MASTER

Analysis of a patient and beam restriction interface

Osaiweran, A.A.H.

Award date:
2008

Link to publication
Abstract

In conventional design methodologies, verification of a particular design is made after implementation and testing. At this point, if flaws are detected then it might become quite expensive to change the design and the architecture of the software system. Model checking techniques allow detection of flaws and defects of a particular design prior implementation and testing. This work is a part of an investigation made at Philips Healthcare to analyze and comprise the ASD methodology that uses the the CSP theory for model checking as a framework for designing software systems with the conventional methods that are currently used targeting the functionality of the Patient and Beam unit.
Acknowledgments

I would like to take the opportunity to thank all people who have in one way or another supported and helped me in any way with my master thesis. I want to thank Jan Borstrok, Marjolein Weiss at Philips Healthcare for their kind support. I would like to thank Aad Mathijsen for his support and help throughout the writing of this thesis. I want to express my sincere gratitude to Marcel Boosten for providing continuous valuable feedbacks during the progress meetings of this work and also to Jos Langen from Verum for his kind support. I want to express my deepest appreciation and special thanks to Nico Franke who is my tutor at Philips Healthcare for sharing his knowledge and accompanying me during lunch breaks. Last but certainly not least I would like to express my gratitude to my gradation supervisor at the TU/e, Jan Friso Groote, for the opportunity to do my master thesis under his guidance. His suggestions, feedbacks and support during my master study are greatly appreciated. Thanks
Contents

1 Introduction .......................... 1
  1.1 Purpose of the assignment ............ 2
  1.2 Structure of this Thesis ............... 3

2 Languages and Tools ..................... 5
  2.1 Sequence-Based Specification .......... 5
  2.2 Analytical Software Design .......... 7
    2.2.1 ASD Design Methodology ............ 7
    2.2.2 Run-to-completion Semantics ......... 8
    2.2.3 Contents of The ASD component ....... 10
    2.2.4 Component Specification in ASD ....... 10
  2.3 Communicating Sequential Processes .... 13
  2.4 FDR ................................ 14

3 The Patient and Beam Unit ............... 17
  3.1 The Patient and Beam Unit ............ 17
  3.2 An Example Execution Scenario ........ 18

4 High level requirements .................. 21
  4.1 Restrictions ........................ 21
    4.1.1 Types of Restrictions .............. 21
    4.1.2 Restriction States .................. 23
  4.2 External Events ...................... 23
  4.3 Informal Requirements ................ 24

5 Software design description ............ 27
  5.1 Design issues ........................ 27
  5.2 The Software Design .................. 29
  5.3 Components descriptions and state machines .... 29
  5.4 Internal Events ...................... 36

6 Modeling and Specification .............. 41
  6.1 Modeling ............................ 41
    6.1.1 Component Down-scaling .......... 42
CONTENTS

6.1.2 Closing the Model ................................................. 43
6.1.3 The Complete Model ............................................. 43
6.1.4 Results from Modeling ........................................... 46

6.2 Specification .......................................................... 47
6.2.1 The Queue Specification ........................................... 47
6.2.2 Binding Bridges with restrictions and UserMessages Components ........................................... 48
6.2.3 Binding the controller with its used components .............. 49
6.2.4 The LimitClient Specification .................................... 50

7 Verification .............................................................. 53
7.1 Checking components Deadlocks and livelocks ...................... 53
7.2 Requirements Verification ............................................ 54
7.3 Results ............................................................... 56

8 Conclusions .............................................................. 59

Appendix A: The Queued Design ........................................ 63
Appendix B: The Formal Requirement Formulas in CSP .............. 65

References ................................................................. 86
Chapter 1

Introduction

Philips Healthcare develops a number of systems that are widely used in medical applications. One of these systems is the Cardio/Vascular X-ray Imaging System. Figure 1.1 shows a basic setup of a Cardio/Vascular X-ray Imaging System. This system is mainly used for acquiring images of human body Cardio or Vascular using X-Ray.

Figure 1.1: Shows a basic setup of a C/V system with two movable stands and a table. For each side of image detectors and X-ray tubes, there exists a bodyguard sensor.

As is seen from the figure above, the system includes some movable parts and offers a number of motorized movements to the clinical user. These movements are speed-motorized movements, which a movement speed is controlled by a joystick or button on the UIModule. The motorized movable parts comprise the table, stands (that hold X-ray...
collimators) and image detectors. The movement of these movable parts should be safe with respect to patients as well as the body of the system. Guidance to the clinical user, by means of user messages or a beeper, should be accurate and should reflect the actual status of the movement.

1.1 Purpose of the assignment

The clinical user may activate joysticks or buttons to move one or more of the movable parts (the stands, detectors or table). While a movable part is moving, some reasons may cause this movement to slow down the speed and possibly to stop. Such reasons are called restrictions. For instance, when a bodyguard sensor of a stand detects an object, it will influence the movement to slow down (to prevent a collision) but if the object becomes too close to the stand then the movement will stop. When the movement is stopped, the clinical user may continue the movement despite this object by applying a joystick double clutch.

For each possible movement, there is a controller that controls user requests and the restrictions that might be imposed on that movement. The movement controller is a part of the responsibilities of the Patient and Beam unit. The Patient and Beam unit is introduced in more detail, with respect to this thesis, in Chapter 3.

This assignment will mainly focus on designing, modeling and verifying the formal behavior of a system that contains a movement controller and its related possible restrictions, insuring that the system will provide its clients (the clinical user or the unit that controls the movement resources) the correct handling. The movement controller considers only one movement in one direction at a time.

For the purpose of modeling and verification, we use formal methods which are mathematical-based techniques for specifying and verifying complex software and hardware systems. One of these formal methods is model checking which is used to verify (check) whether a given model of a software design will satisfy its requirements.

The main advantage of applying model checking is the detection of design flaws upfront the implementation and testing. Moreover it will guarantee verification completeness; if requirements are valid and satisfied in a design model, we are certain that these requirements will not be violated by any possible execution scenario in the original design of course only if the model is a valid representation of the software design under consideration and the requirements are well formulated. Currently there are various model checking tools that are available now and widely used, for instance FDR2[11], mCRL2[5], SPIN[3], UPPAAL[12] and CADP [10].

In this thesis we use the ASD tool supported by Verum Consultants BV to create the system design model based on ASD design Methodology. Using ASD, we create our model using SBS and then generate the corresponding CSP code that will eventually be verified by the FDR tool. Moreover ASD can generate source code in a high-level programming language, such as C++ or Java, automatically from the component specification. We will introduce SBS, ASD, CSP and FDR in more detail in Chapter 2.
1.2 Structure of this Thesis

This thesis consists of eight chapters in total. These chapters are organized as follow:

- Chapter 2 details the languages and tools that are used in this thesis.
- Chapter 3 gives a brief overview about the context of the analyzed system and its related associated sub-units.
- Chapter 4 covers the description of informal requirements of the analyzed system in detail. At the end of the chapter we introduce a list of requirements that should be met by the analyzed system which will be translated and verified in Chapter 7.
- Chapter 5 considers the detailed design of the analyzed system which contains a movement controller and its associated components. Each component is described in detail together with the interaction with other components.
- Chapter 6 describes the steps that have been made to specify and model the behavior of the design given in Chapter 5.
- Chapter 7 covers the steps that have been done to translate and verify the informal requirements introduced in Chapter 4.
- Chapter 8 draws the conclusions of this thesis.
Chapter 2

Languages and Tools

This chapter details the languages and tools that are used throughout this thesis. In Section 2.1 we introduce the concept of Sequence-Based Specification which is the core of ASD design methodology. In Section 2.2 we cover briefly the Analytical Software Design (ASD) methodology, we describe the toolset and we introduce some concepts and terminologies that are related to the methodology. Section 2.3 gives a brief syntax and semantics of the language CSP which is mathematical-based language used to construct the formal specification of our software system behavior.

In Section 2.4 we give an overview of the FDR tool, which provides the necessary automation to check a formal specification written in CSP. Both CSP and FDR are related to the concept of model checking. Model checking is introduced in Chapter 6.

2.1 Sequence-Based Specification

This section introduces Sequence-Based Specification (SBS) which is the basis of the ASD design methodology. To give an overview about SBS in general, we need first to introduce some related concepts. The Box Structure Development Method (BSDM) is a method that uses requirement analysis techniques to create a formal specification that is traceable to the informal requirement. The method involves three main steps: a black box specification which defines a given system in terms of sequences of external events and responses; the transformation of the black box to a state box specification in order to facilitate implementation; the clear box which is the actual implementation of the state box (e.g. C++). The clear box is out of the scope of this thesis.

The black box is developed from informal requirements and defines the external behavior of a system or component by observing its input stimuli and responses. Let $BB : S^* \rightarrow R$ define the black box function that maps a finite sequence of stimuli in $S^*$ to a response in $R$ based on a given set of informal requirements, where $S$ is the set of all stimuli and $R$ is the set of all responses. The set $R$ includes two additional special responses that are called null and $\omega$. The null response is used when a sequence of stimuli will not invoke a response, while the $\omega$ is used when a sequence of stimuli is illegal. In practice, informal requirements could be incomplete or tend to be quite large.
CHAPTER 2. LANGUAGES AND TOOLS

To make the black box method feasible, the sequence-based software specification method is introduced for systematically defining a black box specification that guarantees completeness and traceability with the original informal requirements and the possibility to derive other requirements while specifying system behavior. Briefly the SBS method involves the following steps:

1. Original informal requirements: prepare the list of initial traced requirements.
2. System/component boundary definition: identify system/components interfaces.
3. Itemize stimuli: list all incoming events and denote them by Stimuli.
4. Itemize responses: list all outgoing events and call them Responses.
5. Sequence Enumerations: we start by enumerating stimuli sequences in $S^*$ preserving the order by length. So we start the enumeration process from the empty sequence and we assign responses to each stimulus. While enumerating, sequence equivalences may arise: two sequences $a, b \in S^*$ are equivalent when all nonempty extensions of both of them will result to the same response. We say a sequence $a$ is reducible when there is a previously enumerated sequence $b$ that results to the same response $r$. We say that $a$ is reduced to $b$. Otherwise the new sequence $a$ is irreducible or canonical. In this way we partition the set $S^*$ to a finite set of equivalent classes as defined by the sequence equality property mentioned above. Each partition defines a state in a state machine of the system or component. Figure 2.4.(e) shows an example specification of a component after the enumeration process has been done using the ASD tool. In this example we start from the empty sequence as the first canonical irreducible sequence, and enumerate all stimuli, and then assign responses to these stimuli. We repeat the step whenever we encounter a new sequence that is irreducible until we obtain a complete enumeration. An enumeration is complete and finite when all of the longest sequences are reducible.
6. Derived requirements: initial requirements often tend to be incomplete to tackle complete system behavior. During the enumeration process some stimuli histories might not be anticipated in the initial requirements. For instance, some illegal responses for some given stimuli could be discovered during the enumeration process [4].
7. Canonical Sequence Analysis: the canonical sequence that represents a partition of $S^*$ will also represent a single Mealy machine state[2]. So the set of all canonical sequences for a particular specification of a given system or component will represent Mealy machine states. The arcs between the Mealy machine states will be labeled with a stimulus-response pair (state box representation). The Mealy machine will be used to generate the formal specification (e.g. CSP) or a code in a high-level programming language (e.g. C++).

For more details about SBS, see [4, 7, 8].
2.2 Analytical Software Design

This section briefly covers the Analytical Software Design (ASD) methodology and ASD toolset developed by Verum Consultants BV. We start describing the design methodology. We introduce some concepts that are needed for the following chapters and then we briefly describe the steps applying these concepts using the ASD tool to create components specification.

2.2.1 ASD Design Methodology

ASD employs the component-based software engineering technique for designing software systems that consist of components with well-defined interfaces used for communications among these components. An ASD component represents the behavior of a particular original component of a given software design in a complete or abstracted way using the Sequence-Based Specification technique.

ASD uses SBS for first systematically defining a component specification such that it preserves traceability with original informal requirements and second to eventually generate a formal specification or source code in a high-level language. Although SBS
could be applied on the system as a whole, ASD uses SBS here to create only a single component. Mainly, there are three types of ASD components. The first type is the *ASD design model* which is the complete specification of any given component. The second type is the *ASD interface model* which is an abstraction of an ASD design model such that all events to its lower-level components (used-components) including its internal events are abstracted. The third type is the *usage model* which is used to generate test cases. The usage model is beyond the interest of this thesis.

Once the software design is available, we start to build up the complete model using the bottom-up approach by specifying the behavior of each component separately using SBS. When components have been created, the formal specification of each involved component could be generated separately. Each component could be model checked using the model checking tool once individually and once with its used components.

At this step of model checking, we check whether the component uses its used-interfaces properly. Step 8 in Figure 2.1 shows an example of checking whether component D6 uses the interfaces of D3 and D4 properly, but the interface model of D3 and D4 is equal to their design model in this case. Generally, the check will include whether an individual component or a component and its related used-components that run together in parallel are deadlock and livelock free. We may include an abstracted used-component (the ASD interface model of a particular used component) instead of the component itself (the ASD design model) to avoid state space explosion. We describe the deadlock and the livelock in Section 2.4.

In some cases, we may replace a particular design model of a component with its related used-components at a lower level by only the interface model of that particular design model (in Figure 2.1 we may replace D5, D1 and D2 by ID5). We continue this way of checking components and their related interfaces until we reach the upper level. Depending on the given design, some abstracted components (interface models) could be included in the verification step (for instance ID5 and ID6 instead of all lower level components). The verification step is mainly used to check requirements by formulating some formal properties and test them against the complete model using the model checking tool.

Note that, for some designs there is no need to formulate dedicated properties to verify requirements since these requirements are implicitly valid in the specification of the model. These requirements are already traced during the enumeration step within SBS. However, this is not the case of the design treated in this thesis.

After requirements verification have been done, we can generate a high-level language source code for each component separately.

### 2.2.2 Run-to-completion Semantics

ASD uses the concept of *durative* and *non-durative* actions (see Figure 2.2.(a)). The non-durative action is realized if a client component synchronously calls a method at a server component. While the server component processes the call, the client component will remain blocked (it waits for an acknowledgment or a return value) and it can not invoke
other methods at the same or other server components until the server component has completely process the request. The server can invoke methods at other servers before it unlocks the client. A durative action is a non-durative action but after the return value has been received by the client component, the server may process a durative part and eventually sends asynchronous callbacks to the client. During the durative part of the server, the client component may invoke other methods at the same or other servers.

In some cases, server components may need to send asynchronous callbacks to a blocked
client component. In this case the client component should maintain its own attached queue and while the client is blocked, the server component can send callback events to that queue. After the client has been released, it may process these queued callbacks. We call such a behavior the non-durative actions with callbacks in between (see Figure 2.2(b)).

The run-to-completion semantics means that once a given stimulus is invoked at a particular component, all its corresponding responses will be processed completely in the given order, and all state predicates are updated before the state transition is made (see Figure 2.3). By applying this concept, we can model the behavior of components that run in a single execution thread. In this case a client component is blocked by a server component and the server component may become a client component for other server components that may block it, and so on. Each server will unlock its client after it completely processes the request or invokes other methods at other components.

2.2.3 Contents of The ASD component

After the software design is available, the designer can create components using the ASD toolset. Currently, the ASD toolset is a Microsoft Excel plug-in that allows designers to specify the behavior of components using Microsoft Excel sheets.

An ASD component is represented by a single Excel file. Mainly, any ASD component contains five main user interface sheets (five Excel sheets within an Excel file that correspond to a particular component, see the screenshots depicted in Figure 2.4). The first sheet has the name ‘Interfaces’. This sheet contains the process alphabet (Stimuli and Responses) and channel declarations. Figure 2.4.(c) shows an example of such a sheet. The channels are classified into two types namely the implemented (client) interfaces and the used (server) interfaces. The second sheet is needed for the source code generation which contains configuration settings of the source code that will be generated. The third sheet has the same name as the component name itself and contains the component behavior specification that represents a state machine, see Figure 2.4.(f). In this sheet the enumeration process is used to create a specification of a given component. The fourth sheet is used to assign names to the states that are generated by the enumeration process, see Figure 2.4.(d). The last sheet is used for requirement traceability. This sheet is used to list requirements that will be referenced later during the enumeration process, see Figure 2.4.(e).

2.2.4 Component Specification in ASD

Once the architectural software design is available, we start to model each involved component separately. At this point we might have two choices: create a Mealy state machine of a corresponding component or start to create the component using SBS. If a Mealy machine that describes the behavior of a given component is already created, we translate that machine to the corresponding SBS in the Excel sheets. But if we start to specify a component behavior by using the SBS technique, at the end we will obtain a Mealy machine representation of that component.
In general, to specify a component behavior in ASD (even by having a Mealy machine in prior or by SBS), ASD needs first to identify the component boundary (the interfaces), and itemize all possible events which will be invoked at the component. We call these events stimuli and we refer to their communication channels as client interfaces (or imple-
mented interfaces). The stimuli, in turn, may invoke other events at other components via their interfaces. These events are called Responses and their communication channels are called server interfaces (or used interfaces). Internal events are specified by introducing a dedicated communication channel.

In some cases, a component may need to receive some callback events from other components while it is blocked by a server component. To allow receiving events while a component is blocked, a dedicated queue will be used to store such events until the component is unlocked by its server components (we apply the concept of non-durative actions with callbacks in between). The channel between the queue and the component is called the ClientCB Interface, and its events are used as stimuli for the component, see Figure 2.4.(a) and 2.4.(c).

After the component interfaces, stimuli and the responses have been listed, we can start the enumeration process. The enumeration process is done by first starting from the empty sequence and the list of all possible stimuli and their assigned responses. This will form the initial state of the ultimate state machine. While adding responses to the stimuli other canonical sequences may arise. That is simply because there is a transition to another state if the state machine is known a priori or the sequence is a representation of a requirement in case we follow the SBS technique. In case such a new canonical sequence is needed, the stimuli will be listed again and we add responses to each one of them. We continue adding new canonical sequences or adding references to already generated sequences until the specification is complete.

