Eindhoven University of Technology

MASTER

Formal verification of control software in X-ray systems

Redegeld, J.

Award date:
2016

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.
Formal verification of control software in X-Ray systems

Master’s Thesis
by
Jeroen Redegeld
j.redegeld@student.tue.nl

Committee:

University supervisor:                        Company supervisor:
prof.dr.ir. J.F. Groote                        dr.ir. P.T. Wolkotte
j.f.groote@tue.nl                             pascal.wolkotte@philips.com

Company supervisor:
drs. L.H.F.P.M. Lakeman
lake.lakeman@philips.com

Philips Healthcare, Best, The Netherlands
and
Eindhoven University of Technology, Eindhoven, The Netherlands
January 27, 2014
Version 1.10
## Contents

1 Introduction .................................................. 1
   1.1 Context, X-Ray systems by Philips Healthcare .......... 1
   1.2 Formal methods ........................................... 2
   1.3 Research goals and questions ............................. 3
   1.4 Motivation to use mCRL2 ................................ 4
   1.5 Overview of the remaining chapters ..................... 4

2 The mCRL2 language ........................................... 5
   2.1 Modeling steps ............................................. 5
   2.2 Modeling system behavior ................................ 6
      2.2.1 Actions ............................................... 6
      2.2.2 Data Types ........................................... 7
      2.2.3 Visualizing system behavior ......................... 12
      2.2.4 Behavioral equivalence ................................ 13
      2.2.5 Sequential modeling constructions ................. 14
      2.2.6 Parallel modeling constructions .................... 17
   2.3 Specifying requirements .................................. 18

3 Start, stop and recover use-case ............................ 20
   3.1 Research goals and approach ............................. 20
   3.2 Context .................................................... 21
   3.3 Interface definitions ..................................... 25
      3.3.1 Process side messages ............................... 25
      3.3.2 Controlling side messages ........................... 26
      3.3.3 Timeout event side message ......................... 29
   3.4 Required behavior ........................................ 29
   3.5 Requirements .............................................. 32
   3.6 Modeling ................................................... 34
      3.6.1 mCRL2 model .......................................... 34
      3.6.2 Modal formulas ....................................... 43
      3.6.3 Used software and hardware ......................... 53
   3.7 Conclusions and recommendations ....................... 55
      3.7.1 Use-case ............................................... 55
CONTENTS

3.7.2 Model ................................................. 55
3.7.3 Modal formulas ................................. 57
3.7.4 Recommendations and future work ............... 58

4 X-Ray controller use-case .............................. 63
4.1 Research goal ......................................... 63
4.2 Context ................................................. 64
4.3 Modeling ............................................... 64
4.3.1 mCRL2 model ...................................... 64
4.3.2 Modal formulas .................................... 72
4.4 Conclusions and recommendations ................. 81
4.4.1 Use-case ............................................ 81
4.4.2 Model ............................................... 82
4.4.3 Modal formulas .................................... 84
4.4.4 Recommendations and future work ............... 86

5 Conclusions .............................................. 87

A SSR component documentation .......................... 93

B SSR mCRL2 model ...................................... 94

C SSR modal formulas .................................... 95
C.1 Simple check, response to a Start message .......... 96
C.2 Complex check, response to a Start message .......... 96
C.3 Simple check, response to a Stop message ........... 97
C.4 Complex check, response to a Stop message .......... 97
C.5 Simple check, response to a Recover message ....... 98
C.6 Complex check, response to a Recover message ....... 98
C.7 General check, sending a Crash is allowed ........... 99
C.8 General check, sending a Crash is not allowed ....... 99
C.9 Simple check, receiving a Stop is possible ........... 100
C.10 Complex check, receiving a Stop is possible ......... 100
C.11 General check, occurrence of a deadlock .......... 101

D X-Ray controller component .......................... 102

List of Figures ............................................ 103
List of Tables ............................................. 104
List of Listings ......................................... 105
Bibliography ............................................. 107
CHAPTER

Introduction

This report addresses the results regarding research about the applicability of formal methods in industrial use-cases. An X-Ray system that is developed by Philips Healthcare is used to see if and how the development of an existing system by using formal methods can be enhanced. Section 1.1 introduces Philips Healthcare and gives some background information related to the X-Ray system developed by Philips Healthcare. Research goals and questions are addressed in Section 1.3. Motivations to use mCRL2 as the modeling language and tool are addressed in Section 1.4 and Section 1.5 gives a brief overview of the remaining chapters.

1.1 Context, X-Ray systems by Philips Healthcare

Philips Healthcare delivers advanced solutions for healthcare professionals. The goal is to simplify healthcare by focusing on the people in the care cycle, i.e., patients and care providers. Through combining human insights and clinical expertise, Philips Healthcare aims to improve patient outcomes while lowering the burden on the healthcare system.

One of the products developed by Philips Healthcare is an X-Ray system. This X-Ray system contains both hard- and software. It is generally known that the quality of software is directly related to the quality of a given product, so the quality of the software implemented in the X-Ray system is of great importance with regards to the total product quality of the X-Ray system.

Interventional X-Ray (iXR) is responsible for the development of X-Ray systems used for diagnosis and during interventional treatment of heart and vascular diseases, e.g., dottering, stent placement and coiling. The development of this type of systems is performed in multidisciplinary teams where hardware, mechanics and software come together. Philips Healthcare is the market leader in this field. An example of a Philips X-Ray system is shown
in Figure 1.1. Figure 1.2 shows an image generated by an X-Ray system that shows veins filled with a contrast fluid.

In Cardiovascular minimal invasive interventions, physicians operate the X-Ray system according to a certain procedure. The interaction between the system and the physician during the procedure results in specific state-transitions of the system.

Behavior of a system is the relation between the performed interaction of the system with its environment and the internal status of the system. The internal status that is related to a series of interactions is called a state and the interactions with the systems external environment are called actions. An action is enabled in a given state when this action can be executed in this state. Certain state-transitions are desired, whereas others shall be prevented to guard the system from undesired behavior. This state model is implemented by software. The quality of this software is thereby directly contributing to the overall quality of the product.

1.2 Formal methods

The purpose of applying formal methods is to develop robust software by finding defects and avoiding the introduction of defects in the early design phase. Hence, minimizing the number of defects present in the developed software thereby increasing the overall quality of the developed product [3].

The typical approach is to create an abstract representation of the considered (sub)system. Next, properties are verified in a systematic way using various techniques that are based on mathematics. By systematically verifying properties one can prove that certain properties hold for a given model and use this defect free model in further development. It is possible to cre-
ate a model in the early development phase with the aim to realize a solid foundation out of a system description and a set of requirements, but creating a model from an existing part is also possible. Case studies presented in [3] show that by using formal methods in the early design phase will increase the quality of the software that is being developed. Furthermore, [3] shows that no extra development time is needed and in some cases a speedup in the development time is observed.

Different formal languages have been developed that focus on a specific system type, e.g., event driven or time driven systems. Most formal languages are supported by a tool set that can be used to model and verify requirements. Various tools are freely available, but there are also tools that are commercially available.

1.3 Research goals and questions

The aim of this research is to investigate if and how formal methods can be applied in industrial applications. This is done by investigating the usability of a tool set that supports a formal method using real industrial applications.

The considered software for this research is a subset of the total X-Ray system that focuses on the general control of the followed interventional procedures that are executed by the X-Ray system. The software used in the use-cases is developed without the usage of formal methods. A total of 2 different use-cases have been modeled by using the formal specification language mCRL2 and its related tool set. Models are kept to a minimum and are extended using an iterative approach when more details are needed.

Interesting questions that are addressed are:

- Is it possible to use mCRL2 in the selected use-case and modeling approach and is this useful?
- What is the effort that was needed in order to model a given software part?
- Can existing source code be translated into a model that is suitable for validation and/or verification?
- Are there opportunities to fully automate the translation process from source code to a model?
- Does the selected use-case and modeling approach contribute to the quality of the developed software?
- What is the scalability of the model that is being used for the selected use-case?
- Can a model be used for other purposes than model checking?
Answers to the questions above are given in Chapter 5.

1.4 Motivation to use mCRL2

mCRL2 is the formal modeling tool that is selected for both use-cases. Motivations for this choice are given in this section and are based on [1, 2].

mCRL2 is a powerful tool set that allows users to model system behavior and prove that a specified properties hold for the modeled system. Reasons to select mCRL2 are:

- mCRL2 can be used to model larger systems composed of multiple components placed in parallel. This model can be used to successfully verify system requirements.

- It is possible to formulate tailor made requirements to verify properties that are specific for the modeled system. One can think of orders between events, liveness properties and safety properties.

- mCRL2 contains convenient ways to define data types and standard data types (Booleans, positive natural numbers, natural numbers, integer numbers and real numbers) are predefined including common auxiliary functions on those data types. Verification takes data parameters into account, so correct behavior that is based on parameter values can be verified.

- mCRL2 is developed at Eindhoven University of Technology, so this tool is well known within the Formal System Analysis (FSA) group and direct communication with mCRL2 experts is possible.

1.5 Overview of the remaining chapters

In this section a brief overview of the remaining chapters is given. mCRL2 modeling concepts and constructions that are needed to specify system behavior and requirements are addressed in Section 2. The first use-case considers the start, stop and recover (SSR) component as discussed in Chapter 3. Next, the second use-case is addressed in Chapter 4 and this use-case considers the X-Ray controller. Finally, results and answers to the research questions are presented in Chapter 5.
The mCRL2 language

This chapter introduces the mCRL2 language constructions that are used in the models related to the two selected use cases. Information that is presented in this chapter is based on [1, 2]. Section 2.1 contains a high level overview of the steps required to model and verify a component. Next, modeling constructions are presented in Section 2.2. Verifying requirements is addressed in Section 2.3.

2.1 Modeling steps

This section explains the steps that are required to model and verify system behavior without going into details. Generally one can identify the following steps in the modeling process:

- Create an mCRL2 model. This is an abstract representation of the considered component.
- Define system properties. A property defines parts of required system behavior and by using multiple properties one can define required behavior.
- Verify if the model meets all specified requirements.

The purpose of an mCRL2 model is to describe relevant internal functionality of the considered component. This mCRL2 model is a process specification that uses extended process algebra to describe system behavior. This extended process algebra allows the usage of data parameters in the process specification. Typically the value of a data parameter influences the behavior of the modeled component and by including data parameters in the model one can verify behavior upon parameter values.
CHAPTER 2. THE MCRL2 LANGUAGE

Required behavior of the modeled component is typically divided over multiple requirements. Requirements are specified by using modal formulas that are based on the \( \mu \)-calculus. A modal formula specifies desired or unwanted relations between observable interactions of the modeled component.

Properties are mathematically verified by creating a set of equations out of the model and the modal formula. The solution of this set of equations indicates if the mCRL2 model meets the specified requirement. A typical approach is to check requirements during the realization of the model and refine both the model and the requirements. This iterative approach continues until all important properties are described by means of modal formulas and the used model satisfies all requirements.

2.2 Modeling system behavior

The ability to observe behavior is addressed in Section 2.2.1 and the definition of the data types that might be used in this process are addressed in Section 2.2.2. A convenient way to graphically depict system behavior is presented in Section 2.2.3 and how to determine if two specifications are equivalent is addressed in Section 2.2.4. Sections 2.2.5 and 2.2.6 contain modeling constructions related to sequential behavior and parallel behavior respectively.

2.2.1 Actions

Interaction of a component is communication that is observable on the outside, or interface, of a component and defines system behavior. Interaction of computer systems is typically some form of discrete communication like receiving information from sensors, sending information to actuators or even transmission of data packages. Actions, or messages, are used to model interaction of components. An action is an observable atomic event that does not have a duration. Actions cannot overlap with other actions over time, with the exception of the multi-action that is addressed further on in this section.

An action is identified by a name. Action names are typically related to the purpose or functionality that is triggered by an action, e.g., \textit{send}, \textit{start} or \textit{open}. Optionally, actions can take one or more data parameters of a specified data type, e.g., \textit{send}(45), \textit{start(engine1, engine4)} or \textit{open(allDoors)}.

Actions are defined in the mCRL2 model optionally followed by one or more data types separated by a hash (#). Data types used in the definition indicate the type of parameter that is required when using the action. Means to define data types and standard built-in data types are addressed in Section 2.2.2. The definition of one or more actions is preceded by the keyword \textbf{act}. Actions are separated by a semicolon (;). As a shorthand notation, multiple actions with equal data parameters can be separated by a comma (,). An example of the definition of multiple actions is shown Listing 2.1.
2.2. MODELING SYSTEM BEHAVIOR

<table>
<thead>
<tr>
<th>DESCRIPTION</th>
<th>RANGE</th>
<th>SYMBOL</th>
<th>mCRL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean value</td>
<td>{true, false}</td>
<td>(\mathbb{B})</td>
<td>(\text{Bool})</td>
</tr>
<tr>
<td>Positive natural numbers</td>
<td>{1, 2, \ldots, \infty}</td>
<td>(\mathbb{N}^+)</td>
<td>(\text{Pos})</td>
</tr>
<tr>
<td>Natural numbers</td>
<td>{0, 1, \ldots, \infty}</td>
<td>(\mathbb{N})</td>
<td>(\text{Nat})</td>
</tr>
<tr>
<td>Integer numbers</td>
<td>{−\infty, \ldots, −1, 0, 1, \ldots, \infty}</td>
<td>(\mathbb{Z})</td>
<td>(\text{Int})</td>
</tr>
<tr>
<td>Real numbers</td>
<td>{−\infty, \ldots, \infty}</td>
<td>(\mathbb{R})</td>
<td>(\text{Real})</td>
</tr>
</tbody>
</table>

Table 2.1: Predefined data types that are built-in in the mCRL2 tool set.

Listing 2.1: Defining actions

```plaintext
act send;
start;
%actions require a natural number as argument
openDoor, closeDoor : Pos;
%action requires a natural number and a boolean as argument
transmitNumber : Nat # Bool;
```

It might be the case that two or more actions occur at exactly the same time. This phenomenon is called a multi-action and denoted by separating actions that occur at the same time by the pipe symbol (|), e.g., the occurrence of both actions `callFireBrigade` and `activateSprinklers` is denoted as `callFireBrigade | activateSprinklers`. Multi-actions are typically used when modeling parallel behavior.

2.2.2 Data Types

Data types have to be defined first before they can be used in the modeling process. A data type is defined by a data type specification, that is an equational specification that defines a non-empty, possible infinite set. Furthermore, auxiliary functions like ‘smaller than’ can be defined. Common data types are predefined and a language to define new data types is present in the mCRL2 tool set.

Predefined data types

An overview of standard built-in data types is shown in Table 2.1. The mCRL2 column shows the keyword that is used in both the mCRL2 specification and the modal formulas when this data type is used.

Positive numbers, natural numbers, integer numbers and real numbers are defined such that they contain infinite elements hence mimic the mathematical definition as closely as possible (Table 2.1). Various auxiliary func-
### Chapter 2. The MCRL2 Language

#### Boolean
- `if` (if)
- equality (==)
- logical negation (!)
- negation (−)

#### Positive numbers
- inequality (!=)
- logical and (&&)
- logic implication (=>)

#### Natural numbers
- comparison (<, <=, >=, and >)
- logical or (||)

#### Integers
- maximum (max)
- addition (+)
- subtraction (−)
- multiplication (*)
- successor (succ)
- predecessor (pred)
- integer division (div)
- modulo (mod)
- exponentiation (exp)
- absolute value (abs)
- floor (floor)
- ceiling (ceil)
- rounding (round)

<table>
<thead>
<tr>
<th>Boolean</th>
<th>Positive numbers</th>
<th>Natural numbers</th>
<th>Integers</th>
<th>Reals</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 2.2: Predefined auxiliary functions on built-in data types.
tions on built-in data types are defined as shown in Table 2.2. Casting data types can be achieved by using the $A2B$ construction where $A, B \in \{\text{Pos, Nat, Int, Real}\}$.

User defined data types

Data types are defined by using a basic equational data type specification mechanism in which the different parts are indicated by keywords. A data type is also called a sort and a data type specification starts with the keyword sort followed by the name of the data type. Next, constructor functions are used to define exactly all elements that are part of a data type. Constructor functions are prefixed with the keyword cons and may use recursion hence can define an infinite number of elements.

The example that is shown in Listing 2.2 defines a data type TwoTires (line 2) that contains the elements $\{\text{front, rear}\}$ (line 4). The second data type that is defined in Listing 2.2 is the NaturalNumbers data type. Elements are defined by means of two functions. First, the single element zero is defined on line 10. Next, a function is defined that takes an element of the NaturalNumbers and provides an element that is in the set of elements belonging to the NaturalNumbers data type (line 11).

Listing 2.2: Definition of data types and all there elements

```plaintext
%Define data type TwoTires
sort TwoTires;
%Define 4 elements of the data type TwoTires
cons front, rear : TwoTires;

%Define data type NaturalNumbers
sort NaturalNumbers;
%Define element zero and an update function that both define all
%elements of the data type NaturalNumbers
cons zero : NaturalNumbers;
  succor : NaturalNumbers -> NaturalNumbers;

%Define data type AlarmActive that is of the built-in data type
%booleans
sort AlarmActive = Bool;
%Define actions that require the AlarmActive data type as parameter
act frontDoor, backDoor : AlarmActive;
  alarmOfDoorActive : Nat # AlarmActive;
```

Note that no relations between elements of both data types TwoTires and NaturalNumbers are established yet. In case of the TwoTires data type it might be the case that both elements front and rear are equal. Thus, this data type contains one or two distinct elements. The mapping between elements of the function that is defined on line 11 (Listing 2.2) is not defined. Therefore, the data type NaturalNumbers contains at least one element when
all elements are equal and may contain an infinite number of elements when
all elements are distinct.

It is also possible to define a data type that uses an existing data type. Listing 2.2 shows the definition of the AlarmsActive data type (line 15) that is based on the built-in Boolean data type. Lines 17 and 18 of Listing 2.2 shows that actions that require own defined data types can be defined in the same way as built-in data types. Line 18 also shows that built-in data types can be used in combination with own defined data types.

Auxiliary functions, or mappings, are used to define relations between ele-
ments. Relations can be used to ensure that elements are distinct. Mappings
can be used in both the model and the modal formulas. Predefined auxiliary
functions on built-in data types (Table 2.2) are defined by mappings. A map-
ing is prefixed with the keyword map followed by the name of the mapping
and the required arguments.

A mapping only defines a function and required arguments. Equations
are used to define the relation between elements. Equations are prefixed by
the keyword eqn followed by the relation between the elements. The only
assumption that one can use when defining equations is that the Boolean
data type is defined. Therefore, the Boolean elements true and false exist and
are different. Furthermore, one can use defined Boolean auxiliary functions
(Table 2.2) in equations.

Defining mappings and equations for built-in data types is also possible. One can define specific auxiliary functions that are needed in the modeling
and verification process of the selected system.

