D.5.2 - ADVANCE PROCESS INTEGRATION II

ADVANCE

Grant Agreement:	287563
Date:	3/12/2013
Pages:	30
Status:	Final
Access:	Access List
Reference:	D5.2
Issue:	1

Partners / Clients:

FP7 Framework Programme	**European Union**

Consortium Members:

University of Southampton	**Critical Software Technologies**	**Alstom Transport**	**Systerel**	**Heinrich Heine Universität Düsseldorf**

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

ADVANCE Deliverable D5.2

ADVANCE Process Integration II

Public Document

December 3, 2013

`http://www.advance-ict.eu`

Contributors:

Lukas Ladenberger
John Colley

Reviewers:

Laurent Voisin

Contents

Chapter 1

Preface

This deliverable reports on the final work on combining formal modelling with requirements analysis and safety analysis, including tool support. It ensures that the tools delivered support properly the requirements and safety analysis methods that have been developed and that the accompanying documentation and tutorials are in place. It also reports on initial work on integration of design flows that combine proof and simulation.

In chapter 2 we summarise our earlier work on combining formal modelling with requirements and safety analysis to set the context for the work described in this deliverable.

In chapter 3 we show how our earlier work using System-Theoretic Process Analysis (STPA) has been extended and how the ProR developments of the first reporting period have been used to document this STPA-based approach.

In chapter 4 we report on how a test suite can be developed from a concrete model of a safety-critical system and how a controller for that system, developed using this STPA approach, can be used in ADVANCE multi-simulation.

In chapter 5 we summarise the work done so far for this workpackage and the focus of future work.

Chapter 2

Summary of earlier work

In our earlier work, described in the ADANCE deliverable D5.1, we developed a method for linking formally the safety requirements of a system to its functional requirements.

We investigate the functional requirements using a method that identifies the *system phenomena* and then structures the functional requirements according to these phenomena [YB12]. We then use System-Theoretic Process Analysis (STPA) from Leveson [Lev12], a technique for Hazard Analysis.

STPA has two main steps. The first step is focused on deriving the safety constraints on the system from the potentially hazardous control actions. The second concentrates on determining how unsafe control actions could occur.

- The Functional Requirements are developed using the System Phenomena
- The Safety Requirements are derived from the Controlled Phenomena
- The Safety Constraints are then derived systematically from the Safety Requirements, represented in natural language
- The Safety Constraints are represented formally in the Event-B model as invariants and guards

Using a domestic washing machine in our case study, we explored the application of the method to the washing machine sub-systems. In particular, we applied the method to the door sub-system, as described below.

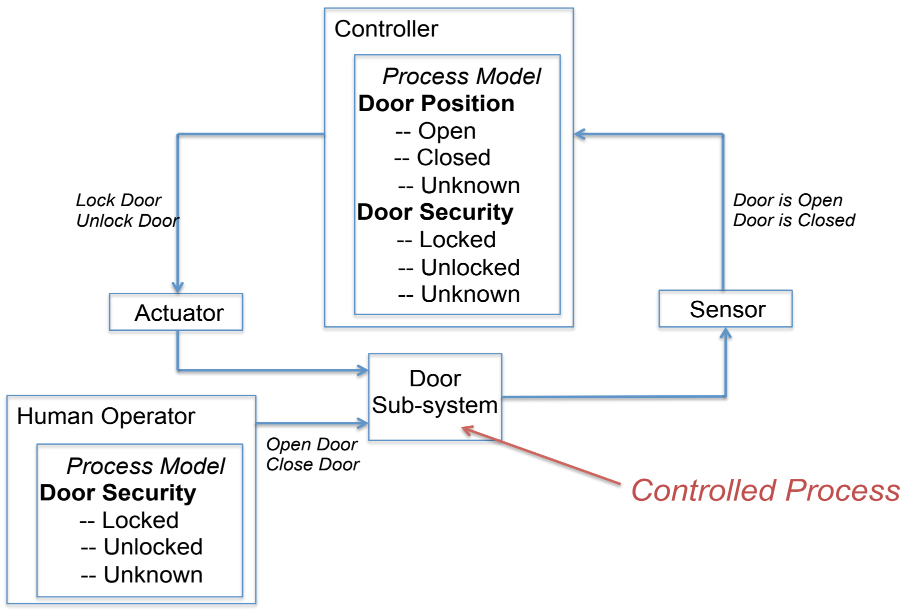


Figure 2.1: The Controlled Door Sub-system

The Door Sub-system

Consider first a model of the Controlled Door Sub-system as shown in Figure 2.1.

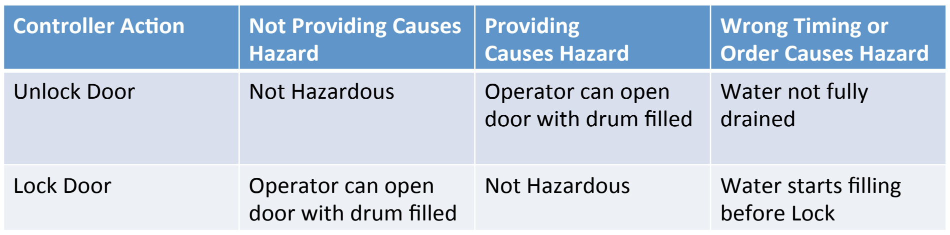
The Main Controller has a *Process Model* of the Door Sub-system. So also does the Human Operator. The Operator can open or close the door directly. The Controller uses an *Actuator* to lock and unlock the door and a *Sensor* to detect whether the door is open or closed.

Step I: Identifying Potentially Hazards Control Actions

For each of the two Controller actions, *Unlock Door* and *Lock Door*, we identify three potential causes of a hazard: *not providing* the action when it should, *providing* the action when it shouldn't and providing the action at the *wrong time* or in the *wrong order*. The results of the analysis are shown in in Figure 2.2.