Note that, to make the specification more consistent, some special responses could be assigned to the stimuli. The special responses are null, blocked and illegal. The null denotes having no response at all (it is possible to receive the input stimulus but just ignore it). The blocked response denotes that it is not possible to invoke a stimulus in a particular state. The illegal response denotes a non-desired event at a particular state.
2.3 Communicating Sequential Processes

Using the ASD toolset, we can create components specification using SBS and later we can generate their corresponding CSP specification automatically. Communicating Sequential Processes (CSP) is a formal specification language for describing the behavior and interactions in complex concurrent systems based on process algebra theory. CSP represents a mathematical framework to specify behavior of systems. Since a vast part of this thesis is related to model and verify the system using CSP, this section describes briefly the syntax and semantics of the CSP language to the extent that is related to this work. The reader may consult [1, 9, 11] for a more extensive description of the language syntax and semantics.

The process

The process expression is basically an event or an operator combined with a process expression. Let $e$ denotes an event, $p$ and $q$ are processes, $a$ is a set of events, and finally $c$ and $c_0$ are communication channels. The following table depicts a list that contains most of the special events and operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STOP</td>
<td>Deadlock. Process does nothing.</td>
</tr>
<tr>
<td>SKIP</td>
<td>Successful Termination.</td>
</tr>
<tr>
<td>$e \rightarrow p$</td>
<td>Simple event prefix. This process will execute the event $e$ and then will behave as the process $p$.</td>
</tr>
<tr>
<td>$p</td>
<td>a$</td>
</tr>
<tr>
<td>$p[c \leftarrow c_0]$</td>
<td>Renaming. Rename all events in channel $c_0$ to $c$.</td>
</tr>
<tr>
<td>$p</td>
<td></td>
</tr>
<tr>
<td>$p\sim</td>
<td>q$</td>
</tr>
<tr>
<td>$p||q$</td>
<td>Parallel operator. Interleaving between events of process $p$ and $q$.</td>
</tr>
<tr>
<td>$p</td>
<td>a|q$</td>
</tr>
<tr>
<td>$p[c \leftrightarrow c_0]q$</td>
<td>Linked parallel operator. $p$ and $q$ will form a chain. They will be connected by the channels $c$ and $c_0$.</td>
</tr>
<tr>
<td>$[ x : a @ p$</td>
<td>Replicated external choice. Substitution of event $x$ in $p$ with an event form the set $a$ at a time and separate them by the $[ ]$ operator.</td>
</tr>
<tr>
<td>$\sim</td>
<td>x : a @ p$</td>
</tr>
<tr>
<td>$| x : a @ p$</td>
<td>Replicated interleave.</td>
</tr>
<tr>
<td>$c?e$</td>
<td>waiting for event $e$ from channel $c$.</td>
</tr>
<tr>
<td>$ce$</td>
<td>out event $e$ to channel $c$.</td>
</tr>
<tr>
<td>$diff$</td>
<td>the set difference.</td>
</tr>
</tbody>
</table>

- Datatypes and Nametypenes

The Datatypes that are used in this thesis are at a simplest level and used to define atomic
CHAPTER 2. LANGUAGES AND TOOLS

constants. Example:

    datatype SimpleColor = Red | Green | Blue | White | Black | Gray

However, a Name type definition consists of a name with a type expression. The ‘.’ and ‘( , , )’ operate on it as type constructors. Example:

    Mono_color = {White, Black, Gray}
    Double_Mono_Color = Mono_color.Mono_color

Example of uses of type Double_Mono_Color is the value White.Black.

- Channels
Consider the following definition:

    datatype SimpleColor = Red | Green | Blue | White | Black | Gray
    Mono_colors = {White, Black, Gray}
    channel tick
    channel c:Mono_colors

We can also define the last channel as:

    channel c : {White, Black, Gray}

Channels form the basis of events. A channel becomes an event when enough values have been supplied to complete it. For example consider the declared channels above. Clearly, tick is an event and c.White is an event as well. When the channel name is used for hiding or sharing, all events involved at that channel will be included.

2.4 FDR

Failures-Divergence Refinement (FDR) is a model checking tool for state machines that provides the necessary automation to model check and verify a given formal specification of a model written in CSP. The tool is able to check the presence and absence of deadlocks, livelocks and determinism of a state machine. The system deadlocks simply when it refuses to do any event. Livelock (hanging) means that the system may enter infinite loop of internal events and may never exit to a visible event. The determinism check is mainly used for checking software security protocols. Moreover, the tool also has the ability to check whether a given property holds in the model by using the refinement check.

To use FDR, the properties and the formal specification of the model must be written in the same language CSP. For verification using FDR, requirements should first be formulated as CSP processes and then they are checked using the refinement check. The concept of refinements is also used to reduce the complexity of the system without degrading the properties of the system. That can be established by the fact that one or more components might be replaced by another simple component. An example is given in section 2.2.1 where we can replace one or more components by a single interface specification. Generally, the refinement should reflect the properties of a system which are important [11]:
“in building bridges it may acceptable to replace a (weaker) aluminium rivet by a (stronger) iron one, but if weight is critical, say in an aircraft, this is not a valid refinement.”

There are mainly three types of refinements supported by FDR:

- **Traces refinement**: each process is represented by a set of finite sequences of events it can perform. The set of traces of a process $P$ is denoted by $\text{traces}(P)$. A process $Q$ is a traces refinement of a process $P$, written $P \subseteq_T Q$, if all possible sequences of events process $Q$ can perform are also possible for process $P$. In Chapter 7, we use traces refinement to verify safety properties. If we consider that process $P$ defines possible safe states (based on requirements) and $Q$ is our system, then $Q$ is a safe implementation as soon as the trace refinement check is satisfied. The trace refinement is defined as follow:

$$P \subseteq_T Q = \text{traces}(Q) \subseteq \text{traces}(P)$$

Informally, if $Q$ can do more than $P$, $Q$ possibly contains undesirable behavior. For some systems, it is useful to check some illegal situations that are defined during the formal specification. If a model contains an illegal event that is already defined during the formal specification, such a check is given by the following assertion:

$$\text{STOP} \subseteq_T \text{Design} \setminus \text{diff}(\text{Events, illegal})$$

Where $\text{Events}$ is the set of all possible events in a given system, "\" is the hide operator, $\text{diff}$ is the set difference, and $\text{Design}$ is the complete formal specification of the system. Informally, if there is a trace in the system that contains the illegal event (the illegal always followed by the STOP event), FDR will show a counterexample of such a trace. Generally this assertion is used in this thesis to check illegal and Queue-Full situations. Noteworthy, it is possible to check whether a given point in the model is reachable by simply injecting an extra event and check it in the same way as given in the above assertion. That is because FDR does not support simulation.

- **Failures refinement**: The distinction between processes can be made not only by the traces of events but also by the set of events that are blocked or performed by the implementation process. Each process is represented by a failure pair $(s, X)$, where $s$ is a finite trace and $X$ is the set of events that can not be executed after $s$. The set of $P$’s failures is given by $\text{failures}(P)$.

$$P \subseteq_F Q = \text{failures}(Q) \subseteq \text{failures}(P)$$

Informally, if $\text{Design}$ can do less (refuses to do more) than the property, we miss some behaviors which are specified in the property. If a process refuses to do any event, it simply deadlocks.
• *Failures-Divergences* refinement: this refinement detects the presence of a livelock state.

\[ P \sqsubseteq_{FD} Q = \text{failures}(Q) \subseteq \text{failures}(P) \land \text{divergences}(Q) \subseteq \text{divergences}(P) \]

Where a *divergence* is a trace during or after the process can enter infinite loop of internal events and there is no way to exit that loop to a visible event.

In Section 7.2 we give an example to illustrate the concept of both the traces and failures refinement.
Chapter 3

The Patient and Beam Unit

3.1 The Patient and Beam Unit

We introduce a general overview of the Patient and Beam unit and some of its associated subsystems that are relevant to this assignment. In general, the Patient and Beam unit is responsible for application related functions such as user messages and user interactions. One additional responsibility is controlling movement restrictions and exceptions.

As is seen from Figure 3.1, the Patient and Beam unit can be represented as a functional layer.

![Diagram of Patient and Beam functionality overview.](image)

Figure 3.1: Patient and Beam functionality overview.

The functional layer interacts and lies between the presentation layer and the technical
CHAPTER 3. THE PATIENT AND BEAM UNIT

layer. The functional layer is responsible for translating user commands issued via the user interface (issued via buttons or joysticks) to the corresponding functions (movements) the system should perform, such as rotate beam, swing beam and tilt table. The function denotes a particular movement.

When a command is issued from the clinical user, the functional layer assigns the responsibility to control the corresponding function to a function controller. The functional layer forwards the function request of that particular user command to the technical layer which will, in turn, observe and monitor movement resources that are imposed on that function. If no movement restriction is active, the functional layer will issue the Start command to the technical layer and the movement will start. During the course of a movement, the technical layer will provide the function controller with the status of its related movement resources. Depending on a resource status and the user requests, the controller will create movement restrictions state machines and provides the correct handling to the clinical user by means of correct user messages or by influencing the movement by setting some values of restriction profiles. The restriction profiles will allow the technical layer to proceed with the movement despite any reasons that cause the movement to stop (override a restricted movement). The movement restrictions are explained in Chapter 3.

Although Patient & Beam is responsible for application functions, it may influence the movement by setting restriction profiles. That happens only in case movement restrictions that cause a movement to stop need to be overridden.

Override a restriction means that the clinical user wants to continue with the movement regardless the restrictions that cause the movement to stop. In this case the movement will continue with a special speed configured in the associated restriction profiles, and then it is up to the technical layer to decide at which speed the movement should continue in Override mode.

At any time, one or more restrictions could be imposed on a movement and each restriction has a speed value in its profile in case the user wants to override it. The technical layer may employ a max or min function to select the proper speed of the movement during Override mode which means that it will select the maximum or minimum speed among all profiles of active restrictions.

3.2 An Example Execution Scenario

In Figure 3.2 we show an example execution scenario for rotating the frontal stand beam. The clinical user may activate the rotate beam frontal joystick. The Patient and Beam unit will translate the command and inform the technical layer to move with a speed that corresponds to the joystick pressure. The technical layer will start rotating the beam but suddenly the detector bodyguard may see an object during the movement. In this case the object will lead the movement to slow down and finally will cause the movement to stop (the detector bodyguard sensor is restricting the movement). To continue the movement again despite that object, the clinical user needs to issue a double clutch request which will lead him to enter the Override mode. The movement in this case will proceed with the speed associated with the detector bodyguard sensor profile and a beeper may start.
In case the object does no longer exist, the movement will exit Override mode and the movement will be back to its normal speed.

Generally, the technical layer is responsible for the real-time related issues of movements, and the Patient and Beam unit is responsible to start or stop the movements, tracking the status of user commands and the status of the movement resources, and then send the correct user messages to the clinical user or provide the correct handling to the technical layer when entering Override mode.
Chapter 4

High level requirements

This chapter will mainly cover the informal requirements that should be met by the system. These requirements are translated to CSP processes for verification in Chapter 7. Before we start, we first need to introduce the meaning of restrictions. A restriction is an abstracted and generalized definition for any reason that causes the movement (of the stands, image detectors or the table) to slow down or possibly to stop. We refer to that reason as a restriction. These restrictions are categorized into four main types: Blocking, Simple, Overridable and StepBack. Each type of restrictions is represented by a state machine. We will address each one of them in this chapter in more detail and introduce restriction state machines in Chapter 5.

4.1 Restrictions

The technical layer is responsible to provide the functional layer with the status of movement resources that are imposed on any particular movement. As mentioned earlier, these resources could be restricted by restrictions of several types and possibly more than one type at a time. These restrictions may originated due to activated sensors (e.g. the sensor sees an object, becomes dirty or defect), defect actuators or reaching a predefined position (e.g. working, parking area or end position). The functional layer is responsible for creating restriction state machines based on the type of the reasons reported from the technical layer. The functional layer is now responsible for maintaining the status of active restrictions together with user requests and provide the user and the technical layer with the correct handling.

4.1.1 Types of Restrictions

The restrictions fall into four categories, namely Blocking, Simple, Overridable and StepBack restrictions.
CHAPTER 4. HIGH LEVEL REQUIREMENTS

Blocking Restrictions

If a Blocking restriction is imposed on a movement, all other restrictions are not allowed to enter Override mode. Moreover, if such a restriction becomes active while some restrictions are in Override mode, they should be forced out from Override mode and the movement should stop. The following list contains examples of possible restrictions that might be imposed on a movement which are of type Blocking:

- Defect: resource failure.
- Parked: the movement is at the end position.
- Not Available: according to a geometry configuration, the movement is not available.
- In Use: the movement is in use by another combined movement.

Simple Restrictions

This type of restrictions will merely be used to send user messages. The speed of the movement will be slowed down by the technical layer. This type of restrictions will not prohibit other restrictions from entering Override mode.

Examples of possible restrictions that may be imposed on a movement which are of type Simple:

- Frontal/lateral stand area: the Frontal/lateral stand is not in its working area.
- Detector area: the rotate detector position is in between the portrait and landscape position with respect to the stand that holds it. In this case the sensors that are fixed on the detector sides will not be able to sniff objects while the stand is moving.
- Table area: the table is outside its working area.
- Not calibrated: the movement speed is not calibrated.

Overridable and StepBack Restrictions

Restrictions of type Overridable will allow the user to override a restricted movement unless a Blocking restriction is active. For some restrictions a StepBack movement (a reverse motorized movement) is required prior to override a restriction. The StepBack movement is performed to free up collisions between stands. For instance, in case the distance between the stand and the detector becomes less than 6 cm, a StepBack movement for the detector will be performed such that the distance is at least 10 cm.

Examples of possible restrictions that may be imposed on a movement which are of type Overridable:

- Table Force sensor: table force sensor is active or defect
- L-arc collision: L-arc collision sensor is active (on-off switch).
4.2. EXTERNAL EVENTS

- Collision prevention: when a collision between the frontal, lateral stand or table is detected.
- Stand/table Bodyguard: when the bodyguard sensor is active or defect.

The Controller

The controller should be able to receive requests from the user as well as movement resource state updates from the technical layer. These events are forwarded to active restriction state machines which will react upon those events and may send callbacks to the controller indicating their current status. Depending on the controller event and the current state of the restriction, the restriction state machine will react by sending user messages, setting movement profiles or it may issue other callback events to the controller which might be broadcasted again to active restrictions.

4.1.2 Restriction States

The following list contains the possible states that may be imposed on a restriction:

- NotPresent: the restriction is not active or it is no longer present. In this state, a resource can be used freely.
- Present: the restriction is about to occur but not restricting the usage of the resource yet, and the speed of the movement is slowing down. For instance, a bodyguard sensor "sees" something during the course of the movement and slows down the speed. Another example is that the lateral stand is not in its working area.
- Restricting: a restriction completely restricts the movement and causes it to stop.
- Override: the movement will continue moving despite the active restricted restriction with a speed associated within its profile and a beeper may start.
- Blocking: the restriction is blocked and overriding restrictions is not allowed. The movement is stopped.

Some of these states contain other sub-states. In general, these states may occur while there is (no)active request from the user (the user is (not)activating the button or the joysticks). In Chapter 5, we will associate idle or active to the restriction state to indicate the actual state of the restriction with respect to the user, and we show the detailed state machines. For instance, IdleRestricting denotes a restricted state while there is no active request from the user.

4.2 External Events

This section covers the events which provided by the clinical user and the technical layer. We will refer to these events later as environment events. These events are exchanged externally between the environment and the controller. At first, there are two events that
indicate the status of the user. These events are broadcasted to all active restrictions.

*SetRequestActive*: Notifies the controller that the user is activating a joystick or pressing a button.

*UnSetRequestActive*: Notifies the controller that the user releases a joystick or button. For each active restriction the technical layer may report the following events:

*SetPresent*: The restriction becomes present and slowing down the movement speed.

*UnSetPresent*: The restriction is released.

*SetRestricting*: The restriction is restricting the movement.

*StepBackPerformed*: The StepBack movement is performed and it is possible to override the restriction.

*SetOverridePendingTimeOut*: The timer of pending Override request is timed out.

*EnableOverride*: Enable Override mode for a specific restriction.

*DisableOverride*: Disable Override mode for a specific restriction.

As a consequence of these events the restriction state machines will react and other internal and callback events might be issued. An example of internal events is the *ForceInOverride* and *ForceOutOverride* events which are used to force the restricted restrictions to go in/out Override mode. The internal events are covered in detail in the following chapter.

### 4.3 Informal Requirements

This section informally explains the requirements (rules) that subsequently will be verified in Chapter 7. These requirement are part of the restriction interface handling document at Philips Healthcare. The requirements are written after the design of the system has been developed.

