We present a tutorial to guide the user through the ADVANCE process flow. We then give a detailed explanation of the formal decomposition process and its role in ADVANCE in enabling cyber-physical system multi-simulation. We also report on how the MC/DC coverage measure can be used in ADVANCE at all stages of model refinement prior to model-based test generation. Finally, we explain how the ADVANCE process can be applied to the DO-178/254 standards for the design assurance of airborne electronic hardware and software.

In Chapter 2 we present the complete ADVANCE Process Flow.

In Chapter 3 we present a tutorial, based on the Aircraft Landing Gear case study, which demonstrates how the ADVANCE process can be applied to a safety critical cyber-physical system.

In Chapter 4 we look at how a system-level Event-B model of a system consisting of multiple physical devices under some coordinated control may be decomposed into sub-models, where these sub-models will represent separate architectural components including the devices, the controller and the signalling mechanisms between them.

In Chapter 5 we report on how MC/DC coverage is used throughout the ADVANCE process flow and how a test suite can be generated from a concrete model of a safety-critical system.

In Chapter 6 we report on how the ADVANCE process can be used in aircraft certification.

In Chapter 7 we present a summary of the ADVANCE process developments.

Chapter 2

The ADVANCE process overview

2.1 Process Summary

We can summarise the ADVANCE process flow as a set of seven steps:-

1. Deriving the Safety Constraints from the Functional Requirements using STPA
2. Modeling the Safety Constraints in Event-B
3. Determining how Unsafe Control Actions could occur
4. Documenting the Requirements and Design Decisions with ProR
5. Refining the model and safety constraints to ensure Control Actions are safe in the presence of Hazards
6. Model-based test generation and MC/DC coverage
7. Shared Event Decomposition

2.1.1 Deriving the Safety Constraints from the Functional Requirements using STPA

After initial studies using simple examples, STPA has now been applied successfully to both industrial case studies, as reported in the final deliverables [Mej14b] and [Rei14]. In both case studies, domain experts were involved in deriving the safety constraints.

2.1.2 Modeling the Safety Constraints in Event-B

Again, in both case studies, the safety constraints were successfully modelled in Event-B as a combination of event guards and invariants, and the invariants proved.

2.1.3 Determining how Unsafe Control Actions could occur

This second phase of the STPA process informs the design process and is used during system development to mitigate or eliminate potential hazards. Domain experts are closely involved during this phase of the process.

2.1.4 Documenting the Requirements and Design Decisions with ProR

ProR has been used in the case studies for documentation and traceability [Mej14b] and [Rei14].

2.1.5 Refining the model and safety constraints to ensure Control Actions are safe in the presence of Hazards

Using the input from the domain experts, the system designers can use the ADVANCE proof facilities to verify that actions issued by the digital controller are safe.

2.1.6 Model-based test generation and MC/DC coverage

During the final year of the project these facilities have been added to ADVANCE to support the link between formal and simulation-based verification. MC/DC coverage can be used throughout the formal model refinement process and to give confidence that the tests generated have adequate coverage.

2.1.7 Shared Event Decomposition

Formal decomposition has been used in both case studies, enabling further refinement of the digital controller implementation and multi-simulation.

Chapter 3

ADVANCE Process Tutorial

3.1 The Landing Gear Case Study

We have chosen the landing gear case study as the basis for this ADVANCE process tutorial because it typifies the kind of safety-critical system that ADVANCE addresses, is easy to understand without necessarily being a domain expert and a comprehensive, well-defined set of requirements is available in the public domain [BW14].

3.1.1 Requirements and Safety Analysis

In the ADVANCE process, requirements and safety analysis are closely integrated. We consider the requirements systematically in terms of the

- Monitored Phenomena
- Controlled Phenomena
- Commanded Phenomena
- Mode Phenomena

The Controlled Phenomena

It is the *Controlled Phenomena* which provides the link to the safety analysis process. Consider the requirements for the *Door Sub-system* of the aircraft landing gear.

- The Controller will open the Doors when the Pilot moves the Lever to Extend or Retract the Landing Gear

- The Controller will then close the Doors when the Landing Gear is fully Extended or Retracted
- The Doors will remain open while the Landing Gear is Extending or Retracting

The ADVANCE process introduces safety analysis at the very beginning, to ensure that safety considerations are addressed as early as possible. We use SystemTheoretic Process Analysis (STPA) [Lev12] which is performed in two phases.

- Identify Potentially Hazardous Control Actions and derive the Safety Constraints
- Determine how Unsafe Control Actions could occur

Identifying Potentially Hazardous Control Actions

The landing gear system has effectively two controllers: the pilot, who has a high-level view of the position of the landing gear, according to the position of the extend/retract handle in the cockpit, and the digital controller, which controls the position of the doors, using the *actuators*, according to the position of the pilot handle and the landing gear. The digital controller monitors the position of the doors and updates the state of its internal process model using the *sensors*. If the sensor values are inconsistent with the process model, the controller can notify the pilot of a potential system failure. The process models are shown in Figure 3.1 below.

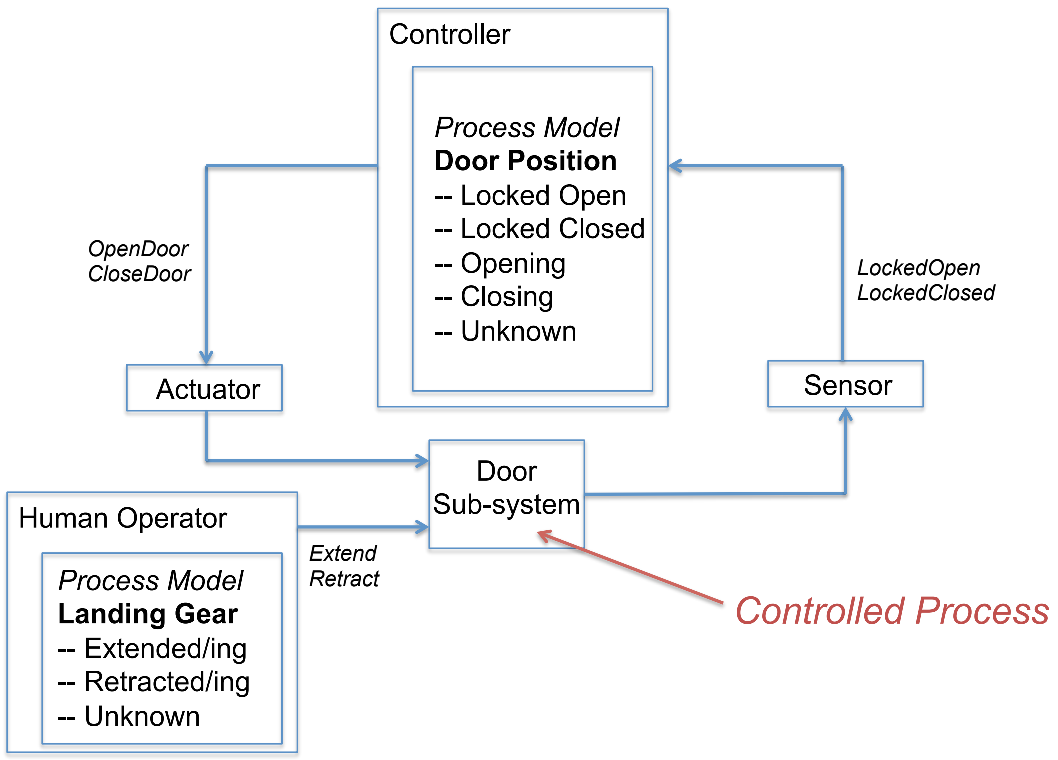


Figure 3.1: The Landing Gear Doors Process Model

For each of the controller door actions, *Open Door* and *Close Door*, we identify in a systematic way, how these actions can be hazardous as shown in Figure 3.2.

Controller Action	Not Providing Causes Hazard	Providing Causes Hazard	Wrong Timing or Order Causes Hazard	Stopped too soon/Applied too long
Open Door	Cannot extend Landing Gear for landing	Not Hazardous	Damage to Landing Gear	Damage to Landing Gear/ Not Hazardous
Close Door	Not Hazardous	Damage to Landing Gear	Damage to Landing Gear	Not Hazardous/ Not Hazardous

Figure 3.2: Safety Analysis for Door Sub-system

The controller not opening the door when it should is hazardous as the landing gear cannot be extended for landing, but opening the door when it shouldn't is not hazardous. Opening the doors after the landing gear has begun to extend or retract is hazardous, as is failing to complete the opening procedure. A similar analysis is then performed on the *Close Door* action.

From this table we are able to derive the natural language safety constraints.

1. If the Landing Gear is Extending, the Door must be Locked Open
2. If the Landing Gear is Retracting, the Door must be Locked Open
3. A *Close Door* command must only be issued if the Landing Gear is Locked Up or Locked Down
4. An *Open Door* command must only be issued if the Landing Gear is Locked Up or Locked Down

3.1.2 Modelling the Landing Gear System

The Abstract Model

We begin with an abstract model of the system which represents just the *gearstate*. The landing gear may be *locked_ up*, *locked_ down* or, because we wish to model the temporal nature of the system, *extending* or *retracting*. Four events define the transitions between these states: *Extend* and *Retract* represent a requested operation, initiated by the pilot, and *CompleteExtend* and *CompleteRetract* are observed when the requested operation is completed.

The abstract model is illustrated by the state machine shown in Figure 3.3 below. Notice that when the landing gear is in the process of *extending* or *retracting*, the pilot can at any time move the landing gear handle position to reverse the command.