Failing to unlock the door is inconvenient but not hazardous. Unlocking the door when the drum is filled is hazardous because the operator will be able to open the door inadvertently and release potentially very hot water. Unlocking the door *before* the drum has been fully drained is also hazardous.

Failing to lock the door when the drum is filled is hazardous, but locking the door when the drum is empty is not. Locking the door *after* the drum



Controller Action	Not Providing Causes Hazard	Providing Causes Hazard	Wrong Timing or Order Causes Hazard
Unlock Door	Not Hazardous	Operator can open door with drum filled	Water not fully drained
Lock Door	Operator can open door with drum filled	Not Hazardous	Water starts filling before Lock

Figure 2.2: Hazards: Door

has started filling is hazardous.

Step II: Deriving the Safety Constraints

Three Safety Constraints can be derived from Figure 2.2

1. The Door must always be locked when there is water in the Drum
2. An *Unlock Door* command must never be issued until the water is fully drained
3. A *Lock Door* command must be issued before starting to fill the Drum

The first is an *Invariant* of the system. The second and third are *Guards* that prevent an operation occurring in an unsafe way. These natural language invariants and guards can then be represented formally in an Event-B model [Abr10].

We will now go on to explore more fully the modelling of the safety constraints of the door sub-system in Event-B.

Chapter 3

Modelling the Door Sub-system in Event-B

3.1 The Abstract Model

> "Any controller - human or automated - needs a model of the process being controlled to control it effectively."
>
> "Accidents can occur when the controller's process model does not match the state of the system being controlled and the controller issues unsafe commands."
>
> Engineering a Safer World, Leveson, 2012

We therefore begin with an abstract model of the process in Event-B to represent the controller's process model shown in Figure 2.1 above, which subsequently we shall refine. Our refinement strategy mirrors the STPA method.

We introduce an Event-B context which defines the *Door Position* as *OPEN* or *CLOSED*. For the *Door State*, however, we introduce three states: *LOCKED*, *UNLOCKED*, or *UNKNOWN*, since we have already identified through STPA that the locking mechanism is the the source of potential hazards which we must mitigate. If the door lock develops a fault during operation and the controller detects this fault, the controller may not know whether the door is actually locked or not. The context is shown below.

CONTEXT DOORC

SETS

$DoorPosition, DoorState$

CONSTANTS

$OPEN, CLOSED$

$LOCKED, UNLOCKED, UNKNOWN$

AXIOMS

`axm1` : $partition(DoorPosition, \{OPEN\}, \{CLOSED\})$

`axm2` : $partition(DoorState, \{LOCKED\}, \{UNLOCKED\}, \{UNKNOWN\})$

END

We then define the abstract machine for the door sub-system, which has two variables, *dpos* and *doorst*, which are initialised to *OPEN* and *LOCKED* respectively.

MACHINE DOORM

SEES DOORC

VARIABLES

$dpos$

$doorst$

INVARIANTS

`inv1` : $dpos \in DoorPosition$

`inv2` : $doorst \in DoorState$

EVENTS

Initialisation

begin

`act1` : $dpos := OPEN$

`act2` : $doorst := UNLOCKED$

end

We can define the door operations as events. The door can be closed if it is open and can be opened if it is closed, but not locked.

Event $CloseDoor \mathrel{\widehat{=}}$
    when
        grd1 : $dpos = OPEN$
    then
        act1 : $dpos := CLOSED$
    end

Event $OpenDoor \mathrel{\widehat{=}}$
    when
        grd1 : $dpos = CLOSED$
        grd2 : $doorst \neq LOCKED$
    then
        act1 : $dpos := OPEN$
    end

The door can be locked if it is closed and unlocked. It can be unlocked if it is locked.

Event $LockDoor \mathrel{\widehat{=}}$
    when
        grd1 : $doorst = UNLOCKED$
        grd2 : $dpos = CLOSED$
    then
        act1 : $doorst := LOCKED$
    end

Event $UnlockDoor \mathrel{\widehat{=}}$
    when
        grd1 : $doorst = LOCKED$
    then
        act1 : $doorst := UNLOCKED$
    end

For STPA we introduce an extra event *DetectDoorFault*. In any good state of the system, if the controller detects a door fault then the door may or may not be locked, which results in the doorst going to *UNKNOWN*.

Event *DetectDoorFault* $\widehat{=}$

  when

    `grd1` : $doorst \neq UNKNOWN$

  then

    `act1` : $doorst := UNKNOWN$

  end

END

The graph for the abstract model, which is produced automatically by ProB [LB08], is shown in Figure 3.1.