The requirements fall into three categories namely controlling, states and stimuli requirements. The restriction in the following list denotes the restriction state machine. The requirements are depicted in the following tables:
4.3. INFORMAL REQUIREMENTS

- Control:

<table>
<thead>
<tr>
<th>No</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>If one Overridable restriction enters the Override state for the first time, all active individual Overridable type restrictions are also in Override mode in case they are in state Restricted and regardless if the Override setting is disabled.</td>
</tr>
<tr>
<td>R2</td>
<td>If one restriction does not allow override, none of Overridable type restrictions are in the Override mode.</td>
</tr>
<tr>
<td>R2Rv</td>
<td>If some restrictions are in Override, none of all active restrictions are in state that does not allow override.</td>
</tr>
<tr>
<td>R3</td>
<td>If a restriction enters the restricting state, none of active restrictions are in Override.</td>
</tr>
<tr>
<td>R4</td>
<td>In case a restriction becomes restricting, the Override mode for all Overridable type restrictions should be exited.</td>
</tr>
<tr>
<td>R4Rv</td>
<td>In case a restriction is resolved during overriding multiple restrictions, the override mode should be maintained for the remaining restrictions.</td>
</tr>
</tbody>
</table>

- States:

<table>
<thead>
<tr>
<th>No</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>R5</td>
<td>If the restriction is in the Restricting state and receives the stimulus ForceInOverride, then the state machine must transit to override mode, in case override is supported.</td>
</tr>
<tr>
<td>R5Rv</td>
<td>If the restriction is in state Override and receives the stimulus ForceOutOverride, the state machine must transit out of override mode.</td>
</tr>
<tr>
<td>R6</td>
<td>If a state machine is in state Override, the state machine is not in state that does not allow override.</td>
</tr>
<tr>
<td>R6Rv</td>
<td>If a state machine is in state that does not allow Override, the state machine is not in the state Override.</td>
</tr>
<tr>
<td>R7</td>
<td>If a state machine is in state Override, the state machine is not in state Blocking.</td>
</tr>
<tr>
<td>R7Rv</td>
<td>If a state machine is in state Blocking, the state machine is not in state Override.</td>
</tr>
<tr>
<td>R8</td>
<td>If a state machine is in state Restricting, the state machine is also in state Present.</td>
</tr>
<tr>
<td>R8Rv</td>
<td>If a state machine is in state Present, the state machine is not in state Restricting or state Override.</td>
</tr>
<tr>
<td>R9</td>
<td>If a state machine is in state Override, the state machine is also in state Present and in state Restricting.</td>
</tr>
<tr>
<td>R10</td>
<td>If a StepBack restriction becomes restricting, Override mode is not allowed until the StepBack movement is performed and the request has been released.</td>
</tr>
</tbody>
</table>
- Stimuli:

<table>
<thead>
<tr>
<th>No</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>R11</td>
<td>If the stimulus \textit{SetNotPresent} is processed, the state machine is not in state \textit{Present}, \textit{Restricting} or \textit{Override}.</td>
</tr>
<tr>
<td>R12</td>
<td>If the stimulus \textit{OverrideNotAllowed} is processed, the state machine is not in state \textit{Override}.</td>
</tr>
<tr>
<td>R13</td>
<td>If the stimulus \textit{SetPresent} is processed, the state machine is not in state \textit{Restricting} or in state \textit{Override}.</td>
</tr>
<tr>
<td>R14</td>
<td>If the stimulus \textit{UnSetRequestActive} is processed, the state machine is not in state \textit{Override}.</td>
</tr>
</tbody>
</table>
Chapter 5

Software design description

Before the design which is introduced in this chapter is created, several designs have been analyzed. One of these designs is a queued design. In Appendix A we illustrate some issues we encountered during analyzing the behavior of such a queued design.

In this chapter we cover the design of the movement controller and its possible associated restrictions state machines. The movement controller merely considers one movement in one direction at a time and it is known upfront in the design phase which movement resources will be imposed on each movement. During the course of a particular movement, the technical layer is responsible to provide the functional layer with actual status of the movement resources.

In practice, it is possible for a movement resource to have more than one restriction at a time. For instance, a sensor sees something during the course of a movement; therefore an Overridable type restriction will be created to slow down the movement speed and possibly override it when it is stopped. However, it could be possible that this particular sensor is used by another movement, so in this case another restriction of type Blocking will be created to prevent any possible motion.

Current design implies a controller that forwards user requests and restriction state updates (the external events) to restrictions state machines via Bridges, see Figure 5.1. We consider here only the generic part of the design. Some details are omitted such as the concrete data about each restriction such as the specific user messages, beeper and the restriction profile. The controller and the Bridges are dispatchers which are used to forward events to/from the restrictions.

In this chapter we cover the detailed design of the restriction controller and its related restrictions state machines. We show the complete design from the architectural point of view which contains the components and the communication channels between them, and then we introduce the state machine of each type of restriction.

5.1 Design issues

1. The controller will create restriction components dynamically when needed. So, in practice, it is possible to have zero or more active restrictions at a time.
2. The input events (stimuli) are identical for all restriction components. They differ only by their implemented interfaces and their id’s. That is, all restriction state machines are able to receive the same set of events. For instance, the events \texttt{ISimple[id].SetRequestActive} and \texttt{IOverridable[id].SetRequestActive} denote an active request from the user sent to both Simple and Overridable restriction state machines. Internally, depending on the type of restriction and its current state, the restriction state machine may react or ignore these events.

3. When creating a restriction, associated \textit{Bridge} and \textit{UserMessages} components will also be created. The Bridge component forwards messages to/from the controller and the restriction state machines. The Bridge is created to include all individual concrete data about the restriction such as the specialized user messages and the concrete profile. The Bridge will derive these data from a concrete class. These specialized data are beyond the interest of this thesis.

4. Depending on the callback event, the Bridge will be responsible for forwarding restriction callbacks to either the controller or to the UserMessages component.

5. The input events (stimuli) for Bridges are identical. They differ only by their implemented interfaces and their id’s.

6. When creating a restriction, an associated UserMessages component will also be created. This component will issue user messages to the UIModule. It is assumed that the UserMessages component and the UIModule are decoupled (the UserMessages component will post a message to the UIModule and the UIModule may process it later.)
5.2 The Software Design

This section explains the software design of the controller with its associated components which will be formally modeled and specified later in Chapter 6. The components that are involved in this design are depicted in Figure 5.1. As is seen, the design contains a controller, Bridges, restrictions state machines, state machines for user messages and a central queue for callbacks. The design shows that it is possible to have more than one active restriction of the same type (a one-to-n relation). For instance, it is possible to have \( n \) Overridable Bridges associated to \( n \) Overridable state machines and \( n \) UserMessages components. The description of these components is given in the following section in more detail.

![Figure 5.1: The software design.](image)

5.3 Components descriptions and state machines

This section describes the design components which are depicted in Figure 5.1, and their responsibilities. The section will cover the detailed state machines for each type of restrictions. Note that, for simplicity reasons and making the state machines readable, we numbered the callback events and we refer to them in the transitions between states. Note also that for any state in the following restriction state machines, it is possible to receive any input stimulus event at any time. The stimuli events that cause the transition between
states are depicted in the figures while it is implied that other events might be received and ignored in the state (events cause self transitions to the same state but they are not depicted in the figures). The description of stimuli and callback events are explained at the end of this chapter.

The controller

The following figure depicts the controller state machine which is responsible for forwarding user requests and state updates events (the environment events) to restrictions via their Bridges. The controller is also responsible to process possible callback events which are issued from restrictions state machines, and possibly, based on them, it broadcasts other internal events to all active restrictions. To clarify the responsibility of the controller, we need to explain it from a broader perspective with respect to the relation with other components.

For any environment event, the event first will enter the central queue, and then the controller will process it until completion. The controller completely finished processing an environment event when the event has been processed by the restriction state machines and all possible related callback events in the central queue are processed as well.

That means that the controller will forward a particular event to Bridges. When the controller invokes a method at a Bridge, the controller will remain blocked until that particular method returns a value (e.g. acknowledgement). When the controller receives the returned value, it knows that the lower level components are completely processed the request and it will process the callback events in the central queue if any.

![Figure 5.2: The controller state machine.](image)

Between invoking a method and waiting for a return value, the controller can not invoke another method at any Bridge. When the controller invokes a method at a Bridge, the Bridge will forward the request to its restriction state machine which will process the event and may send callback events to the central queue targeting its UserMessages component (via the Bridge) or the controller (via the Bridge as well). When a restriction state machine has processed the event and sent the callbacks to the central queue, it
unlocks its Bridge component and the Bridge will unlock the controller. Now, if there is a callback event in the central queue, the event is forwarded to the Bridge which in turn may forward it to the UserMessages component or to the central queue again targeting the controller.

If there is a callback event in the central queue targeting the controller, the controller will process it and may broadcast other events to Bridges and waits for their return values before it processes the next event (run-to-completion). This broadcasted event may cause other callbacks to be generated by restrictions. If it is the case, the behavior repeats itself again until the central queue becomes empty. When the central queue becomes empty, the controller will process the next event from the environment.

Note that, the run-to-completion semantics and non-durative actions with callback events in between are employed in the controller design, so when the controller is blocked by lower level components, the callbacks issued from restrictions targeting the controller are maintained in a queue (the central queue). When the controller broadcasts a specific event to Bridges, the controller remains blocked until all restriction state machines process that event.

When a restriction becomes restricting, the clinical user needs to issue a joystick double clutch (releases a joystick and activate it again within 5 seconds). The controller maintains a timer to allow pending requests to enter Override mode within 5 seconds.

There are two possibilities to design the controller to deal with Override requests as follows. First, if a restriction requests to enter Override mode, the controller will query all active restrictions by checking whether it is allowed to enter override mode or not. If all active restrictions allow Override, the controller will grant the requested restriction to enter Override mode. If one or more restrictions prohibit Override, the requested restriction will also be prohibited to enter Override mode.

Another solution is that instead of query all restrictions to check whether entering Override mode is allowed or not, it is also possible to maintain a counter variable at the controller to control entering Override. The counter basically increases its value by one when receiving SetOverrideNotAllowed callback and will decrease its value by one when receiving SetOverrideAllowed callback from restrictions. The counter will be used here as a guard in the controller to permit or prohibit active restrictions from entering Override mode (it allows override when its value is zero). The SetOverrideRequestAllowed or SetOverrideRequestNotAllowed events will be broadcasted to all active restrictions depending on the value of the guard after issuing the SetRequestActive or UnSetRequestActive events immediately. So, if any restriction is in state Request Override, it can transit immediately to the Override state after it receives both the SetRequestActive and SetOverrideRequestAllowed events.
Blocking Restrictions

The following figure depicts a Blocking restriction state machine. When such a restriction becomes active and restricting, it will issue `SetBlocking` and `SetOverrideNotAllowed` callback events targeting the controller. Other events and callback events that are depicted above are detailed in Section 5.4. The controller, in turn, will prohibit any restriction to enter Override mode and will force the restrictions which are in Override to exit the Override mode. As is seen from Figure 5.3, the state machine will maintain the user requests as well as the restriction states.

Figure 5.3: The Blocking restriction state machine. Light states denote states with active user requests. Gray states denote states with inactive user requests. Events that cause the transitions are depicted while it is assumed that other events will make self transitions to the same state.
5.3. COMPONENTS DESCRIPTIONS AND STATE MACHINES

Simple Restriction

The following diagram depicts a Simple restriction state machine. This restriction is not Overridable and does not prohibit other restrictions from entering Override mode. The

![Simple Restriction State Machine Diagram]

To make the diagram readable, the callbacks are numbered as follow:
1: SetRestrictionPresent
2: UnSetRestrictionPresent
3: SetPresentAction
4: UnSetPresentAction

Figure 5.4: The Simple restriction state machine. Light states denote the states with active user requests. Gray states denote inactive user requests. Events that cause the transitions are depicted while it is assumed that other events will make self transitions to the same state.

The purpose of this restriction is mainly used to issue user messages.

Overridable Restriction

Figure 5.5 depicts an Overridable restriction state machine. When such a restriction is active and restricting, the user can override this restriction unless a Blocking restriction is present or a StepBack movement is needed. The user may enter Override mode by issuing a joystick double clutch. The state of the restriction may change to/from Present, Restricting, not Present and in Override.
Figure 5.5: The Overridable restriction state machine. A light state denotes active user request. A gray state denotes inactive user request. Events that cause the transitions are depicted while it is assumed that other events will make self transitions to the same state.

StepBack Restriction

Figure 5.6 depicts a state machine of an Overridable restriction with a StepBack movement. The user needs to release the button/joystick to perform a StepBack movement before entering override mode.
Figure 5.6: The StepBack restriction state machine. A light state denotes a state with active user request. A gray state denotes a state with inactive user request. Events that cause the transitions are depicted while it is assumed that other events will make self transitions to the same state.
Bridges

Each active restriction component has an associated Bridge component. A Bridge component is used merely to forward requests issued by the controller to its restriction state machine. The restriction in turn may send callback events to the Bridge. Depending on the callback, the Bridge will forward it to the UserMessages component or to the controller. The Bridge state machine consists of two states. One state receives events from the controller and the central queue, and another state that allows only the return values of the queries. The queries are explained in the next section.

\[\text{Figure 5.7: The Bridge state machine.}\]

UserMessages component

For each restriction state machine there is an associated UserMessages component which is responsible to post user messages to the UIModule. The Bridge component is responsible for forwarding events to the UserMessages component.

5.4 Internal Events

In this section the exchange of events between components is explained. We will first list all internal events and explain the purpose of each one of them separately. Later we explain component interactions.

The following events are exchanged internally between the controller and the restriction state machines via their Bridges. These event are identical for all restriction state machines (Overridable, StepBack, Simple and Blocking restrictions) and they differ by their interface id's (the interface name and its id). For simplicity we omit the interface id's during the description of these events later on throughout this thesis. For instance the interface $ICtrOverridableBridge[id]$ becomes $ICtrOverridableBridge$ which denotes the interface between the controller and the Overridable Bridge component, so events can be defined as $IOverridableBridge.InternalEvent$. 
Before we list the internal events, we introduce an example of event exchange between the controller and the Overridable restriction state machine via its Bridge which enforces the Overridable restriction to enter Override mode. The controller will send the event \texttt{ICtrOverridableBridge.ForceInOverride} to the Bridge component (\texttt{ICtrOverridableBridge} is the Overridable Bridge interface), and the Bridge in turn will forward the event \texttt{IOverridable.ForceInOverride} to the Overridable restriction state machine (\texttt{IOverridable} is the Overridable restriction interface). The behavior above repeats itself for each of the following events:

**ForceInOverride**: The restricted restriction should enter override mode even if Override mode is disabled.

**ForceOutOverride**: The restriction should exit Override mode.

**OverrideStopped**: The restriction should exit Override mode.

**SetOverrideRequestNotAllowed**: Entering Override mode is not allowed and if the restriction is in Override mode then it should exit Override mode.

**SetOverrideRequestAllowed**: Entering Override mode is allowed.

**Controlling callback events issued from Restrictions to Bridges**

Active restrictions may send the following callback events to Bridges via the central queue targeting the controller, and the Bridges will forward them to the controller via the central queue as well. These events are associated with the Bridges callback interfaces:

**SetRestrictionRestricting**: notifies the controller that the restriction becomes restricting so other restrictions should be forced out from Override mode.

**UnSetRestrictionRestricting**: notifies the controller that the restriction is not restricting the resource.

**SetRestrictingAction**: inform the controller that the restriction is in active \texttt{Blocking} state.

**UnSetRestrictingAction**: inform the controller that the restriction exits the active \texttt{Blocking} state.

**SetRestrictionBlocking**: notifies the controller that the restriction becomes \texttt{Blocking} so as a consequence entering Override mode is prohibited.

**UnSetRestrictionBlocking**: informs the controller that the restriction is not in the \texttt{Blocking} state and entering Override mode is allowed.

**RequestOverride**: asks the restriction whether it is possible to enter Override mode.

**SetOverrideAllowed**: notifies the controller that the restriction state machine allows Override.

**SetOverrideNotAllowed**: notifies the controller that entering Override mode is prohibited.

**StartOverridePendingTimeOut**: this callback will inform the controller to start its timer.

**StopOverridePendingTimeOut**: inform the controller to stop the timer.

**SetInOverride**: inform the controller that the restriction is in override mode and as a consequence all restrictions should be forced to enter Override mode if they are in the
Restricting state.

**UnSetInOverride**: the restriction exits the override mode.

**User messages Callbacks issued from restrictions to Bridges**

Most of these messages are generated by the restriction when there is active request from the user. Therefore we add the word "Action" to distinguish between controlling events and user messages events. When a Bridge receives these events, it will simply forward it to the UserMessages state machine which in turn will post it to the UIModule.

- **ActionSetRestrictionPresent**: informs the user that the restriction becomes present so as a consequence the movement will slow down.
- **ActionUnSetRestrictionPresent**: informs the user that the restriction is not present.
- **SetActionStepBackPending**: informs the user that StepBack request is pending and the user need to release button/joystick to allow the StepBack movement and proceed with the movement.
- **UnSetActionStepBackPending**: the StepBack request is not in the pending state because the StepBack movement is starting or the restriction no longer restricting the resource.
- **SetActionStepBack**: informs the user that the StepBack movement is now active and is currently waiting for a signal from the StepBack manager via the controller that the StepBack movement has been completed successfully.
- **UnSetActionStepBack**: informs the user that the StepBack request is completed due to accomplishing the request successfully or the restriction is not restricted the resource (the restriction resolved).
- **ActionPerformStepBack**: informs the controller that the StepBack is needed.
- **SetActionOverridePending**: informs the user that the timer is started and the request to override the restriction is pending for 5 seconds.
- **UnSetActionOverridePending**: un-post **SetActionOverridePending** message.

**Between Bridges and UserMessages components**

The following messages will be translated later using the concrete class which contains the specific date about the restriction. (For example, for a detector bodyguard sensor, the event **PostRestrictionActionOverridePendingUIMessage** will be translated to 'Warning: Detector bodyguard is active'.)

- **PostRestrictionActionOverridePendingUIMessage**: informs the user that the request of overriding a restriction is pending, so the user can release and activate the button/joystick to enter Override mode.
- **PostRestrictionActionPresentUIMessage**: informs the user that the restriction becomes Present.
- **PostRestrictionActionStepBackPendingUIMessage**: informs the user that the StepBack request is pending, so the user needs to release the joystick to allow the StepBack movement.
- **PostRestrictionActionStepBackUIMessage**: informs the user that the movement is
currently performing a StepBack movement.

**PostRestrictionOverridePendingUIMessage**: informs the user that it is possible to activate the button/joystick to enter Override mode.

**PostRestrictionOverrideUIMessage**: informs the user that the movement is in Override mode.

**UnPostRestrictionActionOverridePendingUIMessage**: removes the message **PostRestrictionActionOverridePendingUIMessage**.

**UnPostRestrictionActionPresentUIMessage**: removes the message **PostRestrictionActionPresentUIMessage**.

**UnPostRestrictionActionStepBackPendingUIMessage**: removes the message **PostRestrictionActionStepBackPendingUIMessage**.

**UnPostRestrictionActionStepBackUIMessage**: removes the message **PostRestrictionActionStepBackUIMessage**.

**UnPostRestrictionOverridePendingUIMessage**: removes the message **PostRestrictionOverridePendingUIMessage**.

**UnPostRestrictionOverrideUIMessage**: removes the message **PostRestrictionOverrideUIMessage**.

**Queries**

The controller can query a restriction state via Bridges synchronously and the return value that indicate the restriction state will be returned synchronously. For each state in the restriction state machine, the possibility to query is included. These queries will be used mainly in debugging as well as verification. Basically we query all active restrictions before processing a new external event.