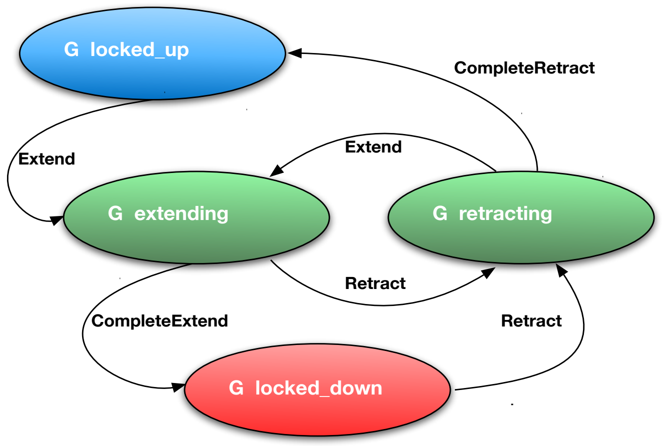


Figure 3.3: The Abstract Model

The First Refinement

We now introduce the door and establish formally the relationship between the gear and the door from the natural language safety constraints.

1. If the Landing Gear is Extending, the Door must be Locked Open
2. If the Landing Gear is Retracting, the Door must be Locked Open
3. A *Close Door* command must only be issued if the Landing Gear is Locked Up or Locked Down
4. An *Open Door* command must only be issued if the Landing Gear is Locked Up or Locked Down

The first two safety constraints are represented by the invariant *inv3* below.

$$\texttt{inv3} : gearstate \in \{extending, retracting\} \Rightarrow doorstate = locked\_open$$

The second two safety constraints are modelled by the guards *grd1* in the *Open* and *Close* events.

```
event  Open ≙
    when
            grd1 : $gearstate \in \{locked\_down, locked\_up\}$
            grd2 : $doorstate \in \{closing, locked\_closed\}$
    then
            act1 : $doorstate := opening$
    end
event  Close ≙
    when
            grd1 : $gearstate \in \{locked\_down, locked\_up\}$
            grd2 : $doorstate \in \{opening, locked\_open\}$
    then
            act1 : $doorstate := closing$
    end
```

Running the Rodin automatic provers establishes that the formal system-level safety constraints are preserved by the refinement. We can represent the refined model as a state machine in terms of the two variables *gearstate* and *doorstate* as shown in Figure 3.4 below.

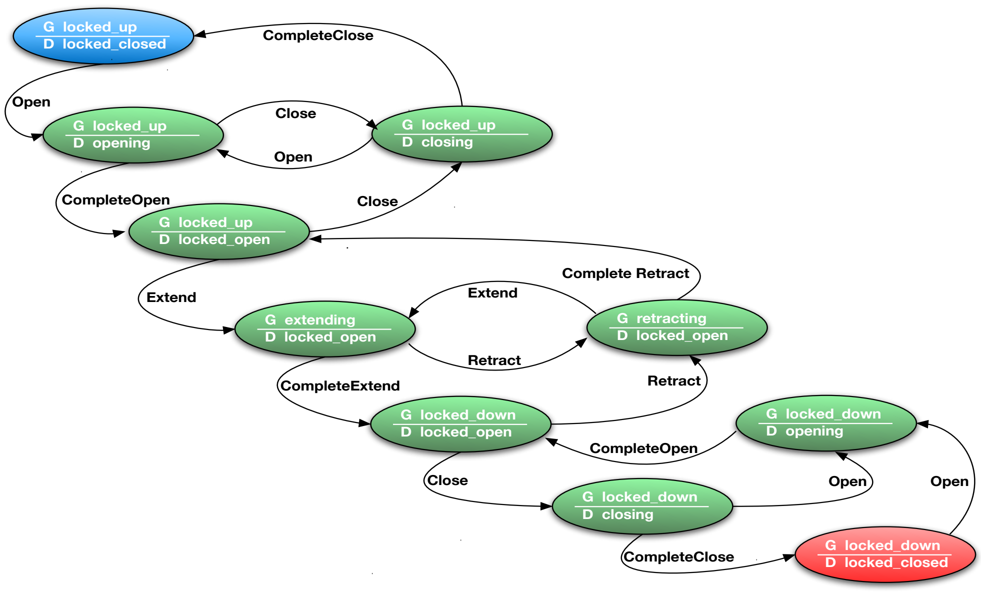


Figure 3.4: The First Refinement

The Second Refinement

In this refinement, we introduce the pilot *handle* and model the synchronous timing and synchronisation necessary to represent the concurrency of the system. We do not at this stage model actual times or delays but simply introduce the notion of a *tick*. For any given tick of the system, the handle may or may not be moved; the digital controller responds to any handle change in the next tick.

To represent the latency of the system in an abstract way, we introduce the event *Idle* as shown in the extended finite state machine, defined in terms of the three variables *gearstate*, *doorstate* and *handle*, in Figure 3.5 below.

Only 12 of the 32 possible states for *(gearstate;doorstate;handle)* are valid. To ensure that the system never makes a transition to one of the 20 invalid states we introduce and prove a set of invariants. For instance, invariant *inv10* below represents the fact that the door cannot be opening if the gear is locked up and the handle is up.

$$\texttt{inv10} : \neg(gearstate = locked\_up \wedge doorstate = opening \wedge handle = UP)$$

We now run the ADVANCE model checker to ensure that all the valid states of the model are reachable and there is no deadlock. At this high level of abstraction an exhaustive model check can be completed in seconds.

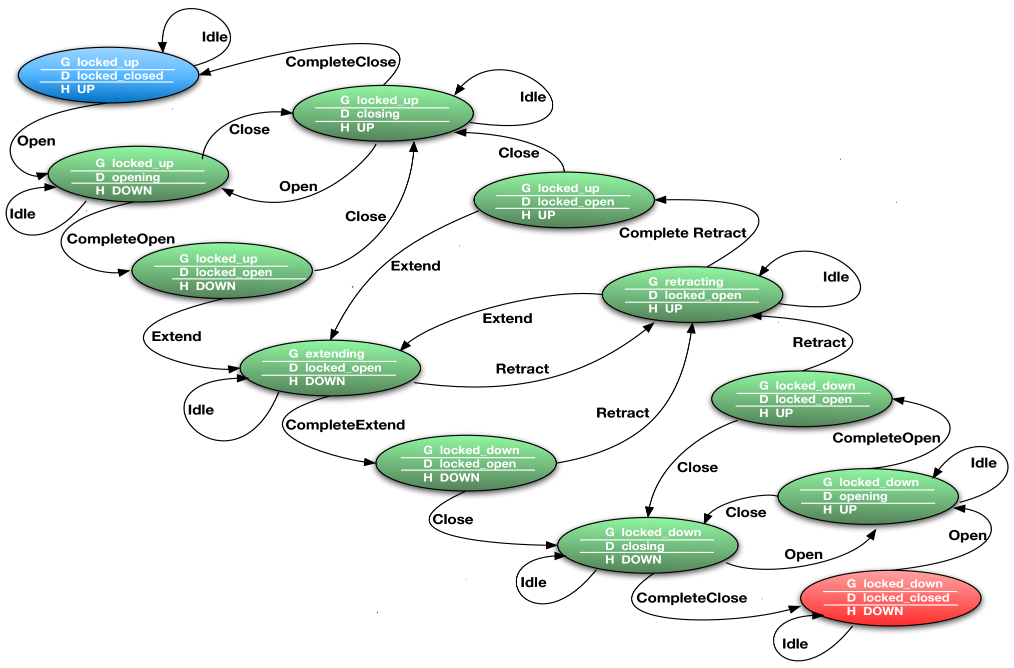


Figure 3.5: The Second Refinement

Introducing the Timing Deadlines

We now introduce the concrete signals between the Controller and the landing gear sub-system as shown in Figure 3.6 below.

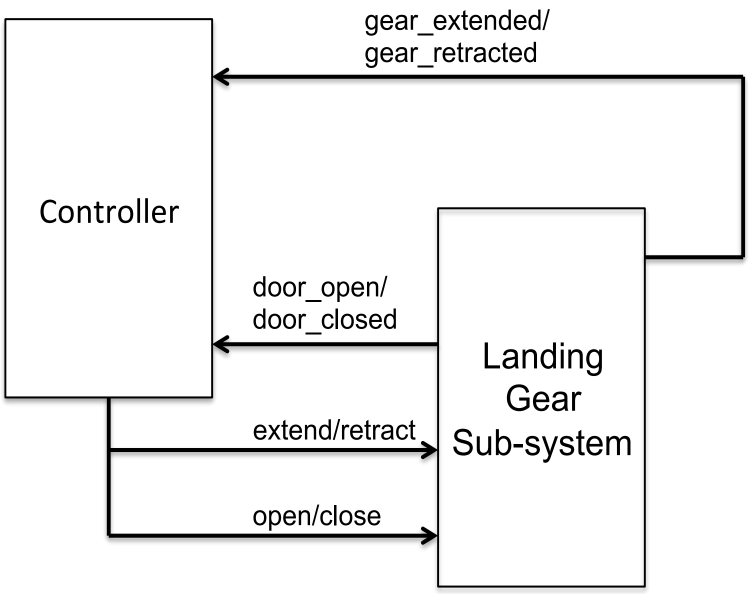


Figure 3.6: The Component View

The landing gear requirements detail a set of timing constraints for each of the mechanical and hydraulic procedures of the extending and retracting sequences. We introduce these constraints, systematically using refinement, as deadlines which refine the abstract *Idle* events.