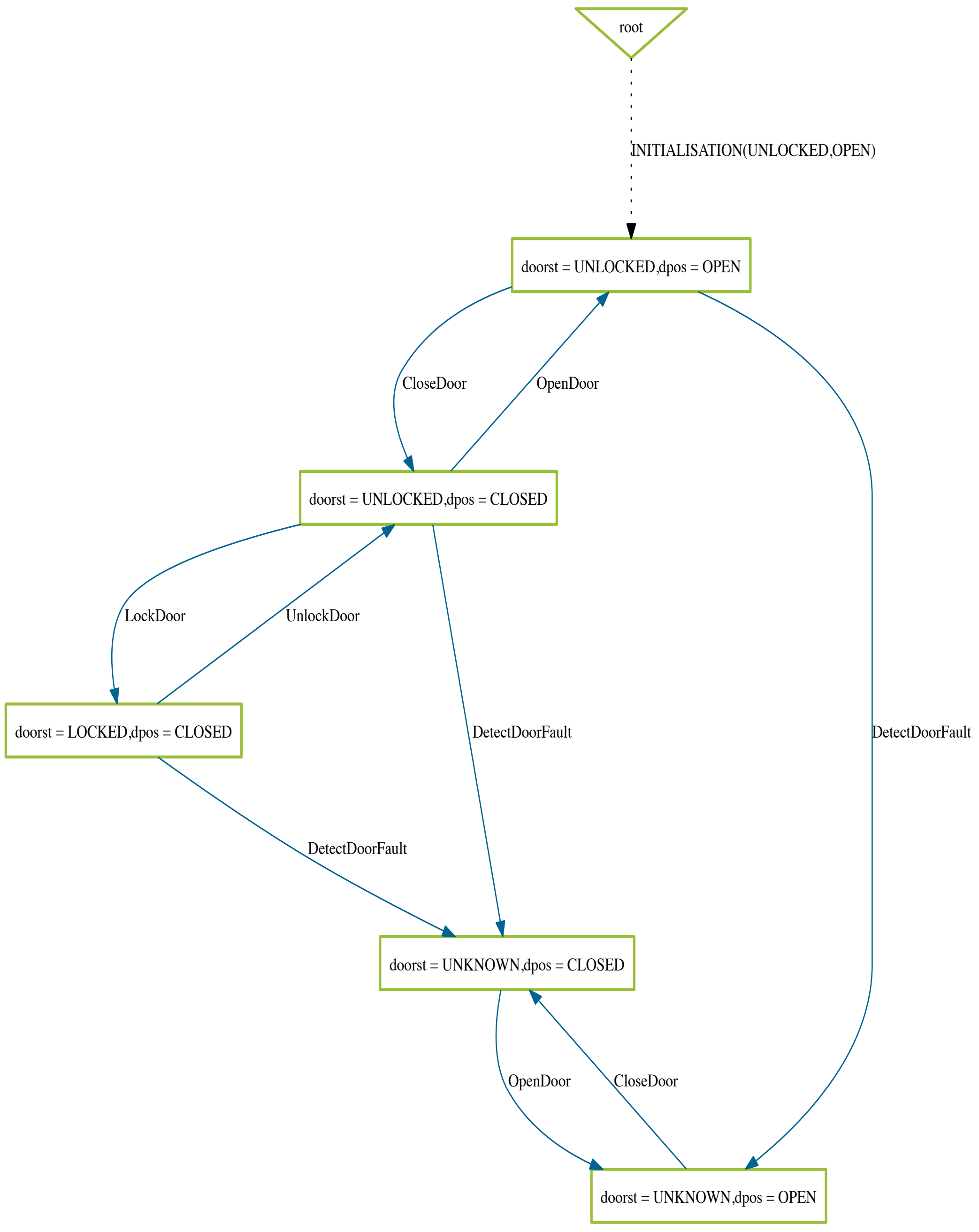

root
INITIALISATION(UNLOCKED,OPEN)
doorst = UNLOCKED,dpos = OPEN
CloseDoor
OpenDoor
doorst = UNLOCKED,dpos = CLOSED
LockDoor
UnlockDoor
doorst = LOCKED,dpos = CLOSED
DetectDoorFault
DetectDoorFault
DetectDoorFault
doorst = UNKNOWN,dpos = CLOSED
OpenDoor
CloseDoor
doorst = UNKNOWN,dpos = OPEN

Figure 3.1: The Abstract Process Model Graph

3.2 The First Refinement: modeling the safety constraints

We now *extend* the Event-B context to introduce the notion of the *Drum-State*. The drum is either *EMPTY*, *FILLING*, *FILLED*, or *EMPTYING*.

CONTEXT LOCKSE1
EXTENDS LOCKSC
SETS

$DrumState$

CONSTANTS

$EMPTY, FILLING, FILLED, EMPTYING$

AXIOMS

axm1 : $partition(DrumState, \{EMPTY\}, \{FILLING\}, \{FILLED\}, \{EMPTYING\})$

END

The abstract process model is *refined*, introducing the variable *drumst* to represent the state of the drum and safety constraints developed using STPA are modelled.

Recall the derived safety constraints for the door sub-system.

1. The Door must always be locked when there is water in the Drum
2. An *Unlock Door* command must never be issued until the water is fully drained
3. A *Lock Door* command must be issued before starting to fill the Drum

We first consider safety constraint *2*. The *Unlock Door* command is represented by the event *UnlockDoor* in the abstract Event-B machine: if the door is locked, then the door can be unlocked. In the refinement of the event we introduce an extra *guard*.

grd2 : $drumst = EMPTY$

This guard is a direct, formal representation of safety constraint *2*: the door cannot be unlocked unless the drum is *EMPTY*, as shown in the refined event below.

Event *UnlockDoor* $\widehat{=}$
refines *UnlockDoor*
  when
    `grd1` : $doorst = LOCKED$
    `grd2` : $drumst = EMPTY$
  then
    `act1` : $doorst := UNLOCKED$
  end

We then consider safety constraint *3*. We introduce a new event, *FillDrum* in the refined machine.

Event *FillDrum* $\widehat{=}$
  when
    `grd1` : $doorst = LOCKED$
    `grd2` : $drumst = EMPTY$
  then
    `act1` : $drumst := FILLING$
  end

The first guard of this event, *grd2*, is an indirect, formal representation of safety constraint *2*: the empty drum cannot be filled unless the door is locked, but the door can only be locked if the event *LockDoor* is activated. Therefore, a *lock door* command must always precede a *fill drum* command.

The safety constraint *1* is an invariant of the system: the door must always be locked when there is water in the drum. and can be represented formally in Event-B thus.