- **IsPresent**: ask the restriction whether it is in state **Present**.
- **IsRestricting**: ask the restriction whether it is in state **Restricting**.
- **IsInOverride**: ask the restriction whether it is in **Override** mode.
- **IsOverrideAllowed**: ask the restriction whether entering Override mode is possible.
- **IsBlocking**: ask the restriction whether it is in Blocking state.
- **IsRequestActive**: ask the restriction whether the user is still activating the button/joystick.

The return values of these queries are as follow:

- **RetPresent**: the restriction is in state **Present**.
- **RetNotPresent**: the restriction is not in state **Present**.
- **RetRestricting**: the restriction is in state **Restricting**.
- **RetNotRestricting**: the restriction is not in state **Restricting**.
- **RetInOverride**: the restriction is in state **Override**.
- **RetNotInOverride**: the restriction is not in state **Override**.
- **RetBlocking**: the restriction is in state **Blocking**.
- **RetNotBlocking**: the restriction is not in state **Blocking**.
- **RetOverrideAllowed**: entering Override mode is allowed.
- **RetOverrideNotAllowed**: entering Override mode is not allowed.
- **RetRequestActive**: the user activates the joystick/button.
**RetRequestNotActive** the user releases the joystick/button.
Chapter 6

Modeling and Specification

In this chapter we explain the steps that have been taken to create the formal model of the software design given in Chapter 5. We select one restriction from each type of restrictions to be involved in the model. The reason of this selection is introduced in Section 6.1. In Section 6.1 we start first to describe the steps that have been done to create the complete setup of the model which includes components specifications and some additional processes that are needed to model the actual behavior of the software design. In Section 6.2 we describe the formal specification of each component and the way to bind them together to construct the formal model in CSP.

6.1 Modeling

Modeling software systems gives the designers a better understanding of the system context as well as system requirements. During modeling the software design, some design defects have been detected before model checking took place. In this section we introduce the steps of constructing our model and we show some design defects that are detected during modeling the behavior of this software system.

Before we start, we need to introduce some concepts. In this chapter, modeling, specifying and verifying systems are heavily related to the concept of model checking which is mainly used to analyze and verify complex distributed systems. In general, this entails a formal model that represents the behavior of the system under consideration. The formal model consists of several components that run in parallel and interact with each other via communication channels.

Model checking tools check the validity of a given requirement by exploring all possible execution scenarios in the formal model. These execution scenarios (state space) represent all possible sequences of events the system can perform. In case the modeled system tends to be quite large, we may encounter a possible state space explosion. An example requirement is that the system must be deadlock and livelock free.

Generally most model checking tools enable model checking without generating the complete state space when verifying requirements. Examples of some techniques that are used to circumvent the state space from exploding are: down-scaling, partial order re-

41
duction, on-the-fly verification and abstraction. Down-scaling means nominating a small number of similar components to be involved in the model and generalize the results to any number of them. Partial order reduction implies that if the execution of actions leads to the same state and no matter at which order they take place, we may consider only one consecutive execution order of these events. On-the-fly verification implies constructing and exploring the state space incrementally, so in case a property is violated, the exploration process will terminate.

Basically, the abstracted model is a simplified reflection of the behavior of a system that ignores irrelevant details which are not related to checked requirements. Depending on the modeled system, we could abstract from internal events within components, variable bounds or even the number of involved components in the model. But it is necessary that the abstraction is sound; after verifying the correctness of a requirement in the abstracted model, this also should hold for the original system[6]. We will use abstraction techniques in this chapter.

6.1.1 Component Down-scaling

The design given in Chapter 5 implies having zero or more possible restriction components of the same type for any given movement (for instance zero or more Overridable restrictions), but including all these components in the model tends to be impractical and may lead to a state space explosion. In practice, some movements may have forty active restrictions in the worst case. Therefore we use the down-scaling technique to reduce the number of involved components in the model without degrading the properties of the system. We nominate only one restriction from each type of restrictions with its related Bridge and UserMessages components to be involved in the model. Such a decision has been made after a careful understanding of the system context. Therefore the initial model contains only one Blocking, Simple, Overridable, StepBack restrictions with their Bridges and UserMessages components, a controller and a central queue. We chose only one restriction from each type of restrictions for the following reasons:

- we chose only one Blocking restriction to be involved in the model since it does not matter if we have one or more active Blocking restrictions; the movement should stop in either case.

- the Simple restriction has no influence on the movement and simply one restriction of this type is sufficient.

- there is a need to have more than one Overridable restrictions. One case is to check if one Overridable type restriction enter Override mode, other Overridable restriction should be forced in to enter Override mode. However we chose one Overridable component since the StepBack restriction also includes the Overridable restriction behavior.

- for the StepBack restriction only one StepBack restriction is needed since for any particular movement the StepBack movement might be done only once.
6.1. MODELING

6.1.2 Closing the Model

Closing the model means, after all components have been checked for deadlocks and livelocks in isolation and formally specified to run together in parallel, we need to close the model by an environment process which behaves chaotically and simulate the behavior of the external world. We add a process to our initial model that simulate the behavior of the environment, and we call it the Rules component (or environment component or even the external world component).

6.1.3 The Complete Model

Figure 6.2 shows the complete setup of components of the formal model. In practice, all these components run in a single execution thread. That means when a single component is processing a request, all other components are blocked.

![Figure 6.1: The complete setup of the model.](image)

Before we start to describe the model, we need to introduce the concept of event synchronization, interleaving and interfering between components. 
Interleaving means that processes run concurrently and independent of each other, and do not interact or share any event. Their behavior is described by all possible combination of events they can perform.

Synchronization means that processes will share and communicate on common events,
and there is no interleaving of events between these components.

We introduce the concept of events *interfering* as processing a new environment event while the components still have to complete processing of an old environment event. An example of interference in our model is that a new environment event is entered to the central queue while a restriction state machine is busy with processing a request and sending callback events to the central queue (the new event is entered to the queue in between processing an old environment event).

When specifying that all involved components will run in parallel, these components will run concurrently and may interleave their events. In practice, the interleaving of events between our design components does not exist since our design runs in a single thread. The system will alternate by the external choice operator and the components run in parallel synchronously. Therefore additional effort is needed to make the model synchronized, prevent any possible interleaving or interfering and model the behavior of the design as a single threaded software design.

To model such a behavior, we employ the non-durative actions with callbacks in between and run-to-completion semantics to specify the behavior between any top level components and its child components (the Bridges with their lower level components and the controller with the Bridges). That will make a synchronous communication between components and prevent the controller with all lower level components from interfering with each other (it is not possible to invoke a method in a Bridge while another Bridge is still processing an old request). However events interfering may occur when a particular component has more than one implemented interface (it is possible at a given state to receive events from more than one communication channels). In our model the interfering might occur between the environment process and the central queue when closing the model.

The environment and LimitClient processes

Before we start, we need to clarify that the main reason to introduce the *LimitClient* process is to restrict the queue and to monitor the behavior of all components to ensure that the components have been completely processed the environment event and its all related callbacks before the next environment event is processed. In the followings we will describe in detail the purpose of introducing the *LimitClient* process.

Based on our model, the environment process is connected to a central queue. If the queue is not bounded, that will lead to generate infinitely many states and the model checker tool can not handle that. Moreover, if the queue is bounded and there is no restriction on the environment process, the queue will become full and the system will deadlock since all places in the queue are occupied even if realizing this situation is difficult in practice. For instance, in our case, the user is very fast to activate and release the joystick to fill up the queue, or objects change their states very fast with respect to sensors.

To circumvent this problem in our model, we introduce a filtering process called *LimitClient* process. All events entering or exiting the queue are passed through this process. The idea is that if a process $p_1$ wants to communicate with another process $p_2$, $p_1$ will communicate first with the filter process and the filter will communicate with $p_2$.
and as a consequence of communication a value of a variable might decrease or increase. This variable is used as a guard to allow or prevent communication with some particular processes. At first, the LimitClient process was simple and maintained a counter (count variable) that increases its value by one when the event is entering the queue and decreases its value by one when the event exiting the queue. The communication between the environment and the queue is guarded by the count variable (the LimitClient process hides the possibility of synchronizing with the environment process if the count variable is not zero). When the count variable is zero, the environment process will be able to synchronize with the LimitClient process which will in turn synchronize with the queue. The queue will synchronize with the LimitClient which will in turn synchronize with the controller. Therefore when the queue is empty, the environment will be allowed to put a new event in the queue. However, there is still some sources of interfering in the model.

![Figure 6.2: The LimitClient State machine.](image)

**Reflecting the actual behavior of the system**

Consider the following scenario. The controller may process an external event and forward it to a restriction state machine via its Bridge. The restriction state machine may send callback events to the central queue and then unlock the Bridge and the controller. Since the queue is not empty, the environment process is still restricted to put a new event in the queue since the count variable is not zero. However, the controller may process the callback events one by one and after the last callback is taken by the controller from the queue, the queue becomes empty and the environment process is now allowed to put a new event in the queue (since count is zero) while the last callback may cause other callbacks to be generated. Therefore the new event arrives while processing an old event. Such behavior can not happen in practice and is considered to be undesirable behavior. We circumvent that by introducing extra synchronization event (we call it the ISwitch event) between the controller and Bridges from one side and the LimitClient process on the other side. The event denotes that the controller or a Bridge has been completely
processed the event or the callback. So when the controller or Bridges finish processing their events, we issue the \textit{ISwitch} event.

We adjust the specification of the count variable in the the \textit{LimitClient} process to cover such a behavior. The \textit{count} variable increases its value by two when a particular event is entering the queue and decreases its value by one if the event is exiting the queue. The controller and the Bridges synchronize with the \textit{LimitClient} process by the \textit{ISwitch} event which decreases the count variable by one. Therefore if the count variable becomes zero, we are certain that the old event is completely processed with all its callback events. The number of generated states in our model is decreased dramatically and the model now reflects the actual behavior of the system. Figure 6.2 shows the \textit{LimitClient} state machine and its formal specification is introduced at the end of the following section.

Note that to model the behavior of an original design, the designer has the freedom to create the components that construct the model which reflect the behavior of the original design. The components of the model are not necessarily be similar to the components of the original design since we are interested in the behavior of the original design. For instance in our model one can incorporate the Bridges to one single component. Another possibility is to incorporate all components to a very large single component since all these components do not interleave with each other, and they act as a large single state machine (a single sequential process). However, the components of our model are similar to the components of the design because we ultimately want to generate a source code of a high-level language automatically from the component specification. Therefore, additional effort is needed to prevent any interfering or interleaving of events between the components in the model.

6.1.4 Results from Modeling

During modeling the behavior of the software design, the following design defect have been detected:

- When multiple Overridable restrictions enter the Override Pending state, they will start a shared timer by issuing the \textit{StartOverridePendingTimeOut} event. However, if one of them is resolved (not restricting any more), it will issue a \textit{StopOverridePendingTimeOut} event and the timer will stop while other Overridable restrictions will remain at the Override Pending state since they are waiting for \textit{SetOverridePendingTimeOut} event while the timer is inactive. Therefore when the clinical user comes later and activates the joystick (say next day or next month), the movement will start to enter Override mode and the beeper will start while he/she should issue a joystick double clutch to enter override.

To circumvent this defect, a counter is maintained at the controller to count the number of current pending requests, so the timer will stop by \textit{StopOverridePendingTimeOut} event only if there is only one active pending request.

- there were two wrong transitions between the Override state and Override Pending state in both Overridable and StepBack restriction state machines when receiving the \textit{ForceOutOverride} event. The transitions have been corrected.
6.2. SPECIFICATION

- missing a StepBack performed state which allows the user to release the joysticks before entering the Override pending state.
- removing an unnecessary redundant stimulus NotRestricting.
- adding a request override State in Overridable restriction state machines which allows entering Override state when all active restrictions allow Override.

6.2 Specification

Each component involved in the model is specified separately as an ASD component by following the steps of SBS mentioned in Chapter 2. Currently, Verum Consultant BV is developing a tool which a designer can specify components in a studio environment which includes all involved components of a model in one file. However, each component state machine given in Chapter 5 was translated to SBS as illustrated in Chapter 2, and later the corresponding CSP code is generated automatically for each one of them.

For confidentiality reasons we are not able to include any generated CSP file, however the algorithm that is used to translate an SBS to equivalent CSP model is given in [7]. Briefly, each canonical sequence in SBS is represented as a state in the formal specification. All variables that are used as predicates are used as parameters in each state. If we consider the example component given in Section 2.2.4., the translation of its SBS to CSP generates four states in total. We can define its top level process as ComponentX_Main which contains the formal specification of all states. This process is used later to run in parallel with other processes.

All these generated files are included in a main CSP file (the manually written CSP) which specify the complete formal model. We introduce some parts of the main CSP file specification, and we omit some parts for confidentiality reasons.

6.2.1 The Queue Specification

The queue is a generic FIF0 queue[9]. We renamed all restrictions and Bridges callbacks to be included in AllQInEvents and AllQOutEvents channels. This will facilitate replicating and hiding these callbacks. All these callbacks pass through the filtering process before they enter the queue, so it is easier to process the callbacks as if they are included in one channel rather than address each one individually. We introduced the purpose of the filtering process in Section 6.1.3.

channel QUEUE_FULL
channel AllQInEvents, AllQOutEvents : CBAllQueueEvents -- all restrictions and Bridges Callbacks

CHASED_CHECK_QUEUE(N, in, out) =
  let
  TAIL = in?x -> (out!x -> TAIL [] in?x -> QUEUE_FULL -> STOP)
  COPY = in?x -> out!x -> COPY
within chase(TAIL [out<->in] ([out<->in] x : <1..N-1> @ COPY))

CBQLen = 6

CentralQueue = CHASED_CHECK_QUEUE(CBQLen, AllQInEvents, AllQOutEvents)

The queue is constructed as a pipeline (n-place buffer). The process COPY is replicated n−1 times constructing n−1 processes that run in parallel as a chain. The processes join each other by the in and out channels forming a single pipe with n−1 places, the internal in and out channels are hidden from the environment. Only the in channel of the Tail process and the out channel of the last COPY process are visible to the external world.

The TAIL process will issue the QUEUE_FULL event when the queue becomes full because it is at the end of the pipe and if it is not possible to forward a new event by the out channel and there is another new input, the queue becomes full. When such event is fired, FDR will show a counter example that describes the scenario which causes the queue to be full. The assertion formula for checking the queue was introduced earlier in Section 2.4.

Note that the restriction callback events are renamed to be included in AllQInEvents channel. For example the Overridable callback event IOverridableCB.SetInOverride is renamed to AllQInEvents.IOverridableCB.SetInOverride. The controller and Bridges callbacks are renamed to be included in AllQOutEvents channel since they receive these events from the queue process. For instance the IOverridableCB.SetInOverride is renamed to AllQOutEvents.IOverridableCB.SetInOverride. The queue process receives these callbacks via AllQInEvents (in) channel and forward them by the AllQOutEvents (Out) channel. Note that renaming all callback events to be included in two channels makes it easier to define the synchronization between components and the queue process rather than introducing each communication channel explicitly.

6.2.2 Binding Bridges with restrictions and UserMessages Components

The following is an example of the formal specification of binding the Simple Bridge component to the Simple and UserMessages components (its used-components). The same specification repeats itself for all other Bridges and their related lower level components.

At first, we start by defining the interfaces, renaming the callback events and run the used components in parallel as follow:

UsedSimpleInterfaces = {{IUIMsgSimple, ISimple}}
UsedSimpleComponents’ = ISimple_Main ||| SimpleUIMessages_Main
UsedSimpleComponents = UsedSimpleComponents’[[x <- AllQInEvents.x | x <- CBSimpleEvents]]

Note that all callback events that are issued from the Simple restriction are renamed to be included in the AllQInEvents channel. The processes ISimple_Main and SimpleUIMessages_Main will run in parallel without any synchronized (shared) events between them (they will synchronize with the Bridge).
6.2. **SPECIFICATION**

Now we combine the Simple Bridge component with its used components and we define the shared events (interfaces) between them (the events in which they will synchronize).

\[
\text{BridgeSimple\_Design}(\text{IG}, \text{IIG}) = ((\text{BridgeSimple\_}(\text{IG}, \text{IIG})[x \leftarrow \text{AllQOutEvents}.x \mid
\text{x} \leftarrow \text{CBSimpleEvents}])\backslash \lbrack \mid \text{UsedSimpleInterfaces}\rbrack \mid \text{UsedSimpleComponents})
\]

The \text{IG} and \text{IIG} variables will define the type of checks for the process with other processes. They might be true or false depending on the type of check. In our model it does not matter which value they hold since we have not define any illegal sequences. Note that the Bridge may receive some callback events from the queue, so we need to rename these callbacks at the Bridges to the \text{AllQOutEvents} channel.

### 6.2.3 Binding the controller with its used components

To bind the \text{ResController} component to its used-Bridge components, we need first to define the controller used interfaces and bind the Bridges (with their used interfaces) to run in parallel together.

\[
\text{UsedCtrInterfaces} = \{\text{ICtrSimpleBridge}, \text{ICtrBlockingBridge}, \text{ICtrStepBackBridge}, \text{ICtrOverridableBridge}, \text{RuleINT}\}
\]

\[
\text{UsedCtrComponents}'(\text{IG}, \text{IIG}) = \text{BridgeStepBack\_Design}(\text{IG}, \text{IIG}) |||
\text{BridgeBlocking\_Design}(\text{IG}, \text{IIG}) |||
\text{BridgeSimple\_Design}(\text{IG}, \text{IIG}) |||
\text{BridgeOverridable\_Design}(\text{IG}, \text{IIG}) ||| \text{Rules\_Main}
\]

\[
\text{UsedCtrComponents}(\text{IG}, \text{IIG}) = \text{UsedCtrComponents}'(\text{IG}, \text{IIG})[\lbrack \mid \text{AllQInEvents}.x \mid
\text{x} \leftarrow \text{CBResControllerEvents}\rbrack]
\]

where the \text{RuleINT} is a channel that contains the \text{Done} event which denotes that the controller has been completely finished forwarding the event and the return values from the Bridges have been successfully received. \text{Rules\_Main} is a process that simulate a chaotic environment (the environment process). Note that all callback events issued from the Bridges that enter the queue are renamed to \text{AllQInEvents}.