Rather than idling indefinitely, the model sets an abstract timer *count* which is decremented if the confirmation signal from the landing gear subsystem has not been received. If confirmation is received before the deadline expires, the operation can complete. If, however, the count reaches zero without confirmation, the controller moves to a fail state and switches on a warning light on the control panel. The general refinement mechanism is shown in Figure 3.7 below.

Proving Liveness

Now that model behaviour is sufficiently constrained by the deadlines, we can prove liveness of the system. For each of the 12 valid states as shown in Figure 3.5 above we introduce an invariant to prove that at least one event is always enabled in that state. For instance, in the state *(locked_ down, locked_ open, DOWN)*, one of the events *Close* or *Retract* is always enabled. Similarly, in state *(locked_ down, locked_ open, UP)*, one of the events *Close* or *Retract* is also always enabled. We describe this using a single *theorem.*

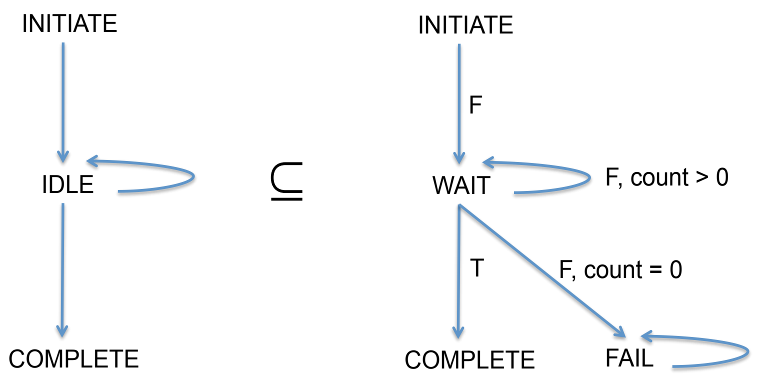


Figure 3.7: Introducing the Deadlines

We write the theorem as an implication, with the current state on the left hand side and the *disjunction of the guards* of the two enabled events on the right hand side as shown below in invariant *inv19*.

$$\begin{aligned}
\texttt{inv19} : (gearstate = locked\_down \land doorstate = locked\_open) \quad \Rightarrow \\
&(gearstate \in \{extending, locked\_down\} \land \\
&\ doorstate = locked\_open \land handle = UP) \quad \lor \\
&(gearstate = locked\_up \land \\
&\ doorstate \in \{opening, locked\_open\} \land handle = UP) \quad \lor \\
&(gearstate = locked\_down \land \\
&\ doorstate \in \{opening, locked\_open\} \land \\
&\ handle = DOWN)
\end{aligned}$$

Ten theorems describe the liveness properties of the system and are proved automatically by the Rodin theorem provers.

3.1.3 Measuring Coverage

The Modified Condition/Decision Coverage (MC/DC) measure introduced in ADVANCE can be used to verify, at every refinement level, that the guards of each event can be independently set to *FALSE*. The facility is also used at every refinement stage to ensure that any *vacuous* guards are eliminated.

3.1.4 Model-based Test Generation

We are now in a position to generate tests from the model that can be used to verify the implementation. Because, the pilot is free to move the handle backwards and forwards with complete freedom, we now need to constrain the model behaviour to ensure that we can generate a test suite of tractable size. We introduce, in a refinement a *ghost variable*, *handle_toggle_count* which limits the number of times the handle can change position. We can then restrict the value of this count in the ADVANCE test generation tool to limit the *scope* of the model search space.

For instance, restricting the count to two, the pilot can only move the handle twice and a total of 48 tests are generated. If the count is restricted to three, 852 tests are generated. In a structured regression suite mechanism, these tests sets can be run first to detect any gross errors. For more thorough testing, the tests generated for higher handle counts can be run - setting the count to 5, for instance, results in 15400 tests. The coverage obtained by each set of tests is measured by using the ProB simulator to run the generated tests against the model.

3.1.5 Decomposition

Formal, *shared event* decomposition is now used to separate the Controller model from its environment for further refinement towards implementation. The decomposed Controller model can also be converted into a Functional Mockup Unit (FMU) for simulation in a continuous representation of the landing gear environment.

Chapter 4

Guidelines for Decomposition of Control System Models

4.1 Introduction

In this chapter we look at how a system-level Event-B model of a system consisting of multiple physical devices under some coordinated control may be decomposed into sub-models, where these sub-models will represent separate architectural components including the devices, the controller and the signalling mechanisms between them. A closed system model in Event-B includes variables representing the software-based controller ('the controller') plus the physical entities in the environment ('the environment') that are monitored and controlled by the controller. At the abstract modelling level it is convenient to allow all variables to be available globally so that a control decision that affects one device can depend on the state of another device, e.g., the landing gear door should only be extending or retracting if the gear door is open. At the more detailed design level we need to model the fact that decisions about which environment phenomenon to control are made by the controller. In order to make *control decisions* (i.e., decisions that control a phenomenon in the environment) the controller typically needs to know the values of the environment variables. At implementation level, the controller will have internal state representing its model of the state of the physical devices received via sensors and signals. The aim is to refine the closed model sufficiently that it may be decomposed into a model of the controller, the environment and the signalling mechanisms through which the controller and the environment interact. In order that the controller has enough information to make a decision, we use refinement to introduce a controller version of each environment variable which will be included in the controller model at decomposition. In an abstract model, a control decision may depend directly on an environment variable. In a refinement, instead of basing a control decision on the environment variable directly, the control decision will be based on the controller version of the variable. The environment variable and its corresponding controller version do not need correspond all the time. However, they do need to correspond when the controller makes a control decision; otherwise the refinement will be unsafe.

We start by presenting a standard technique for syntactically partitioning an Event-B model into several sub-models. An important property of the decomposition technique is that the resulting sub-models can be refined independently of each other. Our decomposition technique will be used to partition the behaviour of agents in a distributed architecture into separate models, including separate models of signalling mechanisms. In order to be able to decompose a model, the model needs to contain enough structure for the variables to be partitioned amongst the sub-models. For example, the model needs to contain a controller's version of an environment variable and needs to contain signalling variables. We present a refinement pattern for introducing variables and events representing the controller's management of its version of the environment variables. We also present a refinement pattern for introducing representations of the signalling mechanism. To manage the complexity of models representing multiple devices, we show how the refinement and decomposition can be applied in a stepwise manner.

4.2 Decomposing machines

In this section, we describe a parallel composition operator for Event-B machines called *shared event composition* [But09]. Machines M and N must not have any common state variables in order to be composed. Instead they interact by synchronising over shared events (i.e., events with common names). They may also exchange parameter values on synchronisation. We look first at basic composition of events and later look at composition of events with shared parameters. We show how model composition may be applied in reverse in order to decompose system models into subsystem models.

4.2.1 Parallel Composition of Machines

In general, an event has the form

$$ev \;=\; \textbf{any}\ x\ \textbf{where}\ G\ \textbf{then}\ S\ \textbf{end}$$

where x is a list of event parameters, G is a list of guards (implicitly conjoined) and S is a list of actions on the machine variables (implicitly simultaneous). We write $G \wedge H$ to join two lists of guards and $S \parallel T$ to join two lists of actions.

To achieve the synchronisation effect between machines, shared events from M and N are composed to form an event that is globally enabled when both constituent events are locally enabled and that has the effect of executing the actions of the constituent events in parallel. Assume that m (resp. n) represents the state variables of machine M (resp. N). Variables m and n are disjoint. We compose an event from M with an event from N with the following form:

$$\begin{array}{rcl} ev_M & = & \textbf{any } y \textbf{ where } G(y,m) \textbf{ then } S(y,m) \textbf{ end} \\ ev_N & = & \textbf{any } z \textbf{ where } H(z,n) \textbf{ then } T(z,n) \textbf{ end} \end{array}$$

The parallel composition of these events is a single event defined as follows:

$$\begin{array}{lcl} ev_M \parallel ev_N & \widehat{=} & \textbf{any } \ y,z \ \textbf{ where} \\ & & \quad G(y,m) \wedge H(z,n) \\ & & \textbf{then} \\ & & \quad S(y,m) \parallel T(z,n) \\ & & \textbf{end} \end{array}$$

This form of composition models synchronisation: the composite system engages in a joint event when both systems are willing to engage in that event. The parallel composition of machines M and N is a machine constructed by composing shared events of M and N and leaving independent events independent. The state variables of the composite system are formed by the union of the variables of M and N.

As an illustration of this, consider machines *V1* and *W1* of Figure 4.1. The machines work on independent variables v and w respectively. Both machines may be composed using an Event-B *composed machine* component as shown in Figure 4.2. Here $VW1$ is defined as a machine that composes $V1$ and $V2$ with events of $VW1$ being defined as compositions of events of $V1$ and $W1$. The A event of the composed machine is defined as the A event of $V1$. This means that, from the point of view of $V1$, A is an independent event in the composition since is not executed jointly with any events of $W1$. Similarly for event C of the composition. The B event of the composed machine is defined as the composition of the B-events of *V1* and *W2*. This means that B becomes a joint event in the composition that is executed jointly by $V1$ and $W1$. The initialisations of *V1* and *W1* are also combined to form the initialisation of *VW1*.

```
machine  V1
variables   v
invariants    $v \in \mathbb{N}$
init   $v := N$
event  $B \mathrel{\widehat{=}}$
   when
          grd1 : $v > 0$
   then
          act1 : $v := v - 1$
   end
event  $A \mathrel{\widehat{=}}$
   begin
          act1 : $v := N$
   end
```

(a) Machine $V1$

```
machine  W1
variables   w
invariants    $w \in \mathbb{N}$
init   $w := 0$
event  $B \mathrel{\widehat{=}}$
   when
          grd2 : $w < M$
   then
          act2 : $w := w + 1$
   end
event  $C \mathrel{\widehat{=}}$
   when
          grd1 : $w > 0$
   then
          act1 : $w := w - 1$
   end
```

(b) Machine $W1$

Figure 4.1: Machines to be composed in parallel.

```
composed machine  VW1
includes  V1, W1
init   $V1.INIT \parallel W1.INIT$
event  $A \mathrel{\widehat{=}} V1.A$
event  $B \mathrel{\widehat{=}} V1.B \parallel W1.B$
event  $C \mathrel{\widehat{=}} W1.C$
```

Figure 4.2: Composition of $V1$ and $W1$.