Listing 2.3: Definition of the TwoTires data type and an auxiliary function

```plaintext
%Define data type TwoTires
sort TwoTires;
%Define 4 elements of the data type TwoTires
cons front, rear : TwoTires;
%define auxiliary functions
map isFrontTire : TwoTires -> Bool;
%define mapping of elements of the auxiliary functions
eqn isFrontTire(front) = true;
    isFrontTire(rear) = false;
```

Listing 2.3 shows the TwoTires data type and an auxiliary function is-
FrontTire (line 6) that requires an element of the TwoTires data type and
returns a Boolean. Relations between the elements of the isFrontTire are
defined by two equations (lines 8 and 9).

Variables can be used in the definition of equations to define an equation
that holds for multiple values. Variable declarations are preceded by the key-
word var. A variable declaration starts with the name of the variable followed
by the data type of the declared variable. Listing 2.4 shows an extension of
the TwoTires data type by defining the mapping isRearTire (line 7). The
variable $t$ (line 9) is used in the equation (line 13) that defines the relation between the elements using the previously defined $isFrontTire$ equation.

Listing 2.4: Definition of the TwoTires data type and an auxiliary function.

```plaintext
%Define data type TwoTires
sort TwoTires;
%Define 4 elements of the data type TwoTires
cons front, rear : TwoTires;
%define auxiliary functions
map isFrontTire : TwoTires -> Bool;
isRearTire : TwoTires -> Bool;
%declare a variable that can be used in equations
var t : TwoTires;
%define mapping of elements of the auxiliary functions
eqn isFrontTire(front) = true;
isFrontTire(rear) = false;
isRearTire(t) = !isFrontTire(t);
```

It is not necessary needed that a function, or constructor, is fully defined. It is simply not known what the output value is when an undefined input is used, but the data type that is returned is specified.

A special construction is available to define structured data types. A structured data type contains a finite number explicitly defined elements and is preceded by the keyword `struct` followed by the elements of the structured data type. The `TwoTires` data type (Listing 2.4) can be defined using a structure as shown in Listing 2.5 (line 3). No equations and mappings are needed to ensure that the two defined elements are distinct since all elements of a structured data type are distinct.

Listing 2.5: Defining structured data types.

```plaintext
%Declare the structured data type TwoTiresNew that contains 2 distinct elements
sort TwoTiresNew = struct frontNew | rearNew;
%Declare the structured data type FourTires that contains 4 distinct elements and include recognizer functions
sort FourTires = struct frontLeft?isFrontLeft | frontRight?isFrontRight |
rearLeft?isRearLeft | rearRight?isRearRight ;
```

The second data type that is defined in Listing 2.5 (lines 7-10) defines the data type `FourTires` that contains four distinct elements. Furthermore, an optional recognizer function such as `isFrontLeft` is defined. A recognizer function takes a single element of the defined data type and maps this to the Boolean data type. A recognizer function returns true iff they are applied to the constructor to which they belong, e.g., $isFrontLeft(frontLeft)=true$ and $isFrontLeft(ft)=false$ for $ft \in \{frontRight, rearLeft, rearRight\}$.

An overview of the different keywords presented in this section related to the definition of data types is given in Table 2.3.
### 2.2.3 Visualizing system behavior

Observable system behavior is often graphically presented. In most cases required behavior that is graphically presented is easier to understand when compared to a textual description. A *Labelled Transition System* (LTS) is a frequently used graphical representation that shows system behavior.

An LTS is a directed labeled graph with a set of states, i.e., graph vertices, and a set of transitions, i.e., graph edges. It contains exactly one initial state that marks the starting point of the LTS and there are zero or more terminating states. A transition between two states denotes the ability to execute an action. The initial state is denoted by a by a small incoming arrow and terminating states are denoted by an extra circle that is placed inside the state.

Figures 2.1 shows two examples of LTSs. Both LTSs can execute an *a* action from the initial state. The left LTS (Figure 2.1a) reaches a *deadlock state* after executing the *a* action meaning that it cannot do any actions afterwards. A deadlock state cannot terminate and has no outgoing transitions. The right LTS (Figure 2.1b) can terminate after performing the *a* action.
2.2. Modeling System Behavior

(a) A deadlock state is reached after executing the $a$ action.
(b) A terminating state is reached after executing the $a$ action.

Figure 2.1: Two simple LTSs that both can execute an atomic action ($a$) from the initial state.

2.2.4 Behavioral equivalence

Different equivalences exist, but only strong bisimulation and branching bisimulation equivalences are addressed. Strong bisimulation equivalence is also called bisimulation equivalence or just bisimulation.

Two strongly bisimilar equivalent processes cannot be distinguished by any realistic form of behavioral observation and is considered to be the finest known equivalence relation [2].

The idea behind strong bisimulation equivalence is to establish an equivalence relation between states. Two states are related if all enabled actions (outgoing transitions in the LTS) of a state are enabled in the other state as well. This means that when considering states $s$ and $t$ one has to verify that all actions enabled in $s$ are also enabled in $t$ and that all actions enabled in $t$ are also enabled in $s$. The second action simulates the first one. Furthermore, the reached states after simulating an action need to be related as well. Two LTSs are bisimilar when there initial states are bisimilar.

Abstraction is used to simplify systems and establish a better understanding of a system. The strategy used is to indicate that an action is non-observable or internal. A non-observable internal action is labeled with the reserved symbol $\tau$ and cannot be observed directly. Figure 2.2a shows an LTS that contains a non-observable $\tau$ transition hence it is not possible to make a distinction between the two LTSs presented in Figure 2.2. A non-observable $\tau$ transition is called inert when it cannot be observed from the outside.

(a) LTS with a non-observable or inert $\tau$ transition.
(b) LTS without a $\tau$ transition.

Figure 2.2: Two branching bisimilar LTSs.

Figure 2.3a shows an LTS that contains a non-inert $\tau$ transition. This LTS can perform an $a$ action using the $\tau$ transition or a $b$ action. After silently executing the $\tau$ action one loses the ability to execute the $b$ action and this is externally observable. This means that one can distinguish behavior of the LTSs presented in Figure 2.3.

An equivalence that incorporates $\tau$ transitions is branching bisimulation
equivalence. States are simulated by a sequence of zero or more internal \( \tau \) transitions instead of a single action. This paradigm is depicted in Figure 2.4. The dashed line annotated with an \( R \) indicates a branching bisimulation relation.

![Figure 2.3: Two non-branching bisimilar LTSs.](image)

### 2.2.5 Sequential modeling constructions

This section addresses different modeling constructions that are part of the mCRL2 process algebra used to model behavior. An important aspect in the discussion of modeling constructions is that an action is a process that terminates afterwards.

Processes can be executed one after another by using the **sequential composition**, notated by \( \cdot \) between processes. The second process can only start when the first process terminates. An LTS of the formula \( a \cdot b \) is shown in Figure 2.5.

The alternative composition is used when a choice between processes can be made, denoted by \( + \) between processes. Only one of the processes placed in the alternative composition will be executed. Figure 2.6 shows two LTSs of process specifications that both contain an alternative composition.
2.2. MODELING SYSTEM BEHAVIOR

The actual choice between the processes is made by the first action in every process. This should be taken into account when modeling an arbitrary choice between parameters also called *non determinism*. The way to model a random choice between multiple actions is to precede all actions with the same reserved dummy action. This dummy action is a non-visible action and can be abstracted away by replacing this action by a $\tau$ action. Because all first actions are the same, a random choice is made between the non-deterministic dummy actions binding the value of the next action. It is not possible to enforce a particular value in the case of communication because the choice is made on forehand when the dummy action is executed.

Figure 2.6: Difference between a deterministic and non-deterministic LTS.

Figure 2.6a shows an LTS of the specification $a + b$ that is deterministic. Every state contains at most one unique outgoing transition, i.e., the initial state contains one outgoing $a$ action and one outgoing $b$ action. The first action can be used to enforce a choice since both actions are different. Figure 2.6b shows an LTS of the specification $(i \cdot a) + (i \cdot b)$ that is non-deterministic. There is at least one state that contains at least two outgoing transitions with the same action name. In this case, the initial state has two outgoing $i$ transitions so a random choice between the two actions is made.

A deadlock is modeled by a special process to ensure that a state does not have any outgoing transitions and cannot terminate. The deadlock process is denoted by the $\delta$ symbol.

Figure 2.7a shows an LTS of the specification $a \cdot b$. This specification terminates when the $b$ action terminates. Thus, the state after the $b$ transition is a terminating state. Figure 2.7b shows the LTS of the specification $a \cdot b \cdot \delta$. After the $b$ action the $\delta$ process is explicitly called hence this specification does not terminate.

The sum operator is a generalization of the alternative composition taking all elements of a data type $D$ into account. The notation that is used for the sum operator is $\sum_{d \in D} p(d)$ where $D$ is a data type, $d \in D$ is the free variable...
CHAPTER 2. THE MCRL2 LANGUAGE

Figure 2.7: Two branching bisimilar LTSs.

and $p$ a process. This means that a random element out of the data set is chosen and that the process $p$ is executed using this element.

A sum can be rewritten using the alternative composition when $D$ contains a finite number of elements. E.g. $\sum_{d \in D} p(d)$ is equivalent to the specification $p(true) + p(false)$. It is allowed to use the sum over an infinite data type, e.g. natural numbers. In the latter case it is not possible to replace the sum using the alternative composition.

The location of the sum operator determines when an element for the free variable, which is introduced by the sum notation, is chosen. The data element is chosen during the occurrence of the first action that is within the scope of the sum operator. When the data element is defined before the actual action a mismatch between the previous selected parameter and the real parameter can occur. This can lead to a deadlock when parallel behavior is modeled. This is similar compared to the alternative composition. This means that the sum should be placed directly before the relevant action. When it is the intention to pick a random element out of a set an extra action should be placed between the sum and the action that takes the data to avoid the enforcement of the selected data parameter.

The conditional operator is comparable with an if-then-else construction in traditional programming languages like C and Java. The notation in mCRL2 is $c \rightarrow p \cdot q$ where $c$ is an expression that evaluates to a Boolean value and both $p$ and $q$ are processes. The term $c$ is data and it is not allowed to use processes in the $c$ term.

When $c$ evaluates to true process $p$ is executed and when $c$ evaluates to false process $q$ is executed. It is allowed to leave the else branch out of the specification. This means that there are no actions that can be performed in the else branch meaning that this is equivalent to the deadlock process $\delta$ in the else branch. An example using the conditional operator is shown in Figure 2.8 where the specification $(\text{queueSize} > 3) \rightarrow \text{openCheckout} \cdot \text{closeCheckout}$ is used.

Figure 2.8: LTS of the specification $(\text{queueSize} > 3) \rightarrow \text{openCheckout} \cdot \text{closeCheckout}$
One can assign a process variable to a process that defines the behavior of a component or (sub)system. This process variable is also called a process and can be used in recursions in the same or other processes. Figure 2.9 shows the LTS of the process \( P = a.b.P \) and recursively calls process \( P \). When one defines the processes \( Q = a.R \) and \( R = b.Q \) and uses process \( Q \) as the initial process the same LTS is obtained, namely the LTS that is shown in Figure 2.9.

It is also possible to add data to process specifications. An example of a process specification using data is \( S(\text{boolVal : } \mathbb{B}) = (\text{boolVal} \rightarrow a \diamond b).S(\neg\text{boolVal}) \).

When this process starts initially as \( S(\text{true}) \) the LTS of Figure 2.9 is obtained.

This simple example shows that there are many different ways of constructing models with equivalent behavior.

Another example that uses data is the specification \( T(n : \mathbb{N}) = c(n) \cdot (n < 3) \rightarrow T(n + 1) \diamond T(0) \). The related LTS is shown in Figure 2.10.

2.2.6 Parallel modeling constructions

It is possible to place multiple processes in parallel and observe interaction between processes. Actions of two processes running in parallel happen independently from each other. Since the order between actions of processes is not fixed it might be that actions from the first process are executed before, after or simultaneous with actions of the second process that is placed in parallel with the first process.

The parallel operator (||) is used to denote that two or more processes are placed in parallel. There are more modeling constructions for parallel modeling, but this section only addresses constructions that have been used in our models.

Multi-actions are observed when processes are placed in parallel. The communication operator is used to rename specific multi-actions to a single action. Data parameters have to be equal when used. The \( \Gamma_C(p) \) notation is used where \( C \) is a set of allowed communications of the form \( a_1 | ... | a_n \rightarrow c \), with \( n > 1 \) and \( a_i \) and \( c \) are action names. For example \( \Gamma_{\{a|b\rightarrow c\}}(a(0)|b(0)) = c(0) \) since data parameters are equal. On the other hand \( \Gamma_{\{a|b\rightarrow c\}}(a(0)|b(1)) = a(0)|b(1) \) because data parameters are not equal.
The communication operator does not forbid the occurrence of a single action that is part of a multi-action used to model synchronization between processes. Only allowing the renamed multi-action and blocking all other undesired messages is achieved by using the allow operator. The notation used is \( \nabla_V(p) \) where \( V \) is a list of actions that is allowed. For example \( \nabla_{\{a|b,b\}}(a|b + a + b) = a|b + b \).

Hiding behavior is done by using the hiding operator \( \tau_I \) where \( I \) is the set of messages that needs to be hidden. For example \( \tau_{\{a\}}(a) = \tau \).

### 2.3 Specifying requirements

The modal \( \mu \)-calculus is used to define system properties by means of modal formulas. The advantage of modal formulas is that a modal formula is not ambiguous, but the downside is that the expressiveness and the somewhat cryptic notation make it hard to understand the precise meaning of a modal formula. Writing the intended property in the form of a modal formula requires some experience. Basic constructions are addressed in this section.

The syntax of the modal \( \mu \)-calculus is given by the following Backus-Naur Form (BNF) equations:

- \( \phi ::= \text{true} \mid \text{false} \mid \neg \phi \mid \phi \land \phi \mid \phi \lor \phi \mid \langle a \rangle \phi \mid [a] \phi \)

The modal formula \( \text{true} \) is true in every state of a process and the modal formula \( \text{false} \) is false in every state of a process. The \( \lor \) (disjunction), \( \land \) (conjunction) and \( \neg \) (negation) have their usual meaning known from logic theory. The \( \langle a \rangle \phi \) is the diamond modality and is valid when one can perform an \( a \) after which \( \phi \) holds. The modal formula \( \langle a \rangle \langle b \rangle \text{true} \) states that it should be possible to observe an \( a \) action that is followed by a \( b \) action. The \( [a] \phi \) is the box modality and states that for every \( a \) action that is possible \( \phi \) should hold afterwards. The modal formula \( [a][b] \text{false} \) states that it is not possible to observe an \( a \) action followed by a \( b \) action. Another example is \( [a]\langle b \rangle \text{true} \) that states that after every \( a \) action it should be possible to execute a \( b \) action.

In most cases a requirement involves the observation of multiple actions. Regular formulas \( R \) are used for this in combination with action formulas \( \alpha \):

- \( \alpha ::= a_1 \mid \cdots \mid a_n \mid \text{true} \mid \text{false} \mid \overline{\alpha} \mid \alpha \land \alpha \mid \alpha \cup \alpha \)

- \( R ::= \epsilon \mid \alpha \mid R \cdot R \mid R^* \mid R^+ \)

The \( a_1\cdots a_n \) is the set of multi-actions that only contains the multi-actions \( a_1\cdots a_n \). The \( \text{true} \) and \( \text{false} \) represent the set of all actions and the empty set respectively. The modal formula \( \langle \text{true} \rangle \langle a \rangle \text{true} \) specifies that after an arbitrary action it should be possible to perform an \( a \) action afterwards. All actions other that the \( a \) action is denoted as \( \overline{a} \). The \( \land \) and \( \cup \) connectives are used to indicate intersection and union of action sets.
The empty sequence of actions is denoted by $\epsilon$. Concatenation in action sequences is denoted by $R \cdot R$ and union of action sequences is denoted by $R + R$. Zero or more repetitions of action sequences is denoted by $R^*$ and one or more repetitions of action sequences is denoted by $R^+$.

More expressiveness is added to the language that is presented so far by introducing minimal and maximal fixed point operators. Minimal and maximal fixed point operators are omitted from this chapter since fixed point modalities are hard to explain. Furthermore, verification can be extended with data parameters and time as well. More information regarding fixed point modalities, data and time can be found in [2].
This chapter addresses research and results of the start, stop and recover (SSR) component. First, Section 3.1 defines relevant questions and the approach used to obtain answers to those questions. Section 3.2 defines the context of the SSR component that is used as a foundation for the SSR components study. Component boundaries are given in Section 3.3 by defining external interaction and required behavior is described in Section 3.4 by means of a conceptual state machine. The SSR components requirements are listed in Section 3.5. The actual model and checks are described in Section 3.6 and results, conclusions and future work are shared in Section 3.7.

3.1 Research goals and approach

This first use-case that is addressed is used to investigate the possibilities to use mCRL2 in an industrial application. The use-case used considers the SSR component that implements strategies used to start, stop and recover X-Ray systems. The advantage of the SSR component is that the implemented functionality and the interaction with its environment are clear. Questions regarding this use-case are:

- How can one create a model of a real life system or component?
- What are the benefits gained from the model?
- How can the model be used after creation, validation and verification?
- Can one tell something about the scalability with respect to the modeled use-case?
The remaining part of this section will address the strategy that is used to model the SSR component.

Rather than using the source code that implements the SSR component designers were interviewed to understand the required behavior and requirements of the SSR component. This led to an informal specification of both system behavior and requirements and can be found in Appendix A. Large parts of this appendix are also presented in this chapter in a more formal way with a higher level of abstraction.

Modeling started with creating an mCRL2 specification that describes the behavior of the SSR component. Next, multiple modal formulas were used to check the requirements that are presented in Section 3.5.

3.2 Context

An X-Ray system is composed out of multiple components. One or more components are hosted by a single process and may rely on other components. To guarantee the availability of components without violating dependencies different strategies to start, stop and recover the X-ray system are used. Those strategies are implemented in the SSR component.

The functionality and used strategies are given by definitions that are used in the remainder of this chapter.

**Definition 3.1** (process). A process hosts one or more components in an isolated environment and has two states, namely:

- *Inactive*, hosted components cannot be used.
- *Active*, hosted components can be used.

Initially a process is inactive and can become active by starting the process. When a process is active it can be stopped or a crash can occur after which, in both cases, the process becomes inactive. This is depicted in Figure 3.1.

![Figure 3.1: LTS depicting the internal states of a process according to Definition 3.1.](image)

A layered style is used to manage starting, stopping and recovering processes. Processes are grouped together in so called levels. Starting, stopping
and recovering are implemented using a level based strategy rather than a process based strategy.

**Definition 3.2** (level). A level is a, possibly empty, set of processes that has three states:

- **Stopped**, all processes that are in this level are inactive.
- **Started**, all processes that are in this level are active.
- **Pending**, not all processes of this level are active or inactive. Thus, this level contains 1 or more active and 1 or more inactive processes.

Note that in case of the empty set a level is per definition started and stopped, but can never be pending.