Now we combine the controller process with its used components, and define the interfaces between them to obtain the complete specification of the model:

\[
\text{Controller}(\text{IG}, \text{IIG}) = \text{ResController\_}(\text{IG}, \text{IIG})[\lbrack \mid \text{AllQOutEvents}.x \mid
\text{x} \leftarrow \text{CBResControllerEvents}\rbrack]
\]

\[
\text{UnqueuedControllerDesign}(\text{IG}, \text{IIG}) = \text{Controller}(\text{IG}, \text{IIG})[\mid \text{UsedCtrInterfaces}\rbrack \mid \text{UsedCtrComponents}(\text{IG}, \text{IIG})
\]

\[
\text{QueuedController}(\text{IG}, \text{IIG}) = \text{UnqueuedControllerDesign}(\text{IG}, \text{IIG})
\]

\[
\text{ControllerDesign}(\text{IG}, \text{IIG}) = \text{QueuedController}(\text{IG}, \text{IIG})[\mid \text{AllQInEvents}, \text{AllQOutEvents}, \text{IRuleINT}, \text{RuleINT}, \text{ISwitch}, \text{IRuleQ}\rbrack \mid \text{LimitClient}(0)
\]
CHAPTER 6. MODELING AND SPECIFICATION

Where the LimitClient process is the filtering process that control and manage the complete behavior of the model.

6.2.4 The LimitClient Specification

The following is the specification of the LimitClient process.

MAX_LIMIT = 1
LimitingEvents = {IRuleINT}
ISwitchEvents = {On}
channel ISwitch : ISwitchEvents
-- some events used for checking bounds of local variables
channel Check_Mon, back_to_top, goToQuery
Mon_Bound=15 -- the counter should not exceed this value, otherwise I miss
-- "ISwitch.On" somewhere
-- receive something from the external world only if the counter is zero
-- (the components are stable).
LimitClient(count) = count<MAX_LIMIT & ([x:LimitingEvents@x ->
LimitClient_1(count))

LimitClient_1(count) =

   count<=Mon_Bound & ([x:diff(\{AllQInEvents\},\{})@x ->
LimitClient_1(count+2))
   []
   count>=0 & ([x:{AllQOutEvents.IRuleCtr}@x ->
LimitClient_1(count-1))
   []
   count>=0 & ([x:{IRuleINT}@x -> DoQueries(count))
   [] count>Mon_Bound & Check_Mon -> STOP

DoQueries(count) =

   count<MAX_LIMIT & goToQuery -> Query_0(count,0)
   []
   count<=Mon_Bound & ([x:diff(\{AllQInEvents\},
   {AllQInEvents.IRuleCtr})@x -> DoQueries(count+2))
   []
   count>=0 & ([x:{AllQOutEvents}]@x -> DoQueries(count-1))
   []
   count>=0 & [RuleINT]@x -> DoQueries(count))
   []
   count>=0 & ISwitch.On -> DoQueries(count-1)
   []
   count<0 & Check_Mon -> STOP
   [] count>Mon_Bound & Check_Mon -> STOP

Query_0(count,0) =
--here we query all restrictions state machines
back_to_top -> LimitClient(count)
)

As is seen from the above specification, the LimitClient process contains three states. The first state allows the environment process to communicate with the LimitClient process and as a consequence an environment event will enter the central queue. The second state (LimitClient.1) allows the callback events of the components excluding the environment process to enter and exit the central queue, and prevents synchronization with the environment process. The state will transit to the third state when it receives the IRuleINT.Done event from the controller which denotes that the controller has been completely forward the request to the Bridges and receive their return values. The third state (DoQueries) allows the callbacks to enter and exit the central queue, and will receive the ISwitch.on event which denotes that a bridge or the controller has been finished processing the callback event. When the count variable becomes zero, the state will transit to the Query state. The fourth state (Query.0) queries the states of the restriction state machine sequentially.
Chapter 7

Verification

In this chapter we shall confine attention to the steps that have been done to verify the behavior of the system. We show briefly the steps that have been made to check deadlocks and livelocks of components, and then we translate the informal requirements given in Chapter 4 to their corresponding CSP processes that are employed for refinement checks. At the end of the chapter, we will shed a light on the description of a part of the generated source code which is currently implemented in the real system. Note that we use ASD tool Version 1.0. for writing components specification, and for verification we used FDR Version 2.82.

7.1 Checking components Deadlocks and livelocks

Each component is checked for deadlocks and livelocks once individually and once with its used lower level components. We start to check the deadlock and livelock freedom of components using the bottom-up approach by following the steps mentioned in Section 2.2.1. So, we start to check all lower level components individually (to check any possible miss-spelling as well), and then each Bridge component with its related lower level components. After all Bridges with their lower level components have been checked, we add the controller and the rest of the components to accomplish the deadlock and livelock checks for our complete model. Note that we have not replaced the design model of the components by their interface models since it is not a valid refinement to replace the Bridges state machines and all their lower level components by the Bridges interface specifications. Replacing the design models of the Bridges by their interface specifications will change the properties of the system, and the behavior of the system will be lost.

The following example shows the assertion formula that is used to check the deadlock and livelock freedom of the ControllerDesign process which represents the complete model:

\[
\text{assert ControllerDesign(true,false)} : \text{[deadlock free [FD]]}
\]

To check the risk of having infinite loops of internal events caused by forcing in/out of Override mode between multiple active Overridable restrictions, we apply the following assertion to check whether the overall system is livelock free.
CHAPTER 7. VERIFICATION

\[ \text{RuleMainEvents} = \{ \text{IRuleQ, AllOutEvents.IRuleCtr} \} \]
\assert \text{ControllerDesign(true,false)} \setminus \text{diff(Events,RuleMainEvents)} \\
:\text{[livelock free [FD]]} \]

Fortunately, the system is livelock free.

7.2 Requirements Verification

Before we start introducing the steps of verifying the requirement, we need to clarify some issues. The input events are identical for all restriction state machines, they differ only by their interface id’s, and at any state in any restriction state machine, it is allowed to receive any input event. So some of these events may cause transitions to other states while others cause a self transition to the same state (there are no illegal sequences). Since all informal requirements are safety requirements, we formulate our informal requirements as safety properties by translating these requirements to their corresponding CSP processes, and we verify them using the traces refinement model supported by FDR. In the following, we introduce a basic example to illustrate the traces and failures refinement models supported by FDR.

Consider the system depicted in Figure 7.1. Let \( a_1 \) denotes pressing a specific button, \( a_2 \) releasing a brake and \( a_3 \) operating a motor. It is important that the brake must be released before the motor is operated otherwise the motor could be damaged. From the figure, one can easily note that there are some undesirable behaviors in the system since the motor can be operated before the brake is released. However we are going to check a property such that when the button is pressed, the brake will be released and the motor will operate ultimately. The following CSP process could be used to check such a property using the traces refinement model, and such undesirable behaviors are detected as follow:

\[ \text{SpecProperty} = a_1 \rightarrow a_2 \rightarrow a_3 \]
\assert \text{SpecProperty} [\text{T= Design} \setminus \text{diff(Events,\{a1,a2,a3\})}] \\

Where \( \text{Design} \) is our system specification, \( \text{Events} \) is the complete set of all events in the model, “\( \setminus \)" is the hiding operator and \( \text{diff} \) is the set difference. Recall that

\[ P \subseteq Q = \text{traces}(Q) \subseteq \text{traces}(P) \]

The property will be violated and some counter examples will be generated because of the following reason:

\[ \text{traces(SpecProperty)} = \{ a_1a_2a_3 \} \]
\[ \text{traces(Design)} = \{ a_1a_2a_3, a_1a_3a_2, a_1a_3a_2a_3 \} \]
\[ \text{traces(Design)} \not\subseteq \text{traces(SpecProperty)} \]

So, these extra traces that the design can perform are considered to be undesirable behavior. An example is the trace \( a_1a_3a_2a_3 \) which shows the possibility to operate the motor.
7.2. REQUIREMENTS VERIFICATION

before releasing the brake. From the above example we can simply conclude that the execution order of events should be preserved in both the formulated property and the formal model.

Now consider the state space depicted in Figure 7.1.(b). At one trace, we have a self loop of $a_2$. By applying the trace refinement above such undesirable behavior could not be detected since both the property and Design traces are identical. However, using the failure refinement model such undesirable behavior could easily be detected.

$\text{SpecProperty} = a_1 \rightarrow a_2 \rightarrow a_3$

assert SpecProperty $[P= \text{Design}\setminus\text{diff}(\text{Events},\{a_1,a_2,a_3\})$

Recall that

$$P \subseteq_F Q = \text{failures}(Q) \subseteq \text{failures}(P)$$

The property will be violated since at the trace $a_1a_2$ the design will refuse to do the set of actions $\{a_1,a_3\}$ and $\{a_1,a_2\}$ while the property as the same trace refuses only to do $\{a_1,a_2\}$, so

$\text{failures}(\text{Design}) \not\subseteq \text{failures}(\text{SpecProperty})$

Back again to our model, the requirements that are given in Chapter 4 are translated to CSP processes and verified using the traces refinement model.

For readability, we only introduce an example requirement to clarify the steps that have been made for verification, and since the specification of the requirements are long to be included in this section, we list the complete formal specification of all requirements in Appendix B.

Now, consider the following informal requirement:

R14: “If the stimulus UnSetRequestActive is processed, the Overridable state machine is not in state Override.”

The corresponding CSP process is as follows:
Rule14StateOverridable = AllQOutEvents.IRuleCtr.UnSetRequestActive ->
    Rule14StateOverridable_a
    []
    IRuleQ.OverridableIsInOverride ->
    Rule14StateOverridable_q

Rule14StateOverridable_a = IRuleQ.OverridableIsInOverride ->
    IRuleQ.OverridableRetNotInOverride ->
    Rule14StateOverridable

Rule14StateOverridable_q = IRuleQ.OverridableRetInOverride ->
    ~| ~|
    |~|
    IRuleQ.OverridableRetNotInOverride ->
    Rule14StateOverridable

Rule14StateOverridable_Events = { AllQOutEvents.IRuleCtr.UnSetRequestActive ,
    IRuleQ.OverridableIsInOverride ,
    IRuleQ.OverridableRetInOverride ,
    IRuleQ.OverridableRetNotInOverride }

assert Rule14StateOverridable [T= ControllerDesign(true,false)\diff(Events, Rule14StateOverridable_Events)]

The process shows that when the *UnSetRequestActive* event is processed then if we query the Overridable restriction state machine, the answer must be that the state machine is not in Override state. Because our design allows querying at any state (it allows also all possible input events at any state), the process shows that it is possible to query at any state but the result is not important (we use the internal choice operator to specify a non-deterministic return value for the query). If we do not add the possibility to query a restriction from the initial state of the property, the property will be violated since it is possible to query any state at any time. However, we specified that in state *Rule14StateOverridable_a* the answer of the query must be *OverridableRetNotInOverride*. So, if there is a trace that contains *OverridableRetInOverride* after processing *UnSetRequestActive* event, a counter example will be generating that describes that trace. Note that all events in the design are hidden except those events which are interesting for the property.

### 7.3 Results

All requirements which are listed on Chapter 4 have been formally verified and no flaws have been detected, see Appendix B for the complete set of the formal requirements. After verification has been accomplished, we generate the corresponding C++ code for each restriction state machine, and we add the generated source code to the current
implementation. The generated C++ code is also tested using the conventional testing approaches and we finally deploy the code in the real system. We applied some scenarios in the real system to check the validity of the generated C++ code. We applied some important scenarios like overriding a restriction and performing a StepBack movement before entering Override mode when multiple Overridable restrictions are active. All these scenarios have been validated in the real system.

The main reason of generating the C++ code only for the restrictions state machines is that for any ASD component that has callback interfaces, a queue implementation will also be created when generating the source code automatically. The generated queues in our design will add unnecessary complexity to the implementation and will create multiple threads which might be troublesome to debug and resolve any possible deadlocks or race conditions with respect with other parts of the Patient and Beam unit which are not formally model checked yet. So a decision has been made to implement the controller and the Bridges manually (since the implementation of them is simple and straightforward) and generate the restriction state machines automatically. That will ensure that the software will run in one single execution thread.

However, a suggestion is introduced to circumvent the problem of generating these queues. The suggestion implies defining the callback interfaces as implemented interfaces when generating the C++ code for components that have callback interfaces. In this suggestion we treat the queue as a process that synchronously communicate with other components, but there still should be some handwritten C++ code to glue up all components together. Throughout this suggestion, we aimed to generate as much code as we can but unfortunately we could not investigate the validity of this suggestion due to time.
Chapter 8

Conclusions

This work is part of an investigation that is made to compare the ASD design methodology with conventional design methodologies. This work is involved in the generic part of the investigation which concerns applying the ASD tool to model check and verify an already existing design. The main aim of the study in general is to investigate whether ASD design methodology will increase the product quality, decrease test and integration efforts and generate a reliable source code automatically. The results of the investigation in general are covered in a separate pilot report. The report addresses some advantages and suggestions to enhance the ASD tool. For confidentiality reason the report is not included in this thesis.

ASD tool allows writing components specification by means of Microsoft Excel sheets. Using Microsoft Excel interface, writing components specification becomes quite easier than using normal text editors. The time and effort of writing components specification using Excel or any text editor is quite identical, but when a user needs to change some parts of the specification (say renaming a particular event), Excel has a plus. However some of the ASD tool functions tend to be quite cumbersome when specifying using Excel. For instance to reorder, delete or add a given list of responses in a given state transition, the user must re-enter the list again. Moreover it is not allowed to include more than ten responses at any state transition. Fortunately in our design we have not encountered a need to include more than ten responses at any state transition, but if it is the case, we may circumvent that by splitting up responses among extra states.

One pros of ASD is that it defines a systematic way that designers should follow to design their software systems, so these designs tend to be quite understandable between designers and most of project stockholders. As I noticed during my work with some designers at Philips, some designs that are made using conventional design methodologies in our department tend to be at some points quite confusing for other designers. Sometimes these designs need more explanations about how they actually work in practice, and some explanations about their real implementation.

Currently ASD checks only whether given components are deadlock and livelock free and checks also illegal sequences which are defined during the enumeration process in SBS. By using that, ASD implies that the requirements are hold during the enumeration.
CHAPTER 8. CONCLUSIONS

process that implies also the requirement traceability, so for some designs there is no need to formulate dedicated properties to verify requirements. ASD consultants try always to create such designs to avoid formulating dedicated CSP properties for verification since most of designers at Philips Healthcare have no experience of CSP. ASD designers (at Verum) are currently trying to find a systematic way which allows unexperienced CSP designers to formulate properties easily for verification.

Nevertheless, we say that these checks (deadlock, livelock and illegal sequences) are fruitful and valuable but also verifying requirements is much better since it allows to formulate in a very direct way what is to be expected from the system. In this way more design flaws might be detected in general than using the built in checking facilities.

Constricting formal models using CSP tends to be quite easy, however formulating properties for verification is quite complex. It is complex in the sense that all trivial cases should be addressed in the property. For example if we consider our model and formulate a liveness property to check it using the failures refinement model in FDR, this property tends to be quite large and approximately equal to the specification of the design. In our model, if we want to check such a property, we need to include all events including all callbacks to the property. An example is that in the empty sequence the design refuses to do all callback events including the return values of the queries, so the property should include all these events and refuse them to make the property proceed. The same behavior repeats itself for any state in the formulated property.

One remark of using FDR is that the toolset does not support a simulation tool which helps designers to execute some scenarios. The tool supports process behavior explorer tool (ProBE) which uses a hierarchical list to display events and states (the guards and local variables have no effects). The simulation tool helps designers to create their formal model more faster in the sense that they can trace events flows as well as simulating some scenarios while constructing their model. In FDR, if we need to check whether a place in the model is reachable, we inject extra event and formulate a property to violate it by the assertion, so FDR generates a counterexample to show the execution scenario that leads to execute such injected event in the model. If the property pass the assertion, that place is not reachable. For example, we use the assertion and traces refinement model to check the Queue.Full event. Moreover during constructing the model, we inject some extra events to check variable bounds as well as some guards and to check whether some places in the model are reachable. When a property is violated, FDR can show more than one counterexample at a time.

To construct our model, we needed to use the down-scaling technique to reduce the number of involved components in the model. So, if there is a 1 to n relationship between components in the design then to create a formal model, the number of n components should be fixed since the model checking tool does not support open systems. Down-scaling is also needed to avoid the state space explosion.

After verification of the requirements was accomplished, we generate the source code of some components. The generated source code was very limited due to the added complexity of some components which introduce a multi-threaded program when generating their source code comparable if their code is written manually. That is because the ASD tool generates a queue for free when a component has a callback interface. Fortunately,
we generate the most important part of the design which includes the restrictions state machines. Since it is necessary for the controller to create and bind restrictions state machines dynamically, the controller source code is written manually. The source code of the Bridge components are also written manually.

In general, the design developed in this thesis is verified and implemented in the real system. During modeling the software design, some flaws have been detected and corrected. During verification, no flaws have been detected. Moreover, we generate the restrictions state machines source code, test the code further using the conventional testing methods, and finally we deploy the code in the real system.

We would like to advice software designers to avoid asynchronous designs. Before this design, some other designs have been analyzed. One of these designs is quite similar to our design but it implies queues for each component that receive callback events (for the controller and the Bridges). For such a design it tends to be quite time consuming to make the design verifiable and to restrict the behavior of these components with respect to their queues. Moreover the design is not valid since there are lots of possibilities to introduce event interference, see Appendix A. For instance, it could be possible at a given time for a Bridge to process a callback event from its queue or to process a new environment event coming from the controller at its waiting state. So, if the technical layer is rapidly reporting changes of resources, the interference of events may occur and the clinical user may experience undesirable behaviors. Moreover such a design will introduce unnecessary complexity by adding a thread for each component. Also the FDR tool reports deadlocks which are caused by continuously processing environment events and leaving the callback events inside the queues intact, so the queues become eventually full and components can not send other callback events to these queues. To circumvent this problem, additional effort should be taken to restrict the environment process as well as the Bridges queue components but we may lose the generation of a valid source code in this case.