The expansion of the composed machine is shown in Figure 4.3. The A and C events are copied from $V1$ and $W1$ respectively. The B event of the expanded machine is formed by combing the guards and the actions of the B events in both $V1$ and $W1$. Note in practice it is not necessary to expand a composed machine. We include the expansion here to help the reader understand the effective meaning of the composed machine.

4.2.2 Synchronous Decomposition

We have presented *VW1* as having been formed from the composition of *V1* and *W1*. We can view the relationship between these machines in another way. Let us suppose we had started with a normal machine, such as the expanded version of *VW1* of Figure 4.3, and decided that we wish to *decompose* it into subsystems. The diagram in Figure 4.4(a) illustrates the dependencies between events and variables in the machine *VW1*. For example, the line from the box indicating event A to the circle indicating variable v represents the fact that event A depends on v, i.e., it may read from and assign to v. The diagram shows that B is the only event that depends on both v and w suggesting that B needs to be a shared event if we are to partition v and w into separate subsystems. This decomposition is illustrated in Figure 4.4(b) where variables v and w of *VW1* are partitioned into subsystems *V1* and *W1* respectively, A is an event of subsystem *V1*, C is an event of subsystem *W1* and B is an event shared by both subsystems.

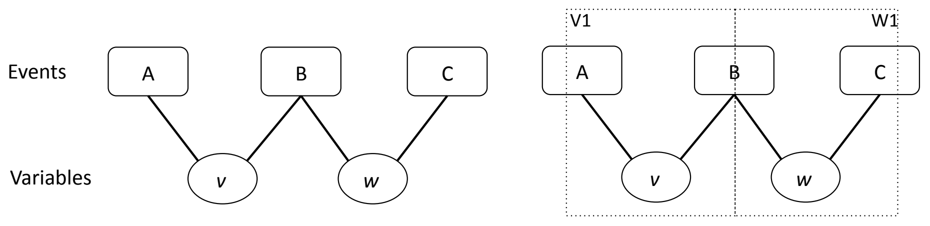
Event B of system *VW1* is partitioned into two parts, one of which will belong in *W1* and the other in *W1*. Event B has an important characteristic that allows it to be partitioned in this way. The guards and actions depend either on v or on w but not both. So, guard $grd1$ and action $act1$ both depend on v only, while guard $grd2$ and action $act2$ both depend on w. This localisation of variable dependency allows us to easily partition the guards and actions of the B event of *VW1* into the separate B events of *V1* and *W1* respectively.

4.2.3 Composition with shared event parameters

We extend the event composition operator to deal with shared event parameters. Events to be fused must depend on disjoint machine variables but they may have common parameters and these common parameters are treated as joint parameters in the composed event. In the following, x represents parameters that are joint across events and y and z are local to their respective

```
machine  VW1
variables   v,  w
invariants   $v \in \mathbb{N}$,  $w \in \mathbb{N}$
init   $v := N$,  $w := 0$
event  $A \mathrel{\widehat{=}}$
    begin
            act1 : $v := N$
    end
event  $B \mathrel{\widehat{=}}$
    when
            grd1 : $v > 0$
            grd2 : $w < M$
    then
            act1 : $v := v - 1$
            act2 : $w := w + 1$
    end
event  $C \mathrel{\widehat{=}}$
    when
            grd1 : $w > 0$
    then
            act1 : $w := w - 1$
    end
```

Figure 4.3: Expansion of $VW1$.



(a) Variable access by events in VW (b) Split events and variables

Figure 4.4: Illustration of decomposition a machine

events:

$$\begin{aligned} ev_M &= \textbf{any } x, y \textbf{ where } G(x,y,m) \textbf{ then } S(x,y,m) \textbf{ end} \\ ev_N &= \textbf{any } x, z \textbf{ where } H(x,z,n) \textbf{ then } T(x,z,n) \textbf{ end} \end{aligned}$$

The composition of these, defined as follows, makes x a single parameter of the composed event:

$$ev_M \parallel ev_N \quad \widehat{=} \quad \begin{array}{l} \textbf{any } \ x, y, z \ \textbf{ where} \\ \quad G(x,y,m) \wedge H(x,z,n) \\ \textbf{then} \\ \quad S(x,y,m) \parallel T(x,z,n) \\ \textbf{end} \end{array}$$

We illustrate the use of shared parameters by extending the machine of Figure 4.3 slightly. Assume that instead of increasing v and decreasing w by 1 in the B event, we modify both v and w by a value i. To do this we give the B event a parameter i which is used to modify the variables as follows:

```
event  $B \mathrel{\widehat{=}}$
    any  $i$
    where
        grd1 : $0 \leq i \leq v$
        grd2 : $w < N$
    then
        act1 : $v := v - i$
        act2 : $w := w + i$
    end
```

Now we partition the guards and actions of B into those that depend on v and those that depend on w giving the following events:

```
event  $B \mathrel{\widehat{=}}$
    any  $i$
    where
        grd1 : $0 \leq i \leq v$
    then
        act1 : $v := v - i$
    end
```

```
event  $B \mathrel{\widehat{=}}$
    any  $i$
    where
        grd1 : $i \in \mathbb{Z}$
        grd2 : $w < N$
    then
        act1 : $w := w + i$
    end
```

The shared parameter i means that both of these events will agree on the amount by which v and w are respectively decreased and increased when they synchronise. In the left hand sub-event, the guard $grd1$ constraints the value of the parameter based in the state variable v. In the right-hand sub-event, the value of i is not constrained other than a typing guard ($i \in \mathbb{Z}$). This means that the left-hand sub-event can be viewed as outputting the value i

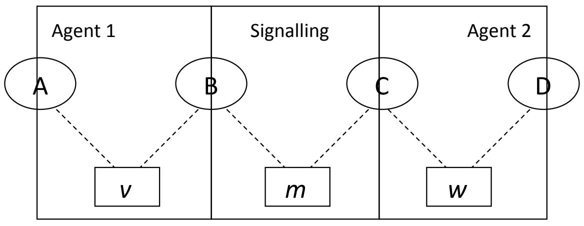


Figure 4.5: Decomposition with asynchronous middleware

while the right-hand sub-event accepts the value i as an input.

4.2.4 Independent refinement of subsystems

Shared event composition of Event-B machines is also monotonic w.r.t. refinement. This means that when we decompose a system into parallel subsystems, the subsystems may be refined and further decomposed independently [But09]. This is a major methodological benefit, helping to modularise the design and proof effort.

4.2.5 Asynchronous Decomposition

Instead of decomposing a model into two subsystems that synchronise directly with each other, we may decompose into three subsystems as illustrated in Figure 4.5. In this decomposition the two agents do not synchronise directly with each other. Instead they interact indirectly through a signalling subsystem. Each agent synchronises directly and separately with the signalling subsystem and this is used to model asynchronous communication between the agents. This form of asynchronous communication via signals can be used to model many distributed systems, including cyber-physical systems consisting of physical and controller components. In order to be able to decompose in this way, we will need to apply refinement steps that enable the agents to be decomposed into asynchronous subsystems and this is the subject of the next section.

4.3 Introducing Controller version of Environment Variables

We will first illustrate the pattern for introducing controller versions of environment variables through an example of a very simple landing gear controller for an aircraft. In this model the landing gear is either *up* or *down*. The aircraft also has a door protecting the landing gear compartment and a safety requirement is that the gear may only make a transition from *up* to *down* or vice versa when the door is open. The Event-B model is shown in Figure 4.6. The model contains a variable for the door state and a variable for the gear state. It also contains four events for opening and closing the door and for raising and lowering the gear. The door events are very simple and are independent of the gear state. The gear events include a guard to ensure that the gear only changes when the door is open, capturing the above mentioned requirement.

Our aim is to decompose this system model to an architecture consisting of the following components:

- A model of the (physical) door
- A model of the (physical) landing gear
- A model of the controller
- A model of the signalling between the door and the controller
- A model of the signalling between the landing gear and the controller

To achieve this, we need to introduce events and variables to represent the controller's behaviour and state explicitly. We will focus on the door-opening behaviour initially. In the refinement the controller should initiate the opening of the door which in turn will trigger the environment to open the door. After the door has opened in the environment, the controller should register this change in its internal state. To achieve this we introduce a new variable *doorstateC* representing the controllers version of the *doorstate* variable. We also introduce new events to represent the controller triggering the door transition and representing the controller registering that the transition has taken place in the environment.