**Definition 3.3** (directly dependent level). Let \( L = \{1, ..., L\} \) be a non-empty set of levels and let \( l, l' \in L \). Then \( l' \) is directly dependent on \( l \) iff \( l' \) contains a process that relies on a process located in \( l \). That is \( l' \) contains a process that relies on a process located in \( l \).

**Definition 3.4** (indirectly dependent level). Let \( L = \{1, ..., L\} \) be a non-empty set of levels and \( l, l' \in L \). Then \( l' \) is indirectly dependent on \( l \) iff there exists a finite set of levels \( \{l_i \in L | 0 \leq i \leq n\} \) with \( n \geq 0 \) for which it holds that:

- \( l_0 \) is directly dependent on \( l \), and
- \( l_{i+1} \) is directly dependent on \( l_i \), and
- \( l' \) is directly dependent on \( l_n \).

Figure 3.2 shows a set of levels with direct and indirect dependencies. The arrow heads of Figure 3.2 indicate a ‘depends on’ relation, so process 2A depends on process 1A in Figure 3.2. There are three direct dependencies (Definition 3.3) present in Figure 3.2, namely level 4 that is directly dependent on level 3, level 4 that is directly dependent on level 2 and level 2 that is directly dependent on level 1. Level 4 is indirectly dependent (Definition 3.4) on level 1 by using level 2. This is the only indirect dependency that is present in Figure 3.2.

**Definition 3.5** (valid set of levels). Let \( L = \{1, ..., L\} \) be a non-empty set of levels and let \( l, l' \in L \). Then \( L \) is a valid set iff levels are ordered such that \( l < l' \) when satisfying:

1. \( l \) is not directly dependent on \( l' \), and
2. \( l \) is not indirectly dependent on \( l' \).
Definition 3.5 states that the ordering of levels is transitive and that levels may only depend on lower located levels. Figure 3.2 is an example of a valid set of levels. It is not necessarily needed for a level to depend on a lower located level as level 3 in Figure 3.2 shows. This means that it could be the case that a given set of levels has multiple valid orderings. For example, swapping levels 2 and 3 in Figure 3.2 would still be a valid set. Starting, stopping and recovering needs to be performed in such a way that those dependencies are not violated.

Definition 3.6 (start strategy). Let \( \mathcal{L} = \{1, \ldots, L\} \) be a valid set of levels that needs to be started and let \( l, l' \in \mathcal{L} \) for which \( l < l' \) holds. Levels are started in ascending order starting with the lowest level \( 1 \in \mathcal{L} \). It is allowed to start level \( l' \) iff \( l \) is started.

Definition 3.7 (stop strategy). Let \( \mathcal{L} = \{1, \ldots, L\} \) be a valid set of levels that needs to be stopped and let \( l, l' \in \mathcal{L} \) for which \( l < l' \) holds. Levels are stopped in descending order starting with the highest level \( L \in \mathcal{L} \). It is only allowed to stop level \( l \) iff \( l' \) is stopped.

Starting a level is only allowed when all lower levels are started and stopping a level is only allowed when all higher levels are stopped (Definitions 3.6 and 3.7 respectively).

Definition 3.8 (crashed level). Let \( l \) be a level that is started or pending. Then \( l \) is a crashed level when it contains a process that is inactive due to a crash. \( l \) remains crashed until either \( l \) is started or stopped.
Definition 3.9 (Relevant crash). Let $\mathcal{L} = \{1, ..., L\}$ be a valid set of processes and let $l \in \mathcal{L}$. Then the crash of level $l$ is a relevant crash iff the crash of $l$ occurred when all levels in $\mathcal{L}$ are started and no other levels are crashed.

Definition 3.10 (lowest crashed level). Let $\mathcal{L} = \{1, ..., L\}$ be a valid set of levels and let $l \in \mathcal{L}$ be a crashed level. $l$ is the lowest crashed level iff there is no other crashed level $l' \in \mathcal{L}$ such that $l' < l$ holds.

Definition 3.11 (recovery strategy). Let $\mathcal{L} = \{1, ..., L\}$ be a valid set of levels and let $l_i \in \mathcal{L}$ be the lowest crashed level. Recovery is performed in three consecutive steps:

1. The set $\{l_{i+1}, ..., L\}$ is stopped according to the stop strategy,

2. next $l_i$ is recovering by starting all inactive processes located in this level,

3. next the set $\{l_{i+1}, ..., L\}$ is started according to the start strategy.

Definition 3.12 (Fatal crash). Let $\mathcal{L} = \{1, ..., L\}$ be a valid set of levels that is being recovered. A fatal crash occurs iff the crash occurs while being in step 2 or 3 of the recovery strategy that is defined in Definition 3.11.

A crash during the recovery procedure (Definition 3.11) is only relevant when it occurs during the restart of levels (steps 2 and 3 of Definition 3.11). A crash during the stopping of levels, that is step 1 of Definition 3.11, can only change the lowest crashed level and does not interfere with the current active recovery strategy. Receiving a crash while levels are being restarted implies that a crash occurred in a level that is restated. It might be the case that another level depends on the crashed level, so the crash is considered to be a fatal crash (Definition 3.12).

The SSR component and all other necessary components that are needed to start the rest of the system are started upon booting. The SSR component receives instructions to start, stop or recover the system when this is appropriate. This decision is made in another part of the system and is considered to be an external event for the SSR component. The SSR component informs the rest of the system about the state of the system by means of messages that are send by the SSR component. Other components are informed if starting, recovering or stopping succeeded or failed and when a crash occurred.
3.3. INTERFACE DEFINITIONS

External interfaces of the SSR component are split into three groups, namely
the process side, the controlling side and the timeout event side. Messages
related to the process side consider messages regarding single levels and pro-
cesses and are defined in Section 3.3.1. The controlling side contains messages
of a higher abstraction level that are related to starting, stopping and recov-
ering the X-Ray system and are defined in Section 3.3.2. The occurrence of a
timeout is handled by the timeout event (Section 3.3.3). A graphical overview
of all three groups is shown in Figure 3.3.

3.3.1 Process side messages

Messages that are part of the process side, that is messages on the left of the
SSR component in Figure 3.3, are defined in this section.

- **Start a level**
  The SSR component sends a `StartLevel(Level)` message to start all in-
active processes of the level that is given as a parameter. As a response the
`Starting(ListOfProcesses)` message is received by the SSR component
with as parameter a list of processes that will be started. An overview
of messages that are related to starting a level are shown in Table 3.1.
Note that this message is also used to start all crashed processes of the
level that is given as a parameter as stated in Definition 3.11.

- **Stop a level**
  The SSR component sends a `StopLevel(Level)` message to stop active
processes of the level that is given as a parameter. As a response the
`Stopping(ListOfProcesses)` message is received by the SSR component
CHAPTER 3. START, STOP AND RECOVER USE-CASE

Request message that is sent by the SSR component

\textit{StartLevel(}\text{Level}\text{)}

\text{Level} \quad \text{Level that should be started}

Reply message that is received by the SSR component

\textit{Starting(}\text{ListOfProcesses}\text{)}

\text{ListOfProcesses} \quad \text{List of process ids that are being started}

Table 3.1: Message used to start a single level.

Request message that is sent by the SSR component

\textit{StopLevel(}\text{Level}\text{)}

\text{Level} \quad \text{Level that should be stopped}

Reply message that is received by the SSR component

\textit{Stopping(}\text{ListOfProcesses}\text{)}

\text{ListOfProcesses} \quad \text{List of process ids that are being stopped}

Table 3.2: Messages used to stop a level.

with as parameter a list of processes that are being stopped. Messages related to stopping a level are shown in Table 3.2.

- \textbf{Event messages}

Event messages are thrown by the subsystem that is attached to the process side of the SSR component to inform the SSR component about a status change of a process. When a process is started the \textit{ProcessStarted( Id)} message is received with the process id of the started process as a parameter. When a process is stopped the \textit{ProcessStopped(Id)} message is received with the process id of the stopped process as a parameter. The \textit{ProcessCrashed(Level, Id)} is sent when the crash of a process is detected and includes the process id and the level at which this process is located as parameters. Table 3.3 contains all event messages of the process side.

3.3.2 Controlling side messages

Messages that are part of the controlling side, that is messages on the right of the SSR component in Figure 3.3, are defined in this section.

- \textbf{Starting the system}

The SSR component receives the \textit{Start(MinLevel, MaxLevel, Timeout)} message when it should start the set of levels \{\text{MinLevel, ..., MaxLevel}\}
### 3.3. INTERFACE DEFINITIONS

**Event messages sent to the SSR component**

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ProcessStarted(Id)</code></td>
<td>Id of the process that is started</td>
</tr>
<tr>
<td><code>ProcessStopped(Id)</code></td>
<td>Id of the process that is stopped</td>
</tr>
<tr>
<td><code>ProcessCrashed(Level, Id)</code></td>
<td>Level that contains the crashed process, Id of the process that is crashed</td>
</tr>
</tbody>
</table>

Table 3.3: Incoming event message related to single processes.

**Request message that is received by the SSR component**

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Start(MinLevel, MaxLevel, Timeout)</code></td>
<td>MinLevel - lowest level to start, MaxLevel - highest level to start, Timeout - maximum time that is available to complete the start sequence</td>
</tr>
</tbody>
</table>

Table 3.4: Message used to start the X-Ray system.

According to the start strategy (Definition 3.6) within the given timeout time that is given as the third parameter. If all levels are started within the given time limit the `Started` message is sent by the SSR component. The `StartFailed` message is sent when not all levels are started within the given timeout time. Note that both `Started` and `StartFailed` messages are responses to the `Start(MinLevel, MaxLevel, Timeout)` message. Messages related to starting the X-Ray system are shown in Table 3.4.

**Stopping the system**

The SSR component receives the `Stop(Timeout)` message when the SSR component should stop all levels within the given timeout time (third parameter) according to the stop strategy (Definition 3.7). That is stop the set of levels `{MinLevel, ..., MaxLevel}` previously started by the `Start(MinLevel, MaxLevel, Timeout)` message. If all levels are stopped within the given time limit the `Stopped` message is sent by the SSR component. The `StopFailed` message is sent in case not all levels are stopped.
CHAPTER 3. START, STOP AND RECOVER USE-CASE

- Requested message that is received by the SSR component

  \[ \text{Stop} (\text{Timeout}) \]
  \[ \text{Timeout} \]
  Maximum time that is available to complete the stop sequence

- Reply messages that are sent by the SSR component (just one message is send upon a single request message)

  \[ \text{Stopped} \]
  \[ \text{StopFailed} \]

Table 3.5: Messages used to stop the X-Ray system.

- Request message that is received by the SSR component

  \[ \text{Recover} (\text{Timeout}) \]
  \[ \text{Timeout} \]
  maximum time that is available to complete the stop sequence

- Reply messages that are sent by the SSR component (just one message is send upon a single request message)

  \[ \text{Recovered} \]
  \[ \text{RecoverFailed} \]

Table 3.6: Messages used to recover the X-Ray system.

within the given timeout time. Note that both \text{Stopped} and \text{StopFailed} messages are responses to the \text{Stop} (\text{Timeout}) message. Messages related to stopping the X-Ray system are shown in Table 3.5.

- Recovery of the system

  The SSR component receives the \text{Recover} (\text{Timeout}) message when the SSR component should initiate the recovery strategy (Definition 3.11) of the set of levels \{\text{MinLevel}, ..., \text{MaxLevel}\} previously started by the \text{Start(MinLevel, MaxLevel, Timeout)} message. Recovering should be performed within the given timeout parameter. The \text{Recovered} message is sent when recovering succeeded within the given timeout time. If not all levels are recovered within the given timeout time the \text{RecoverFailed} message is sent by the SSR component. Note that both \text{Recovered} and \text{RecoverFailed} messages are responses to the \text{Recover} (\text{Timeout}) message. Messages related to recovering the X-Ray system are given in Table 3.6.

- Event messages

  The SSR component throws the \text{Crash} event when a relevant process
### 3.4. REQUIRED BEHAVIOR

Event message sent by the SSR component

| Crash |

Table 3.7: Outgoing event message related to the complete X-Ray system.

Event message sent to the SSR component

<table>
<thead>
<tr>
<th>ReceiveTimeout(Timeout)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timeout</td>
</tr>
<tr>
<td>The time value of the expired timeout</td>
</tr>
</tbody>
</table>

Table 3.8: Incoming event message related to a timeout.

Crash is detected (Definition 3.9). Observing a crash is done by receiving the `ProcessCrashed(Level, Id)` message that is part of the process side. Table 3.7 shows the crash message.

#### 3.3.3 Timeout event side message

There is one message that is related to receiving a timeout as depicted in Figure 3.3. This message is received when a timeout occurred. The actual timer used to keep track of the timeout time is not implemented in the model and is considered to be external. Detailed timing information may cause a state space explosion if a large set of possible timeout values are modeled hence using abstraction avoids this problem. The timeout message that is used will be discussed next:

- **Timeout message**
  The timeout event `ReceiveTimeout(Timeout)` is received when the timeout time of the current controlling side operation, that is starting, stopping or recovering the X-Ray system, has been expired. Table 3.8 shows this message.

#### 3.4 Required behavior

Behavior of the SSR component is described by a high level conceptual state machine that is shown in Figure 3.4. This state machine is not the real state space, but a higher level state machine to understand the functionality of the SSR component. Next a description of all states in the state space is given and outgoing transitions per state are addressed:

- **Idle**
  The X-Ray system is booted, but the X-Ray software is not started yet. There are no messages sent to the SSR component and the SSR component did not send any messages to its outside world.
The following outgoing transition is possible from the idle state:

- A $\text{Start}($MinLevel, MaxLevel, Timeout$)$ message is received and the Starting state is reached (transition 1 of Figure 3.4).

**Starting**

The set of levels $\mathcal{L} = \{\text{MinLevel}, ..., \text{MaxLevel}\}$, which is received via the $\text{Start}($MinLevel, MaxLevel, Timeout$)$ message, is started according to the start strategy (Definition 3.6). Not all levels in $\mathcal{L}$ are started and no crash or timeout has occurred during the start-up procedure. A crash or timeout is observed by receiving a $\text{ProcessCrashed}($Level, Id$)$ or $\text{ReceiveTimeout}($Timeout$)$ respectively.

The set $\mathcal{L}$ is used in the remainder of the conceptual state machine description with the bounds imposed by the $\text{Start}($MinLevel, MaxLevel, Timeout$)$ message. Note that this message can only be received once while being in the Idle state fixing the set of levels.

The starting state has the following outgoing transitions:

- A $\text{Stop}($Timeout$)$ message is received. The current starting procedure is stopped and the Stopping state is reached (transition 2 of Figure 3.4).

- All levels in $\mathcal{L}$ are started. No crash or timeout occurred during the execution of the starting strategy. The $\text{Started}$ message is sent to notify the controlling side and the Started state is reached (transition 3 of Figure 3.4).

- A $\text{ProcessCrashed}($Level, Id$)$ message or $\text{ReceiveTimeout}($Timeout$)$ message is received meaning that either a crash occurred or a timeout occurred during the start-up procedure respectively. In both
3.4. **REQUIRED BEHAVIOR**

cases it is assumed that the system is, per definition, irrecoverable. The message *StartFailed* is sent to notify the controlling side and the Irrecoverable error state is reached (transition 4 of Figure 3.4).

- **Started**
  All levels of $\mathcal{L}$ are successfully (re)started. No crash was detected during the start-up procedure (Definition 3.6) or all errors have been recovered according to the recovery procedure (Definition 3.11) without the occurrence of a fatal crash (Definition 3.12).

  The Started state has the following outgoing transitions:
  
  - A *Stop(Timeout)* message is received and the Stopping state is reached (transition 5 of Figure 3.4).
  
  - A *ProcessCrashed(Level, Id)* message is received indicating the occurrence of a relevant crash (Definition 3.9). The message *Crashed* is sent to notify the controlling side and the Recoverable error state is reached (transition 6 of Figure 3.4).

- **Stopping**
  All started and pending levels are being stopped using the stop strategy (Definition 3.7). Not all levels in $\mathcal{L}$ are stopped yet and no timeout has occurred. A timeout is observed by receiving the *ReceiveTimeout(Timeout)* message.

  The Stopping state has the following outgoing transitions:
  
  - All levels in $\mathcal{L}$ have been stopped within the given timeout time and the *Stopped* message is sent to notify the controlling side (transition 7 of Figure 3.4).
  
  - A *ReceiveTimeout(Timeout)* message is received meaning that a timeout occurred during the stopping procedure meaning that not all levels are stopped. The *StopFailed* message is sent to notify the controlling side and the Stop failed state is reached (transition 8 of Figure 3.4).

- **Stopped**
  All levels of $\mathcal{L}$ are successfully stopped.

  The Stopped state has no outgoing transitions.

- **Stop failed**
  Not all levels of $\mathcal{L}$ are successfully stopped.

  The Stop failed state has no outgoing transitions.

- **Irrecoverable error**
  The X-Ray system cannot be recovered and levels that are started and
processes that are active will remain active unless they crash by means of a \texttt{ProcessCrashed(Level, Id)} message.

The Irrecoverable error state has the following outgoing transition:

- A \texttt{Stop(Timeout)} message is received and the Stopping state is reached (transition 9 of Figure 3.4).

• Recoverable error

The X-Ray system can be recovered, but the recovering procedure is not started yet. Levels that are started and processes that are active will remain active unless they crash by means of a \texttt{ProcessCrashed(Level, Id)} message.

The Recoverable error state has the following outgoing transitions:

- A \texttt{Recover(Timeout)} message is received and the Recovering state is reached (transition 10 of Figure 3.4).
- A \texttt{Stop(Timeout)} message is received and the Stopping state is reached (transition 11 of Figure 3.4).

• Recovering

The X-Ray system is recovering by using the recovery strategy (Definition 3.11).

The Recovering state has the following outgoing transitions:

- All levels in \( L \) are recovered. The \texttt{Recovered} message is sent to notify the controlling side and the Started state is reached (transition 12 of Figure 3.4).
- A \texttt{Stop(Timeout)} message is received and the Stopping state is reached (transition 13 of Figure 3.4).
- A fatal crash (Definition 3.12) occurred. The \texttt{RecoverFailed} message is sent to notify the controlling side and the Recoverable error state is reached (transition 14 of Figure 3.4).

3.5 Requirements

This section contains the requirements for the SSR component.

1. The SSR component is only allowed to send messages when the X-Ray system is in the starting, recovering or stopping state of the conceptual state machine (Figure 3.4 in Section 3.4) or in case a relevant crash (Definition 3.9) occurred.
2. After receiving a \textit{Start(MinLevel, MaxLevel, Timeout)} message a response message \textit{Started} or \textit{StartFailed} has to be sent unless the starting procedure is interrupted before sending a response message by receiving a \textit{Stop(Timeout)} message. In that case it is not allowed to send either a \textit{Started} or \textit{StartFailed} message.