`inv2` : $drumst \neq EMPTY \Rightarrow doorst = LOCKED$

When we run the Rodin [ABH^{+}10] provers, however, we find that this safety invariant is not preserved by the event *DetectDoorFault*. From STPA, if the door sub-system develops a fault during normal operation and the controller detects the fault, the controller will not know whether the door is locked or not. If the drum is not empty when the fault arises and the door is indeed no longer locked, then the system does not preserve the safety invariant. We can, however, say that under normal operation in the absence of faults, the following invariant,

`inv2` : $drumst \neq EMPTY \Rightarrow doorst \neq UNLOCKED$

is preserved and the Rodin provers confirm this. We must therefore make a design decision. Do we wish to eliminate the hazard or mitigate it when we detect it? If a *fail safe* door subsystem component was selected for the design, the controller would always know whether the door was locked or not. When a door fault was detected the state of the door would go to $LOCKED$, the hazard would be eliminated and the invariant

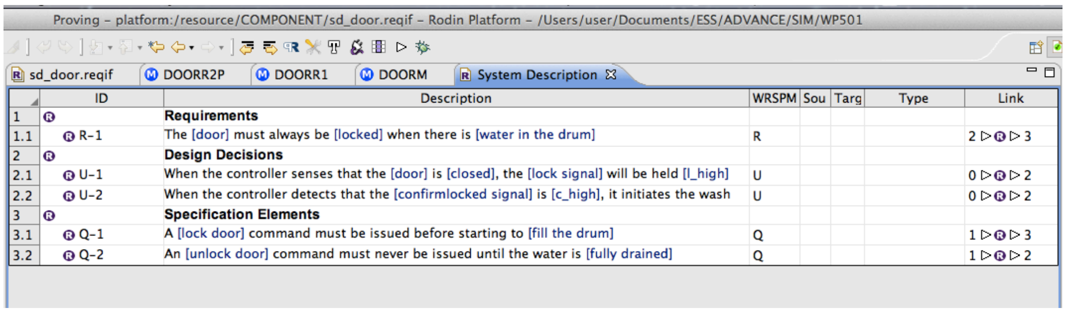
$$\texttt{inv2} : drumst \neq EMPTY \Rightarrow doorst = LOCKED$$

would be preserved by the system. A fail-safe washing machine door subsystem would, however, be very inconvenient to a domestic user, so we choose to *mitigate* the hazard rather than *eliminate* it and use the invariant.

$$\texttt{inv2} : drumst \neq EMPTY \Rightarrow doorst \neq UNLOCKED$$

We will then, in a subsequent refinement have to develop a mitigation strategy, and prove that the design implements the mitigation strategy under all possible failure circumstances.

We document the safety analysis using ProR [Jas10] within Rodin as shown in Figure 3.2.



Proving – platform:/resource/COMPONENT/sd_door.reqif – Rodin Platform – /Users/user/Documents/ESS/ADVANCE/SIM/WP501

	ID	Description	WRSPM	Sou	Targ	Type	Link
1		**Requirements**					
1.1	R-1	The [door] must always be [locked] when there is [water in the drum]	R				2 ▷ ▷ 3
2		**Design Decisions**					
2.1	U-1	When the controller senses that the [door] is [closed], the [lock signal] will be held [l_high]	U				0 ▷ ▷ 2
2.2	U-2	When the controller detects that the [confirmlocked signal] is [c_high], it initiates the wash	U				0 ▷ ▷ 2
3		**Specification Elements**					
3.1	Q-1	A [lock door] command must be issued before starting to [fill the drum]	Q				1 ▷ ▷ 3
3.2	Q-2	An [unlock door] command must never be issued until the water is [fully drained]	Q				1 ▷ ▷ 2

Figure 3.2: Documenting the Safety Analysis in ProR

3.3 The Second Refinement: Determining how Unsafe Control Actions could Occur

Following the second STPA step we determine how unsafe control actions could occur with respect to the hazard: *the door is open and there is water in the drum*, as shown in Figure 3.3.

- The Controller issues the lock but it is not received
- The Actuator fails and the door is not locked
- The Door is not properly closed
- Door Closed is signaled when the door is not properly closed

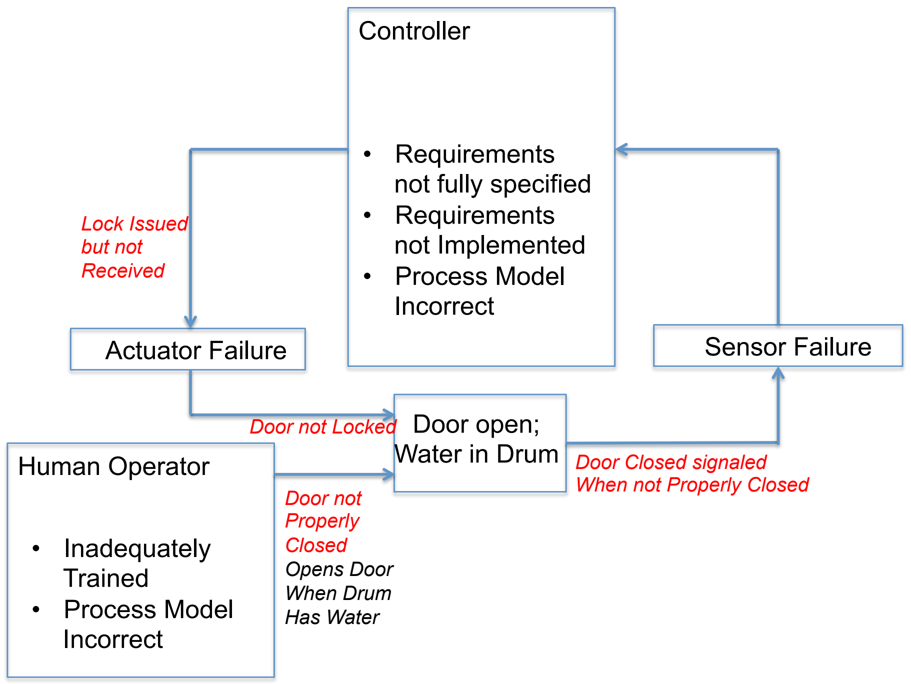


Figure 3.3: How Unsafe Control Actions can Occur

We now make the design decision to select a pre-certified door component which uses electo-mechanical interlocks as shown in Figure 3.4 below. The door component is supplied with the following *contract*.

> If *lock* is held *high* then *confirmlocked high* indicates that the door is properly shut and locked.

The door component also provides a parallel circuit with a signal which the controller can use to determine whether the door is closed or not. It has the following contract.

> If the *signal* is *high* then the door is *closed*.