The relationship between the door opening events is illustrated by the ERS (Event Refinement Structure) diagram in Figure 4.7. Here the single *OpenDoor* event in the abstract model is replaced by three events in the refinement and these three refinement events occur in the sequential order shown in the diagram (left to right). We use a naming convention to distinguish events of the controller (ending in 'C') from events of the environment (ending in 'E'). The dashed lines indicate that *OpenDoorStartC* and *OpenDoorFinC* are new events (refining *skip*). The solid lines indicates that *OpenDoorE* is a refinement of the abstract *OpenDoor* event. When the controller initiates the opening of the door (*OpenDoorStartC* event), the

```
machine  SimpleLG1
variables  doorstate, gearstate
invariants

          inv1 : doorstate ∈ {closed, open}
          inv2 : gearstate ∈ {up, down}
events
init  begin

                act1 : doorstate := closed
                act2 : gearstate := down
       end
event   DoorOpen ≙
       when

                grd1 : doorstate = closed
       then

                act1 : doorstate := open
       end
event   DoorClose ≙
       when

                grd1 : doorstate = open
       then

                act1 : doorstate := closed
       end
event   GearUp ≙
       when

                grd1 : gearstate = down
                grd2 : doorstate = open
       then

                act1 : gearstate := up
       end
event   GearDown ≙
       when

                grd1 : gearstate = up
                grd2 : doorstate = open
       then

                act1 : gearstate := down
       end
```

Figure 4.6: Simple landing gear.

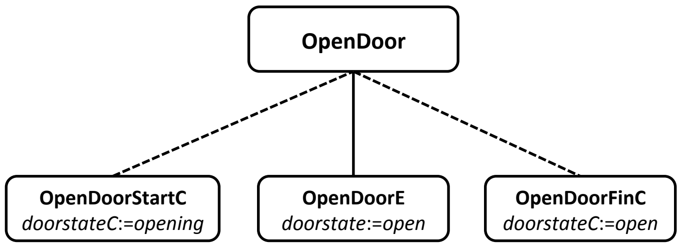


Figure 4.7: Refining the *OpenDoor* Event

controller variable *doorstateC* is assigned the value *opening* meaning the controller does not yet know whether the door is indeed open. The environment variable *doorstate* is then assigned the value *open* by the environment event *OpenDoorE* . After the door has opened in the environment, the controller variable is updated by the *OpenDoorFinC* event as the controller now knows that the door is open in the environment. The Event-B specification of the door opening events is shown in Figure 4.8.

In the events of Figure 4.8 there is still a direct dependency between the controller events and the environment events: environment event *DoorOpenE* is guarded by a condition on the controller variable *doorstateC* and similarly controller event *DoorOpenFinC* is guarded by a condition on the environment variable *doorstate*. We will make this dependency indirect in the next section by introducing an explicit signalling mechanism between the controller and the door.

Before this we turn our attention to the *GearUp* and *GearUp* events. In the abstract model (Figure 4.6), both these events are guarded by the value of *doorstate* directly. In the refinement we replace *doorstate* by *doorstateC* as shown in Figure 4.9. This represents the fact that the decision about whether it is safe to move the landing gear is based on the controllers model of the state of the gear rather than the actual state of the gear in the environment. To justify the correctness of this replacement of the environment variable by the controller variable, we need to provide and verify the invariant shown in Figure 4.6 that specifies that the refined *grd2* entails the corresponding abstract *grd2* for both events. This invariant captures the key property of *doorstateC*: when its value is used to make a control decision about the landing gear, then its value corresponds to the value of the environment variable that it shadows. There are states when *doorstateC* and *doorstate* differ, e.g., when *doorstateC=opening*, but that does not matter since, in those states, the value of *doorstateC* is not used to make a control decision about the landing gear.

All of the proof obligations for the refined model of Figures 4.8 and 4.9

```
event  $DoorOpenStartC \;\widehat{=}$
    when
            grd1 : $doorstateC = closed$
    then
            act1 : $doorstateC := opening$
    end
event  $DoorOpenE \;\widehat{=}$
  refines  $DoorOpen$
    when
            grd1 : $doorstate = closed$
            grd2 : $doorstateC = opening$
    then
            act1 : $doorstate := open$
    end
event  $DoorOpenFinC \;\widehat{=}$
    when
            grd1 : $doorstate = open$
            grd2 : $doorstateC = opening$
    then
            act1 : $doorstateC := open$
    end
```

Figure 4.8: Refined Door Opening Events.

```
invariants

        inv3 : $doorstateC = open \Rightarrow doorstate = open$
event  $GearUp \widehat{=}$
   refines  $GearUp$
      when

         grd1 : $gearstate = down$
         grd2 : $doorstateC = open$
      then

         act1 : $gearstate := up$
      end
event  $GearDown \widehat{=}$
   refines  $GearDown$
      when

         grd1 : $gearstate = up$
         grd2 : $doorstateC = open$
      then

         act1 : $gearstate := down$
      end
```

Figure 4.9: Refined Gear Movement Events.

are proved (automatically) by the Rodin provers provided *inv*3 of Figure 4.9 is included in the model. Note that the following invariant about the door closed state also holds, though it is not required to prove the refinement (because the gear control events are not enabled when the door is closed):

$$doorstateC = closed \;\Rightarrow\; doorstate = closed$$

4.4 Introducing Explicit Signalling

Consider again the door opening events of Figure 4.8: environment event *DoorOpenE* is guarded by a condition on the controller variable *doorstateC* and similarly controller event *DoorOpenFinC* is guarded by a condition on the environment variable *doorstate*. In this section we introduce an explicit signalling mechanism between the controller and the door. The signalling is a shared resource between the door and the controller and will be used to replace the direct dependency between the controller events and environment variable (and between the environment event and controller variable).

First we present a general pattern for introducing signalling between pairs of events. Suppose we have a pair of events $Ev1$ and $Ev2$ where execution of $Ev1$ may result in $Ev2$ being enabled. We wish to be able to decompose the model so that $Ev1$ appears in one agent, *Agent1*, and $Ev2$ appears in another agent, *Agent2*. We also require that *Agent1* and *Agent2* do not synchronise with each other directly but interact indirectly through a signalling agent (as in Figure 4.5).

Figure 4.10 provides a schematic representation of the way in which a signalling mechanism may be introduced as a refinement of this pair of events. In Figure 4.10 the abstract event $Ev1$ sets variable $v1$ to the value E. Because event $Ev2$ is guarded by $v1 = E$, execution of $Ev1$ may result in $Ev2$ being enabled. One possibility would be to treat $v1$ as a resource shared by both agents. However, assume that we require $v1$ to be part of *Agent1* as it is used by other events of that agent and not be a shared resource between the agents. Under these constraints, decomposition of $Ev1$ and $Ev2$ into two non-synchronising agents is not possible because $Ev2$ depends on variable $v1$. We overcome this by introducing a signalling variable *sig* that acts as a shared resource between both agents. In the refined events of Figure 4.10, $Ev1$ sets the signal variable to a value that enables $Ev2$ and the abstract guard $v1 = E$ of $Ev2$ is replaced by a guard on the signal variable. Thus, in the refinement, the sequential dependency between $Ev1$ and $Ev2$ is achieved via the shared *sig* resource rather than via the variable $v1$ intended for *Agent1*. The $Ev2$ event also resets the signal variable to a value representing the absence of a signal (*noSIG*) indicating that *Agent2* has received the signal.

Abstract events:

$$\begin{array}{lcl} Ev1 & \widehat{=} & \textbf{when } G1 \textbf{ then } A1 \;||\; v1 := E \textbf{ end} \\ Ev2 & \widehat{=} & \textbf{when } v1 = E \wedge G2 \textbf{ then } A2 \textbf{ end} \end{array}$$

Refined events with signalling:

$$\begin{array}{lcl} Ev1 & \widehat{=} & \textbf{when } G1 \textbf{ then } A1 \;||\; v1 := E \;||\; sig := SIG \textbf{ end} \\ Ev2 & \widehat{=} & \textbf{when } sig = SIG \wedge G2 \textbf{ then } A2 \;||\; sig := noSIG \textbf{ end} \end{array}$$

`inv` : $sig = SIG \;\Rightarrow\; v1 = E$

Figure 4.10: Pattern for introducing signalling.

To ensure the correctness of this refinement pattern, an invariant is required that specifies a relationship between the value of the signal variable and variable $v1$ as shown in Figure 4.10.

In the case that there are multiple sequentially dependent pairs of events from *Agent1* to *Agent2*, then we can use the same signalling variable for those event pairs. For example, in the refined landing gear model, *DoorOpenE* depends on *DoorOpenStartC* and *DoorCloseE* depends on *DoorCloseStartC*. We can group these pairs because

- the signalling is in the *same direction* (from Controller to Door), and
- the pairs are *mutually exclusive*, i.e., the controller will not initiate door opening and closing at the same time.

If there is sequential dependency between the agents in the other direction, as is often the case, then we introduce a separate shared signalling variable using the same refinement pattern.

We apply these signalling guidelines and pattern to the refined landing gear model. The refinement step that introduces the signalling has two additional variables, one to represent actuation signals from the controller to the door and another to represent confirmation signals back from the door to the controller:

`inv1` : $todoorsig \in SIGNAL$

`inv2` : $fromdoorsig \in SIGNAL$

SIGNAL has three possibles values representing (i) no signal present, (ii) a signal to indicate actuation/confirmation of door opening, and (iii) a signal to indicate actuation/confirmation of door closing:

$$\texttt{axm1} : partition(SIGNAL, \{noSIG\}, \{openSIG\}, \{closeSIG\})$$

To apply the pattern, we pair and group the events as follows:

Group1:	(*DoorOpenStartC*, *DoorOpenE*)
	(*DoorCloseStartC*, *DoorCloseE*)
Group2:	(*DoorOpenE*, *DoorOpenFinC*)
	(*DoorCloseE*, *DoorCloseFinC*)

Group1 represents signalling from the controller to the environment which is achieved using the *todoorsig* variable. *Group2* represents signalling from the environment to the controller which is achieved using the $fromdoorsig$ variable.