3. Sending the \textit{Started} message is only allowed after receiving a \textit{Start(MinLevel, MaxLevel, Timeout)} message and a successful start of all levels without the occurrence of a crash, a timeout or a stop request. So the \textit{ProcessCrashed(Level, Id)} message, \textit{ReceiveTimeout(Timeout)} message or \textit{Stop(Timeout)} message respectively are not received when levels are being started.

4. Sending the \textit{StartFailed} message is only allowed after receiving a \textit{Start(MinLevel, MaxLevel, Timeout)} after which a crash or timeout occurred observed by receiving a \textit{ProcessCrashed(Level, Id)} or \textit{ReceiveTimeout(Timeout)} message respectively. One of those messages must be received before all levels are started and it is not allowed to interrupt the start-up procedure by receiving a \textit{Stop(Timeout)} message.

5. After receiving a \textit{Stop(Timeout)} message a response message \textit{Stopped} or \textit{StopFailed} has to be sent.

6. A \textit{Stopped} message may only be sent after receiving a \textit{Stop(Timeout)} message and when all levels are stopped within the given timeout time. That is the message \textit{ReceiveTimeout(Timeout)} is not observed.

7. A \textit{StopFailed} message may only be sent after receiving a \textit{Stop(Timeout)} message and when not all levels are stopped within the given timeout time. That is the \textit{ReceiveTimeout(Timeout)} message is received before all levels are stopped.

8. After receiving a \textit{Recover(Timeout)} message a response message \textit{Recovered} or \textit{RecoverFailed} has to be sent unless the recovering procedure is interrupted before sending a response message by receiving a \textit{Stop(Timeout)} message. In that case it is not allowed to either send a \textit{Recovered} or \textit{RecoverFailed} message.

9. Sending the \textit{Recovered} message is only allowed after receiving a \textit{Recover(Timeout)} message and a successful restart of all levels without the occurrence of a fatal crash (Definition 3.12), a timeout or a stop request. Receiving a \textit{ReceiveTimeout(Timeout)} or \textit{Stop(Timeout)} message indicates that a timeout or a stop request occurred respectively.

10. Sending the \textit{RecoverFailed} message is only allowed after receiving a \textit{Recover(Timeout)} after which a fatal crash (Definition 3.12) or a timeout, observed by receiving the \textit{ReceiveTimeout(Timeout)} message, occurred.
The occurrence of either a relevant crash or a timeout must be received before the recovering procedure is completed and it is not allowed to interrupt the recovering procedure by receiving a Stop(Timeout) message.

11. The Crash message has to be send if and only if a relevant crash occurred (Definition 3.9).

12. Starting the next level by means of the StartLevel(Level) message is only allowed when all processes of the previous lower level are started (Definition 3.6).

13. Stopping the next level by means of the StopLevel(Level) message is only allowed when all processes of the previous higher level are stopped (Definition 3.7).

14. It should always be possible to receive a Stop(Timeout) message after receiving a Start(MinLevel, MaxLevel, Timeout) message and before a Stop(Timeout) message was received.

15. No other actions can be sent or received after sending the Stopped message, so the reached state is a deadlock state. After receiving a StopFailed message it is only possible to receive ProcessStopped(Id) messages. In all other cases no deadlock may occur.

### 3.6 Modeling

The construction of the model is addressed in Section 3.6.1. First, the decomposition between processes is addressed and next the actual processes are addressed in more detail. Section 3.6.2 introduces the different types of modal formulas that have been used and shows specific checks in more detail. The last section, Section 3.6.3, gives some information regarding the software and hardware that have been used.

#### 3.6.1 mCRL2 model

Functionality of the SSR component is split into logical fragments by using multiple mCRL2 processes. There is only one process active at a given time, so no parallelism or message synchronization is used in this model. The active process may activate another process by calling this process directly by its name thereby deactivating itself.

A high level overview of the dependencies among processes is given in Figure 3.5. The Start system component receives the Start(MinLevel, MaxLevel, Timeout) message to initiate the start-up according to the start strategy (Definition 3.6 and transition 1 of Figure 3.5). Starting and stopping levels is performed in the Start levels and Stop level process groups respectively. Each group contains 3 processes:
3.6. MODELING

![High level diagram showing that relations between processes.](image)

- **Start/stop next level**
  Checks if there is a next level that should be started/stopped or if the currently active starting, stopping or recovering strategy is completed (Definitions 3.6, 3.7 and 3.11 respectively).

- **Start/stop single level**
  Checks if all processes of a level are started/stopped.

- **Start/stop single level timeout handler**
  Implements the expected behavior upon a timeout while levels are being started/stopped.

The *System started* process is used when all levels are started.

The transition between the Start levels group and the Stop levels group (transition 2 of Figure 3.5) is needed since it should be possible to interrupt the start-up procedure and initiate a system stop. The transition between the Stop levels group and the Start levels group (transition 5 of Figure 3.5) is only used when the X-Ray system is recovering using the recovery strategy (Definition 3.11).

After a successful start or recovery procedure the System started process is reached because all levels have been started (transition 3 of Figure 3.5). The relation between the System started process and Stop levels process group (transition 4 of Figure 3.5) is used when the X-Ray system is recovered after the occurrence of a relevant crash (Definition 3.9) or when the X-Ray system is stopped by using the stop strategy (Definition 3.7).

Next, the different processes of the model are addressed. Note that all actions are prefixed with ‘Act_’ to distinguish actions from processes.

The first part of the model is shown in Listing 3.1. The *Level* and *Id* data types are used to model levels and processes respectively and are defined by using the built in positive numbers (Listing 3.1, lines 1 and 2 respectively). Timeout values are abstracted away by using a *Timeout* sort with a single element reducing the complexity of the SSR component (Listing 3.1, line 3).

Bookkeeping related to processes of a level that is being started or stopped are stored in lists. Extra properties on those lists are needed to be able to determine if all processes of a level are started or stopped. The list needs to be ordered and the order that is chosen is to start with the smallest element.
Furthermore, the occurrence of an element out of the set $Id$ is limited to at most one. The extra properties used are needed to ensure that the list of processes is unique for a given set of processes. This property is required to compare the list of processes that has to be started and the list of processes that is started.

The $insert$ function shown in Listing 3.1 lines 7-12 facilitates the insertion of a single element while maintaining the order of the list and ensuring that every element occurs at most one time in the entire list. The $sorted$ function (Listing 3.1, lines 15-20) tests if a given list with elements out of the set $Id$ is either sorted or not by means of returning the Boolean true or false receptively. This function is used to ensure that only correctly formatted lists are received as parameter of the $Starting(ListOfProcesses)$ message or the $Stopping(ListOfProcesses)$ message.

Finally, two variables that hold upper bounds on the number of processes and levels are defined to bound the model (Listing 3.1, lines 23-27). A finite number of levels and processes are required to be able to check requirements.

Listing 3.1: Used data types, auxiliary functions and bounds

```plaintext
sort Id = Pos;
sort Level = Pos;
sort Timeout = struct timeout;

%%% Auxiliary functions to handle lists
map insert : Pos # List(Pos) -> List(Pos); %The list contains at most one instance of a given element
var n, n':Pos;
l:List(Pos);
eqn insert(n, []) = [n];
insert(n, n'|>l) = if (n==n', n'|>l,
    (if (n<n', n>|>n'|>l, n'|>insert(n,l))) );

%%% Upper bounds on infinite data types
map MAX_ID : Id;
eqn MAX_ID = 2;
map MAX_LEVEL : Level;
eqn MAX_LEVEL = 2;
```

Starting the system by means of receiving the $Start(MinLevel, MaxLevel, Timeout)$ message is done in the $StartSystem$ process that is shown in Listing 3.2. The lowest level in the set of levels that has to be started is started by calling the $StartNextLevel$ process (Listing 3.2, line 7).
3.6. MODELING

- **toStartLevel**  
  Next level that should be started

- **minLevel**  
  Lower bound on the set of levels that should be started

- **maxLevel**  
  Upper bound on the set of levels that should be started

- **notRecovering**  
  True when starting and false when recovering

| Table 3.9: Parameters of the *StartNextLevel* process.

**Listing 3.2:** The *StartSystem* process of the SSR component

```plaintext
StartSystem =
{
  sum min, max:Level.
  (min<=max && min<=MAX_LEVEL && max<=MAX_LEVEL) ->
  (Act_Start(min, max, timeout).
  StartNextLevel(min, min, max, true)
  )
};
```

The *StartNextLevel* process (Listing 3.3) ensures that levels are started according to the start strategy (Definition 3.6) using the parameters that are shown in Table 3.9. Note that the *minLevel* and *maxLevel* parameters contain the values that were received by the *Start(MinLevel, MaxLevel, Timeout)* message earlier.

**Listing 3.3:** The *StartNextLevel* process of the SSR component

```plaintext
StartNextLevel(toStartLevel, minLevel, maxLevel:Level,
notRecovering:Bool) =
{
(toStartLevel <= maxLevel) ->
  (Act_StartLevel(toStartLevel).
  StartSingleLevel(toStartLevel, minLevel, maxLevel,
    notRecovering, [], []
  )
  )
}
<>

notRecovering ->
  (Act_Started.SystemStarted(minLevel, maxLevel)
  <> Act_Recovered.SystemStarted(minLevel, maxLevel)
  )
};
```

The *toStartLevel* is started by sending the *StartLevel(Level)* with the *toStartLevel* as parameter if this level is within the bounds (Listing 3.3, line 6). All levels have been started when the *toStartLevel* is outside the bounds. In that case a *Started* or *Recovered* message is sent depending on the value of the *notRecovering* (Listing 3.3, lines 12-14).
Starting a single level is done in the \textit{StartSingleLevel} process that is shown in Listing 3.4. This process checks if all processes belonging to a level are actually started and uses the parameters that are shown in Table 3.10. The \textit{minLevel} and \textit{maxLevel} parameters contain the values that were received by the \textit{Start(MinLevel, MaxLevel, Timeout)} message earlier.

\textbf{Listing 3.4: The} \textit{StartSingleLevel} \textbf{process of the SSR component}

\begin{verbatim}
StartSingleLevel(currentLevel, minLevel, maxLevel: Level, 
   notRecovering: Bool, toStartList, startedList:List(Id)) = 
{
  notRecovering ->
  { 
    \textbf{sum} crshLevel:Level, crshId:Id.
    (crshLevel<=MAX_LEVEL \&\& crshId<=MAX_ID) ->
    { 
      Act_ProcessCrashed(crshLevel,crshId).
      (crshLevel)>=minLevel \&\& crshLevel<=maxLevel) ->
      { 
        Act_StartFailed.Act_Stop(timeout).
        StopNextLevel(currentLevel, minLevel, maxLevel, true, 1)
      } 
    } 
  } <> StartSingleLevel() 
}

<>(
  \textbf{sum} crshLevel:Level, crshId:Id.
  (crshLevel)<=MAX_LEVEL \&\& crshId<=MAX_ID) ->
  { 
    Act_ProcessCrashed(crshLevel,crshId).
    (crshLevel)>=minLevel \&\& crshLevel<=maxLevel) ->
    { 
      Act_RecoverFailed.
      { 
        Act_Recover(timeout).
        StopNextLevel(currentLevel, minLevel, maxLevel, 
          false, crshLevel)
      } 
    } +
  } 
  Act_Stop(timeout).
  StopNextLevel(currentLevel, minLevel, maxLevel, 
    true, 1)
}

<>(StartSingleLevel()) 
}
\end{verbatim}
The first part implements the crash handling while a level is being started (Listing 3.4, lines 3-45). First, behavior related to starting the X-Ray system (Definition 3.6) is described (Listing 3.4, lines 6-18). This corresponds with
CHAPTER 3. START, STOP AND RECOVER USE-CASE

<table>
<thead>
<tr>
<th>currentLevel</th>
<th>Level that is currently started</th>
</tr>
</thead>
<tbody>
<tr>
<td>minLevel</td>
<td>Lower bound on the set of levels</td>
</tr>
<tr>
<td>maxLevel</td>
<td>Upper bound on the set of levels</td>
</tr>
<tr>
<td>notRecovering</td>
<td>True when starting and false when recovering</td>
</tr>
<tr>
<td>toStartList</td>
<td>List containing all processes that should be started</td>
</tr>
<tr>
<td>startedList</td>
<td>List containing all actually started processes</td>
</tr>
</tbody>
</table>

Table 3.10: Parameters of the StartSingleLevel process.

the starting state of Figure 3.4 (Section 3.4). Next, behavior that is related to step 3 of Definition 3.11, that is starting of the levels belonging to the recovery strategy, is modeled (Listing 3.4, lines 20-43). This behavior corresponds to the recovering state of Figure 3.4 (Section 3.4). For both cases an extra check that tests if the crash occurred in a level that is actually started is added and if not this crash message is ignored and no further actions are taken (Listing 3.4, lines 11 and 25).

Receiving a ProcessStarted(Id) indicates that a process has been started. This process is added to the list of processes that are actually started by using the insert function (Listing 3.4, lines 47-61). The list of all processes that are being started is received by means of the Starting(ListOfProcesses) message (Listing 3.4, lines 63-77). This message should be received at least one time before the level is actually considered to be started. A level is started when the received list of all processes that are being started is equal to the list of processes. This test uses the fact that a list is unique for a given set of processes that are actually started. When a level is not considered to be started yet the Start single level process is recursively called with the correct parameters (Listing 3.4, lines 57-58 and 74). The previously introduced StartNextLevel process is called when the current level has been started (Listing 3.4, lines 55-56 and 72-73).

The start of a level can be interrupted by receiving a Stop(Timeout) or ReceiveTimeout(timeout) message (Lines 3.4, lines 79-83 and 85-95). The Stop(Timeout) initiates the stop strategy (Definition 3.7). A StartFailed or RecoverFailed message is sent in case of a timeout when the start or recovery strategy is active (Definitions 3.6 and 3.11). To be able to receive messages that are related to the aborted start of this level an auxiliary process StartSingleLevelTimeoutHandler is started as described below.

After all levels are started the SystemStarted process, that is shown in Listing 3.5, is called. The parameters that are shown in Table 3.11 contain the level bounds that were previously received by the Start(MinLevel, MaxLevel, Timeout) message. This process ensures that it is possible to stop the X-Ray system by means of a Stop(Timeout) message (Listing 3.5, lines 3-6). Detection of a relevant crash (Definition 3.9) and the recoverable error state
3.6. MODELING

<table>
<thead>
<tr>
<th>minLevel</th>
<th>Lower bound on the set of levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxLevel</td>
<td>Upper bound on the set of levels</td>
</tr>
</tbody>
</table>

Table 3.11: Parameters of the SystemStarted process.

of Figure 3.4 (Section 3.4) are also implemented in the SystemStarted process (Listing 3.5, lines 8-31). The occurrence of a crash that is not related to a level that is started is ignored.

Listing 3.5: The SystemStarted process of the SSR component

```plaintext
SystemStarted(minLevel, maxLevel:Level) =
{
    Act_Stop(timeout).
    StopNextLevel(maxLevel, minLevel, maxLevel, true, 1)
} +
{
    sum crshLevel:Level, crshId:Id.
    (crshLevel<=MAX_LEVEL && crshId<=MAX_ID) ->
    Act_ProcessCrashed(crshLevel, crshId).
    (crshLevel>=minLevel && crshLevel<=maxLevel) ->
    Act_Crash.
    (Act_Recover(timeout).
        StopNextLevel(maxLevel, minLevel, maxLevel,
                        false, crshLevel)
    ) +
    (Act_Stop(timeout).
        StopNextLevel(maxLevel, minLevel, maxLevel, true, 1)
    )
} <> SystemStarted()
};
```

When a timeout occurred during the start-up of a level the StartSingleLevelTimeoutHandler process, which is shown in Listing 3.6, is called. After a timeout it should be possible to receive a start notification by means of a ProcessStarted(Id) message of the level that was in the starting phase (Listing 3.6, lines 4-11). This message is accepted, but no further actions are taken. Another action that can be received is the Stop(Timeout) message to stop the system according to the stop strategy (Definition 3.7 and Listing 3.6, lines
CHAPTER 3. START, STOP AND RECOVER USE-CASE

<table>
<thead>
<tr>
<th>currentLevel</th>
<th>Level that is currently started</th>
</tr>
</thead>
<tbody>
<tr>
<td>minLevel</td>
<td>Lower bound on the set of levels</td>
</tr>
<tr>
<td>maxLevel</td>
<td>Upper bound on the set of levels</td>
</tr>
<tr>
<td>notRecovering</td>
<td>True when starting and false when recovering</td>
</tr>
</tbody>
</table>

Table 3.12: Parameters of the StartSingleLevelTimeoutHandler process.

13-16). It is also possible to initiate a new recover attempt (Definition 3.11) by means of the Recover(Timeout) message if a relevant crash (Definition 3.9) occurred (Listing 3.6, lines 18-32).

Listing 3.6: The StartSingleLevelTimeoutHandler process of the SSR component

```
StartSingleLevelTimeoutHandler(currentLevel, minLevel,
                               maxLevel, notRecovering
                               = Bo) =

    (sum rcvdId:Id.
       (rcvdId <= MAX_ID) ->
       Act_ProcessStarted(rcvdId)
    ).

    StartSingleLevelTimeoutHandler()
    +
    Act_Stop(timeout).
    StopNextLevel(currentLevel, minLevel, maxLevel, true, 1)
    +
    !notRecovering ->
    {
        %This StartSingleLevelTimeoutHandler process allows the
        %reception of a Act_ProcessStarted message. It is assumed that
        %this started process belongs to the level received via the
        %Act_Started message that is timed out and called the
        %StartSingleLevelTimeoutHandler process. Receiving the
        %Act_ProcessStarted message after the timeout is assumed to be
        %incorrect hence this level is recovered.
        Act_Recover(timeout).
        StopNextLevel(currentLevel, minLevel, maxLevel, false,
                       currentLevel)
    }
```

Processes related to the stopping levels process group that is shown in Figure 3.5 are similar to the processes used in the start levels process group of Figure 3.5.
3.6. MODELING

A difference between the StopNextLevel process with respect to the StartNextLevel process is that this process implements the transition between step 1 and step 2 of Definition 3.11, that is the point where the last level is stopped and levels should be (re)started. This process contains an extra parameter that is used to keep track of the lowest crashed level (Definition 3.10).

The StopSingleLevel is used to monitor if all processes that are part of a level are stopped by using two lists. This is the same strategy that is implemented in the StartSingleLevel process that was described before. Some extra bookkeeping that is related to either applying the stop or recovery strategy (Definitions 3.7 and 3.11) are added to this process. Timeout handling is in case of stopping levels also implemented by means of an auxiliary process, namely the StopSingleLevelTimeoutHandler process.

The StopNextLevel process, StopSingleLevel process and StopSingleLevelTimeoutHandler process are shown in Appendix B that contains the complete SSR component model code.

3.6.2 Modal formulas

This section describes the modal formulas that were used to verify the requirements (Section 3.5). Properties were checked using different modal formulas with different levels of accuracy. The used modal formulas are first generally introduced by means of an abstract example and there general properties. Next, the used modeling approach is addressed and the last part of this section addresses the individual modal formulas in more detail.