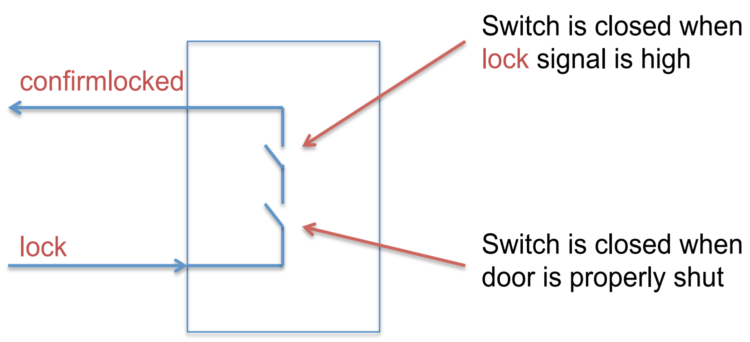

confirmlocked
lock
Switch is closed when lock signal is high
Switch is closed when door is properly shut

Figure 3.4: A Door Component with Interlocks

We have already made the decision to *mitigate* the hazard: *the door is open and there is water in the drum*, rather than eliminate it. We now refine this design decision.

> When the Controller detects that the *confirmlocked* signal is low while its internal process model says it should be high, it must
>
> - Post a warning on the control panel
> - Empty the drum if necessary
> - Prevent the user from initiating another wash

We then implement the design in this second refinement of our Event-B model, introducing the concrete signals *confirmlocked* and *lock*. We can use the variable *dpos* from the abstract model to represent the *door closed* signal.

We therefore represent two components in the refinement, the *Controller* and the *Door*, which communicate using the signals *dpos*, *confirmlocked* and *lock* as shown in Figure 3.5.

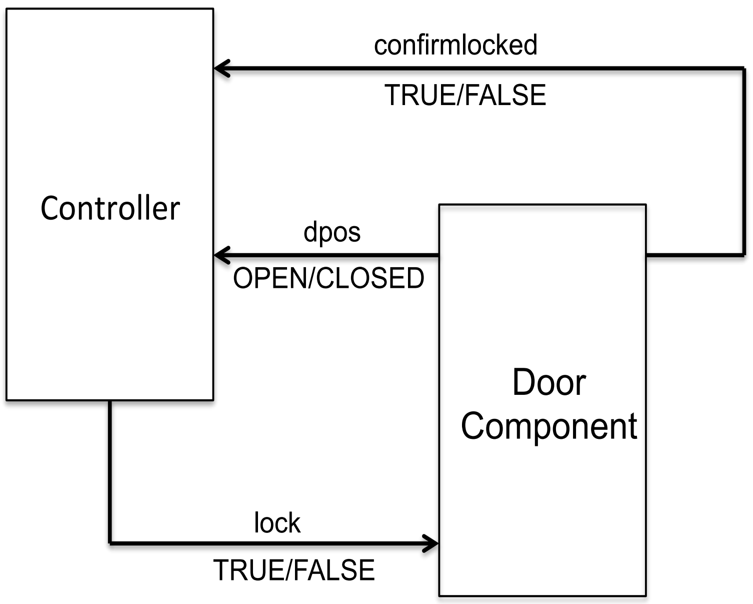


Figure 3.5: The Component View

The abstract event *CloseDoor* is renamed to indicate that it is an action of the *door component*. Otherwise it is unchanged.

Event $DoorSubsystemCloseDoor \mathrel{\widehat{=}}$
refines $CloseDoor$
  when
    grd1 : $dpos = OPEN$
  then
    act1 : $dpos := CLOSED$
  end

We then data refine the *OpenDoor* event; the abstract guard

grd2 : $doorst \neq LOCKED$

is replaced by two concrete guards

grd2 : $lock = FALSE$
grd3 : $confirmlocked = FALSE$

together with the *gluing invariant*

inv7 : $lock = FALSE \Rightarrow doorst \neq LOCKED$

The refined event is shown below.

Event *DoorSubsystemOpenDoor* $\widehat{=}$
refines *OpenDoor*
when
grd1 : $dpos = CLOSED$
grd2 : $lock = FALSE$
grd3 : $confirmlocked = FALSE$
then
act1 : $dpos := OPEN$
end

We now refine the abstract event *LockDoor*. In fact, because of the synchronisation required between the components, we need three events: the new event, *ControllerIssueLock* which refines *skip* and the events *ControllerCompleteLock* and *ControllerCompleteLockFail*.

The controller issues a lock if the controller's process model is in state *UNLOCKED*, the door position is *CLOSED* and the *confirmlock* signal is *FALSE*. The event sets the *lock* signal to *TRUE* and then *waits* for a response from the door component.

Event *ControllerIssueLock* $\widehat{=}$
when
grd1 : $doorst = UNLOCKED$
grd2 : $dpos = CLOSED$
grd3 : $confirmlocked = FALSE$
grd4 : $controllerwait = FALSE$
grd5 : $lock = FALSE$
grd6 : $controllerlockwarning = FALSE$
then
act1 : $lock := TRUE$
act2 : $controllerwait := TRUE$
end

A new door component event *DoorSubsystemLock* is introduced which detects that the *lock* signal is set to *TRUE* and then *non-deterministically* sets the *confirmlocked* signal to *TRUE* or *FALSE*. It also sets an internal variable *lockfailure* to *TRUE*. In this way the event models a possible failure in the lock.

```
Event  DoorSubsystemLock ≙
    any
        pfail
    where
        grd1 : $pfail \in BOOL$
        grd2 : $lock = TRUE$
        grd3 : $confirmlocked = FALSE$
        grd4 : $lockfailure = FALSE$
        grd5 : $doorsubsystemwait = FALSE$
    then
        act1 : $confirmlocked := pfail$
        act2 : $lockfailure := bool(pfail = FALSE)$
        act3 : $doorsubsystemwait := TRUE$
    end
```

The controller can now use its *process model* to determine whether the locking procedure has been successful or not. If the signal *confirmlocked* is set to *TRUE* by the door component, the controller knows that the door is locked and the event *ControllerCompleteLock*, which refines *LockDoor* is activated.

```
Event  ControllerCompleteLock ≙
refines  LockDoor
    when
        grd1 : $doorst = UNLOCKED$
        grd2 : $lock = TRUE$
        grd3 : $dpos = CLOSED$
        grd4 : $confirmlocked = TRUE$
        grd5 : $controllerwait = FALSE$
    then
        act1 : $doorst := LOCKED$
    end
```

The controller updates the state of its internal process model to *LOCKED*.