Based on this grouping, the refined door opening and closing events that include the signalling are derived by application of the signal-introduction pattern of Figure 4.10. The refined door opening events are shown in Figure 4.11. When the controller initiates the door opening, an *openSIG* is sent from the controller to the door by the *DoorOpenStartC* event. The guard of the environment event *DoorOpenE* that refers to the controller variable is replaced by a signalling guard. Similarly the *DoorOpenE* event sends an *openSIG* signal to indicate the the door is now open and this in turn signals the *DoorOpenFinC* event.

4.4.1 Avoiding signal confusion

Gluing invariants required by the signal-introduction pattern are presented in Figure 4.12. Invariants $inv3$ to $inv6$ are required because we replace guards on controller or environment variables by guards on the appropriate signal; so these invariants describe relations between the value of a signal and the corresponding controller or environment variable whose value the signal represents.

The other two invariants of Figure 4.12 describe conditions under which the signal variables should have the value *noSIG*. Invariant $inv7$ states that both the *to* and the *from* signals cannot have a signal simultaneously, i.e., one of the two must have the value *noSIG*. The reason that the system satisfies this invariant is because of a certain protocol: once the controller has sent a signal to the door, it does not send another signal until the door

```
event  $DoorOpenStartC \;\widehat{=}$
  refines  $DoorOpenStartC$
    when
         grd1 : $doorstateC = closed$
    then
         act1 : $doorstateC := opening$
         act2 : $todoorsig := openSIG$
    end
event  $DoorOpenE \;\widehat{=}$
  refines  $DoorOpenE$
    when
         grd1 : $doorstate = closed$
         grd2 : $todoorsig = openSIG$
    then
         act1 : $doorstate := open$
         act2 : $todoorsig := noSIG$
         act3 : $fromdoorsig := openSIG$
    end
event  $DoorOpenFinC \;\widehat{=}$
  refines  $DoorOpenFinC$
    when
         grd1 : $fromdoorsig = openSIG$
         grd2 : $doorstateC = opening$
    then
         act1 : $doorstateC := open$
         act2 : $fromdoorsig := noSIG$
    end
```

Figure 4.11: Door Opening Events with Signalling.

invariants

$$
\begin{array}{ll}
\texttt{inv3} & : todoorsig = openSIG \;\Rightarrow\; doorstateC = opening \\
\texttt{inv4} & : fromdoorsig = openSIG \;\Rightarrow\; doorstate = open \\
\texttt{inv5} & : todoorsig = closeSIG \;\Rightarrow\; doorstateC = closing \\
\texttt{inv6} & : fromdoorsig = closeSIG \;\Rightarrow\; doorstate = closed \\
\texttt{inv7} & : todoorsig = noSIG \lor fromdoorsig = noSIG \\
\texttt{inv8} & : doorstateC \in \{open, closed\} \;\Rightarrow\; fromdoorsig = noSIG
\end{array}
$$

Figure 4.12: Invariants for Signalling.

has responded. Later we will see that it is possible to allow for more liberal protocol where the controller can send another signal without waiting for a response from the device (e.g., in case of a timeout or in case the previous signal needs to be overridden).

An interesting question is how does the need for $inv7$ manifest itself in terms of proof. Without $inv7$ it cannot be proved that $inv3$ is preserved by *DoorOpenFinC*: this event sets *doorstateC* to a value different to *opening* and in this case, to preserve $inv3$, *todoorsig* should be different from *openSIG*. Invariant $inv7$ together with $grd1$ of *DoorOpenFinC* ensure that the value of *todoorsig* is different from *openSIG*. For similar reasons, $inv7$ is also required to prove that *CloseOpenFinC* preserves $inv5$.

Invariant $inv8$ states that if the controller believes the door is open or closed, then there should not be any outstanding signal from the door to the controller. A state violating this invariant could lead to a hazardous state where the controller has the wrong view of the door state. The verification need for $inv8$ arises from the need to prove that the the controller initiation events (*DoorOpenStartC, DoorCloseStartC*) preserve invariant $inv7$.

All of the proof obligations for the refined model of Figure 4.11 are proved (automatically) by the Rodin provers when all the invariants of Figure 4.12 are included in the model.

4.4.2 Parameterising the signal events and decomposing models

Recall from Section 4.2.2 that in order to decompose a machine, we identify how the variables should be partitioned amongst the sub-components. Based on this partition, events that depend on variables in more than one partition need to be decomposable, that is, each guard and action of an event should depend only on variables of a single sub-component. Being decomposable means that an event can be syntactically decomposed into several sub-events, one for each sub-component on which it depends.

The introduction of the signalling mechansim to the simple landing gear example means that the events of the model have enough structure to be able to decompose it into three sub-components as follows:

- *Door*, with variable *doorstate*
- *Signals*, with variables *todoorsig, fromdoorsig*
- *Controller*, with variable *doorstateC*

Each event either depends on variables of *Controller* and *Signals* (and not *Door*) or on variables of *Door* and *Signals* (and not *Controller*). All of the events are syntactically decomposable based on this partitioning of the variables amongst the sub-components.

For example, consider the *DoorOpenStartC* event of Figure 4.11: $grd1$ and $act1$ depend on variable *doorstateC* and can be used to construct an event of the *Controller* sub-component while $act2$ depend on variable *todoorsig* and can be used to construct an event for the *Signals* sub-component. However, a property of this decomposition of the landing gear model is that the behaviour of the events in the *Signals* sub-component depends on particular signal values (e.g., $act2$ of *DoorOpenStartC* depends on the value *openSIG*). We would prefer that the behaviour of the signalling mechanism is independent of the values of the signals since its role is simply to pass signals between the controller and the door. The behaviour of only the controller and the door should depend on the signal values.

We can achieve this by introducing the signal value as an explicit parameter of the events, with the value of the signal parameter being determined by the controller (or the door) and simply used by signalling mechanism without interpretation. Figure 4.13 presents a pattern for introducing a parameter to represent some expression appearing in an event. The figure shows that an expression E appearing in the guards and events may be abstracted by a new parameter y by adding a guard $y = E$ and replacing the occurrences of E by y.

Figure 4.14 illustrates the result of applying the parameter introduction pattern to two events of the controller. In both events, a parameter *sig* is introduced and a guard added to define the value of *sig* for the event. Occurrences of the signal value (i.e., *openSIG, closeSIG* are replaced by a reference to *sig*. The means that the parts of the events that are used to

Abstract event that depends on expression E:

$$Ev \;\; \widehat{=} \;\; \textbf{any } x \textbf{ where } G(E) \textbf{ then } A(E) \textbf{ end}$$

Refined event with additional parameter represent expression E:

$$Ev \;\; \widehat{=} \;\; \textbf{any } x, y \textbf{ where } y = E \wedge G(y) \textbf{ then } A(y) \textbf{ end}$$

Figure 4.13: Pattern for representing expressions as parameters.

construct the sub-events for the signalling sub-component are independent of the value of the signal. Furthermore for both events of Figure 4.14, the sub-events for the signalling sub-component are identical, i.e., $act2$ is the same in both events. This means that we can use a single sub-event in the signalling subcomponent to represent the contribution it makes to both the events of Figure 4.14.

After introducing the signal values as event parameters we decompose the model into the three desired sub-models. The controller events *DoorOpen-StartC, DoorOpenFinC, DoorCloseStartC, DoorCloseFinC* are syntactically split into sub-events for the controller and sub-events for the signalling component. Likewise the door *DoorOpenE, DoorCloseE* are syntactically split into sub-events for the controller and sub-events for the signalling component. The composed machine of Figure 4.15 shows how the three sub-components are brought together and appropriate combinations of events from the sub-components are composed to form refinement of the system-level events.

Some of the events of the sub-models are shown in Figures 4.16, 4.17 and 4.18. The opening events of the controller sub-model are shown in Figure 4.16, the opening event of the door sub-model is shown in Figure 4.17, and the events of the signalling sub-model are shown in Figure 4.18.