Two different modal formula constructions have been used in the modeling process. First, a relative simple construction is introduced that is considered to be less accurate. This modal formula construction is called the simple check. The second modal formula contains more details and is more accurate. This modal formula construction is called the complex check.

An abstract example of the simple modal formula is shown in Listing 3.7. This check states that the \( b \) action is inevitable after observing an \( a \) action. A level of fairness plays an important role in this formula because it could be that the \( b \) action is infinitely deferred by executing an infinite loop. This modal formula allows the execution of an infinite loop after observing the \( a \) action and before observing the \( b \) action and does not distinct actions that may or may not be executed infinity many times. Note that the actions \( a \) and \( b \) can also be a set of actions.

Listing 3.7: Simple inevitability check

\[
\text{% a: the triggering action(s)}
\text{% b: the target action(s)}
\text{[true*}.a!\{b\}]<true*.b\text{true}
\]

The complex modal formula that is shown in Listing 3.8 is an inevitability check that uses \( \mu \)'s and \( \nu \)'s. This modal formula also checks that the \( b \) action
is inevitable after observing an action. The difference is that observable actions executed between an action and a action are explicitly added to sets of actions that may be executed a finitely or infinitely number of times. The set contains actions that may be executed infinitely many times and actions not in the set and not in the set may be executed finitely many times. In this case one knows for what actions the notion of fairness is needed since the set of actions that can be performed infinitely many times is stated explicitly. Note that the actions and can also be sets of actions.

Listing 3.8: Complex inevitability check

```plaintext
% a: the triggering action(s)
% b: the target action(s)
% c: the infinite allowable action(s)

[true*.a]
mu X.

{ nu Y.

% finite looping over all actions except the actions listed below
[ !(b || c) ]X

&&

% infinite looping over all actions listed below
[ c ]Y

&&

% the target action b is reachable from the current state
%(and no deadlock)
<true*.b>true

}
```

Complex checks were initially used because they are very precise, but this construction is not favorable for the readability and the constructability of the modal formulas. The modal formulas become large and debugging the modal formulas turns out to be complex hence time consuming. After that simpler and less precise checks were used to investigate the difference between the two versions.

All actions that are allowed between the trigger action and the target action are placed in the set of finitely allowed actions, that is in the cycle. The formula is checked and a counterexample is generated in case the formula turned out to be false. This counterexample is used to analyze the defect and correct the model and/or the modal formula accordingly. Multiple inconsistencies between the model and the modal formulas have been resolved using this strategy. Typically multiple iterations are needed before a modal formula hold for a given model.
A counter example is generated by the tool set (pbes2bool tool with option -c), but cannot be used directly because the counter example does not contain external actions that lead to the violation. Internal process variables are used to keep track of the internal state of the model and the values of those process variables are listed for every state that is reached. Variable names are not included in the provided counter example. This counterexample is manually converted into a list of actions by using the lpsxsim simulator. This tool shows all internal process variables and the changes of those internal process variables upon execution of all enables external actions. By matching the changes in the process variables to the changes in the counterexample one can manually convert the counterexample to a list of actions that lead to an illegal state. It may be the case that different actions are enables that result in an equivalent process variable update.

Due to time limitations and the choice to use the modal formulas that contained the $\mu$ and $\nu$ constructions not all requirements that were presented in Section 3.5 are translated into modal formulas hence are not checked. The detailed construction that is the result of using $\mu$’s and $\nu$’s in modal formulas and the large amount of counter examples that had to be analyzed result in a large time consumption. Table 3.13 shows an overview of all checks and there status.

All checks that are constructed by using a complex modal formula (Listing 3.8) were also checked by using a simple modal formula (Listing 3.7) so a comparison between the two can be made. Next, all finished checks will be addressed.

The simple modal formula for the second requirement (Section 3.5) is given in Listing 3.9. This check uses the same concept that is shown in Listing 3.7, but to maintain readability multiple lines are used. The complex modal formula is shown in Listing 3.10 and follows the concept of Listing 3.8.

Listing 3.9 states that after observing a Start(MinLevel, MaxLevel, Time-out) message either a Started, StartFailed or Stop(Timeout) message has eventually to follow. Receiving the Stop(Timeout) messages before all levels are started aborts the start-up meaning that neither the Started nor the StartFailed message may be sent. This is also observed in the starting state of the conceptual state machine (Figure 3.4 of Section 3.4).

Listing 3.9: Simple check for requirement 2

```plaintext
%After receiving a Start(MinLevel, MaxLevel, Timeout) message a %response message Started or StartFailed has to be sent unless the %starting procedure is interrupted before sending a response message %by receiving a Stop(Timeout) message. In that case it is not %allowed to send either a Started or StartFailed message.

[true*]
forall min, max:Level, timeout:Timeout.
{
```

<table>
<thead>
<tr>
<th>Check number</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Not checked</td>
</tr>
<tr>
<td>2</td>
<td>Checked, passed using a simple and complex check</td>
</tr>
<tr>
<td>3</td>
<td>Check not completed</td>
</tr>
<tr>
<td>4</td>
<td>Not checked</td>
</tr>
<tr>
<td>5</td>
<td>Checked, passed using a simple and complex check</td>
</tr>
<tr>
<td>6</td>
<td>Not checked</td>
</tr>
<tr>
<td>7</td>
<td>Not checked</td>
</tr>
<tr>
<td>8</td>
<td>Checked, passed using a simple and complex check</td>
</tr>
<tr>
<td>9</td>
<td>Not checked</td>
</tr>
<tr>
<td>10</td>
<td>Not checked</td>
</tr>
<tr>
<td>11</td>
<td>Checked, passed fragmented using two checks</td>
</tr>
<tr>
<td>12</td>
<td>Check not completed</td>
</tr>
<tr>
<td>13</td>
<td>Check not completed</td>
</tr>
<tr>
<td>14</td>
<td>Checked, passed using a simple and complex check</td>
</tr>
<tr>
<td>15</td>
<td>Checked, passed using a single check</td>
</tr>
</tbody>
</table>

Table 3.13: Status of the modal formulas.

```plaintext
Act_Start(min, max, timeout).
!(
  (exists timeout2:Timeout.Act_Stop(timeout2)) ||
  Act_Started ||
  Act_StartFailed
)*
<
true*
{
  (exists timeout2:Timeout.Act_Stop(timeout2)) ||
  Act_Started ||
  Act_StartFailed
}
>true
```

The complex modal formula (Listing 3.10) also shows that infinitely looping over the actions `ProcessStarted(Id)`, `Starting(ListOfProcesses)` or `ProcessCrashed(Level, Id)` is allowed and a level of fairness only applies to those actions.
Listing 3.10: Complex check for requirement 2

%After receiving a Start(MinLevel, MaxLevel, Timeout) message a
%response message Started or StartFailed has to be sent unless the
%starting procedure is interrupted before sending a response message
%by receiving a Stop(Timeout) message. In that case it is not
%allowed to send either a Started or StartFailed message.

[true]
forall min, max:Level, timeout:Timeout.
{
  %whenever the Act_Start command is received with arbitrary
  %variables
  [Act_Start(min, max, timeout)]
  mu X.
  {
    nu Y.
    {
      %finite looping over all actions except actions listed
      %below allowed
      ![ (exists timeout2:Timeout.Act_Stop(timeout2)) ||
         Act_Started ]
      Act_StartFailed ]
      (exists rcvdId:Id.Act_ProcessStarted(rcvdId)) ||
      (exists rcvdList:List(Id).Act_Starting(rcvdList)) ||
      (exists rcvdLevel:Level, rcvdId:Id.
       Act_ProcessCrashed(rcvdLevel, rcvdId))
    }X
    &&
    %infinite looping actions listed below allowed
    {
      (exists rcvdId2:Id.Act_ProcessStarted(rcvdId2)) ||
      (exists rcvdList2:List(Id).Act_Starting(rcvdList2)) ||
      (exists rcvdLevel2:Level, rcvdId2:Id.
       Act_ProcessCrashed(rcvdLevel2, rcvdId2))
    }Y
    &&
    %target actions should be reachable (and no deadlock)
    true.
    [
      (exists timeout2:Timeout.Act_Stop(timeout2)) ||
      Act_Started ]
    Act_StartFailed
    >true
  }
}
}
The simple modal formula for the fifth check is shown in Listing 3.11 and the complex modal formula is shown in Listing 3.12.

Listing 3.11 does not contain any exceptions, so this means that the either a Stopped or StopFailed message is eventually sent after receiving a Stop(Timeout) message. The behavior that is checked in this modal formula corresponds with the behavior of the stopping state of the conceptual state machine (Figure 3.4 in Section 3.4).

Listing 3.11: Simple check for requirement 5

```plaintext
[true]
forall timeout:Timeout.
{
  Act_Stop(timeout).
  !(Act_Stopped || Act_StopFailed)
}
<true.
{
  Act_Stopped || Act_StopFailed
}
>true
}
```

Listing 3.12 that contains the complex modal states that loops actions ProcessStarted(Id), ProcessCrashed(Level, Id), ProcessStopped(Id) and Stopping(ListOfProcesses) are allowed and a level of fairness needs to be taken into account for those actions.

Listing 3.12: Complex check for requirement 5

```plaintext
[true]
forall timeout:Timeout.
{
  %when the Act_Stop command is received with an arbitrary timeout
  %variable [Act_Stop(timeout)]
  mu X.
  (nu Y.
```
Requirement 8 checks the response after receiving a *Recover(Timeout)* message and is related to the recovering state of the conceptual state machine (Figure 3.4 of Section 3.4). Both simple and complex modal formula follow the same paradigm compared to checks 2 and 5. After receiving a *Recover(Timeout)* message it holds that either a *Recovered, RecoverFailed* or *Stop(Timeout)* message is eventually received. The complex modal formula allows infinite loops with the *ProcessStarted( Id ), ProcessCrashed( Level, Id ), ProcessStopped( Id ), Starting( ListOfProcesses)* and *Stopping( ListOfProcesses)* are allowed. The simple version of the modal formula that checks requirement 8 is given in Appendix C.5 and the related complex modal formula is given in Appendix C.6.

The eleventh check is divided into two separate checks. First it is checked
when the \textit{Crash} message needs to be sent as shown in Listing 3.13. The second modal formula checks that the \textit{Crash} message is indeed not sent when it is prohibited as shown in Listing 3.14.

The \textit{Crash} message needs to be sent when a relevant crash (Definition 3.9) occurred and no other action may be observed. An extra check is added by using the \texttt{val} statement to limits this check to started levels only (Listing 3.13, line 11).

\begin{lstlisting}[language=PLA] Listing 3.13: Allowed check of requirement 11 %The Crash message has to be send if and only if a relevant crash occurred (See definition relevant crash in report). [true*] forall min, max:Level, timeout:Timeout. { [Act_Start(min, max, timeout).true*. (Act Started||Act_Recovered)] forall level:Level, id:Id. { val(level>=min && level<=max) => %When a Act Started or Act_Recovered is received an Act Crash can only be sent after receiving a Act_ProcessCrashed and no other action is possible [Act_ProcessCrashed(level, id)] } <Act_Crash>true && ![Act_Crash]false } } 
\end{lstlisting}

The second modal formula (Listing 3.14) checks that the \textit{Crash} message is not sent when the \texttt{ProcessCrashed(Level, Id)} message is not considered to be a relevant crash (Definition 3.9).

\begin{lstlisting}[language=PLA] Listing 3.14: Prohibited check of requirement 11 %The Crash message has to be send if and only if a relevant crash occurred (See definition relevant crash in report). %When the last action is neither a Act Started nor a ActRecovered [true*. (!Act Started && !Act_Recovered)] %an Act Crash is not sent after receiving a Act_ProcessCrashed exists level2:Level ,id2:Id. { [Act_ProcessCrashed(level2, id2). Act Crash]false } 
\end{lstlisting}

The simple modal formula that checks the fourteenth requirement is shown in Listing 3.15 and the related complex modal formula is shown in Listing 3.16.
The requirement states that stopping the X-Ray system by receiving the \textit{Stop(Timeout)} should always be possible if no stop has been issued yet. All arrows going to the stopping state that is in the conceptual state machine visualize (Figure 3.4 of Section 3.4) visualizes this requirement.

The simple check (Listing 3.15) shows that there is no exception that cancels the need to observe either the \textit{Stopped} or \textit{StopFailed} message.

**Listing 3.15: Simple check for requirement 14**

```plaintext
%It should always be possible to receive a Stop(Timeout) message after receiving a Start(MinLevel, MaxLevel, Timeout) message and before a Stop(Timeout) message was received.

forall min, max: Level, timeout: Timeout.
{
  [true]*
  forall min, max: Level, timeout: Timeout.
  {
    Act_Start(min, max, timeout).
    !(exists timeout2: Timeout. Act_Stop(timeout2))
    [true]*
    <true*
    (exists timeout3: Timeout. Act_Stop(timeout3))
    >true
```

The complex modal formula (Listing 3.16) shows that the set of actions that may be executed for an infinite number of times after observing the \textit{Start(MinLevel, MaxLevel, Timeout)} messages and before observing either the \textit{Stopped} or \textit{StopFailed} message is relatively large and contains 12 different messages.

**Listing 3.16: Complex check for requirement 14**

```plaintext
%It should always be possible to receive a Stop(Timeout) message after receiving a Start(MinLevel, MaxLevel, Timeout) message and before a Stop(Timeout) message was received.

forall min, max: Level, timeout: Timeout.
{
  [true]*
  forall min, max: Level, timeout: Timeout.
  {
    %When the Act_Start command is received with arbitrary variables [Act_Start(min, max, timeout)]
    mu X.
    {
      nu Y.
      {
        %finite looping over all actions excepts actions listed below
        ![exists timeout2: Timeout. Act_Stop(timeout2)) ||
```
The fifteenth check contains three parts and comments about the used constructions as shown in Listing 3.17.

Listing 3.17: Check for requirement 15

```plaintext
%No other actions can be sent or received after sending the Stopped message, so the reached state is a deadlock state. After receiving a StopFailed message it is only possible to receive ProcessStopped(Id) messages. In all other cases no deadlock may

%target action should be reachable (no deadlock)
<
  true*.
  (exists timeout3:Timeout.Act_Stop(timeout3))
>true
```

The fifteenth check contains three parts and comments about the used constructions as shown in Listing 3.17.
%occur.

[true*]
(
%In all cases it has to hold that after a Act_StopFailed message a Act_ProcessStopped message is the only message that can be received. Even after receiving multiple Act_ProcessStopped messages

[Act_StopFailed]
{
  %exists recvId:Id.Act_ProcessStopped(recvId))>true

  %true*:!(%exists recvId:Id.Act_ProcessStopped(recvId))false
}

%In all cases it has to hold that after a Act_Stopped no other action is enabled
[

%all other actions do not result in a deadlock
[
    ![
        Act_Stopped
        Act_StopFailed
    ]
]

3.6.3 Used software and hardware

This section contains information regarding the hardware and software that have been used to verify the requirements and benchmark figures.

All tests have performed on a higher end Lenovo notebook computer (W520) with an Intel I7 processor (2630QM running at 2.00 GHz), 20 gigabyte of RAM memory (1333 DDR3) and a standard 7200 RPM 16 MB cache hard disk (Seagate ST9500420AS). During the execution of the benchmark tests all unnecessary programs were closed and no other activities were performed. No parallel execution of tools was used and a Windows PowerShell script was used to run all tools in the correct order.

Timing measures were obtained by starting a timer, calling a tool and stopping the timer. One has to be aware of the overhead that is included in the benchmark figures like disk access overhead while reading and writing files to the hard disk. All intermediate files are stored onto the hard disk and the next tool reads the information from the hard drive. Tool set information is
shown in Listing 3.18 and detailed operating system and hardware information are shown in Listing 3.19.

Listing 3.18: Toolset information

```plaintext
PS D:\> mcrl22lps --version
mcrl22lps mCRL2 toolset 201310.0 (Release)
Copyright (c) 2013 Technische Universiteit Eindhoven.
This is free software. You may redistribute copies of it under the
terms of the Boost Software License
<http://www.boost.org/LICENSE_1_0.txt>.
There is NO WARRANTY, to the extent permitted by law.
Written by Jan Friso Groote.
```

Listing 3.19: Operating system and hardware information

```plaintext
PS D:\> systeminfo

Host Name: S108183
OS Name: Microsoft Windows 7 Enterprise
OS Version: 6.1.7601 Service Pack 1 Build 7601
OS Manufacturer: Microsoft Corporation
OS Configuration: Member Workstation
OS Build Type: Multiprocessor Free
Registered Owner: TU/e
Registered Organization: TU/e
Product ID: 00392-918-5000002-85982
Original Install Date: 17-8-2011, 0:34:49
System Boot Time: 14-11-2013, 12:00:35
System Manufacturer: LENOVO
System Model: 4282AJ9
System Type: x64-based PC
Processor(s): 1 Processor(s) Installed.
[01]: Intel64 Family 6 Model 42 Stepping 7 GenuineIntel ~2001 Mhz
BIOS Version: LENOVO 8BET60WW (1.40 ), 6-12-2012
Windows Directory: C:\Windows
System Directory: C:\Windows\system32
Boot Device: \Device\HarddiskVolume1
System Locale: en-us;English (United States)
Input Locale: en-us;English (United States)
Time Zone: (UTC+01:00) Amsterdam, Berlin, Bern,
Rome, Stockholm, Vienna
Total Physical Memory: 20.363 MB
Available Physical Memory: 16.880 MB
Virtual Memory: Max Size: 40.725 MB
Virtual Memory: Available: 37.106 MB
Virtual Memory: In Use: 3.619 MB
Page File Location(s): C:\pagefile.sys
Domain: campus.tue.nl
Logon Server: \CAMPUSDC3
```
3.7 Conclusions and recommendations

Section 3.7.1 gives a short explanation why this use-case is suitable for modelling. Observations related to the used model and modal formulas are addressed in Sections 3.7.2 and 3.7.3 respectively. Recommendations and future work that followed from this use-case are presented in Section 3.7.4.

3.7.1 Use-case

The system bounds of the SSR component are well defined and general required behavior is known. Messages needed to interact with the external environment, that is communication that crosses the system bounds, is known and the set of actions that is needed is minimal. Those properties turned out to be favorable in the modeling process since this information is used to define the messages crossing the component bounds and the required behavior upon sending and receiving those messages.

3.7.2 Model

The construction of the used model is, despite the relatively large line count (412 lines), simple. Declared actions are kept to a minimum and synchronization between actions is not needed. Multiple processes are used to fragment the total functionality of the SSR component in a logic way creating different levels of abstraction. Relevant data is passed to the next process as parameters that are part of the process call. This eliminates facilities for global variable management like a dedicated variable process that can be queried for the required data by means of communicating messages.