If, however, the signal *confirmlocked* is not set to *TRUE* by the door component, the controller knows that the door locked may have failed and the event *ControllerCompleteLockFail*, which refines *DetectDoorFailure* is activated instead.

```
Event   ControllerCompleteLockFail ≙
refines DetectDoorFault
        when
                grd1 : doorst = UNLOCKED
                grd2 : lock = TRUE
                grd3 : dpos = CLOSED
                grd4 : confirmlocked = FALSE
                grd5 : controllerwait = FALSE
        then
                act1 : lock := FALSE
                act2 : controllerlockwarning := TRUE
                act3 : doorst := UNKNOWN
        end
```

The controller updates the state of its internal process model to *UNKNOWN* and sets *controllerlockwarning* to *TRUE*.

The event *Update* controls the synchronisation between the controller and the door.

Event *Update* $\widehat{=}$

  when

    grd1 : $controllerwait = TRUE$

    grd2 : $doorsubsystemwait = TRUE$

  then

    act1 : $controllerwait := FALSE$

    act2 : $doorsubsystemwait := FALSE$

  end

Update is a formal representation of the synchronisation mechanism used in ADVANCE multi-simulation.

If the controller completes the locking successfully, it can proceed to fill the drum.

Event *ControllerFillDrum* $\widehat{=}$

refines *FillDrum*

  when

    grd1 : $doorst = LOCKED$

    grd2 : $drumst = EMPTY$

    grd3 : $confirmlocked = TRUE$

  then

    act1 : $drumst := FILLING$

  end

We now refine the safety invariant. For the abstract model, we proved the invariant

inv2 : $drumst \neq EMPTY \Rightarrow doorst \neq UNLOCKED$

In this refinement the invariant is re-cast in terms of the concrete signals.

$\texttt{inv8} : drumst \neq EMPTY \Rightarrow (lock = TRUE \land confirmlocked = TRUE) \lor lockfailure = TRUE$

If the lock is sound then *confirmlocked* will be *TRUE*. If, however, at any stage, *confirmlocked* changes to *FALSE* when the controller's process model indicates that the door is *LOCKED*, the controller must mitigate the hazard appropriately.

We introduce a new event *DoorSubsystemFail* which is always enabled when the door is *LOCKED* and can, non-deterministically set *confirmlocked* to *FALSE*

Event *DoorSubsystemFail* $\widehat{=}$

    when

        $\texttt{grd1} : lock = TRUE$
        $\texttt{grd2} : confirmlocked = TRUE$
        $\texttt{grd3} : lockfailure = FALSE$
        $\texttt{grd4} : doorst = LOCKED$

    then

        $\texttt{act1} : confirmlocked := FALSE$
        $\texttt{act2} : lockfailure := TRUE$

    end

If *DoorSubsystemFail* is activated, *lockfailure* is also set to *TRUE*.

Controller events are introduced which use its process model to detect and mitigate the hazard. For instance, the event *ControllerWash* is accompanied by an event *ControllerAbortWash*. The guards of this pair of events are identical except that in *ControllerAbortWash* we have the guard

$\texttt{grd2} : confirmlocked = FALSE$

rather than

$\texttt{grd2} : confirmlocked = TRUE$

At every node of the control graph, where the drum is not *EMPTY* a pair of events must be implemented; one to handle the good behaviour and one to detect and mitigate the hazard.

We must now *prove* that the controller mitigates the hazard correctly.

Informally,

> if *lockfailure* is non-deterministically set to *TRUE* when the drum is not *EMPTY* then *either* a controller event is enabled that will set *controllerlockwarning* to *TRUE or controllerlockwarning* has already been set to *TRUE*.

Formally,

inv13 : $lockfailure = TRUE \wedge drumst \neq EMPTY \Rightarrow (lock = TRUE \wedge doorst = LOCKED) \vee controllerlockwarning = TRUE)$

We prove that this invariant is preserved by all the events of the the refined machine. (It should be noted that the analysis of the guards of the enabled controller events that mitigate the hazard was performed manually in order to derive the formal invariant. It would be very valuable if Rodin were to provide a mechanism which performed this analysis automatically).

In fact, we can extend the invariant to ensure that the controller always sets the door state in it's internal process model to *UNKNOWN*. The guard of *ControllerIssueLock*

grd1 : $doorst = UNLOCKED$

ensures that another wash cannot be initiated if a door fault is detected.

Validating the Concrete Model

A graph of the concrete machine is extracted using ProB and used to validate the model against the requirements. (The graph is included here, Figure 3.6, to illustrate it's level of complexity.)