4.5 Pattern for non-linear controller decomposition

The model of door movements used in the previous sections is simplistic in a number of aspects. Firstly it assumes that the physical door will transition from open to close instantaneously. A more realistic model would assume that a door movement takes time. We will represent this by having two events

```
event  DoorOpenStartC ≙
  refines  DoorOpenStartC
    any
        sig
    where
          grd1 : $sig = openSIG$
          grd2 : $doorstateC = closed$
    then
          act1 : $doorstateC := opening$
          act2 : $todoorsig := sig$
    end
event  DoorCloseStartC ≙
  refines  DoorCloseStartC
    any
        sig
    where
          grd1 : $sig = closeSIG$
          grd2 : $doorstateC = open$
    then
          act1 : $doorstateC := closing$
          act2 : $todoorsig := sig$
    end
```

Figure 4.14: Adding signalling parameters to events.

composed machine CompositeLG
refines SimpleLG3
includes Controller, Door, Signals
init $Controller.INIT \ \| \ Door.INIT \ \| \ Signals.INIT$

event $DoorOpenStartC \ \widehat{=}$
    $Controller.DoorOpenStartC \ \| \ Signals.ControllerSendSignal$
event $DoorOpenE \ \widehat{=}$
    $Door.DoorOpenE \ \| \ Signals.DoorReceiveSendSignal$
event $DoorOpenStartC \ \widehat{=}$
    $Controller.DoorOpenfinC \ \| \ Signals.ControllerReceiveSignal$

event $DoorCloseStartC \ \widehat{=}$
    $Controller.DoorCloseStartC \ \| \ Signals.ControllerSendSignal$
event $DoorCloseE \ \widehat{=}$
    $Door.DoorCloseE \ \| \ Signals.DoorReceiveSendSignal$
event $DoorCloseStartC \ \widehat{=}$
    $Controller.DoorClosefinC \ \| \ Signals.ControllerReceiveSignal$

event $GearUp \ \widehat{=} \ Controller.GearUp$
event $GearDown \ \widehat{=} \ Controller.GearDown$

Figure 4.15: Composition of controller, door and signals.

```
event   DoorOpenStartC ≙
    any
        sig
    where
        grd1 : sig = openSIG
        grd2 : doorstateC = closed
    then
        act1 : doorstateC := opening
    end
event   DoorOpenFinC ≙
    any
        sig
    where
        grd1 : sig = openSIG
        grd2 : doorstateC = opening
    then
        act1 : doorstateC := open
    end
```

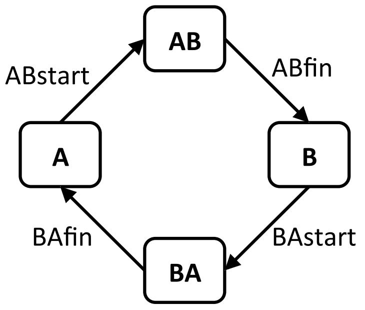
Figure 4.16: Opening events of the controller machine.

```
event   DoorOpenE ≙
    any
        sig1, sig2
    where
        grd1 : sig1 = openSIG
        grd1 : sig2 = openSIG
        grd2 : doorstate = closed
    then
        act1 : doorstate := open
    end
```

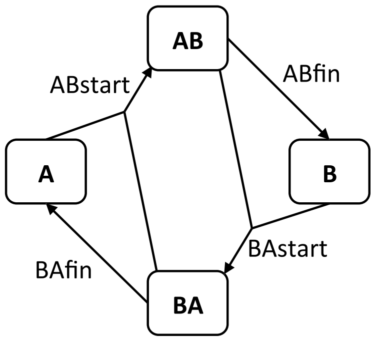
Figure 4.17: Opening event of the door machine.

```
event  $ControllerSendSignal \;\widehat{=}$
    any
          $sig$
    where
          grd1 : $sig \in SIGNAL$
    then
          act2 : $todoorsig := sig$
    end
event  $DoorReceiveSendSignal \;\widehat{=}$
    any
          $sig1, sig2$
    where
          grd1 : $todoorsig = sig1$
          grd2 : $sig2 \in SIGNAL$
    then
          act2 : $todoorsig := noSIG$
          act3 : $fromdoorsig := sig2$
    end
event  $ControllerReceiveSignal \;\widehat{=}$
    any
          $sig$
    where
          grd1 : $sig \in SIGNAL$
          grd1 : $fromdoorsig = sig$
    then
          act2 : $fromdoorsig := noSIG$
    end
```

Figure 4.18: Events of the signalling machine.



(a) Linear ordering



(b) Non-linear ordering

Figure 4.19: State machines for discrete device.

for a door movement: one to represent the point at which the door starts a transition (e.g., *OpenStart*) and another to represent the point at which the transition finishes (e.g., *OpenFin*). Figure 4.19(a) presents a generic pattern for such behaviour as a state machine. Here, A and B are the stable states (e.g., *open*, *closed*) while AB and BA represent intermediate transitioning states (e.g., *opening*, *closing*).

Figure 4.19(a) is itself simplistic in that the ordering of the events is purely linear. The linear ordering does not allow for the possibility of reversing a transition while that transition is in progress. For example, while a door is closing, we might want to start re-opening it before it finishes closing, e.g., because an open button is pressed. The state machine of Figure 4.19(b) addresses this: as well as allowing a transition into an intermediate state to come from a stable state, it allows that transition to come from another intermediate states, e.g., the *ABstart* event, which sets the state to be AB, is enabled when the state is either A or BA.

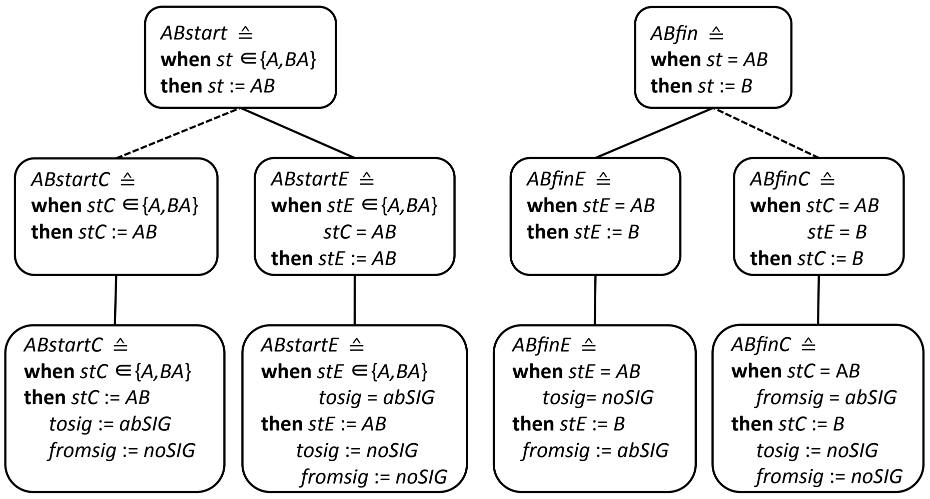
As with the door mechanism in Section 4.3, we treat the state variable represented by Figure 4.19(b) as a model of the physical device and we use a refinement step to introduce a variable representing the controllers version of the device state. After introducing the controller variable, we use a further refinement step to introduce the explicit signalling as described in Section 4.4. This is presented in Figure 4.20 which covers both refinement steps for the *ABstart* and *ABfin* events (the *BAstart* and *BAfin* events are treated in the same way). The top layer in Figure 4.20 specifies the abstract events (note that *ABstart* is enabled when the state is A or BA following Figure 4.19(b)). The middle layer shows the refined events where the controller version of the state variable, stC, is introduced (for clarity, the abstract state variable st is renamed to stE to indicate that it is an environment variable). Corresponding to each abstract variable at the top level, there are two events in the refinement, one representing a controller event and the other representing an environment event (with the controller events being new events refining *skip*). The invariants for the refinement steps are also shown in Figure 4.20. As with the refinement of the door control in Section 4.3, we have that when the controller variable is in a stable state (A or B), then the controller and environment values agree and thus is safe for the control to make a critical decision based on the value of its variable.

The bottom layer of Figure 4.20 specifies the refinement of the controller and environment events in which the signalling mechanism is introduced. This allows any dependency by controller events on environment variables to be removed and similarly for environment events and controller variables. The invariants for this refinement specify correspondences between the signal values on the state of the agent sending the signal, e.g., if the signal to the device is $abSIG$ then the state of the controller is AB (inv3.1); if the signal from the device is $abSIG$ then the device has reached state B. These invariants are used to verify the correctness of the guard replacements in the refined events, e.g, $inv3.1$ justifies the replacement of the guard $stC = AB$ by $tosig = abSIG$ in the lowest level *ABstartE* event. The events at the bottom layer are in a form that allows them to be decomposed. Prior to that, we can introduce explicit parameters to represent the signals exchanged by the decomposed events using the technique described in Section 4.4. After this, the model is decomposed into three sub-models, *Controller*, *Signals* and *Device*, such that:

- stC is placed in *Controller*, stE is placed in *Device* and $tosig$ and $fromsig$ are placed in *Signals*,
- Events *ABstartC* and *ABfinC* are decomposed into *Controller* parts and *Signals* parts,
- Events *ABstartE* and *ABfinE* are decomposed into *Device* parts and *Signals* parts.

4.5.1 Dealing with Errors

All real control systems have to deal with errors arising in the environment such as failure of a mechanical component or loss or delay of signals. In some cases, errors will be detected through sensors and this can be treated as another form of signal from the environment with an appropriate response from the controller such as the use of a backup mechanism or through the



Invariants for first refinement:

inv2.1 : $stE = st$

inv2.2 : $stC \in \{A, B\} \;\Rightarrow\; stC = stE$

Invariants for second refinement:

inv3.1 : $tosig = abSIG \;\Rightarrow\; stC = AB$

inv3.2 : $fromsig = abSIG \;\Rightarrow\; stE = B$

inv3.2 : $tosig = baSIG \;\Rightarrow\; stC = BA$

inv3.2 : $fromsig = baSIG \;\Rightarrow\; stE = A$

Figure 4.20: Refinement steps for non-linear controller.

transition to a failsafe mechanism. This can be modelled using appropriate environment and controller events, e.g., detection of an obstacle while a door is closing could result in a signal to the controller to revert to the *opening* state. In other cases, the controller may enter an error state because a signal was not received from a device by a particular time. Such a timeout can be modelled as a controller event that causes the controller to enter an error state in the absence of an expected signal from the environment. At the abstract level this can be done without any explicit timing. In a more detailed refinement, the timeout delay can be modelled by using a clock or counter that causes the timeout to trigger after an appropriate number of time steps – unless the expected response is received beforehand.