In general we would like to suggest designers when constructing their formal models to consider components that have more than one implemented interface if they implement single threaded designs. When such components exist, these components might be a possible source of event’s interfering which may not exist in the original design. For instance if we consider our design, there is a risk of event interfering at Bridges components because clearly any Bridge component has more than one communication channel namely the interface with the controller and the interface with the queue since the queue is considered to be a process as well. So, for some designs if there is a state which allows processing events from more than one interface, this state needs a careful concern. Another example is the central queue itself since it implements all callback interfaces and the environment interface. Such consideration helps to reflect the actual behavior of the original design.

When closing a given model by an environment process, a queue might be implemented to store incoming events. We introduced some problems that are caused by closing our model in Section 6.1.2. We circumvent these problems by introducing a filtering process which allows the callback events to pass through it. We restrict the environment process by a guard in the filtering process which allows or denies synchronization between the filtering process and the environment process. Of course we can synchronize events between the
environment process and the controller directly without introducing a queue in between them by having the assumption that our system is faster than our environment, but the filtering process is still needed since it organize all activities in our model.

In general, analyzing software designs by means of model checking allows the designers to detect flaws of designs upfront implementation and testing. Modeling behaviors of software designs gives designers a better understanding of the targeted system context and some flaws could be detected during modeling and before verification. Model checking imposes designers to be more precise when they create their informal requirements since these requirements will be translated formally and verified later against the formal model.

Since model checking techniques are new in the field of software engineering and the world is currently shifting toward component-based software engineering, we would like to suggest Philips Healthcare to create initially a small unit of some people who are interested in this field. The unit will be responsible for model checking and verifying the designs provided by various departments. That is because we believe that modeling and verifying designs are valuable when they take place at earlier design stages and they could be realized in a short period of time. The main purpose of introducing such a unit is not only to detect earlier design defects but also to ultimately provide the customers of Philips with highly sophisticated and reliable products by using the power of model checking techniques.
Appendices

Appendix A: The Queued Design

Via the following design, it is possible to generate the source code of all restriction state machines as well as their Bridges. However, we need to analyze this design before we decide to implement it.

From the figure below, one may notice that the Bridges and the controller have more than one implemented interface (their implemented interfaces and the callback interfaces attached to the queues). As we concluded by this thesis, such interfaces might be a source of interfering between events and may cause undesirable behaviors.

![Figure 1: The software design.](image)

When we model checked this design using FDR, the tool reports deadlocks caused by filling up the capacity of the Bridges queues. That is because there is always a possibility to process events coming from the controller or to process the callback events in the queue at the waiting state (consider the external choice operator here). So even if we restrict the behavior of queues, we realized that it is not a valid design to implement because interfering of events is possible and that may cause undesirable behaviors for clinical
users. One example is that the technical layer might rabidly report changes of movement resources and user requests, so new environment events may be injected while components are busy processing and old environment event. An example is as follow: when the run to completion is fulfilled and all clients have been unlocked by their server components including the controller, the controller may forward the next event while there are some callback events inside the queues. The behavior repeats itself and the queue ultimately becomes full.

In the following we give an example scenario of an undesirable behavior. Consider that the blocking restriction becomes active and restricting. The controller forwards the \textit{SetRestriction Restricting} event to the blocking restriction. When the blocking restriction receives the event via its Bridge, it will send callback events to its queue targeting the controller to inform the controller that no movement is allowed (the \textit{SetOverride NotAllowed} callback event). So, after the blocking restriction send these callback events to the queue, it unlocks the Bridge which also unlocks the controller. The controller is now ready to receive a new environment event while there are some callbacks in the queue need to be processed by the Bridge. However consider that the controller receives the \textit{SetRequest Active} event from the environment and forwards it to the restrictions. If there is an Overridable restriction waiting to enter Override mode, it will enter Override mode immediately when it receive the \textit{SetRequest Active} event since Override is allowed by the controller and the controller has no idea that the movement is currently prohibited since the callback events of the Blocking restrictions are still intact in the queue. In such a case the clinical user may encounter a fast flashing messages in the screen or a unexpected fast beeper.

We can circumvent this problem by splitting up the waiting state of bridges to two state, one to receive events from the controller and another from the queue. However we need to add extra events with the callbacks to indicate that the queue is now empty. Adding such event will restrict the source code generation of the restriction state machines since additional overhead should be maintained to deploy the code in the current implementation.

We realized that it will take additional efforts to restrict the queues specially the controller queue, but we may lose the code generation of the components since we need to introduce these extra synchronization events. Another solution is to introduce a filtering process to each queue.

Event if we do that, the generated source code will add unnecessary complexity by introducing a multi-threading program and that makes testing these components using conventional methods quite difficult. Moreover, again this is not a valid design due to events interfering.
Appendix B: The Formal Requirement Formulas in CSP

This appendix covers the formal specification of the informal requirements given in Chapter 4. We use the traces refinement model to verify these requirements.

\[
\text{Rule1ControlOverridable} = \begin{align*}
&\text{IRuleQ.OverridableIsInOverride} \rightarrow \text{Rule1ControlOverridable}_q \\
&\text{IRuleQ.OverridableIsRestricting} \rightarrow \text{IRuleQ.OverridableRetRestricting}\rightarrow \\
&\text{IRuleQ.OverridableIsInOverride} \rightarrow \text{IRuleQ.OverridableRetNotInOverride} \\
&\text{IRuleQ.OverridableIsRestricting} \rightarrow \text{IRuleQ.OverridableRetRestricting} \\
&\text{IRuleQ.OverridableIsInOverride} \rightarrow \text{IRuleQ.OverridableRetInOverride} \\
&\text{IRuleQ.OverridableIsRestricting} \rightarrow \text{IRuleQ.OverridableRetNotRestricting} \rightarrow \\
&\text{Rule1ControlOverridable}_a \\
&\text{Rule1ControlOverridable}_b = \\
&\text{IRuleQ.OverridableIsRestricting} \rightarrow \text{IRuleQ.OverridableRetRestricting} \\
&\text{IRuleQ.OverridableIsInOverride} \rightarrow \text{IRuleQ.OverridableRetInOverride} \\
&\text{IRuleQ.OverridableIsRestricting} \rightarrow \text{IRuleQ.OverridableRetNotRestricting} \\
&\text{Rule1ControlOverridable}_q = \begin{align*}
&\text{IRuleQ.OverridableRetInOverride} \rightarrow \text{Rule1ControlOverridable} \\
&\text{IRuleQ.OverridableRetNotInOverride} \rightarrow \text{Rule1ControlOverridable} \\
&\text{Rule1ControlOverridable}_q\_restricting = \begin{align*}
&\text{IRuleQ.OverridableRetRestricting} \rightarrow \text{Rule1ControlOverridable} \\
&\text{IRuleQ.OverridableRetNotRestricting} \rightarrow \text{Rule1ControlOverridable} \\
\end{align*} \\
&\text{Rule1ControlOverridable}_\text{Events} = \{ \text{AllQOutEvents, ICtrlStepBackBridgeCB.SetInOverride}, \text{IRuleQ.OverridableIsInOverride}, \text{IRuleQ.OverridableRetInOverride}, \text{IRuleQ.OverridableRetNotInOverride}, \text{IRuleQ.OverridableIsRestricting}, \text{IRuleQ.OverridableRetRestricting}, \text{IRuleQ.OverridableRetNotRestricting} \}
\end{align*}
\]
Rule2Control =  
AllQOutEvents.ICtrStepBackBridgeCB.UnSetOverrideAllowed -> Rule2Control_a  
AllQOutEvents.ICtrBlockingBridgeCB.UnSetOverrideAllowed -> Rule2Control_a  
IRuleQ.OverridableIsInOverride -> Rule2Control_q  
IRuleQ.StepBackIsInOverride -> Rule2Control_q_SB  
AllQOutEvents.ICtrStepBackBridgeCB.SetOverrideAllowed -> Rule2Control

Rule2Control_a = IRuleQ.OverridableIsInOverride -> Rule2Control_a  
IRuleQ.OverridableRetNotInOverride -> IRuleQ.StepBackIsInOverride -> IRuleQ.OverridableRetNotInOverride -> Rule2Control

Rule2Control_q = IRuleQ.OverridableRetInOverride -> Rule2Control_q  
|~|  
IRuleQ.OverridableRetNotInOverride -> Rule2Control

Rule2Control_q_SB = IRuleQ.StepBackRetInOverride -> Rule2Control_q_SB  
|~|  
IRuleQ.StepBackRetNotInOverride -> Rule2Control

RuleControl2_Events =
AllQOutEvents.ICtrStepBackBridgeCB.UnSetOverrideAllowed, 
IRuleQ.OverridableIsInOverride, IRuleQ.StepBackIsInOverride, 
IRuleQ.OverridableRetInOverride, AllQOutEvents.ICtrStepBackBridgeCB.SetOverrideAllowed, 
IRuleQ.OverridableRetNotInOverride, AllQOutEvents.ICtrBlockingBridgeCB.UnSetOverrideAllowed, 
IRuleQ.StepBackRetInOverride, IRuleQ.StepBackRetNotInOverride

assert Rule2Control [T= ControllerDesign(true,false)]\diff(Events,RuleControl2_Events)

-----------------------------------------------------------------------------------------------

-- here it is useful to see how both will go to override by the counter examples. just put the Not and that's it

Rule2ControlRV = AllQOutEvents.ICtrStepBackBridgeCB.SetInOverride -> Rule2ControlRV_a  
AllQOutEvents.ICtrBlockingBridgeCB.SetInOverride -> Rule2ControlRV_b  
IRuleQ.BlockingIsOverrideAllowed -> Rule2ControlRV_q  
IRuleQ.StepBackIsOverrideAllowed -> Rule2ControlRV_q_SB
Rule2ControlRV_a =
AllQOutEvents.ICtrOverridableBridgeCB.SetInOverride -> Rule2ControlRV_c
  []
  IRuleQ.BlockingIsOverrideAllowed -> Rule2ControlRV_q
  []
  IRuleQ.StepBackIsOverrideAllowed -> Rule2ControlRV_q_SB

Rule2ControlRV_b =
AllQOutEvents.ICtrStepBackBridgeCB.SetInOverride -> Rule2ControlRV_c
  []
  IRuleQ.BlockingIsOverrideAllowed -> Rule2ControlRV_q
  []
  IRuleQ.StepBackIsOverrideAllowed -> Rule2ControlRV_q_SB

Rule2ControlRV_c =
IRuleQ.BlockingIsOverrideAllowed ->
IRuleQ.BlockingRetOverrideAllowed ->
IRuleQ.StepBackIsOverrideAllowed ->
IRuleQ.StepBackRetOverrideAllowed -> Rule2ControlRV

Rule2ControlRV_q =
IRuleQ.BlockingRetOverrideAllowed -> Rule2ControlRV
  |~|
  IRuleQ.BlockingRetOverrideNotAllowed -> Rule2ControlRV

Rule2ControlRV_q_SB =
IRuleQ.StepBackRetOverrideAllowed -> Rule2ControlRV
  |~|
  IRuleQ.StepBackRetOverrideNotAllowed -> Rule2ControlRV

Rule2ControlRV_Events =
{ AllQOutEvents.ICtrOverridableBridgeCB.SetInOverride, AllQOutEvents.ICtrStepBackBridgeCB.SetInOverride, IRuleQ.BlockingIsOverrideAllowed, IRuleQ.BlockingRetOverrideAllowed, IRuleQ.StepBackIsOverrideAllowed, IRuleQ.StepBackRetOverrideAllowed, IRuleQ.BlockingRetOverrideNotAllowed, IRuleQ.StepBackRetOverrideNotAllowed }

assert Rule2ControlRV [T= ControllerDesign(true,false)] \diff(Events,Rule2ControlRV_Events)

------------------------------------------------------------------------------------------------------------------------

Rule3Control =
IRuleQ.OverridableIsInOverride -> Rule3Control_q
  []
  IRuleQ.StepBackIsInOverride -> Rule3Control_q_SB
  []
  AllQOutEvents.ICtrOverridableBridgeCB.SetRestrictionRestricting
  -> Rule3Control_a
  []
  AllQOutEvents.ICtrStepBackBridgeCB.SetRestrictionRestricting
  -> Rule3Control_a

Rule3Control_a =
IRuleQ.OverridableIsInOverride -> IRuleQ.OverridableRetNotInOverride
IRuleQ.StepBackIsInOverride -> IRuleQ.StepBackRetNotInOverride

67
Rule3Control_q = 
IRuleQ.OverridableRetInOverride -> Rule3Control
| ~ |
IRuleQ.OverridableRetNotInOverride -> Rule3Control

Rule3Control_q_SB = 
IRuleQ.StepBackRetInOverride -> Rule3Control
| ~ |
IRuleQ.StepBackRetNotInOverride -> Rule3Control

Rule3Control_Events = {
IRuleQ.OverridableIsInOverride, 
IRuleQ.OverridableRetInOverride, IRuleQ.OverridableRetNotInOverride, 
IRuleQ.StepBackIsInOverride, IRuleQ.StepBackRetInOverride, 
IRuleQ.StepBackRetNotInOverride, 
AllQOutEvents.ICtrOverridableBridgeCB.SetRestrictionRestricting, 
AllQOutEvents.ICtrStepBackBridgeCB.SetRestrictionRestricting }

assert Rule3Control [T= 
ControllerDesign(true,false)|diff(Events,Rule3Control_Events)]

Rule4ControlOverridable = 
AllQOutEvents.IRuleCtr.OverridableSetPresent -> Rule4ControlOverridable_a
| ~ |
IRuleQ.OverridableIsInOverride -> Rule4ControlOverridable_q

| ~ |
AllQOutEvents.IRuleCtr.OverridableSetRestricting -> Rule4ControlOverridable
| ~ |
IRuleQ.StepBackIsInOverride -> Rule4ControlOverridable_q_stepback

| ~ |
AllQOutEvents.IRuleCtr.StepBackSetPresent -> Rule4ControlOverridable_aSB

| ~ |
AllQOutEvents.IRuleCtr.StepBackSetRestricting -> Rule4ControlOverridable

Rule4ControlOverridable_a = 
AllQOutEvents.IRuleCtr.OverridableSetRestricting -> 
Rule4ControlOverridable_b
| ~ |
IRuleQ.OverridableIsInOverride -> IRuleQ.OverridableRetNotInOverride -> 
Rule4ControlOverridable_a
| ~ |
AllQOutEvents.IRuleCtr.OverridableSetPresent -> Rule4ControlOverridable_a
| ~ |
AllQOutEvents.IRuleCtr.StepBackSetPresent -> Rule4ControlOverridable_aSB
| ~ |
IRuleQ.StepBackIsInOverride -> IRuleQ.StepBackRetInOverride -> 
Rule4ControlOverridable_a
| ~ |
IRuleQ.StepBackIsInOverride -> IRuleQ.StepBackRetNotInOverride -> 
Rule4ControlOverridable_a
| ~ |
AllQOutEvents.IRuleCtr.StepBackSetRestricting -> Rule4ControlOverridable_a

68
Rule4ControlRVOverridable =
AllQOutEvents.ICtrOverridableBridgeCB.SetInOverride ->
AllQOutEvents.ICtrStepBackBridgeCB.SetInOverride ->
IRuleQ.OverridableIsInOverride -> IRuleQ.OverridableRetNotInOverride ->
IRuleQ.StepBackIsInOverride -> IRuleQ.StepBackRetInOverride ->
Rule4ControlRVOverridable_a
AIIQOutEvents.ICtrStepBackBridgeCB.SetInOverride ->
AIIQOutEvents.ICtrOverridableBridgeCB.SetInOverride ->
IRuleQ.OverridableIsInOverride -> IRuleQ.OverridableRetNotInOverride ->
IRuleQ.StepBackIsInOverride -> IRuleQ.StepBackRetInOverride ->
Rule4ControlRVOverridable
AIIQOutEvents.ICtrStepBackBridgeCB.SetInOverride ->
AIIQOutEvents.ICtrOverridableBridgeCB.SetInOverride ->
IRuleQ.OverridableIsInOverride -> Rule4ControlRVOverridable_a
AIIQOutEvents.ICtrStepBackBridgeCB.SetInOverride ->
IRuleQ.OverridableIsInOverride -> Rule4ControlRVOverridable_q
AIIQOutEvents.ICtrStepBackBridgeCB.SetInOverride ->
IRuleQ.OverridableIsInOverride -> Rule4ControlRVOverridable_q
AIIQOutEvents.IRuleCtr.OverridableSetPresent -> Rule4ControlRVOverridable
AIIQOutEvents.IRuleCtr.StepBackSetPresent -> Rule4ControlRVOverridable_a

Rule4ControlRVOverridable_a =
AIIQOutEvents.IRuleCtr.OverridableSetPresent ->
IRuleQ.OverridableIsInOverride -> IRuleQ.OverridableRetNotInOverride ->
IRuleQ.StepBackIsInOverride -> IRuleQ.StepBackRetInOverride ->
Rule4ControlRVOverridable
AIIQOutEvents.IRuleCtr.OverridableSetPresent ->
IRuleQ.OverridableIsInOverride -> IRuleQ.OverridableRetNotInOverride ->
IRuleQ.StepBackIsInOverride -> IRuleQ.StepBackRetNotInOverride ->
Rule4ControlRVOverridable
AIIQOutEvents.IRuleCtr.OverridableSetPresent ->
IRuleQ.OverridableIsInOverride -> Rule4ControlRVOverridable_q

Rule4ControlRVOverridable_b =
IRuleQ.OverridableIsInOverride -> IRuleQ.OverridableRetNotInOverride ->
Rule4ControlRVOverridable

Rule4ControlRVOverridable_q =
IRuleQ.OverridableRetNotInOverride -> Rule4ControlRVOverridable
IRuleQ.OverridableSetPresent -> Rule4ControlRVOverridable