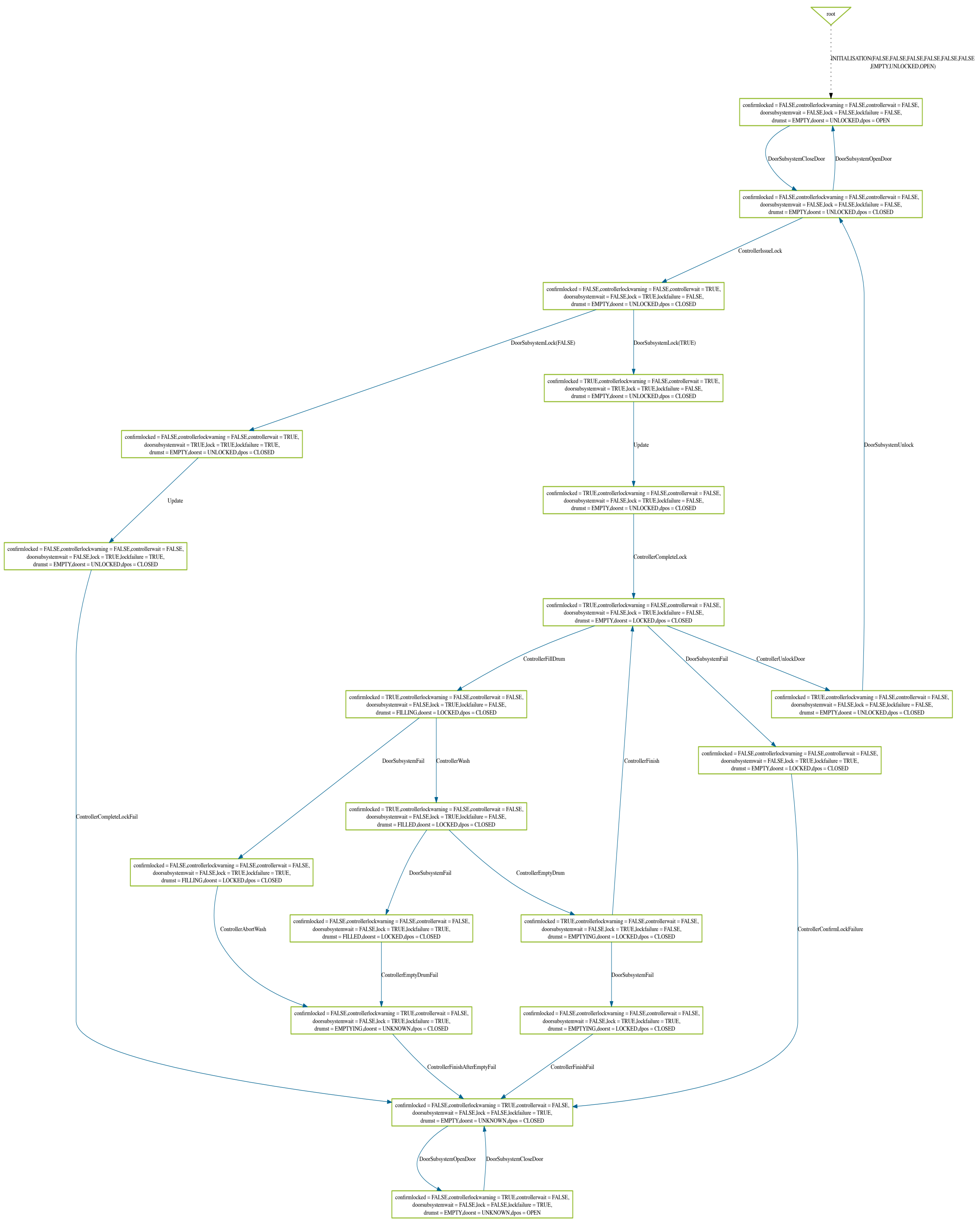

root
INITIALISATION(FALSE,FALSE,FALSE,FALSE,FALSE,FALSE ,EMPTY,UNLOCKED,OPEN)
confirmlocked = FALSE,controllerlockwarning = FALSE,controllerwait = FALSE, doorsubsystemwait = FALSE,lock = FALSE,lockfailure = FALSE, drumst = EMPTY,doorst = UNLOCKED,dpos = OPEN
DoorSubsystemCloseDoor
DoorSubsystemOpenDoor
confirmlocked = FALSE,controllerlockwarning = FALSE,controllerwait = FALSE, doorsubsystemwait = FALSE,lock = FALSE,lockfailure = FALSE, drumst = EMPTY,doorst = UNLOCKED,dpos = CLOSED
ControllerIssueLock
confirmlocked = FALSE,controllerlockwarning = FALSE,controllerwait = TRUE, doorsubsystemwait = FALSE,lock = TRUE,lockfailure = FALSE, drumst = EMPTY,doorst = UNLOCKED,dpos = CLOSED
DoorSubsystemLock(FALSE)
DoorSubsystemLock(TRUE)
confirmlocked = TRUE,controllerlockwarning = FALSE,controllerwait = TRUE, doorsubsystemwait = TRUE,lock = TRUE,lockfailure = FALSE, drumst = EMPTY,doorst = UNLOCKED,dpos = CLOSED
DoorSubsystemUnlock
confirmlocked = FALSE,controllerlockwarning = FALSE,controllerwait = TRUE, doorsubsystemwait = TRUE,lock = TRUE,lockfailure = TRUE, drumst = EMPTY,doorst = UNLOCKED,dpos = CLOSED
Update
Update
confirmlocked = TRUE,controllerlockwarning = FALSE,controllerwait = FALSE, doorsubsystemwait = FALSE,lock = TRUE,lockfailure = FALSE, drumst = EMPTY,doorst = UNLOCKED,dpos = CLOSED
confirmlocked = FALSE,controllerlockwarning = FALSE,controllerwait = FALSE, doorsubsystemwait = FALSE,lock = TRUE,lockfailure = TRUE, drumst = EMPTY,doorst = UNLOCKED,dpos = CLOSED
ControllerCompleteLock
confirmlocked = TRUE,controllerlockwarning = FALSE,controllerwait = FALSE, doorsubsystemwait = FALSE,lock = TRUE,lockfailure = FALSE, drumst = EMPTY,doorst = LOCKED,dpos = CLOSED
ControllerFillDrum
DoorSubsystemFail
ControllerUnlockDoor
confirmlocked = TRUE,controllerlockwarning = FALSE,controllerwait = FALSE, doorsubsystemwait = FALSE,lock = TRUE,lockfailure = FALSE, drumst = FILLING,doorst = LOCKED,dpos = CLOSED
confirmlocked = TRUE,controllerlockwarning = FALSE,controllerwait = FALSE, doorsubsystemwait = FALSE,lock = FALSE,lockfailure = FALSE, drumst = EMPTY,doorst = UNLOCKED,dpos = CLOSED
confirmlocked = FALSE,controllerlockwarning = FALSE,controllerwait = FALSE, doorsubsystemwait = FALSE,lock = TRUE,lockfailure = TRUE, drumst = EMPTY,doorst = LOCKED,dpos = CLOSED
DoorSubsystemFail
ControllerWash
ControllerFinish
ControllerCompleteLockFail
confirmlocked = TRUE,controllerlockwarning = FALSE,controllerwait = FALSE, doorsubsystemwait = FALSE,lock = TRUE,lockfailure = FALSE, drumst = FILLED,doorst = LOCKED,dpos = CLOSED
confirmlocked = FALSE,controllerlockwarning = FALSE,controllerwait = FALSE, doorsubsystemwait = FALSE,lock = TRUE,lockfailure = TRUE, drumst = FILLING,doorst = LOCKED,dpos = CLOSED
DoorSubsystemFail
ControllerEmptyDrum
ControllerAbortWash
confirmlocked = FALSE,controllerlockwarning = FALSE,controllerwait = FALSE, doorsubsystemwait = FALSE,lock = TRUE,lockfailure = TRUE, drumst = FILLED,doorst = LOCKED,dpos = CLOSED
confirmlocked = TRUE,controllerlockwarning = FALSE,controllerwait = FALSE, doorsubsystemwait = FALSE,lock = TRUE,lockfailure = FALSE, drumst = EMPTYING,doorst = LOCKED,dpos = CLOSED
ControllerConfirmLockFailure
ControllerEmptyDrumFail
DoorSubsystemFail
confirmlocked = FALSE,controllerlockwarning = TRUE,controllerwait = FALSE, doorsubsystemwait = FALSE,lock = TRUE,lockfailure = TRUE, drumst = EMPTYING,doorst = UNKNOWN,dpos = CLOSED
confirmlocked = FALSE,controllerlockwarning = FALSE,controllerwait = FALSE, doorsubsystemwait = FALSE,lock = TRUE,lockfailure = TRUE, drumst = EMPTYING,doorst = LOCKED,dpos = CLOSED
ControllerFinishAfterEmptyFail
ControllerFinishFail
confirmlocked = FALSE,controllerlockwarning = TRUE,controllerwait = FALSE, doorsubsystemwait = FALSE,lock = FALSE,lockfailure = TRUE, drumst = EMPTY,doorst = UNKNOWN,dpos = CLOSED
DoorSubsystemOpenDoor
DoorSubsystemCloseDoor
confirmlocked = FALSE,controllerlockwarning = TRUE,controllerwait = FALSE, doorsubsystemwait = FALSE,lock = FALSE,lockfailure = TRUE, drumst = EMPTY,doorst = UNKNOWN,dpos = OPEN