The strategy used to keep track of the processes belonging to a single level that is started or stopped is a bit complex at first glance. First sets were used for the process bookkeeping because the advantage of a set is that an element occurs at most one time and there is no order between the elements in a set. Although the mCRL2 tools support sets, using a set in this case led to unbounded behavior while linearizing the model meaning that no requirements could be checked. Using an ordered list is the solution to this problem. Checking properties on a list is no problem, but the used lists have to obey to some extra properties as shown in Listing 3.1 (Section 3.6.1). After defining the extra functions on lists those functions can be used in a straightforward way in the model.

State and transition sizes of various model configurations are graphically depicted in Figures 3.6 and 3.7 respectively. The color of the bar is indicates the number of levels used and the number of processes per level are plotted on the x-axis. The logarithmic y-axis shows the number of states and transitions a model contains. Figures 3.6a and 3.7a show state and transition sizes of the
CHAPTER 3. START, STOP AND RECOVER USE-CASE

Figure 3.6: Scaling of the number of states of the SSR component.

Figure 3.7: Scaling of the number of transitions of the SSR component.
First the state space is considered when the number of processes per level are scaled. The number of states of the full model (Figure 3.6a) scales approximately exponentially, but the scaling of the bisimilar model (Figure 3.6b) is worse than exponential. When the state space is considered while scaling the number of levels one can see that for both the full model and the bisimilar model a better scaling compared to an exponential scaling. The slopes of the obtained state space points are larger compared to obtained slopes of the bisimilar state space points. This means that the state space reduction of the bisimilar model is better than a simple down scaling of the state space.

Next, the transition space is considered when the number of processes per level are scaled. The number of transitions present in the full model (Figure 3.7a) scale slightly better than an exponential relation. The bisimilar model (Figure 3.7b) scales slightly worse than an exponential relation. When the transition space is considered while scaling the number of levels it is clear that the transition space scales better than an exponential for both the full and the bisimilar model. In this case it also holds that slopes of the bisimilar model are smaller compared to slopes of the full model. So transitions are also not scaled by a factor but scales even better.

### 3.7.3 Modal formulas

Initially complex modal formulas (Listing 3.8) were used to check requirements. This turned out to be time consuming because the generated counter example is manually transformed to a list of actions by using the *lpsxsim* simulator as shown in Section 3.6.2. Due to the strictness of the complex modal formulas multiple iterations hence multiple counterexamples were needed.

During the analysis of counter examples it was observed that counterexamples frequently contained parts that are irrelevant because the modal formula is valid in the remainder of the trace, hence are not related to a trace that leads to an illegal state. Furthermore, the length of a counterexample could span dozens of lines. Deleting irrelevant parts from the counterexample makes the generation of the list of actions leading to an illegal state easier. A script was used to automate this process.

Another observation related to the complex modal formulas (Listing 3.8) is that counterexamples are typically constructed by detecting loops that can be executed infinitely many times that contains at least one action that is in the $\mu$-cycle. This means that it is not allowed to execute this action infinitely many times. A counter example that is related to such a complex modal formula contains multiple variables used to indicate the different iterations. The introduced nested $\mu$ and $\nu$ have their own variables and by introducing iterative behavior in the modal formulas more variables are introduced. Determining to what variable an action belongs can become difficult when multiple
variables are used. A counterexample of a complex check contains multiple traces. Most traces lead to the same defect, so one trace was analyzed and resolved. Next a new counter example was generated in case the modal formula failed again. This counterexample was analyzed using the same strategy.

Running times for all finished checks using different configurations of the model are collected. Figures 3.8 and 3.9 show running times of both simple and complex checks related to requirements 5 and 14 respectively. The bar colors indicate the number of levels used and the x-axis depicts the number of processes per level. Running time related to solving a generated formula is shown on the logarithmic y-axis. Separate graphs are used to show running times of simple and complex checks solved with the \textit{pbes2bool} and \textit{pbespgsolve} tools respectively.

In all measurements the \textit{pbespgsolve} tool is faster when compared to the \textit{pbes2bool} tool. In most cases it is observed that evaluating the simple check is faster compared to the required evaluation time of the complex check. This holds for check 5 as well (Figure 3.8). However, evaluating the simple version of the modal formula is not necessarily faster than evaluating the complex version of the modal formula as evaluation times related to check 14 shows (Figure 3.9).

### 3.7.4 Recommendations and future work

This section contains some suggestions that can be used to improve this use-case and the global modeling approach. First, improvements to the model and benefits of those changes are shared. Next, used strategies in the creation and verification of checks by means of modal formulas are addressed. After addressing the modal formulas more thoughts about bench marking and performance are shared. The last item that is addressed in this section considers the script that has been used for this use-case.

One of the modeling decisions is to model function calls asynchronously. The reasoning behind this is that other external components may violate the interface contract hence it might be possible to receive different actions between sending a request and receiving its return message. The SSR component is implemented in a synchronous way hence it is technically impossible to observe other actions in between a request and its return. Nevertheless, those cases are implemented in the model, hence making the model more complex but do not add value when the synchronous architecture is considered. Modeling components synchronously when this assumption can be made is preferred to avoid extra complexity.

The SSR components model could be simpler if external messages were modeled in a synchronous way hence reducing the state space. The number of allowed actions between a function call and its return value is reduced to one hence simplify behavior and reducing state space. The implementation uses synchronous communication as well, so this would be a valid assumption.
3.7. CONCLUSIONS AND RECOMMENDATIONS

Figure 3.8: Scaling of the time that is needed to solve checks related to requirement 5.
CHAPTER 3. START, STOP AND RECOVER USE-CASE

Figure 3.9: Scaling of the time that is needed to solve checks related to requirement 14.

(a) Simple check solved with pbes2bool
(b) Simple check solved with pbespgsolve
(c) Complex check solved with pbes2bool
(d) Complex check solved with pbespgsolve
Starting and stopping a level can be simplified when one uses synchronous communication. In that case it is not possible to receive a `ProcessStarted(Id)` message between sending the `StartLevel(Level)` message and receiving the `Starting(ListOfProcesses)` message. Furthermore, it is not possible to receive the `ProcessStopped(Id)` message between the `StopLevel(Level)` message and `Stopping(ListOfProcesses)` message. This means that an implementation that uses a single list could be used instead the two lists that are currently needed due to the non-synchronous modeling approach used.

Proper error handling can be implemented when external interaction is modeled by synchronous communication. One knows the list of processes that is either started or stopped before receiving `ProcessStarted(Id)` messages or `ProcessStopped(Id)` messages respectively. In case of the start of a single level one could store the returned list of processes that will be started. The bookkeeping strategy can be simple by removing processes that are started from the list. A level is only started when the list is empty and proper error handling regarding a start notification of a process that is not in the list can be implemented. A dedicated message used to indicate the reception of an illegal action can be introduced including a strategy used to handle the error like ignore and continue or a deadlock. Storing information regarding levels and processes that are started can also be used to improve the detection of illegal actions. A crash of a non-started process can be detected and appropriate actions can be taken.

Dividing requirements into sub-requirements can be beneficial to enhance the readability of a modal formula, so this is advised. Debugging a smaller modal formula or temporarily deactivate parts of a modal formula is considered to be a good approach. Pinpointing a defect is in most cases easier when the problem is as small as possible. A modal formula may contain multiple errors, so by iteratively sharpening a modal formula errors can be solved in a systematic way.

When complex checks are needed it is advisable to first verify if a simple version of the check holds and solve detected problems. The simple version of a counter example contains less variables because $\nu$’s and $\mu$’s are not used. When detailed guarantees are needed one can construct a complex modal formula after that without detecting mistakes that were resolved by using the simple modal formula.

The benchmark results presented in this report are from single measurements only. To rule out differences in overhead it is recommended to run the set of tests multiple times and use the average of the measurements. Furthermore, one should check for large difference between the equivalent measurements to obtain accurate measurements. It is also recommended to use the built in timing information of the tool set to obtain more accurate figures excluding overhead not directly related to the actual computations. Another benchmark measurement that can be obtained is memory usage.

One could verify running times of the mCRL2 tool set using a Linux op-
erating system and a Linux install. The toolset running on operating systems other than Windows contain the compiling rewriter that is in most cases more efficient compared to the default rewriter. It is hard to give an estimation regarding the gained performance because the construction of the model is related to the gain, so there are no standard measures to relate equivalent hardware and the same tool set version to different operating platforms. There are no benchmark figures of the mCRL2 tool set that has been used showing the performance between different operating systems as far as the author knows. Testing different tool set options might squeeze out a better performance as well.

The scripts used for this use-case were written in Windows PowerShell. This powerful scripting language was used to run multiple instances of tools with a throttle mechanism to avoid system overloading, parameters to influence script behavior, coupling with a spreadsheet to automate data collecting and more. The disadvantage is that this script is not suitable for non-Windows machines. Therefore using the universities supercomputer that has more resources compared to a decent desktop computer in combination with the PowerShell script was not an option since those machines are Linux based. Using a scripting language that supports different platforms such as Python is recommended when different systems with various platforms can be used.
CHAPTER 4

X-Ray controller use-case

This chapter is dedicated to the modeling process and results of the second use-case. The research goal of this use-case is addressed in Section 4.1 and background information regarding the modeled X-Ray controller is given in Section 4.2. The modeling procedure and used modal formulas are addressed in Section 4.3 by means of abstract examples. Finally, Section 4.4 contains conclusions and suggestions for future work related to the X-Ray controller.

4.1 Research goal

The aim of the second use-case is to find potential flaws in existing source code that is developed without the usage of formal methods. The strategy used to create the model is to stay as close as possible to the existing code. If an issue is found by verifying requirements one can easily map the potential problem to the related parts in the actual code due to the transparent mapping.

Interesting research questions regarding this use-case are:

- How can one create an mCRL2 model that represents behavior that is extracted out of existing C++ code?

- Can the model be used to find issues in the C++ code or prove that the chosen implementation satisfies specified requirements?

- Are there opportunities to fully automate the C++ to mCRL2 conversion and what C++ constructions may be difficult to translate into mCRL2 code?

- How can one verify that the C++ to mCRL2 model transformation is correct?

- How can one detect parts of C++ code that are never executed?
Designers were interviewed to define system specific requirements that are present in the selected use-case. This information is verified by creating specific checks. Needed software parts were added to the model and checks were added accordingly using an iterative approach.

4.2 Context

The X-ray controller is a high level controller that controls the global state of the X-Ray system. Detailed information and user interaction, e.g. movement of the detector or the status of a pedal, are not implemented in the X-Ray controller. The X-Ray controller uses dependencies between the system state and received events to determine the desired system response. Furthermore, the task of the X-Ray controller is to block illegal system state transitions to prevent the occurrence of incorrect behavior.

Low level user input is translated to higher level information by other components not implemented in the X-Ray controller and those events act as input information for the X-Ray controller. Changes in other components may trigger an event that is sent to the X-Ray controller. The X-Ray controller is an event driven component. Events are handled sequentially and cannot be interrupted.

The X-Ray controller is implemented by means of single threaded C++ code, so there is no parallelism involved in this use-case. Function calls are implemented by means of synchronous calls, so the caller blocks until the call returns. This means that the model executes in a sequential way as well and that function calls are also implemented in a synchronous way eliminating difficult asynchronous constructions. Hence, only one event can be active at a time.

4.3 Modeling

Strategies to model different C++ parts are presented in Section 4.3.1. Small abstract examples are used to indicate the used approach. Next, Section 4.3.2 shows the ideas used in the definition of modal formulas. The first part contains information about modal formulas that were automatically defined by a script and the second part is related to manually defined modal formulas. This use-case uses the same hard- and software that is used for the SSR component use-case and detailed information can be found in 3.6.3.

4.3.1 mCRL2 model

The choice to stay close to the C++ code led to a systematic translation of the C++ code into a model. Only relevant parts of the C++ code are added to the model. This section explains the following specific modeling approached
that have been used to translate specific C++ source code constructions into an mCRL2 model:

- Modeling the reception of external triggers in a synchronous and blocking way.
- Modeling internal function calls.
- Modeling reading and writing of internal class variables.
- Modeling pointers of an abstract base class type.

Every function that is in the C++ code is modeled by a dedicated process to allow a one-to-one source code to model translation. Functions that are triggered by an external trigger are implemented differently than functions that are called internally in the C++ source code. First, functions that are triggered by external events are addressed and next modeling internal function calls.

External events are handled in a synchronous way one at a time and cannot be deferred once started. Every external event is serviced by a single function in the source code, so every external event has one process that is activated when the external event is triggered. Only one process that accepts an external request as trigger can be activated at a time. All processes that are triggered by an external request are placed in an alternative composition to ensure that only one single process can be activated at any time. This alternative composition is implemented into a dedicated process that is called recursively after completing the running request to ensure that a new request can be received after servicing the previous one. No message synchronization is used in this approach. An abstract example of this approach is shown in Listing 4.1. Note that the messages used to trigger an external event may also include data parameters.

Listing 4.1: Modeling the reception of external triggers.

```plaintext
act %External triggers
EXTERNrcv_Message1, EXTERNrcv_Message2;

proc RequestHandler1 =
{
  %First action needs to be receiving the external trigger
  EXTERNrcv_Message1.
  <<<Body of the process omitted>>>}

proc RequestHandler2 =
{
  %First action needs to be receiving the external trigger
```
It should always be possible to call a function that is not triggered by an external event, so processes that model such an internal function should always be enabled. All processes used to model internal functions are initially placed in parallel to guarantee their availability. Communication among dedicated messages by means of synchronization is used to enforce the order between the calling and called process to ensure a sequential and blocking function call. Processes that model an internal function start and end with a dedicated message that synchronizes with the function call and the end of the function. Both messages may contain data to model parameters belonging to the function call and a return type respectively. The internal and external messages have to occur simultaneously and are renamed by the communication operator \textit{comm}. Listing 4.2 shows an example of this strategy. More processes implementing internal functions and data can be added in a straightforward way.

\textbf{Listing 4.2: Modeling internal function calls.}

```plaintext
act %FC = FunctionCall
%internal messages
  Int_FC1Req, Int_FC1Return;
%external messages
  Ext_FC1Req, Ext_FC1Return;
%communication messages
  Act_FC1Req, Act_FC1Return;

proc InternalFunction1 =
  ( %Receive function call
    Int_FC1Req.
    <<<Body of the process omitted>>>  
    %Send return value/synchronize
```
Variables present in a class are modeled by using a dedicated process that holds the value of this variable. Communication among dedicated set and get messages using synchronization are used to read and write a variable. The variable process is placed in parallel in the init section of the model together with the initial values of the variables that are modeled by this process. Listing 4.3 shows an mCRL2 model specification that implements a dedicated variable process. Using standard build in data types or own defined data types is allowed and multiple variables can be modeled in the same variable process.

The example of Listing 4.3 changes the value of the modeled variable from 0 up to and including 10 with steps of 1 by reading the old value and writing back an updated value. Initially the modeled variable contains the value 5.

Listing 4.3: Modeling class variables.
Modeling a function call based on an abstract base class requires extra bookkeeping. An object can be of the abstract class type, but the actual executed function is based on the concrete class that is being used. One needs to keep track of the used concrete class in the model to determine the correct version of the function that is called. This concept is explained by means of an example.

Let say that an abstract base class `Display` is used to implement output interfaces for a given application. This class contains a `print(string message)`
function used to display information and is redefined in the concrete classes that are based on the Display class as shown in Listing 4.4.

Listing 4.4: Definitions of the abstract base class Display and its derived concrete classes.

```cpp
using namespace std;
#include <string.h>
#include <iostream>

class Display { //Abstract base class
public:
    virtual void print(string message) = 0;
};
class HDMI : public Display { //Concrete HDMI class
public:
    void print(string message) {
        cout << "This is the HDMI class" << endl;
        cout << "Given argument: " << message << endl << endl;
    }
};
class VGA : public Display { //Concrete VGA class
public:
    void print(string message) {
        cout << "This is the VGA class" << endl;
        cout << "Given argument: " << message << endl << endl;
    }
};
```

One can create a pointer of the abstract base class and let this pointer point to a concrete class that uses the abstract class as a base. The print function can be called using this pointer and the correct version of the print function is executed based on the concrete type. Listing 4.5 shows a code snippet that illustrates this concept and Listing 4.6 shows the output of this snippet.

Listing 4.5: Using a pointer of the abstract base type.

```cpp
void sendMessage (Display *dp, string message) {
    if (dp != NULL)
        dp->print(message);
    else
        cout << "ERROR: No display selected!" << endl << endl;
}

int main(int argc, char* argv[])
{
    //pointer to abstract base class
    Display *dp;
    //concrete instances
    HDMI displayHDMI;
```
CHAPTER 4. X-RAY CONTROLLER USE-CASE

VGA displayVGA;

//reference pointing to HDMI class
dp = &displayHDMI;
dp->print("First test");

//reference pointing to VGA class
dp = &displayVGA;
dp->print("Second test");

//calling a function that requires a Display pointer
sendMessage(&displayHDMI, "First function call");
sendMessage(&displayVGA, "Second function call");
dp = NULL;
sendMessage(dp, "Third function call");
return 0;
}

Listing 4.6: Output of Listing 4.5

D:\>Example.exe
This is the HDMI class
Given argument: First test

This is the VGA class
Given argument: Second test

This is the HDMI class
Given argument: First function call

This is the VGA class
Given argument: Second function call

ERROR: No display selected!

The pointer of the abstract base class is modeled by means of a dedicated
sort that contains all different concrete classes that are based on the abstract
base class. Optionally an extra element is added to indicate that the given
pointer is not initialized if that is used in the C++ code. A model that im-
plements the sendMessage(Display *dp, string message) function is shown in
Listing 4.7. Note that trivial parts based on previously presented approaches
are omitted as indicated.

Listing 4.7: Model containing a pointer of an abstract base class.

%All concrete Display classes
sort Display = struct
HDMI?IsHDMI |
VGA?IsVGA |
noDisplay?IsNoDisplay; %NULL in C++

%Abstract message type
sort Message = struct message;
%Internal action
internal;
%Trigger action
EXTERNrcv_SendMessage : Display # Message;
%Response actions
EXTERNsnd_DisplayError;
EXTERNsnd_HDMI_Print : Message;
EXTERNsnd_VGA_Print : Message;
<<<Messages related to print functions calls omitted>>>

proc
sendMessage =
{
%receiving display pointer
sum dp:Display,
sum m:Message.
EXTERNrcv_SendMessage(dp, m).
%check if pointer is initialized
(!IsNoDisplay(dp)) ->
{
%hard coded dynamic function calls
(?
IsHDMI(dp) ->
{
Ext_HDMI_PrintReq(m).
Ext_HDMI_PrintReturn
}
<> internal
).
<<<Equivalent IsVGA case omitted>>>
}
<> EXTERNALsnd_DisplayError
).,
sendMessage
);
<<<Processes implementing print functions omitted>>>

init
hide
{ internal },
{
allow
{
%internal message
internal,
%external messages
EXTERNrcv_SendMessage,
EXTERNsnd_DisplayError,
EXTERNsnd_HDMI_Print,
EXTERNsnd_VGA_Print,
4.3.2 Modal formulas

The model has to obey some standard properties and because of the systematic construction of the model those properties can be checked in a systematic way as well. A script is used to generate so called automatic checks to verify the translation from the C++ code into a working model. Next, requirements that were discussed with designers were checked by so called manual checks.