Rule4ControlRVOverridable_q_sb =
IRuleQ.StepBackRetInOverride -> Rule4ControlRVOverridable
IRuleQ.StepBackRetNotInOverride -> Rule4ControlRVOverridable

Rule4ControlRVOverridable_Events = {
AIIQOutEvents.IRuleCtr.StepBackSetPresent,
AIIQOutEvents.ICtrOverridableBridgeCB.SetInOverride ,
AIIQOutEvents.ICtrStepBackBridgeCB.SetInOverride,
assert Rule4ControlRVOverridable\[T=ControllerDesign(true,false)\] \diff(Events,Rule4ControlRVOverridable_Events)

Rule5StateOverridable =
ICtrOverridableBridge.ForceInOverride -> Rule5StateOverridable_a
       IRuleQ.OverridableIsInOverride -> Rule5StateOverridable_a
       IRuleQ.OverridableIsRestricting -> Rule5StateOverridable q
       IRuleQ.OverridableRetRestricting -> Rule5StateOverridable q_restricting

Rule5StateOverridable_a =
IRuleQ.OverridableIsRestricting -> IRuleQ.OverridableRetRestricting
   -> Rule5StateOverridable_b
       IRuleQ.OverridableIsInOverride -> IRuleQ.OverridableRetNotInOverride
   -> Rule5StateOverridable

Rule5StateOverridable_b =
IRuleQ.OverridableIsInOverride -> IRuleQ.OverridableRetInOverride
   -> Rule5StateOverridable

Rule5StateOverridable_q =
IRuleQ.OverridableRetNotInOverride -> Rule5StateOverridable
       IRuleQ.OverridableRetNotRestricting -> Rule5StateOverridable

Rule5StateOverridable q_restricting=
IRuleQ.OverridableRetRestricting -> Rule5StateOverridable
       IRuleQ.OverridableRetNotRestricting -> Rule5StateOverridable

Rule5StateOverridable_Events = {
       IRuleQ.OverridableIsInOverride, IRuleQ.OverridableIsRestricting, IRuleQ.OverridableRetRestricting, IRuleQ.OverridableRetNotRestricting, IRuleQ.OverridableRetInOverride, IRuleQ.OverridableRetNotInOverride

assert Rule5StateOverridable\[T=ControllerDesign(true,false)\] \diff(Events,Rule5StateOverridable_Events)

Rule5StateOverridableRV =
ICtrOverridableBridge.ForceOutOverride -> Rule5StateOverridableRV_a
       IRuleQ.OverridableIsInOverride, IRuleQ.OverridableIsRestricting, IRuleQ.OverridableRetRestricting, IRuleQ.OverridableRetNotRestricting, IRuleQ.OverridableRetNotInOverride

Rule5StateOverridableRV_a =
IRuleQ.OverridableIsInOverride -> Rule5StateOverridableRV_q
       IRuleQ.OverridableIsRestricting, IRuleQ.OverridableRetRestricting, IRuleQ.OverridableRetNotRestricting, IRuleQ.OverridableRetNotInOverride

Rule5StateOverridableRV q =

Rule5StateOverridableRV_q =
IRuleQ.OverridableRetInOverride <-> Rule5StateOverridableRV

Rule5StateOverridableRV Events = {
ICtrOverridableBridge.ForceOutOverride , IRuleQ.OverridableIsInOverride,
IRuleQ.OverridableRetInOverride , IRuleQ.OverridableRetNotInOverride }
Rule6StateOverridableRV_q =
IRuleQ.OverridableRetInOverride -> Rule6StateOverridableRV
|~|
IRuleQ.OverridableRetNotInOverride -> Rule6StateOverridableRV

Rule6StateOverridableRV_q_SB =
IRuleQ.StepBackRetOverrideAllowed -> Rule6StateOverridableRV
|~|
IRuleQ.StepBackRetOverrideNotAllowed -> Rule6StateOverridableRV

Rule6StateOverridableRV_Events = {
IRuleQ.OverridableIsInOverride, IRuleQ.OverridableRetInOverride,
IRuleQ.OverridableRetNotInOverride,
IRuleQ.StepBackIsOverrideAllowed, IRuleQ.StepBackRetOverrideAllowed,
IRuleQ.StepBackNotOverrideAllowed,
AllQOutEvents.ICtrStepBackBridgeCB.UnSetOverrideAllowed
}

assert Rule6StateOverridableRV [T=ControllerDesign(true,false)](Events,Rule6StateOverridableRV_Events)

-----------------------------------------------------------------------------------------

Rule7StateOverridable =
IRuleQ.OverridableIsInOverride -> Rule7StateOverridable_q
IRuleQ.BlockingIsBlocking -> Rule7StateOverridable_q_SB
|\|
IRuleQ.OverridableRetNotInOverride -> Rule7StateOverridable

Rule7StateOverridable_q = IRuleQ.OverridableRetInOverride -> Rule7StateOverridable
|~|
IRuleQ.OverridableRetNotInOverride -> Rule7StateOverridable

Rule7StateOverridable_q_SB =
IRuleQ.BlockingRetBlocking -> Rule7StateOverridable
|~|
IRuleQ.BlockingRetNotBlocking -> Rule7StateOverridable

Rule7StateOverridable_Events = {
IRuleQ.BlockingIsBlocking, IRuleQ.OverridableIsInOverride,
IRuleQ.OverridableRetInOverride, IRuleQ.OverridableRetNotInOverride,
IRuleQ.BlockingIsBlocking, IRuleQ.BlockingRetBlocking, IRuleQ.BlockingRetNotBlocking
}

assert Rule7StateOverridable [T=ControllerDesign(true,false)](Events,Rule7StateOverridable_Events)

-----------------------------------------------------------------------------------------

Rule7StateOverridableRV_q =
IRuleQ.OverridableIsInOverride -> Rule7StateOverridableRV_q
|\|
IRuleQ.BlockingIsBlocking -> Rule7StateOverridableRV_q_SB
|\|
AllQOutEvents.ICtrBlockingBridgeCB.SetRestrictionRestricting

Rule7StateOverridableRV_a =
IRuleQ.BlockingIsBlocking -> IRuleQ.BlockingRetNotBlocking ->
IRuleQ.OverridableIsInOverride -> IRuleQ.OverridableRetInOverride ->
Rule7StateOverridableRV

Rule7StateOverridableRV_q_SB =
IRuleQ.BlockingRetBlocking -> Rule7StateOverridableRV
|~|
IRuleQ.BlockingRetNotBlocking -> Rule7StateOverridableRV

Rule7StateOverridableRV_Events = {
IRuleQ.OverridableIsInOverride, IRuleQ.OverridableRetInOverride,
IRuleQ.OverridableRetNotInOverride, IRuleQ.BlockingIsBlocking,
IRuleQ.BlockingRetBlocking, IRuleQ.BlockingRetNotBlocking
}

assert Rule7StateOverridableRV [T=ControllerDesign(true,false)](Events,Rule7StateOverridableRV_Events)

-----------------------------------------------------------------------------------------
IRuleQ.BlockingIsBlocking \rightarrow \text{IRuleQ.BlockingRetBlocking} \\
IRuleQ.OverridableIsInOverride \rightarrow \text{IRuleQ.OverridableRetNotInOverride} \\
\text{Rule7StateOverridableRV}$

\text{Rule7StateOverridableRV}_{q} = \\
\text{IRuleQ.OverridableRetInOverride} \rightarrow \text{Rule7StateOverridableRV} \\
\text{IRuleQ.OverridableRetNotInOverride} \rightarrow \text{Rule7StateOverridableRV}$

\text{Rule7StateOverridableRV}_{SB} = \\
\text{IRuleQ.BlockingRetBlocking} \rightarrow \text{Rule7StateOverridableRV} \\
\text{IRuleQ.BlockingRetNotBlocking} \rightarrow \text{Rule7StateOverridableRV}$

\text{Rule7StateOverridableRV}_\text{Events} = \\
\text{IRuleQ.OverridableIsInOverride}, \text{IRuleQ.OverridableRetInOverride}, \text{IRuleQ.OverridableRetNotInOverride}, \text{IRuleQ.BlockingIsBlocking}, \text{IRuleQ.BlockingRetBlocking}, \text{IRuleQ.BlockingRetNotBlocking}, \text{AllQOutEvents.ICtrBlockingBridgeCB.SetRestrictionRestricting} \\
\text{assert Rule7StateOverridableRV \{T=ControllerDesign(true, false)\}diff(Events, Rule7StateOverridableRV}_\text{Events)} \\
---------------------------------------------------------------

\text{Rule8StateOverridableX} = \\
\text{AllQOutEvents.ICtrOverridableBridgeCB.SetRestrictionRestricting} \rightarrow \text{Rule8StateOverridableX}_a \\
\text{IRuleQ.OverridableIsPresent} \rightarrow \text{Rule8StateOverridableX}_q \\
\text{Rule8StateOverridableX}_a = \text{IRuleQ.OverridableIsPresent} \rightarrow \text{IRuleQ.OverridableRetPresent} \rightarrow \text{Rule8StateOverridableX} \\
\text{Rule8StateOverridableX}_q = \text{IRuleQ.OverridableRetPresent} \rightarrow \text{Rule8StateOverridableX} \\
\text{IRuleQ.OverridableRetNotPresent} \rightarrow \text{Rule8StateOverridableX}$

\text{Rule8StateOverridableX}_\text{Events} = \\
\text{AllQOutEvents(ICtrOverridableBridgeCB.SetRestrictionRestricting, IRuleQ.OverridableIsPresent, IRuleQ.OverridableRetNotPresent, IRuleQ.OverridableRetPresent)} \\
\text{assert Rule8StateOverridableX \{T=ControllerDesign(true, false)\}diff(Events, Rule8StateOverridableX}_\text{Events)} \\
---------------------------------------------------------------

\text{Rule8StateOverridable} = \text{AllQOutEvents.IRuleCtr.OverridableSetRestricting} \rightarrow \text{Rule8StateOverridable}_a \\
\text{IRuleQ.OverridableIsPresent} \rightarrow \text{Rule8StateOverridable}_q \\
\text{Rule8StateOverridable}_a = \text{IRuleQ.OverridableIsPresent} \rightarrow \text{IRuleQ.OverridableRetPresent} \rightarrow \text{Rule8StateOverridable} \\
\text{Rule8StateOverridable}_q = \text{IRuleQ.OverridableRetPresent} \rightarrow \text{Rule8StateOverridable} \\
\text{IRuleQ.OverridableRetNotPresent} \rightarrow \text{Rule8StateOverridable}$

\text{Rule8StateOverridable}_\text{Events} = \\

74
IRuleQ.OverridableIsPresent, AllQOutEvents.IRuleCtr.OverridableSetRestricting, IRuleQ.OverridableRetNotPresent, IRuleQ.OverridableRetPresent

assert Rule8StateOverridable [T=ControllerDesign(true,false)] \diff(Events,Rule8StateOverridable_Events)

-----------------------------------------------------------------------------------------
Rule8StateOverridableRv =
AllQOutEvents.IRuleCtr.OverridableSetPresent \rightarrow Rule8StateOverridableRv_a
                            
IRuleQ.OverridableIsRestricting \rightarrow Rule8StateOverridableRv_q.Restricting
IRDQ.OverridableIsInOverride \rightarrow Rule8StateOverridableRv_q.Override

Rule8StateOverridableRv_a = IRuleQ.OverridableIsRestricting \rightarrow IRuleQ.OverridableRetNotRestricting \rightarrow IRuleQ.OverridableIsInOverride \rightarrow IRuleQ.OverridableRetNotInOverride \rightarrow Rule8StateOverridableRv

Rule8StateOverridableRv_q.Restricting =
IRDQ.OverridableRetRestricting \rightarrow Rule8StateOverridableRv
IRDQ.OverridableRetNotRestricting \rightarrow Rule8StateOverridableRv

Rule8StateOverridableRv_q.Override =
IRDQ.OverridableRetInOverride \rightarrow Rule8StateOverridableRv
IRDQ.OverridableRetNotInOverride \rightarrow Rule8StateOverridableRv

Rule8StateOverridableRv_Events =
IRDQ.OverridableIsInOverride, IRuleQ.OverridableRetInOverride,
IRDQ.OverridableRetNotInOverride, IRuleQ.OverridableRetNotRestricting, IRuleQ.OverridableRetRestricting,
IRuleQ.OverridableIsRestricting, AllQOutEvents.IRuleCtr.OverridableSetPresent

assert Rule8StateOverridableRv [T=ControllerDesign(true,false)] \diff(Events,Rule8StateOverridableRv_Events)

-----------------------------------------------------------------------------------------
Rule9StateOverridable =
IRuleQ.OverridableIsPresent \rightarrow Rule9StateOverridable_q
IRDQ.OverridableIsRestricting \rightarrow Rule9StateOverridable_q.Restricting
IRDQ.OverridableIsInOverride \rightarrow Rule9StateOverridable_q.Override

Rule9StateOverridable_a = IRuleQ.OverridableIsPresent \rightarrow IRuleQ.OverridableRetPresent \rightarrow IRuleQ.OverridableIsRestricting \rightarrow IRuleQ.OverridableRetRestricting \rightarrow IRuleQ.OverridableIsInOverride \rightarrow IRuleQ.OverridableRetInOverride \rightarrow Rule9StateOverridable

Rule9StateOverridable_q = IRuleQ.OverridableRetPresent \rightarrow Rule9StateOverridable
IRDQ.OverridableRetNotPresent \rightarrow Rule9StateOverridable

Rule9StateOverridable_q.Restricting =
IRDQ.OverridableRetRestricting \rightarrow Rule9StateOverridable
IRDQ.OverridableRetNotRestricting \rightarrow Rule9StateOverridable

Rule9StateOverridable_q.Override =
IRDQ.OverridableRetInOverride \rightarrow Rule9StateOverridable
IRDQ.OverridableRetNotInOverride \rightarrow Rule9StateOverridable

75
IRuleQ.OverridableRetInOverride -> Rule9StateOverridable

IRuleQ.OverridableRetNotInOverride -> Rule9StateOverridable

Rule9StateOverridable_Events = {RULE9EVENTS}

assert Rule9StateOverridable [T=ControllerDesign(true,false)] \(\text{diff} \) (Events,Rule9StateOverridable_Events)

-----------------------------------------------------------------------------------------

Rule10StateOverridable =

AllQOutEvents.ICtrStepBackBridgeCB.SetRestrictionRestricting -> Rule10StateOverridable_a

[]

IRuleQ.StepBackIsOverrideAllowed -> Rule10StateOverridable_q

[]

AllQOutEvents.IRuleCtr.StepBackSetStepBackPerformed -> Rule10StateOverridable

[]

AllQOutEvents.IRuleCtr.StepBackSetPresent -> Rule10StateOverridable

[]

AllQOutEvents.IRuleCtr.StepBackUnSetPresent -> Rule10StateOverridable

Rule10StateOverridable_a =

IRuleQ.StepBackIsOverrideAllowed -> IRuleQ.StepBackRetOverrideNotAllowed -> Rule10StateOverridable_b

[]

AllQOutEvents.IRuleCtr.StepBackSetPresent -> Rule10StateOverridable

[]

AllQOutEvents.IRuleCtr.StepBackUnSetPresent -> Rule10StateOverridable

Rule10StateOverridable_b =

AllQOutEvents.IRuleCtr.StepBackSetStepBackPerformed -> Rule10StateOverridable

[]

IRuleQ.StepBackIsOverrideAllowed -> IRuleQ.StepBackRetOverrideNotAllowed -> Rule10StateOverridable_b

[]

AllQOutEvents.IRuleCtr.StepBackSetPresent -> Rule10StateOverridable

[]

AllQOutEvents.IRuleCtr.StepBackUnSetPresent -> Rule10StateOverridable

Rule10StateOverridable_q =

IRuleQ.StepBackRetOverrideAllowed -> Rule10StateOverridable

[|]

IRuleQ.StepBackRetOverrideNotAllowed -> Rule10StateOverridable

Rule10StateOverridable_Events = {RULE10EVENTS}

assert Rule10StateOverridable [T=ControllerDesign(true,false)] \(\text{diff} \) (Events,Rule10StateOverridable_Events)

-----------------------------------------------------------------------------------------
Rule11StateOverridable = IRuleQ.OverridableIsPresent -> Rule11StateOverridable_q

IRuleQ.OverridableIsRestricting -> Rule11StateOverridable_q_Restricting

IRuleQ.OverridableIsInOverride -> Rule11StateOverridable_q_Override

AllOutEvents.IRuleCtr.OverridableUnSetPresent -> Rule11StateOverridable_a

Rule11StateOverridable_a = IRuleQ.OverridableIsPresent -> IRuleQ.OverridableResetNotPresent ->
IRuleQ.OverridableIsRestricting -> IRuleQ.OverridableResetNotRestricting ->
IRuleQ.OverridableIsInOverride -> IRuleQ.OverridableResetNotInOverride->

Rule11StateOverridable

Rule11StateOverridable_q = IRuleQ.OverridableResetPresent -> Rule11StateOverridable

IRuleQ.OverridableResetNotPresent -> Rule11StateOverridable

Rule11StateOverridable_q_Restricting =
IRuleQ.OverridableResetRestricting -> Rule11StateOverridable

IRuleQ.OverridableResetNotRestricting -> Rule11StateOverridable

Rule11StateOverridable_q_Override =
IRuleQ.OverridableResetInOverride -> Rule11StateOverridable

IRuleQ.OverridableResetNotInOverride -> Rule11StateOverridable

Rule11StateOverridable_Events =
AllOutEvents.IRuleCtr.OverridableUnSetPresent, IRuleQ.OverridableResetInOverride, IRuleQ.OverridableResetNotInOverride, IRuleQ.OverridableResetRestricting, IRuleQ.OverridableResetNotRestricting, IRuleQ.OverridableResetPresent, IRuleQ.OverridableResetNotPresent, IRuleQ.OverridableReset

assert Rule11StateOverridable [T= ControllerDesign(true,false)]\diff(Events,Rule11StateOverridable_Events)

-----------------------------------------------------------------------------------------