Typically, an error will arise when a controller is in an intermediate state between stable states, e.g., states AB and BA. Once the controller state has been introduced, an abstract (unguarded) error event could have is guards strengthened as follows:

```
event  Error ≙
    when
        grd1 : $st \in \{AB, BA\}$
    then
        act1 : $warning := TRUE$
    end
```

4.6 Stepwise decomposition

When we have a system model consisting of a number of devices that need to be controlled in a coordinated manner, decomposition can be achieved in stages. Typically the target architecture will be such that the devices will be independent from each other and will only interact via the controller. For such an architecture, a way to proceed, starting with a high level model of the system, is to extract each device out from the system model, one device at a time. To extract a device model from the system model, we introduce controller versions of the device state, then introduce signalling mechanisms using the techniques already outlined. Then we decompose the system model to a device model, a signalling model and a residual model. The residual model may contain a mix of controller and environment variables and events (not including variables and events of the extracted device). The residual model can then be further refined in the same way so that the next device

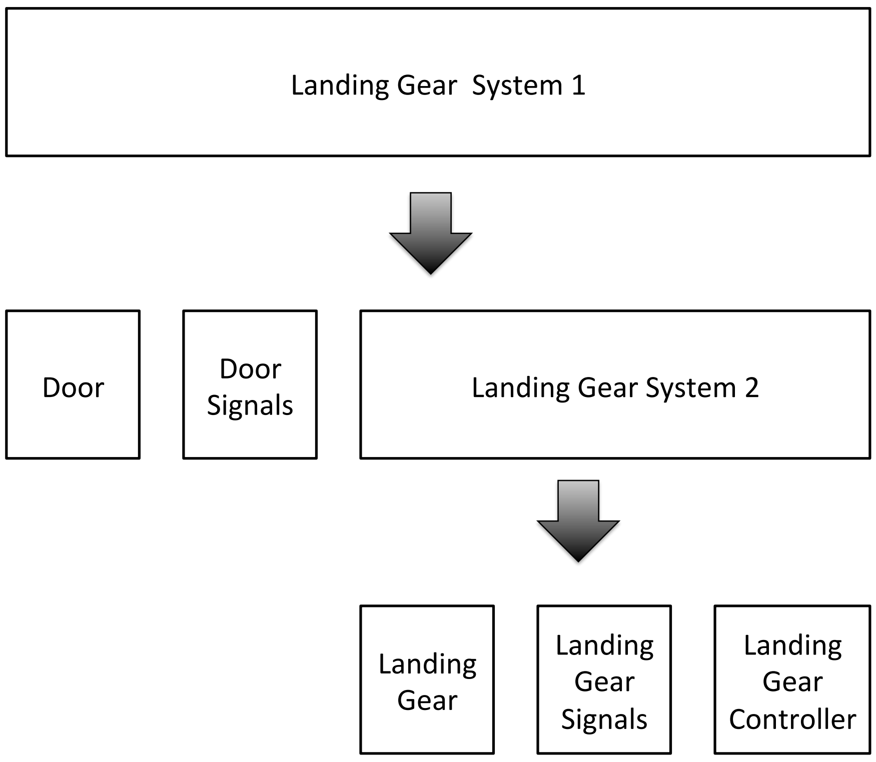


Figure 4.21: Stepwise decomposition for landing gear.

can be extracted, and so on until the residual model only contains controller variables and events representing the controller-only behaviour.

This form of stepwise refinement and decomposition based on device extraction is illustrated in Figure 4.21 for the landing gear example. The top box represents the model of the landing gear system (*Landing Gear System 1*). The first goal is to extract the door model. This is achieved by refinement steps to introduce the controller version of the door variables followed by the signalling mechanisms. The refined model is then decomposed into three sub-components: *Door*, *Door Signals* and *Landing Gear System 2*. The next decomposition stage is to extract the landing gear model from the system. Again this requires introduction of the controller version of the landing gear variables and of the signalling system. The refinement of *Landing Gear System 2* can then be decomposed into three components representing the device, the signalling for the landing gear and the residual model. In this case the residual model (*Landing Gear Controller*) only contains controller variables and events and can thus be treated as the specification of the controller software.

4.7 Multiple instances of device

Systems involving multiple instances of the same kind of device can be treated using the techniques outlined above. For example, a train door controller will control a collection of doors or a railway zone controller will control a collection of points and signals. The use of a state variable to model the state of an individual device can be lifted to a collection of similar devices by using functions from device instances, e.g., instead of $st \in ST$, we have:

$$st \in Device \rightarrow ST$$

The invariants used for introducing controller copies of environment variables can be lifted, e.g.,

$$\forall d \cdot d \in Device \wedge stC(d) \in \{A, B\} \;\Rightarrow\; stC(d) = stE(d)$$

Similarly the invariants and variables for the signalling introduction may be lifted to collections of similar devices.

Alternatively we can use disjoint sets to represent the set of devices in each state, as follows:

$$partition(Device, A, B, AB, BA)$$

Here, for example, $A \subseteq Device$ represents the set of devices in the A state. An invariant specifying the relationship between controller and environment state can be represented as follows:

$$A_C \subseteq A_E \quad \wedge \quad B_C \subseteq B_E.$$

(The set of devices that the controller believes are in the A state is a subset of the devices that are actually in the A state, etc.) Similarly, disjoint sets can be used for different kinds of messages, e.g.,

$$partition(tosig, abSIG, baSIG, noSIG)$$

For stepwise decomposition, we extract out a model representing the collection of similar devices, rather than each individual device, along with a lifted signalling sub-component representing the signalling between the group of extracted devices and the residual system.

4.8 Concluding

To summarise, we presented techniques for stepwise decomposition of control systems involving coordination by a controller of a number of devices. The key techniques are:

- Identification of a device (or collection of similar devices) to be extracted from the model,
- Introduction by refinement of the controller version of the environment variables representing the state of the device(s),
- Introduction by refinement of the signalling mechanism between the device(s) and the residual system,
- Decomposition of the refined system model into sub-models representing the extracted device(s), the signalling between the device(s) and the residual system.

This process is then repeated on the residual system model to extract the next device(s) until all devices have been extracted and the residual model represents the model of the controller.

Chapter 5

Model-based Testing and MC/DC Coverage

5.1 Introduction

Testing and coverage are the mechanisms in the ADVANCE process that link formal, proof-based verification with traditional, simulation-based verification techniques. By introducing testing and coverage early in the process at the formal model development stage, testability issues can be addressed much earlier to inform the design process.

5.2 MC/DC Coverage

The Modified Condition/Decision Coverage (MC/DC) measure introduced in ADVANCE can be used to verify, at every refinement level, that the guards of each event can be independently set to *FALSE*. Since, ultimately the refined model will be used to implement the digital controller, it is good practice to ensure that at even the high levels of abstraction, the controller will be *testable*.The facility is also used at every refinement stage to ensure that any *vacuous* guards are eliminated.

5.3 Model-based Testing

The ADVANCE project has developed two mechanisms for model-based testing: *constraint-based* and *scope-based*, which are described in the deliverable [Leu14]. In the first, the user provides a constraint and the tool finds a test that meets that constraint. In the second, the model checker is used to explore the state space within a *scope* provided by the user and to generate tests for all paths within that scope. Cyber-physical systems are difficult to test, and by using constraints and scoping, a set of tests can be generated which can be targeted at particular aspects of the system functionality. The coverage obtained by each set of tests is measured by using the ProB simulator to run the generated tests against the model.

Chapter 6

The ADVANCE Process in Aircraft Certification

6.1 Introduction

The ADVANCE process has been applied to certification in the railway signalling domain as reported in the deliverables [Mej14b] and [Mej14a]. The ADVANCE process is, however, also applicable to aircraft certification.

6.2 DO-178C and DO-254

The ADVANCE process supports both DO-254 and DO-178C flows for design assurance of airborne electronic hardware and software. Both standards support the use of formal methods and require MC/DC structural coverage of implementations at the highest safety criticality levels. DO-178C introduces two annexes which are particularly relevant when considering the impact that the ADVANCE process could have in this area, DO-331 addressing Model-based development and verification and DO-333 addressing Formal Methods to complement testing. The ADVANCE process combines formal and simulation-based verification with MC/DC coverage closure in a way that is well-suited to airborne system verification and sign-off, providing traceable model-based development and verification from conception to certification.

Chapter 7

Summary

We have shown that we have developed a method in ADVANCE for capturing System Requirements and providing traceability between those requirements and a formal specification of the system in Event-B. We have also developed a method for capturing Safety Requirements, using STPA, which is integrated with Functional Requirement capture and uses the ProR facility. We have also shown how this method, based on formal modelling and proof, can be integrated into the ADVANCE multi-simulation and test method. These techniques have been applied to both ADVANCE case studies.

Bibliography

[But09] Michael Butler. Decomposition Structures for Event-B. In *Integrated Formal Methods 2009*, pages 20–38, 2009.

[BW14] Frédéric Boniol and Virginie Wiels. The landing gear system case study. In *ABZ 2014: The Landing Gear Case Study*, pages 1–18. Springer, 2014.

[Leu14] Michael Leuschel. D4.4 methods and tools for simulation and testing iii. Technical report, University of Dusseldorf, 2014.

[Lev12] N.G. Leveson. *Engineering a safer world: Systems thinking applied to safety.* MIT Press (MA), 2012.

[Mej14a] Luis-Fernando Mejia. D1.4 certification strategy. Technical report, Alstom Transport, 2014.

[Mej14b] Luis-Fernando Mejia. D1.5 full case study. Technical report, Alstom Transport, 2014.

[Rei14] Jose Reis. D2.4 full application in the smart energy domain. Technical report, CSWT, 2014.