**Automatically generated checks**

A syntactically correct model is used to extract all used messages and a prefix notation is used to identify the functionality of a given action. An overview of the different messages used is shown in Table 4.1. The following types of checks are automatically generated:

1. A message must be observable, so the LTS of the model contains at least one transition that is related to this message that is reachable from the initial state. This check is typically used to verify the presence of external triggers and communication messages.

2. A message may not be observable, so the LTS does not contain a transition that is related to this message that is reachable from the initial state. This check is typically used to verify that internal and external messages are not used autonomous, that is without the communication operator.

3. After sending a request message it is only allowed to receive the related response message, so no other messages should be enabled. This check is used to verify that external calls are modeled in a blocking way.

4. After sending a request message a response message will eventually follow. This is used when an internal function is called. The previous check cannot be used since the called function is executed, but the called function should always be possible to terminate and return its value to the caller.
4.3. MODELING

Interfacing with its external environment

• Receiving a message from the external world, a trigger
• Sending a message to the outside world
• Receiving a return message after sending a message to the outside world

Internal function calls

• Messages used in processes that model internal functions. Those messages are called internal messages since they are located inside the process that models the related internal function.
• Messages used to model calls to internal functions. Those messages are called external messages since they are located outside the process that models the related internal function.
• Action message, communication between the internal and external messages

Setting and getting variable values

• Internal message used in the variable process.
• External message used to model reading or writing a class variable.
• Action message, communication between the internal and external messages.

Table 4.1: Different message types used in the X-Ray controller model.

Next, a detailed description of the different automatic checks follows including an example modal formula.

Actions used to communicate with the outside world and communication actions used to model function calls and class variables (Section 4.3.1) have to be used at least once. The reasoning behind this is that a message can be omitted without changing the modeled behavior when this message is not used. The first automatic check is used to verify this property and an example of this check is given in Listing 4.8.

Listing 4.8: Modal formula that checks if an action is used.

```plaintext
1 %Observing an a action has to be possible
2 <>true.a=true
```
CHAPTER 4. X-RAY CONTROLLER USE-CASE

Internal and external messages used in modeling function calls and class variables may not be observed autonomously. The simultaneous occurrence of both internal and external actions is the only case that is allowed using those actions. The communication operator is used to rename this event, so observing either the internal or external message is not allowed. The second check is used to verify this requirement and an example of the used modal formulas is shown in Listing 4.9.

Listing 4.9: Modal formula that checks if an action is not used.

1 %observing an a action is not possible
2 [true*.a]false

After sending out an external request message that requires the reception of a return value one has to block and wait for this return value. This means that no other actions are enabled while waiting for the response message. The third check is used to verify this requirement and an example is shown in Listing 4.10.

Listing 4.10: Modal formula that checks that the reception of a return value occurs in a blocking fashion.

1 %after receiving a Request
2 [true*.request.tau*]
3 {
4    %receiving a return is possible
5    <tau*.return>true &
6    %receiving a message other than return is not allowed
7    [tau*.!(return || tau)]false
8  }

Check four is used for internal function calls. The caller blocks upon a function call but the called function is allowed to run so the third check cannot be used to verify this requirement. The fourth check verifies if the called function is always able to terminate. The used modal formula verifies if there is always a path to the termination message after receiving the function call message and before observing this termination message as shown in Listing 4.11. Note that this check allows infinitely many executions of a loop between the function call and its related return message.

Listing 4.11: Modal formula that checks if a given function is always able to terminate.

1 %after a trace that contains a function call without the related function return
2 [true*.functionCall.!(functionReturn)*]
3 %it should be possible to follow a path to the related function
4 [true*.functionReturn>true

Typical errors found by using the automatic checks and the deadlock check are communication mistakes in function calls and class variable usage. In most
cases this leads to a deadlock and by using the *lps2lts* tool with the options
-D to detect a deadlock and -t to store traces to a deadlock one could find
the error without manually creating a trace to a deadlock. Typical mistakes
are using the internal and/or external messages in the wrong place, absence
or mistakes in the allowed and communication statements in the initialization
process and not adding the process that contains the modeled function to the
initialization process or omitting the recursive call of the modeled function.

There is one case where the deadlock check revealed a potential error that
is related to changes in the current active acquisition, e.g., a fluoroscopy to ex-
posure transition. It could be the case that the measurement type that should
be started next is not initialized. The C++ implementation first stopped the
currently active measurement type before checking the ability to start the
next measurement type. In case of the fluoroscopy to exposure transition this
means that the fluoroscopy measurement is stopped first. Next, one once to
start the exposure measurement, but this is not possible and no measurement
is active afterwards. This results in a deadlock of the model.

The first automatic check is used to detect functions that are not called.
It turned out that it is valuable to detect code that is not executed. In this
particular use-case it was observed that class variables are sometimes changed
in parts that are not modeled. In most cases those changes are related to
initialization of different system parts. Changing internal class variables by
assuming that certain components are operational enabled large parts of the
code. In some cases it turned out that a function is not called because this
function is called outside the part modeled. In that case a function and all its
related messages are commented out to avoid the generation of failing checks.

Another issue found by using the first automatic check are class variables
that are not read or written. The latest model does not contain any variables
that are not read, but there are some variables that are not written. This
concerns variables that are edited in another part of the code that is outside
the modeling scope. Non used messages intended to change class variables
are commented out to avoid the generation of failing automatic checks after a
careful analysis.

**Manually generated checks**

Manual checks that were verified are typical safety requirements stating that
certain behavior is not allowed. A description of the checks and the used
modal formulas can be found in Appendix D. Manually created modal formulas
are presented by means of an abstract version of the X-Ray controller and
abstract modal formulas because the model and modal formulas used contain
confidential information.

The abstract X-Ray controller controls a measurement system that oper-
ates using a non-preemptive measurement sequence as shown in Figure 4.1.
Acquisition settings need to be applied and a measurement type needs to be
selected before a measurement can be started. A measurement run is started by sending a dedicated message (\textit{start}). Once a run is started it cannot be aborted or preempted and a dedicated message is used to notify the system about the completion of a measurement (\textit{completed}). A dedicated event message is used to indicate whether or not the generated output can be processed. Table 4.2 defines the messages used in this abstract example.

The following requirements are addressed in this section:

- It is not allowed to change acquisition settings during a measurement.

- A measurement may only be started when the generated output can be processed as notified by an event message that is sent to the X-Ray controller upon a change.

- It is not allowed to have more than one measurement type that is active if an acquisition is started.

- A measurement may only be started when the generated output can be processed. The X-Ray controller receives a notification when the processability of the output changes and can use a polling schema to receive information about the ability to process the output.

The first requirement states that it is not allowed to change acquisition settings during a measurement. This means that the \texttt{acquisitionSettings(Bool)} message may not be observed during an acquisition. That is observing the \texttt{acquisitionSettings(Bool)} message after sending the \textit{start} message and before receiving the \textit{completed} message is not allowed. Listing 4.12 shows the check that is used to verify this requirement.

Listing 4.12: Changing settings during a measurement is not allowed.

\begin{lstlisting}[
  language=Java,
  caption={Changing settings during a measurement is not allowed.},
  label={lst:change-settings}]
true.*.
%after a start
start.
%followed by actions other than runFinished
\end{lstlisting}
4.3. MODELING

Changing acquisition settings

\[
\text{acquisitionSettings}(\text{Bool})
\]

Change the validity of the settings (send)

**Validity of the applied settings:**

- **true:** Settings are valid
- **false:** Settings are invalid

Activate measurement type

\[
\text{activateType1}(\text{Bool}) \quad \text{activateType2}(\text{Bool})
\]

Change the activity of type 1 (send)

**Activity of the measurement type:**

- **true:** Measurement type is active
- **false:** Measurement type is inactive

Acquired data can be processed

\[
\text{outputEnabled}(\text{Bool})
\]

Output can be captured (receive)

**Status of the receiving party:**

- **true:** Output can be processed
- **false:** Output cannot be processed

Acquisition triggers

\[
\text{start} \quad \text{completed}
\]

- Start a measurement
- Measurement is completed

Table 4.2: Messages used in the abstract example.

\[
\begin{align*}
&! (\text{completed}) \star. \\
&% \text{it is not allowed to observe the following action} \\
&(\text{exists} \ \text{newSettings}:\text{Bool}. \text{acquisitionSettings}(\text{newSettings})) \\
&\] false
\]

The second check verifies that starting a measurement is only allowed when the generated output can be processed. This means that it is not allowed to start a measurement after receiving the `outputEnabled(false)` message while the `outputEnabled(true)` message has not been observed afterwards. Listing 4.13 contains the modal formula used to check this requirement.

Listing 4.13: It is not allowed to start a measurement when the output cannot be processed.
true*
% when the generated output cannot be processed
outputEnabled(false).
% and the output remains disabled!
(outputEnabled(true) )*. 
% it is not allowed to start a measurement
start
]false

The third requirement states that it is not allowed that both measurement
types are active simultaneously during a measurement. However, during the
setup procedure it is allowed to have both outputs active. This means that
starting a measurement by means of the start message is only allowed when
at most one measurement type is selected since it is not allowed to change
the activity of a measurement type according to the first check. The modal
formula used to check this requirement is shown in Listing 4.14.

Listing 4.14: It is only allowed to start a measurement when at most one
measurement type is active.

% initially both outputs are inactive
nu X(type1Active:Boolean=false, type2Active:Boolean=false).
{ % changing activity of the first measurement type
  forall newOutput1:Boolean.
  { [activateType1(newOutput1)]
    X(newOutput1, type2Active)
  }
  % changing activity of the second measurement type
  forall newOutput2:Boolean.
  { [activateType2(newOutput2)]
    X(type1Active, newOutput2)
  }
  % other actions do not change the activity of the outputs
  ![ (exists out1:Boolean. activateType1(out1)) ||
    (exists out2:Boolean. activateType2(out2)) ]
  X(type1Active, type2Active)
  % starting is only allowed when at most one measurement type is
  % selected
  [start]}
The fourth check looks similar to the second check that states that starting a measurement is only allowed when the output can be processed. In the next section some extra properties are added to the abstract example to be able to explain this check and the fault detected by this check.

Besides the event message `outputEnabled(Bool)` another message `pollOutputReq` is defined that is used to poll if the generated output can be processed. The message `pollOutputReturn(Bool)` is used to return whether or not the produced output can be processed and is received in a blocking fashion. The extension is depicted in Figure 4.2 and Table 4.3 shows the definition of this extra added message. The modified check that also contains this new message is show in Listing 4.15.

Listing 4.15: Starting is only allowed when the generated output can be processed.
Acquired data can be processed

- **pollOutputReq**: Send a poll request regarding the ability to process the output (send)
- **pollOutputReturn(Bool)**: Returned message containing if the output can be processed (received)
- **Bool**: Status of the receiving party:
  - *true*: Output can be processed
  - *false*: Output cannot be processed

Table 4.3: Extra messages added to abstract example.

The fourth check leads to the detection of a bug that is also present in the C++ code that was used for this use-case. The abstract example is used to explain the type of error that is found by giving an equivalent example where this bug occurs. Remember that events are handled in a sequential, blocking and non-preemptive way and that requests are not atomic hence it takes time to complete a request. Next, two components are defined that use the abstract example to give a more detailed description of the identified error.

It is assumed that there is an external component called the **Device Handler** that monitors the ability to process the output and interacts with the user. The Device Handler sends the **start** message to the X-Ray Controller if the user requests to do so. The Device Handler knows if the generated output information can be processed and informs the X-Ray Controller upon a change by sending the **outputEnabled(Bool)** message. When a **pollOutputReq** message is sent by the X-Ray Controller a **pollOutputReturn(Bool)** message will be returned by the Device Handler.

If settings are changed by means of the **changeSettings(Bool)** message the Device Handler is polled to get information regarding the ability to process the generated output information. This information is internally used in the X-ray Controller while settings are being changed. On the other hand, when the **start** message is handled by the X-Ray Controller it checks if the output can be processed by using the latest received and processed **outputEnabled(Bool)** event message.

The bug that is found leads to a typical race condition. The Device Handler sends a **start** message when the X-Ray Controller is applying settings. After queuing the **start** message an error occurred such that generated output cannot be processed. The event message **outputEnabled(false)** is sent and placed in the queue after the **start** message that is still pending. When the **start** message is serviced the old value that is stored in the X-Ray Controller is used stating that the output can be processed and the measurement is started while the output cannot be processed hence violates the requirement. Figure 4.3 contains a
4.4. CONCLUSIONS AND RECOMMENDATIONS

sequence diagram that illustrates this error.

Observing implicit assumptions

Implicit assumptions made during the coding phase let to failing manual checks. A counterexample typically contains a sequence of actions leading to undesired behavior. This behavior is in most cases not realistic due to the implicit assumptions, for example, one may assume strict alternation between the receptions of two messages while this is not enforced in the modeled code. Implicit assumptions where implemented in the model to verify if requirements hold if those assumptions are valid.

4.4 Conclusions and recommendations

Suitability issues for this second use-case are addressed in Section 4.4.1. Observations related to the creation of the model and the modal formulas are presented in Sections 4.4.2 and 4.4.3 respectively. Recommendations and future work regarding this use-case are addressed in Section 4.4.4.

4.4.1 Use-case

Defined requirements determine bounds on the model. Only C++ code that is needed to be able to verify defined requirements is added to the model. Requirements were added in an iterative way. This means that the model is
iteratively expanded and model bounds change when more C++ components are introduced in the model. Thus, fully understanding the complete model is harder when compared to the first use-case.

The C++ code contains a communication scheme that is used to implement communication between sub-components. This scheme provides a convenient way to define used messages since those interfaces are logically defined separating sub-components.

4.4.2 Model

The first part of this section contains answers to research questions related to the model that were given at the beginning of this chapter. Next, the second part of this section contains other observations and some statistics of the model used.

Answers to research questions

**How can one create an mCRL2 model that represents behavior that is extracted out of existing C++ code?**

The used strategy that models every function as a separate process (Section 4.3.1) turned out to be feasible in combination with the selected use-case. Most source code constructions can be translated into mCRL2 equivalent construction. A good understanding of the modeled code and assumptions used helps to speed up the modeling process when one converts source code into a model. Modeling decisions related to parts that cannot be translated into mCRL2 equivalent code in a straightforward way and C++ code that is irrelevant is easier when implemented functionality is known.

The disadvantage is that source code contains a lot of details. Source code details are also present in the model due to the direct source code to model translation. Including detailed information in the model leads to a large model and an expansion with respect to the line count of the used C++ files. The final model for this use-case contains over 2600 lines in total and around 800 lines (30%) are used to define messages and communication between messages. Generic figures cannot be given since the structure of the source code determines this ratio.

**Can the model be used to find issues in the C++ code or prove that the chosen implementation satisfies specified requirements?**

Yes, the created model can be used to verify system requirements. Manual checks (Section 4.3.2) where used to verify system properties and let to the identification of a bug present in the source code. More potential issues were found, but by using assumptions on the environment of the X-Ray controller during development those issues are not considered to be flaws. Those assumptions where added to the model and correct behavior is only guaranteed when the assumptions are valid (Section 4.3.2). Furthermore, automatically
generated checks (Section 4.3.2) where used to detect functions and variables that where not used.

The advantage of the chosen modeling approach that there is a direct mapping between the C++ functions and the mCRL2 processes. Mapping a counterexample to the existing C++ code is straightforward since names used in function calls and accessing class variables are linked to the C++ code. Thus, mapping errors found in the model is straightforward. C++ specific requirements like public, private and protected access to class objects are not checked.

Manually converting source code into an mCRL2 specification during active development is discouraged. A minor change in the source code implies a revision of the model. Keeping the model up-to-date can become a challenge or even a burden.

Analyzing the number of states and transitions present in the LTS of the model reveal some interesting reduction figures. The LTS that is created by the \textit{lps2lts} tool generates an LTS that contains 2,210,100 states and 4,055,716 transitions. The branching bisimilar LTS contains 149,096 states and 181,444 transitions. This is a 93.3\% reduction in the number of states and a 95.5\% reduction in the number of transitions.

An even larger reduction can be achieved by hiding internal function calls and accessing class variables. This would be a logical reduction since internal function calls and class variables are also hidden in the C++ implementation. The branching bisimilar LTS of this model contains 19,464 states and 38,108 transitions, a reduction of 99.1\% in both states and transitions. The sizes of the branching bisimilar LTSs are small and such small numbers indicate that the mCRL2 tool set is capable to verify requirements of this use-case.

The used modeling approach leads to a large amount of messages used to model the internal construction of the source code. Both a process that models an internal function and a single class variable (Section 4.3.1) require 6 messages. The model contains 218 messages and 177 (81.2\%) messages are used to model class variables and internal function calls. This means that a large amount of small functions that are sequentially executed require many messages when the modeling approach that is presented in this chapter is used. Thus, this explains the large reduction in states and transitions when internal behavior is abstracted away. The following message categories can be identified:

- 10 messages (4.6\%) are used to model external triggers.
- 20 messages (9.2\%) can be sent by the X-Ray controller to inform other components or to request data.
- 11 messages (5.0\%) can be received by the X-Ray controller containing requested information.
• 75 messages (34.4%) are used to model class variables.

• 102 messages (46.8%) are used to model internal function calls.

Are there opportunities to fully automate the C++ to mCRL2 conversion and what C++ constructions may be difficult to translate into mCRL2 code?

Partly, the X-Ray controller use-case shows that a systematic translation contains large parts that can be automated.

A transparent mapping between C++ functions and mCRL2 processes is used and can be automatically generated. A significant amount of mistakes related to defining, linking and using messages and recursive process calls can be avoided when automatically generated. Furthermore, class variables of standard data types are modeled in a straightforward way and can be automatically translated to mCRL2 code. Benefit is that the chance to make conversion errors is lowered, debugging times are reduced and the conversion is speeded up.

Automating the creation of the content that need to be placed inside the bodies of all defined processes is more challenging. Modeling constructions used to model different C++ constructions that are compact tend to explode in the number of lines that are needed. Encountered C++ constructions that expand when modeled in mCRL2 code are class variables, function calls and pointers to abstract base classes. The C++ language contains more powerful constructions like dynamic creation of objects, passing by value, passing by reference, passing by address and dynamic arrays that require proper modeling translations.

Other observations

The model cannot be used as an abstract representation of the modeled source code. Details are also present in the model and the model contains a large amount of overhead. This means that the model is not a compact representation that can be used as a reference.

Modeling constructions presented in this chapter are merely suitable to model sequential code. Calls to an active function are blocked until the function is completed hence sent its return message.

4.4.3 Modal formulas

The first part of this section contains answers to the research questions that were posed at the beginning of this chapter and the second part contains other observations.
4.4. CONCLUSIONS AND RECOMMENDATIONS

Answers to research questions

How can one verify that the C++ to mCRL2 model transformation is correct?
Automatically generated checks based on basic properties were used to verify source code to model transformation. Actions are named by using dedicated prefixes related to the functionality of the action. Actions are extracted from a syntactically correct model and their functionality is determined by detecting the prefix used. Four different types of automatically generated checks were used (Section 4.3.2) and a total of 389 checks are automatically generated.