Rule12StateOverridable = ICtrOverridableBridge.SetOverrideRequestNotAllowed -> Rule12StateOverridable_a

IRuleQ.OverridableIsInOverride -> Rule12StateOverridable_q

Rule12StateOverridable_a = IRuleQ.OverridableIsInOverride ->
IRuleQ.OverridableResetNotInOverride -> Rule12StateOverridable

Rule12StateOverridable_q = IRuleQ.OverridableResetInOverride -> Rule12StateOverridable

IRuleQ.OverridableResetNotInOverride -> Rule12StateOverridable

Rule12StateOverridable_Events = { ICtrOverridableBridge.SetOverrideRequestNotAllowed,
IRuleQ.OverridableResetInOverride, IRuleQ.OverridableResetNotInOverride, IRuleQ.OverridableReset }

assert Rule12StateOverridable [T= ControllerDesign(true,false)]\diff(Events,Rule12StateOverridable_Events)

-----------------------------------------------------------------------------------------

Rule13StateOverridable = IRuleQ.OverridableIsRestricting ->

Rule13StateOverridable_q_Restricting

- Rule13StateOverridableIsRestricting
- Rule13StateOverridableIsInOverride

Rule13StateOverridable_a = IRuleQ.OverridableIsRestricting -> IRuleQ.OverridableIsInOverride

Rule13StateOverridable_q_Restricting =
- IRuleQ.OverridableIsRestricting
- IRuleQ.OverridableRetNotRestricting

Rule13StateOverridable_q_Override =
- IRuleQ.OverridableRetInOverride

Rule13StateOverridable_Events =
- AllQOutEvents.IRuleCtr.OverridableSetPresent
- IRuleQ.OverridableIsInOverride
- IRuleQ.OverridableRetInOverride
- IRuleQ.OverridableRetNotInOverride
- IRuleQ.OverridableRetNotRestricting
- IRuleQ.OverridableRetRestricting
- IRuleQ.OverridableIsRestricting

assert Rule13StateOverridable [T=ControllerDesign(true,false)] \diff(Events,Rule13StateOverridable_Events)

Rule14StateOverridable =
- AllQOutEvents.IRuleCtr.UnSetRequestActive

Rule14StateOverridable_a =
- IRuleQ.OverridableIsInOverride

Rule14StateOverridable_q =
- IRuleQ.OverridableRetInOverride

Rule14StateOverridable_Events =
- AllQOutEvents.IRuleCtr.UnSetRequestActive
- IRuleQ.OverridableIsInOverride
- IRuleQ.OverridableRetNotInOverride

assert Rule14StateOverridable [T=ControllerDesign(true,false)] \diff(Events,Rule14StateOverridable_Events)

--The individual checks for the StepBack restriction

Rule5StateStepBack =
- ICtrStepBackBridge.ForceInOverride

Rule5StateStepBack_a =
- ICtrStepBackBridge.ForceInOverride

Rule5StateStepBack_q_restricting =
- IRuleQ.StepBackIsRestricting

78
Rule5StateStepBack_a = IRuleQ.StepBackIsRestricting \rightarrow IRuleQ.StepBackNotRestricting 
\rightarrow Rule5StateStepBack_b
IRuleQ.StepBackIsRestricting \rightarrow IRuleQ.StepBackNotRestricting 
\rightarrow Rule5StateStepBack

Rule5StateStepBack_b = IRuleQ.StepBackIsInOverride \rightarrow IRuleQ.StepBackRetNotInOverride 
\rightarrow Rule5StateStepBack

Rule5StateStepBack_q = IRuleQ.StepBackRetInOverride \rightarrow Rule5StateStepBack 
\mid IRuleQ.StepBackRetNotInOverride \rightarrow Rule5StateStepBack

Rule5StateStepBack_q_restricting = IRuleQ.StepBackRetRestricting \rightarrow Rule5StateStepBack 
\mid IRuleQ.StepBackRetNotRestricting \rightarrow Rule5StateStepBack

Rule5StateStepBack_Events = 
ICtrStepBackBridge.ForceInOverride, IRuleQ.StepBackIsInOverride, 
IRuleQ.StepBackRetInOverride, IRuleQ.StepBackIsRestricting, 
IRuleQ.StepBackRetRestricting, IRuleQ.StepBackRetNotRestricting, IRuleQ.StepBackRetNotInOverride

assert Rule5StateStepBack[T=ControllerDesign(true,false)]\diff(Events,Rule5StateStepBack_Events)

-----------------------------------------------------------------------------------------------

Rule5StateStepBackRV = ICtrStepBackBridge.ForceOutOverride \rightarrow Rule5StateStepBackRV_a 
\mid IRuleQ.StepBackIsInOverride \rightarrow Rule5StateStepBackRV_q

Rule5StateStepBackRV_a = IRuleQ.StepBackIsInOverride \rightarrow IRuleQ.StepBackRetNotInOverride 
\rightarrow Rule5StateStepBackRV

Rule5StateStepBackRV_q = IRuleQ.StepBackRetInOverride \rightarrow Rule5StateStepBackRV 
\mid IRuleQ.StepBackRetNotInOverride \rightarrow Rule5StateStepBackRV

Rule5StateStepBackRV_Events = 
ICtrStepBackBridge.ForceOutOverride, IRuleQ.StepBackIsInOverride, 
IRuleQ.StepBackRetInOverride, IRuleQ.StepBackRetNotInOverride

assert Rule5StateStepBackRV[T=ControllerDesign(true,false)]\diff(Events,Rule5StateStepBackRV_Events)

-----------------------------------------------------------------------------------------------

Rule6StateStepBack = IRuleQ.StepBackIsInOverride \rightarrow Rule6StateStepBack_q 
\mid IRuleQ.StepBackIsOverrideAllowed \rightarrow Rule6StateStepBack_q_SB 
\mid AllOutEvents.ICtrStepBackBridgeCB.SetInOverride \rightarrow Rule6StateStepBack_a

79


assert Rule6StateStepBack [T=ControllerDesign(true,false)]\diff(Events,Rule6StateStepBack_Events)
-------------------------------------------------------------------------------------------------------------------------

Rule6StateStepBackRV_a = IRule1.StepBackIsInOverride -> Rule6StateStepBackRV_q

Rule6StateStepBackRV_q = IRule1.StepBackRetInOverride -> Rule6StateStepBackRV


assert Rule6StateStepBackRV [T=ControllerDesign(true,false)]\diff(Events,Rule6StateStepBackRV_Events)
-------------------------------------------------------------------------------------------------------------------------

Rule7StateStepBack = IRule1.StepBackIsInOverride -> Rule7StateStepBack_q

Rule7StateStepBack_q_SB = IRule1.BlockingIsBlocking -> Rule7StateStepBack_q

80
AllOutEvents.ICtrStepBackBridgeCB.SetInOverride -> Rule7StateStepBack_a

Rule7StateStepBack_a = IRuleI.BlockingIsBlocking -> IRuleI.BlockingNotBlocking ->
IRuleI.StepBackIsInOverride -> IRuleI.StepBackRetInOverride -> Rule7StateStepBack

Rule7StateStepBack_q = IRuleI.StepBackRetInOverride -> Rule7StateStepBack

|~|
IRuleI.BlockingRetNotBlocking -> Rule7StateStepBack

Rule7StateStepBack_q_SB=
IRuleI.BlockingNetBlocking -> Rule7StateStepBack

|~|
IRuleI.BlockingNetNotBlocking -> Rule7StateStepBack

Rule7StateStepBack_Events = {
IRuleI.BlockingIsBlocking, IRuleI.BlockingNotBlocking, IRuleI.StepBackIsInOverride, IRuleI.StepBackRetInOverride, IRuleI.BlockingNetBlocking, IRuleI.BlockingNetNotBlocking, AllQOutEvents.ICtrStepBackBridgeCB.SetInOverride }

assert Rule7StateStepBack [T=ControllerDesign(true,false)](Events,Rule7StateStepBack_Events)

-----------------------------------------------------------------------------------------

Rule7StateStepBackRV=
IRuleI.StepBackIsInOverride -> Rule7StateStepBackRV_q

|~|
IRuleI.BlockingIsBlocking -> Rule7StateStepBackRV_q_SB

|~|
AllOutEvents.ICtrBlockingBridgeCB.SetRestrictionRestricting -> Rule7StateStepBackRV_a

Rule7StateStepBackRV_a =
IRuleI.BlockingIsBlocking -> IRuleI.BlockingNetBlocking ->
IRuleI.StepBackIsInOverride -> IRuleI.StepBackRetNotInOverride ->
Rule7StateStepBackRV

Rule7StateStepBackRV_q =
IRuleI.StepBackRetInOverride -> Rule7StateStepBackRV

|~|
IRuleI.StepBackRetNotInOverride -> Rule7StateStepBackRV

Rule7StateStepBackRV_q_SB=
IRuleI.BlockingNetBlocking -> Rule7StateStepBackRV

|~|
IRuleI.BlockingNetNotBlocking -> Rule7StateStepBackRV

Rule7StateStepBackRV_Events = {
IRuleI.StepBackIsInOverride, IRuleI.StepBackRetInOverride, IRuleI.BlockingIsBlocking, IRuleI.BlockingNetBlocking, IRuleI.BlockingNetNotBlocking, AllQOutEvents.ICtrBlockingBridgeCB.SetRestrictionRestricting}

assert Rule7StateStepBackRV [T=ControllerDesign(true,false)](Events,Rule7StateStepBackRV_Events)

-----------------------------------------------------------------------------------------

Rule8StateStepBackX = AllOutEvents.ICtrStepBackBridgeCB.SetRestrictionRestricting ->
RuleStateStepBackX_a

IRuleQ.StepBackIsPresent -> RuleStateStepBackX_q

|~|
IRuleQ.StepBackRetNotPresent -> RuleStateStepBackX

RuleStateStepBackX_Events = {
  AllOutEvents.IRuleCtr.StepBackBridgeCB.SetRestrictionRestricting, IRuleQ.StepBackIsPresent, IRuleQ.StepBackRetNotPresent, IRuleQ.StepBackRetPresent }

assert RuleStateStepBackX [T=ControllerDesign(true,false)]\diff(Events,RuleStateStepBackX_Events)

-----------------------------------------------------------------------------------------

RuleStateStepBack =

AllOutEvents.IRuleCtr.StepBackSetRestricting -> RuleStateStepBack_a
|\|
IRuleQ.StepBackIsPresent -> RuleStateStepBack_q


RuleStateStepBack_q = IRuleQ.StepBackRetPresent -> RuleStateStepBack
|~|
IRuleQ.StepBackRetNotPresent -> RuleStateStepBack

RuleStateStepBack_Events = {
  IRuleQ.StepBackIsPresent, AllOutEvents.IRuleCtr.StepBackSetRestricting, IRuleQ.StepBackRetNotPresent, IRuleQ.StepBackRetPresent }

assert RuleStateStepBack [T=ControllerDesign(true,false)]\diff(Events,RuleStateStepBack_Events)

-----------------------------------------------------------------------------------------

RuleStateStepBackRv = AllOutEvents.IRuleCtr.StepBackSetPresent -> RuleStateStepBackRv_a
|\|
IRuleQ.StepBackIsRestricting -> RuleStateStepBackRv_q_Restricting


RuleStateStepBackRv_q_Restricting = IRuleQ.StepBackRetRestricting -> RuleStateStepBackRv

RuleStateStepBackRv_q_Override = IRuleQ.StepBackRetInOverride -> RuleStateStepBackRv

RuleStateStepBackRv_Events = {
  IRuleQ.StepBackIsInOverride, IRuleQ.StepBackRetInOverride,

82

assert Rule8StateStepBack_Rv [T=ControllerDesign(true,false)]diff(Events,Rule8StateStepBack_Rv_Events)

-----------------------------------------------------------------------------------------

Rule9StateStepBack = IRuleQ.StepBackIsPresent -> Rule9StateStepBack_q
IRuleQ.StepBackIsRestricting -> Rule9StateStepBack_q_Restricting
IRuleQ.StepBackIsInOverride -> Rule9StateStepBack_q_Override
AllQOutEvents.ICtrStepBackBridgeCB.SetInOverride -> Rule9StateStepBack_a

Rule9StateStepBack_a =
IRuleQ.StepBackIsPresent -> IRuleQ.StepBackIsRestricting
IRuleQ.StepBackIsRestricting -> IRuleQ.StepBackIsInOverride
IRuleQ.StepBackIsInOverride -> Rule9StateStepBack q
AllQOutEvents.ICtrStepBackBridgeCB.SetInOverride -> Rule9StateStepBack_a

Rule9StateStepBack_q = IRuleQ.StepBackIsPresent -> Rule9StateStepBack
IRuleQ.StepBackIsRestricting -> Rule9StateStepBack
IRuleQ.StepBackIsInOverride -> Rule9StateStepBack

Rule9StateStepBack_q_Restricting = IRuleQ.StepBackRetRestricting -> Rule9StateStepBack
IRuleQ.StepBackRetNotRestricting -> Rule9StateStepBack

Rule9StateStepBack_q_Override = IRuleQ.StepBackRetInOverride -> Rule9StateStepBack
IRuleQ.StepBackRetNotInOverride -> Rule9StateStepBack


assert Rule9StateStepBack [T=ControllerDesign(true,false)]diff(Events,Rule9StateStepBack_Events)

-----------------------------------------------------------------------------------------

Rule10StateStepBack = AllQOutEvents.ICtrStepBackBridgeCB.SetRestrictionRestricting

Rule10StateStepBack_a =
IRuleQ.StepBackIsOverrideAllowed -> Rule10StateStepBack_b
AllQOutEvents.IRuleCtr.StepBackIsStepBackPerformed -> Rule10StateStepBack_a
AllQOutEvents.IRuleCtr.StepBackSetPresent -> Rule10StateStepBack_a
AllQOutEvents.IRuleCtr.StepBackUnSetPresent -> Rule10StateStepBack_a

Rule10StateStepBack_a =
IRuleQ.StepBackIsOverrideAllowed -> Rule10StateStepBack_a
IRuleQ.StepBackSetStepBackPerformed -> Rule10StateStepBack_a
AllQOutEvents.IRuleCtr.StepBackSetPresent -> Rule10StateStepBack_a
AllQOutEvents.IRuleCtr.StepBackUnSetPresent -> Rule10StateStepBack_a

Rule10StateStepBack_b =
assert Rule11StateStepBack [T=
ControllerDesign(true,false)]\text{diff(Events,Rule11StateStepBack_Events)}

-----------------------------------------------------------------------------------------

Rule12StateStepBack =
ICtrStepBackBridge.SetOverrideRequestNotAllowed \rightarrow Rule12StateStepBack.a
\]
IRuleQ.StepBackIsInOverride \rightarrow Rule12StateStepBack_q

Rule12StateStepBack.a = IRuleQ.StepBackIsInOverride \rightarrow IRuleQ.StepBackNetInOverride \rightarrow
Rule12StateStepBack

Rule12StateStepBack_q = IRuleQ.StepBackRetInOverride \rightarrow Rule12StateStepBack
|~|
IRuleQ.StepBackRetNotInOverride \rightarrow Rule12StateStepBack

Rule12StateStepBack_Events =
{ ICtrStepBackBridge.SetOverrideRequestNotAllowed,IRuleQ.StepBackRetInOverride ,
IRuleQ.StepBackRetNotInOverride,IRuleQ.StepBackIsInOverride }

assert Rule12StateStepBack [T=
ControllerDesign(true,false)]\text{diff(Events,Rule12StateStepBack_Events)}

-----------------------------------------------------------------------------------------

Rule13StateStepBack =
IRuleQ.StepBackIsRestricting \rightarrow Rule13StateStepBack_q_Restricting\[
IRuleQ.StepBackIsInOverride \rightarrow Rule13StateStepBack_q_Override\[
AllQOutEvents.IRuleCtr.StepBackSetPresent \rightarrow Rule13StateStepBack.a

Rule13StateStepBack.a = IRuleQ.StepBackIsRestricting \rightarrow IRuleQ.StepBackNetNotRestricting \rightarrow
IRuleQ.StepBackIsInOverride \rightarrow IRuleQ.StepBackNetNotInOverride \rightarrow
Rule13StateStepBack

Rule13StateStepBack_q_Restricting =
IRuleQ.StepBackRetRestricting \rightarrow Rule13StateStepBack
|~|
IRuleQ.StepBackRetNotRestricting \rightarrow Rule13StateStepBack

Rule13StateStepBack_q_Override =
IRuleQ.StepBackRetInOverride \rightarrow Rule13StateStepBack
|~|
IRuleQ.StepBackRetNotInOverride \rightarrow Rule13StateStepBack

Rule13StateStepBack_Events =
{ AllQOutEvents.IRuleCtr.StepBackSetPresent, IRuleQ.StepBackIsInOverride, IRuleQ.StepBackNetInOverride ,
IRuleQ.StepBackNetNotRestricting, IRuleQ.StepBackRetRestricting, IRuleQ.StepBackIsRestricting}

assert Rule13StateStepBack [T=
ControllerDesign(true,false)]\text{diff(Events,Rule13StateStepBack_Events)}

-----------------------------------------------------------------------------------------

Rule14StateStepBack =
AllQOutEvents.IRuleCtr.UnSetRequestActive \rightarrow Rule14StateStepBack.a
\]
IRuleQ.StepBackIsInOverride \rightarrow Rule14StateStepBack_q

Rule14StateStepBack.a = IRuleQ.StepBackIsInOverride \rightarrow IRuleQ.StepBackNetNotInOverride \rightarrow
Rule14StateStepBack

85
Rule14StateStepBack_q =
  IRuleQ.StepBackInOverride -> Rule14StateStepBack
  | ~
  IRuleQ.StepBackNotInOverride -> Rule14StateStepBack

Rule14StateStepBack_Events = {
  AllOutEvents.IRuleCtr.UnSetRequestActive, IRuleQ.StepBackInOverride , 
  IRuleQ.StepBackNotInOverride, IRuleQ.StepBackIsInOverride }

assert Rule14StateStepBack [T= ControllerDesign(true, false)] diff(Events, Rule14StateStepBack_Events)

-----------------------------------------------------------------------------------------
Bibliography