Figure 3.6: The Concrete Graph

Chapter 4

Integrating Proof and Simulation

In this initial work, we show how an STPA-based model development with proof can be integrated in the design flow with the multi-simulation and test capabilities that are being developed in WP4 in ADVANCE [Col11]. The first step is test generation.

4.1 Test Suite Generation

ProB will be used to generate a set of tests which, in the first instance, *cover* all the transitions of the graph shown in Figure 3.6 above. Since the *guards* of the controller events are all *conjunctions* of simple expressions, coverage of all the controller transitions equates directly to full MC/DC [Chi01] coverage of the controller function with respect to the door subsystem. These tests can be used to drive ADVANCE multi-simulation.

The next step is to perform formal Event-B decomposition [But09].

4.2 Concrete Model Decomposition

The concrete machine can now be decomposed into three distinct machines: the controller, the door component and the communication/synchronisation mechanism. The controller machine can then be used as a Functional Mockup Unit (FMU) [BOA$^+$11] in an Event-B multi-simulation.

4.3 Multi-simulation

The communication/synchronisation mechanism is performed by the ADVANCE multi-simulation *master* and an FMU of the door component, possibly provided by the door component vendor, replaces the Event-B model. The FMUs that represent the controller and the door components are then simulated using Event-B multi-simulation using the tests generated in ProB as the stimuli. Coverage of the controller FMU is verified using ProB.

Chapter 5

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 which is integrated with Functional Requirement capture and uses the ProR facility. We have shown, using Event-B refinement and proof, how a controller can be developed for a safety-critical system which incorporates an STPA process model of the system to control the good behaviour of the system, detect faults in the system and mitigate hazards that arise from these faults. Event-B refinement ensures that the design decisions are clearly and formally represented and Event-B proof has been used to ensure that the safety constraints, represented as invariants, are preserved and the hazards correctly mitigated.

We have also performed some initial work to show how this method, based on formal modelling and proof, can be integrated into the ADVANCE multi-simulation and test method.

In future work, we will explore more fully the use of ProR to document the STPA-based method and to use our case study to validate the ADVANCE multi-simulation and test method. We shall also validate the code generation capability being developed in WP4. In parallel we are applying the method to our two ADVANCE case studies.

Bibliography

[ABH$^+$10] J.-R. Abrial, M. J. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010.

[Abr10] J.-R. Abrial. *Modeling in Event-B – System and Software Engineering.* Cambridge University Press, 2010.

[BOA$^+$11] Torsten Blochwitz, M Otter, M Arnold, C Bausch, C Clauß, H Elmqvist, A Junghanns, J Mauss, M Monteiro, T Neidhold, et al. The functional mockup interface for tool independent exchange of simulation models. In *Modelica'2011 Conference, March*, pages 20–22, 2011.

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

[Chi01] John J Chilenski. An investigation of three forms of the modified condition decision coverage (mcdc) criterion. Technical report, DTIC Document, 2001.

[Col11] J. Colley. D4.1 specification of multi-simulation framework. Technical report, December 2011.

[Jas10] M. Jastram. ProR, an open source platform for requirements engineering based on RIF. *SEISCONF*, 2010.

[LB08] M. Leuschel and M. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.

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

[YB12] S. Yeganefard and M. Butler. Control systems: Phenomena and structuring functional requirement documents. 2012.