It turned out that automatically generated checks were helpful to detect modeling errors and gain confidence in the created model. Various modeling errors, mostly related to message usage, communication between messages and process recursion, have been detected and resolved by using automatically generated checks. Although automatic checks are useful, it is not guaranteed that the source code to model transformation is fault free.

How can one detect parts of C++ code that is never executed?
Automatically generated checks were used to verify if a process is used and if a variable is read or written. The message that triggers such an action needs to be present in the model and is checked by a modal formula that is shown in Listing 4.8 (Section 4.3.2). The automatically generated check will fail if a variable is not used or when a function is not called and is an indication of code that is not executed.

Other observations

The running time per modal formula depends on the modal formula that is addressed. Typical running times that have been observed per modal formula are in the range of 5 to 15 minutes, but peaks up to 30 minutes have also been observed. Note that those figures are just rough estimations and that multiple modal formulas were simultaneous checked while other programs were using resources as well. Benchmark results should be collected and analyzed first before any conclusion can be drawn with respect to resource and timing usage. Comprehensive benchmark figures are not collected due to project time restrictions.

The state and transition sizes of the branching bisimilar LTSs (Section 4.4.2) suggest that improvements to running times can be achieved. Using different tools and/or options should improve the performance of the tool set allowing verification of larger models. For example, the presence or absence of a message can be verified by using the \texttt{lps2lts} tool with the option \texttt{-a} too detect actions. A list of all actions can be given and only a single execution of the \texttt{lps2lts} tool is needed instead of the 142 separate modal formulas used to verify the presence or absence of messages. Creating an LTS using the latest version of the mCRL2 toolset takes approximately 30 minutes including disk
access to save the generated LTS in .aut format.

A script with a throttling mechanism is used to parallelize the verification of all modal formulas. An important observation is that the memory consumption per modal formula increased during the expansion of the model. The upper bound on the number of checks was lowered from 8 to 5 to prevent memory swapping on the hard disc leading to an system overload while a total of 20 gigabyte of RAM was totally available in the used system (detailed hard- and software information can be found in Section 3.6.3).

Creating a trace that lead to an illegal state is time wise an expensive procedure. A trace that contains 100 actions or more is no exception for this use-case due to the detailed model that is used. Furthermore, a large number of internal variables are used by mCRL2 to keep track of the internal state of the X-Ray controller. Adding more details to the model will increase the number of internal variables making it harder to manually convert a given counter example to a trace. Creating a trace out of a counter example follows the same paradigm presented in Section 3.6.2.

4.4.4 Recommendations and future work

The study of this use-case shows that there are opportunities to automate the creation of a model out of existing C++ code that is not optimized for modeling purposes. Creating the used framework in an automated way would speed up the model creation. Debugging an automatically generated framework is not needed when one assumes that the framework and all needed messages are defined correctly.

Results of the X-Ray controller use-case show that there are opportunities to automate parts of the C++ source code to mCRL2 model translation. More research is needed to establish if it is feasible to model specific object oriented constructions and to propose proper modeling constructions. It might even be needed to introduce new mCRL2 language features to be able to translate specific C++ language constructions into an equivalent mCRL2 code that works in most general cases. The constructions used have to be suitable for requirement verification.

Performance of the tool set can be tweaked by using different tools and/or options. Currently no guidelines or best practice guides exists. Furthermore, benchmarks should be collected and other use-cases can be analyzed to determine the feasibility of this approach. It is also recommended to verify the performance of the tool set using a Linux version in combination with the compiling rewriter.
Conclusions

This chapter contains answers to the research questions presented in Section 1.3.

**Is it possible to use mCRL2 in the selected use-case and modeling approach and is this useful?**

**SSR component use-case**

Yes, it turned out that the first use-case can be modeled using a relative large but simple model (Section 3.7.2). The modeled SSR component is well defined and its behavior does not change over time (Section 3.7.1). The modeling approach that is used is suited to create and verify a model using an informal specification. Constructing modal formulas used in the verification process should be kept as simple as possible to minimize verification times (Section 3.7.3). Thus, trading in accuracy to reduce verification time when requirements allow this should be considered. This model can be used in the coding phase and acts as documentation.

**X-Ray controller use-case**

Yes, mCRL2 can be used in the X-Ray controller use-case. The chosen strategy to translate C++ source code into an mCRL2 specification (Section 4.4.2) without abstracting away a large amount of details can be applied in a systematic way and the resulting model can be used to verify automatically and manually generated modal formulas (Section 4.4.3). Common translation properties are verified by automatically generated modal formulas and system properties are verified by manually defined modal formulas. Translating C++ source code directly into mCRL2 code results in a large mCRL2 specification that is not suited for documentation purposes, but analyzing state and transition sizes indicate that the generated model can be used to verify requirements by using mCRL2 tools (Section 4.4.2).

**What is the effort that was needed in order to model a given software part?**
SSR component use-case

It turned out that the constructed model is simple (Section 3.7.2) and can be simplified when environmental constrains are incorporated into the model (Section 3.7.4). A first outline of the model could be realized within 2 to 3 weeks when required functionality is known.

Requirements can be verified with precise and more complicated checks or with simpler less precise checks (Section 3.7.3) and the time that is needed to verify requirements depends on the level of detail that is used. Multiple inconsistencies in the informal specification have been discovered by modeling and verifying the SSR component.

X-Ray controller use-case

Most source code constructions can be translated to mCRL2 equivalent constructions that require more lines of text (Section 4.4.2). Object and pointer constructions are harder to model and require detailed knowledge of the modeled code (Section 4.3.1). The construction of the model is error prone hence a significant time is needed to correct typical conversion mistakes (Section 4.3.1). Absence of sub-components needed for initialization typically leads to parts of the model that are not used and are detected by using automatically generated checks (Section 4.4.3).

Time needed to model a given software part depends on the code complexity and various constructions used. The X-Ray controller use-case shows that a significant amount of time is needed to model software and resolve modeling errors. Time that is spend on the creation and verification cannot be used to speed up the creation of source code as in the first use-case hence this approach is not (yet) suited for large scale industrial applications.

Keeping the model up-to-date with the source code that is translated is more challenging compared to the first use-case since source code is frequently changed during active development. The model for the first use-case needs to be validated and verified before the model is translated into source code. Hence it is likely that less updates to the code are needed for similar components.

Can existing source code be translated into a model that is suitable for validation and/or verification?

Yes, the X-Ray controller use-case shows that it is possible to convert existing C++ source code into an mCRL2 specification and use this specification to verify requirements. The modeling construction used is only suited to model single threaded code as explained in Section 4.3.1. Translating the C++ source code into equivalent mCRL2 constructions is for most source code constructions straightforward (Section 4.4.2). Object oriented construction and pointer constructions are harder to model (Section 4.4.2). The model contains significantly more lines compared to the modeled C++ source code (Section 4.4.2).

Are there opportunities to fully automate the translation process from source code in a model?
Partly, the X-Ray controller use-case shows that there are opportunities to automate the creation of a model out of existing C++ source code as stated in Section 4.4.2. A framework of processes that are used to model functions and class variables can be generated. All messages needed to call functions and to manipulate class variables can be defined and linked by means of the \textit{comm} operator automatically. The actual modeling code related to standard type class variables can be automatically generated as well.

The translation of a function body is harder since various C++ constructions are difficult or even impossible to model. Typical compact C++ object oriented constructions expand in the used number of lines since a brute force modeling schema is used (Sections 4.3.1 and 4.4.2). More research regarding useful modeling strategies for various C++ constructions like dynamic object creation, properly modeling pointers and polymorphism is needed to propose proper modeling strategies (Section 4.4.4).

Other challenges that one may encounter are:

- Conversion of C++ source files requires the implementation of the front-end of a compiler.
- Conversion of source code that is intended for different architectures, e.g., single-threaded versus multi-threaded.
- Automatic detection of external interaction.

Does the selected use-case and modeling approach contribute to the quality of the developed software?

\textit{SSR component use-case}

Yes, when the model is used in the coding phase one can create higher quality software with fewer bugs. The first use-case shows that modeling is beneficial to detect and resolve issues in the modeled system (Section 3.6.2). The challenge is to find a balance between including or abstracting details in both the model and the checks.

Although creating a model and verifying requirements takes time it is believed that a solid model reduced the coding phase and could even reduce the development time as shown by [3]. Experience is needed to speed up the modeling process. Creating a model and translating requirements in appropriate modal formulas is not a trivial process and requires practice. Furthermore, a solid understanding of the modeled component and its requirements is vital in order to create a model and verify related properties.

\textit{X-Ray controller use-case}

Yes, it is possible to translate existing C++ source code into an mCRL2 model and use this model to find potential flaws or even bugs in existing source code (Section 4.4.2). When those detected issues are resolved in the original code one can improve the quality of the software.
Translating existing C++ source code manually into a usable mCRL2 model is time consuming (Section 4.4.2). Extra checks are needed to verify the source code to model transformation (Section 4.4.3). This modeling approach is not feasible for industrial applications due to the time that is needed. An automatic C++ source code to mCRL2 model code conversion could reduce modeling time and make this approach more attractive, but this is not a straightforward conversion (Sections 4.4.2 and 4.4.4).

**What is the scalability of the model that is being used for the selected use-case?**

**SSR component use-case**

Benchmark figures show that the full version of the SSR component scales approximately exponential in the number of states and transitions when the number of processes per level are changed. The bisimilar equivalent version of the SSR component shows that scaling of states and transitions is slightly worse than an exponential relation. The number of states and transitions scales better compared to an exponential relation in both the full and the bisimilar model when the number of levels are changed. State and transition space are scale better than reduction by a factor. This can be seen in Section 3.7.2 and in Figures 3.6 and 3.7.

The same observations are made with respect to time needed to evaluate checks. When the number of processes per level are increased running times scale approximately exponential. Required time scales better than an exponential scaling when the number of processes per level are changed. Running times of the \textit{pbespgsolve} tool are in all cases faster compared to the \textit{pbes2bool} tool when solving the same formula. The disadvantage of the \textit{pbespgsolve} tool is that this tool is not able to produce a counter example.

**X-Ray controller use-case**

No extensive benchmark results have been collected for this second use-case, but scaling in the number of states and transitions when using a branching bisimilar LTS are large (Section 4.4.2). When internal messages are not hidden a reduction of 93.3% in states (from 2,210,100 states to 149,096 states) and 95.5% in transitions (from 4,055,716 transition to 181,444 transitions) is achieved. When hiding all internal messages of 99.1% in both states and transitions is (19,464 states and 38,108 transitions) achieved. Modeling components in a sequential way limits the number of different possibilities hence limits the state and transition size. Furthermore, a total of 177 out of 218 messages are related to internal function calls and class variables. This is 81.2% of the total message count and this explains the large reduction of states and transitions when internal communication is hidden (Section 4.4.2).

A large amount of messages are needed to model internal function calls and class variables (Section 4.4.2). Furthermore, object oriented properties can be hard to model and expand rapidly in the required number of lines needed in the mCRL2 model. The model becomes large and cluttered and multiple files that where directly related to the C++ files where used to maintain overview.
Can a model be used for other purposes than model checking?

**SSR component use-case**
In the SSR component use-case the model acts as a blueprint that can be used in future development. This blueprint guarantees that verified properties are correctly implemented in the model. This verified model can be used in the coding phase. Translating the mCRL2 model into executable source code should be straightforward hence implementing complex constructions should be less error prone since large parts can be copied directly from the model that is proven to be correct. This paradigm is also proposed in [3].

Another application of the model is educational purposes. Different levels of abstraction can be distinct in the model used. Strategies used to start, stop and recover the X-Ray system are implemented on a level bases. Dedicated parts of the model are related to starting and stopping single levels including detailed information that is related to processes (Section 3.6.1). The model can be used to document the functionality and implementation using different levels of abstraction.

**X-Ray controller use-case**
The X-Ray controller model contains a large amount of details. Furthermore, large parts of the model are related to the extra messages that are required to model internal functions and class variables (Section 4.3.1). This makes the model hard to read hence making this model inappropriate for other applications other than verifying properties (Section 4.4.2).
Appendices

This is the publicly available version of the report. Appendices contain confidential information and omitted from the public report as requested by Philips Healthcare.
SSR component documentation
APPENDIX B

SSR mCRL2 model
SSR modal formulas
C.1 Simple check, response to a \textit{Start} message (requirement two)

C.2 Complex check, response to a \textit{Start} message (requirement two)
C.3 Simple check, response to a Stop message (requirement five)

C.4 Complex check, response to a Stop message (requirement five)
C.5 Simple check, response to a Recover message (requirement eight)

C.6 Complex check, response to a Recover message (requirement eight)
C.7 General check, sending a \textit{Crash} is allowed
(requirement eleven)

C.8 General check, sending a \textit{Crash} is not allowed
(requirement eleven)
C.9 Simple check, receiving a *Stop* is possible
(requirement fourteen)

C.10 Complex check, receiving a *Stop* is possible
(requirement fourteen)
C.11 General check, occurrence of a deadlock
(requirement fifteen)
X-Ray controller component
List of Figures

1.1 X-Ray system developed by Interventional X-Ray, Philips Healthcare. 2
1.2 X-Ray image of veins filled with contrast fluid. ................. 2

2.1 Two simple LTSs that both can execute an atomic action \((a)\) from the initial state. ............................................. 13
2.2 Two branching bisimilar LTSs. ........................................ 13
2.3 Two non-branching bisimilar LTSs. ................................. 14
2.4 Principle of branching bisimulation. ................................. 14
2.5 LTS with a sequential composition \((a \cdot b)\). ....................... 15
2.6 Difference between a deterministic and non-deterministic LTS. . 15
2.7 Two branching bisimilar LTSs. ................................. 16
2.8 LTS that visualize a conditional operator. .......................... 16
2.9 LTS of two alternating actions. .................................... 17
2.10 LTS that contains one action with data. .......................... 17

3.1 LTS depicting the internal states of a process according to Definition 3.1. ............................................. 21
3.2 Schematic overview of the processes that are divided into separate levels. ............................................. 23
3.3 Interactions with the SSR component. ............................... 25
3.4 Conceptual state machine of the SSR component. .................. 30
3.5 High level diagram showing that relations between processes. .... 35
3.6 Scaling of the number of states of the SSR component. .......... 56
3.7 Scaling of the number of transitions of the SSR component. ....... 56
3.8 Scaling of the time that is needed to solve checks related to requirement 5. ............................................. 59
3.9 Scaling of the time that is needed to solve checks related to requirement 14. ............................................. 60

4.1 Interactions with the abstract measurement component. ......... 76
4.2 Interactions with the extended abstract measurement component. 79
4.3 Sequence diagram showing the found race condition. ............. 81
List of Tables

2.1 Predefined data types that are built-in in the mCRL2 tool set. . . 7
2.2 Predefined auxiliary functions on built-in data types. . . . . . . . 8
2.3 Overview of keywords for data type definitions. . . . . . . . . . . 12
3.1 Message used to start a single level. . . . . . . . . . . . . . . . . . 26
3.2 Messages used to stop a level. . . . . . . . . . . . . . . . . . . . . . 26
3.3 Incoming event message related to single processes. . . . . . . . 27
3.4 Message used to start the X-Ray system. . . . . . . . . . . . . . . 27
3.5 Messages used to stop the X-Ray system. . . . . . . . . . . . . . 28
3.6 Messages used to recover the X-Ray system. . . . . . . . . . . . . 28
3.7 Outgoing event message related to the complete X-Ray system. . 29
3.8 Incoming event message related to a timeout. . . . . . . . . . . . . 29
3.9 Parameters of the StartNextLevel process. . . . . . . . . . . . . . 37
3.10 Parameters of the StartSingleLevel process. . . . . . . . . . . . . 40
3.11 Parameters of the SystemStarted process. . . . . . . . . . . . . . 41
3.12 Parameters of the StartSingleLevelTimeoutHandler process. . . 42
3.13 Status of the modal formulas. . . . . . . . . . . . . . . . . . . . . . 46
4.1 Different message types used in the X-Ray controller model. . . 73
4.2 Messages used in the abstract example. . . . . . . . . . . . . . . . 77
4.3 Extra messages added to abstract example. . . . . . . . . . . . . . 80
# List of Listings

2.1 Defining actions ........................................ 7  
2.2 Definition of data types and all there elements .......... 9  
2.3 Definition of the TwoTires data type and an auxiliary function 10  
2.4 Definition of the TwoTires data type and an auxiliary function 11  
2.5 Defining structured data types. .......................... 11  

3.1 Used data types, auxiliary functions and bounds ............ 36  
3.2 The `StartSystem` process of the SSR component ............ 37  
3.3 The `StartNextLevel` process of the SSR component ......... 37  
3.4 The `StartSingleLevel` process of the SSR component ....... 38  
3.5 The `SystemStarted` process of the SSR component .......... 41  
3.6 The `StartSingleLevelTimeoutHandler` process of the SSR com- 
    ponent ........................................ 42  
3.7 Simple inevitability check ................................ 43  
3.8 Complex inevitability check ............................... 44  
3.9 Simple check for requirement 2 ............................ 45  
3.10 Complex check for requirement 2 .......................... 46  
3.11 Simple check for requirement 5 ............................ 48  
3.12 Complex check for requirement 5 .......................... 48  
3.13 Allowed check of requirement 11 ........................... 50  
3.14 Prohibited check of requirement 11 ........................ 50  
3.15 Simple check for requirement 14 ........................... 51  
3.16 Complex check for requirement 14 .......................... 51  
3.17 Check for requirement 15 ................................. 52  
3.18 Toolset information ..................................... 54  
3.19 Operating system and hardware information ................. 54  

4.1 Modeling the reception of external triggers. ................ 65  
4.2 Modeling internal function calls. .......................... 66  
4.3 Modeling class variables. ................................. 67  
4.4 Definitions of the abstract base class `Display` and its derived  
    concrete classes. ................................... 69  
4.5 Using a pointer of the abstract base type. .................. 69  
4.6 Output of Listing 4.5 .................................... 70
4.7 Model containing a pointer of an abstract base class. . . . . . . 70
4.8 Modal formula that checks if an action is used. . . . . . . . . . 73
4.9 Modal formula that checks if an action is not used. . . . . . . . 74
4.10 Modal formula that checks that the reception of a return value
occurs in a blocking fashion. . . . . . . . . . . . . . . . . . . . . . . 74
4.11 Modal formula that checks if a given function is always able to
terminate. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
4.12 Changing settings during a measurement is not allowed. . . . . 76
4.13 It is not allowed to start a measurement when the output cannot
be processed. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
4.14 It is only allowed to start a measurement when at most one
measurement type is active. . . . . . . . . . . . . . . . . . . . . . . 78
4.15 Starting is only allowed when the generated output can be pro-
cessed. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
Bibliography

